RICE UNIVERSITY

Performance Analysis of Symbolic Reachability Algorithms in Model Checking

by

Zijiang Yang

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Dr. Moshe Vardi, Noah Harding Professor,
Chair
Computer Science

Dr. Devika Subramanian, Associate
Professor
Computer Science

Dr. Peter Varman, Associate Professor
Electical and Computer Engineering

Houston, Texas

sometime, 2050

Performance Analysis of Symbolic Reachability Algorithms in Model Checking

By

Zijiang Yang

A THESIS
Submitted in partial fulfillment of the requirements
for the degree of
MASTER OF SCIENCE IN COMPUTER SCIENCE

RICE UNIVERSITY
1999

This thesis, "Performance Analysis of Symbolic Reachability Algorithms in Model Checking", is hereby approved in partial fulfillment of the requirements for the Degree of MASTER OF SCIENCE IN COMPUTER SCIENCE.

Department of Computer Science

Dr. Moshe Vardi, Noah Harding Professor
Chair, Computer Science

Dr. Devika Subramanian, Associate Professor
Computer Science

Dr. Peter Varman, Associate Professor
Electrical and Computer Engineering

Date

Performance Analysis of Symbolic Reachability Algorithms in Model Checking

Zijiang Yang

Abstract

Simulation used to be the most common technique to test the correctness of a system. However, the complexity and size of digital systems has increased drastically during the last few years, which exposes increasingly intractable problems faced by simulation. In order to solve the problems, formal verification was introduced to provide a different approach to validate a system. The approach allows one to reason formally about the behavior of systems. At the early stage, formal verification suffered from a low degree of automation and required a fair amount of knowledge from the designer to exploit the power. Model checking has been proposed to make the verification procedure fully automatic. However, the state explosion problem restricts the application of model checking to small systems. Of the different techniques to solve this state explosion problem, one that provided a major breakthrough was that of using techniques to manipulate states symbolically with *Binary Decision Diagrams*(BDDs).

Symbolic model checking is the most practical approach to formal verification today. However, the algorithms that manipulate BDDs in symbolic model checking are heuristic, which makes the analysis of time and memory requirements very difficult. Since the size of BDDs is closely related with the time and memory requirements of the algorithms in symbolic model checking, research [15, 20, 31] has been conducted on the size of BDDs. Unfortunately, the research fails to provide enough meaningful results. This dissertation proposes a more practical approach to study the performance of BDDs. First we use BDDs to represent sets of states that are randomly chosen from a state space. Then we study how the number of states affects the size of BDDs. Based on our research, we provide a formula to calculate the size of BDDs that representing states randomly chosen from a state space.

Having a better understanding of BDDs, we are able to compare the main algorithms used in symbolic model checking. Roughly speaking, model checking is

performed as an exhaustive search. As the algorithms in reachability computation mature, it gets more and more difficult to get unambiguous algorithmic results; that is, one rarely gets results of the form “Algorithm A dominates Algorithm B”. Rather we get results of the form “sometimes Algorithm A is better and sometimes Algorithm B is better”. Therefore, we need to do experiments to compare symbolic reachability algorithms. In symbolic model checking, there are two kinds of properties: safety properties and liveness properties. We compare different reachability algorithms for checking safety properties, and point out the best approach. We also discuss the current reachability algorithms for checking liveness properties. Based on the discussion, we design a new heuristic algorithm that outperforms the existing algorithms.

Our work reveals that it is absolutely critical for verification tools to have built-in performance instrumentation. The built-in evaluation is important not only for the developers of the tools, but also for the users, who need extensive performance data in order to tailor the verification tool to the design at hand. What is also needed is an understanding of how algorithmic performance depends on various attributes of the design. Currently, such an understanding is sorely lacking, which renders performance optimization of verification tools a black art.

Acknowledgments

I am indebted to Professor Moshe Vardi for giving me the opportunity to join the formal verification group. From him I learned many things from the theory of verification to how to speak English, which I will treasure for the rest of my life. His encouragement and insight of research made the whole period of my working more enjoyable. This work would not have been possible without his constant support and help. Special thanks go to Dr. Kathi Fisler. Her office is always open to me. She has shown me the beauty of a carefully thinking and clearly explained ideas in research. Sincere thanks go to Professor Devika Subramanian and Professor Peter Varman for being my committee members. Also, I want to thank Dr. Jan Hewitt and Ms. Iva Jean Jorgensen for reading my whole thesis and pointing out so many grammar errors.

I am grateful to my wife, Kelly, for sharing with me this experience from beginning to end. For her I have my most sincere love and gratitude.

Contents

Illustrations

Tables

Chapter 1

INTRODUCTION

Conventionly, simulation has been the principal method used in the hardware industry to detect errors in designs and implementations. However, the dramatic growth in complexity of systems has exposed two increasingly intractable problems with simulation. These include creating a simulation regime to expose any logical errors in the design and interpreting the vast quantity of simulation output to determining whether incorrect behavior has occurred. Because of these intractable problems, formal verification was introduced to provide a different way to reason about the correctness of systems.

1.1 Formal Methods

How does formal verification work? In general, a design has a specification for the desired behavior. The implementation obtained at some point in the design process has to be consistent with the original specifications. Therefore, formal verification needs to formally verify that an implementation satisfies the specification. The fact that the verification has to be formal requires that formalism be used to express both the implementation and specification.

In hardware design, an implementation consists of a description of the actual design that is to be verified. There are many description languages such as VHDL and Verilog. The choice depends on which levels of the hardware abstraction hierarchy we want to model. At the circuit level, we need to choose the description language that is able to model such properties as analog behavior, hazard and meta-stable conditions, and different logic families. At the switch level, we need to choose one that is able to model such properties as signal strengths, threshold effects, and propagation delays. At the gate level, we need to choose one that is able to model such properties as composition, hierarchy, and gate delays. Finally, at the register-transfer level, we need to choose one that is able to model such properties as nondeterminism, concurrency, and hierarchy. The choice can compromise the validity of the verification results obtained. For example, a switch level description may not be able to catch analog errors such as the threshold drops.

A specification is a description of the intended behavior of a design. Various formalisms have been used to represent specifications. They are classified in two categories. The first category is logics, which includes propositional logic, first-order logic, higher-order logic, modal logic and mu-calculus. The second one is automata theory. The most popular one being used today to represent specifications is modal logic. Modal logic is able to reason about time without introducing time into the expression explicitly, which enables verification of very important correctness properties such as safety and liveness properties.

Formal verification needs to prove that an implementation satisfies a specification. The forms of proof are listed as follows, with model checking being the most popular one today [13].

- *Theorem-proving.* The relationship between a specification and an implementation is regarded as a theorem in logic, to be proved within the context of a proof calculus, where the implementation provides axioms and assumptions that the proof can draw upon.
- *Model checking.* The specification is in the form of a logic formula, the truth of which is determined algorithmically with respect to a semantic model provided by an implementation.
- *Equivalence checking.* The equivalence of a specification and an implementation is checked.
- *Language containment.* The language representing an implementation is shown to be contained in the language representing a specification.

1.2 Thesis Motivation

Although the capabilities of formal verification in detecting incorrect designs has been proven for a long time, its main drawback was its low degree of automation. It required a fair amount of knowledge from the designer to exploit the power of formal verification.

Model checking has been proposed to make the verification procedure fully automatic. In the original implementation of model checking algorithm, transition relations were represented explicitly by adjacency lists. The model checking systems were able to check state transition graphs with between 10^4 and 10^5 states at a rate about 100 states per second for typical formulas [8]. For concurrent systems with small numbers of processes, the number of states is usually fairly small, and the approach is quite practical. However, from the time in which one person could handle the complexity of a system until today, almost every stage in the design flow has radically changed. Today, almost all systems have many concurrent parts and the number of states is too large to be handled by model checking with explicit transition relations.

Of the different techniques to improve model checking, one that provided a major breakthrough was that of manipulating states symbolically with binary decision diagrams(BDDs) [3, 22]. The complexity of these algorithms depends on the size of BDD representation of the transition relation, and the length of the logic formulae that express the specifications. BDDs provide a canonical form for Boolean formulas that is often substantially more compact than conjunctive or disjunctive normal form. Because the symbolic representation captures some of the regularity in the state space determined by the designs, it is possible to verify systems with a large number of states.

However, most algorithms in symbolic model checking are heuristic. After submitting the formal description of a design and some desired properties to currently available verification tools, one cannot predict how long it will take to get any results. Some time a large design may take several minutes to be verified, while some time a relatively small design may take hours. Even worse, some designs are reported to be non-testable at the end after a long wait because of limited resources such as inadequate memory. The main reason behind this is little knowledge about the size of BDDs. For example, given the number of states that need to be represented by a BDD, no theories are known to tell the size of the BDD until it is built. Since the size of BDDs is directly related to the time and memory requirements of the algorithms in symbolic model checking, we cannot predict the time and memory complexity of those algorithms until we have a good knowledge of the size of the BDDs. Getting a better understanding of the size of BDDs is one of the primary goals of our research.

The other goal is to analyze the algorithms for reachability computations that are based on BDD manipulation. Roughly speaking, model checking is performed as an exhaustive search. As the algorithms for reachability computations mature, it gets more and more difficult to get unambiguous algorithmic results; that is, one rarely gets results of the form "Algorithm A dominates Algorithm B". Rather, we get results of the form "sometimes Algorithm A is better and sometimes Algorithm

B is better". What is needed is an understanding of how algorithmic performance depends on various attributes of BDDs. Currently, such an understanding is sorely lacking, which makes performance optimization of verification tools very difficult.

1.3 Thesis Overview

Chapter 2 lays the groundwork for the whole thesis. The basic concepts and algorithms in model checking are reviewed. Different approaches used in the finite state machine traversal, the key algorithm for model checking, are discussed. Then the data structure and relevant algorithms of Binary Decision Diagrams are recalled, which enables further discussion on how to implement the finite state machine traversal algorithm symbolically.

The first step of most verification tools in model checking is to translate the design description languages into transition relations represented by BDDs. We avoid this step for the following two reasons. First, different description languages favor designs in different fields. For example, the hardware description language Verilog may not be suitable to describe protocols. If we choose Verilog as our front-end, our results may not be valid for a different front-end. Second, we don't need to delve into the vast details of description languages. Therefore, we introduce our own general model for transition relations and relative theories about that model in Chapter 3.

In symbolic model checking, BDDs are used to represent sets of states and transition relations. However, few theories are known on the study of BDDs. In Chapter 4, we introduce the approaches we use in our research about the size of BDDs and state relative results.

In Chapter 5, the dynamic properties on the size of BDDs are studied. We use a symbolic traversal algorithm to compute the states reachable from an initial set. We offer an empirical formula to calculate the number of iterations needed in reachability computation. We also offer measurements of the size of BDDs during the reachability computation.

Two very important correctness properties in practical applications are safety properties and liveness properties. In Chapter 6, the safety property is appropriately modeled and related BDD behaviors are measured. In Chapter 7, the liveness property is studied. We recall three heuristic algorithms, Emerson-Lei, Catch-Them-Young and One-Way-Catch-Them-Young, and compare their efficiency in the view of BDD usage.

Finally, Chapter 8 is devoted to concluding remarks.

Chapter 2

Model Checking

Research in formal verification falls within various categories [19]. The most general form is this: given two formulae f, g in a reasonably expressive logic, each representing models or properties, prove "$f \Rightarrow g$". This most general form has been studied since before 1960, under the name automated theorem proving. Automated theorem proving models the system and the specification in a suitable logic, and derives a proof that the original system implies the specification. This method is known to be very powerful. However, most often the question "$f \Rightarrow g$" in a general context is undecidable. Thus, the automated theorem proving method requires extensive user involvement.

Model checking is the most practical approach to formal verification today. It is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. Typically, after users provide a high level description of the model and the specifications to be checked, the model checking algorithm will either terminate with a true answer or give a counter-example that shows why the specification is not satisfied. The procedure shows that model checking has two important advantages over automated theorem proving. First, the procedure of model checking is highly automated. There is no user involvement after the checking has begun. Second, model-checkers automatically produce error tracks when the property being checked fails.

The rest of this chapter is organized as follows: we first use a simple example to show how model checking works in Section 2.1. Then in Section 2.2 we talk about the basic algorithms used in reachability analysis for model checking. In Section 2.3 we briefly review the data structure of Binary Decision Diagrams. And last in Section 2.4 we discuss how to integrate the symbolic methods in model checking.

2.1 What is Model Checking?

Two general approaches to model checking are used in practice [6]. The first is called *temporal logic model checking.* In this approach, specifications are expressed in a temporal logic and the systems are modeled as finite state transition systems. A search procedure is used to check if the finite state transition system P is a model for a given specification T, *i.e.*, $P \models T$. The second one is called *language containment.* In this approach, both the specification and the system are modeled as automata. Given a system modeled by automaton P and a specification modeled by automaton T, model checking in this context consists of the automaton language containment test: $\mathcal{L}(P) \subset \mathcal{L}(T)$. Vardi and Wolper [30] have shown how the temporal logic model checking problem could be formulated in terms of language containment between automata, thus relating the two model checking approaches.

Following is an example using the approach of temporal logic model checking. The meaning of a temporal logic formula is determined by a labeled state-transition graph called a *Kripke structure* [17]. A Kripke structure is defined as $K = \langle S, R, L \rangle$, where S is the set of states; $R \subseteq S \times S$ is the transition relation; and $L : S \to \mathcal{P}(AP)$ is a function that labels each state with a set of atomic propositions true in that state. $K, s \models f$ means that the formula f is true in state s of model K. Figure 2.1 shows a Kripke structure, in which $L(s_1) = \{p\}$ and $L(s_i) = \emptyset$, where $0 \leq i \leq 6, i \neq 1$. We want to check whether $K, s_6 \models \mathbf{EF}p$, where $\mathbf{EF}p$ is a *computation tree logic*(CTL) formula. CTL is a subset of modal logic defined by Clark and Emerson [5]. It is currently the most popular form of temporal logic for formal hardware verification with model checking. EFp specifies that the property p will hold at some state on a computation path. $K, s_0 \models \mathbf{EF}p$ if and only if there exists a path $q = (s_0, s_1, ...)$ in K such that $p \in L(s_i)$, where s_i is a state on q. Similarly, $K, s_0 \models \mathbf{EX}p$ if and only if there exists a path $q = (s_0, s_1, ...)$ in K such that $p \in L(s_1)$. For more detailed discussion on temporal logic, see [11].

Given a formula f, the model checking algorithm [8] operates by labeling each state s with the set $label(s)$, which consists of sub-formulae of f that are true in s. Initially, $label(s)$ is just $L(s)$. At the ith iteration, sub-formulae with $i-1$ nested CTL operators are processed. When a sub-formula is processed, it is added to the labeling of the states in which the sub-formula is true. At the end of the algorithm, $K, s \models f$ if and only if $f \in label(s)$. This procedure is guaranteed to terminate because Kripke structures are finite. As to our example, $\mathbf{EF}p$ is the only non-atomic sub-formula.

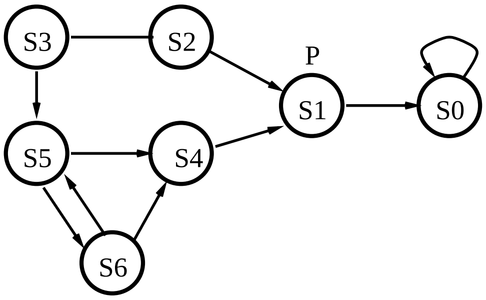


Figure 2.1 A Kripke structure K.

The procedure of labeling is based on the following formula: $\mathbf{EF}p = p \vee \mathbf{EXEF}p$. The algorithm labels s_1 first because $L(S_1) = \{p\}$. Next it labels s_2 and s_4 simultaneously since $K, s_2 \models \mathbf{EX}p$ and $K, s_4 \models \mathbf{EX}p$. The algorithm proceeds by labeling s_3, s_5 and s_6. Since s_6 is labeled, $K, s_6 \models \mathbf{EF}p$. The number of iterations is less than or equal to the diameter of K. It is observed that reachability analysis plays a central role in the example shown above. The technical challenge in model checking is in devising algorithms and data structures that allow us to handle large search spaces during reachability analysis [6]. In the following sections we recall the algorithms and data structure used in reachability analysis today.

2.2 Finite State Machine Traversal

Since the model in model checking is finite, we shall concentrate on finite state machines(FSM), for which the reachability analysis is called FSM traversal. We define a finite-state machine(FSM) as a 6-tuple $\langle I, O, \lambda, S, \delta, S^0 \rangle$, where I and O are the input and output alphabet respectively, λ is the output function mapping I to O, S is the set of states, δ is the transition function mapping $S \times I$ to S and S^0 is the set of initial states. A state s_k is reachable from state s_0 if there exists a sequence of states $s_1, ..., s_{k-1}$ and a sequence of inputs $i_0, ..., i_{k-1}$, such that $s_l = \delta(s_{l-1}, i_{l-1})$, where $0 \leq l \leq k$. The problem of traversing a FSM is to enumerate all of its reachable states. Since in this problem we are only concerned with the transition function δ, we ignore the output function λ in the rest of this dissertation.

We can derive a simple algorithm for the traversal of a FSM based on depth-first search. For simplicity, we initially assume that there is a single initial state, $S = \{s_0\}$. In this algorithm, we need to take care of two state sets and a single state: a set of states that have been reached, a set of states that need to be processed, and a current state. The set of states to be processed are stored in a stack. Initially, the set of states that need to be processed is s_0, the set of reached states is empty, and the current states is s_0.

The processing of the current state s proceeds as follows. First, remove the current state s from the stack and add it to the set of reached states. Then evaluate $\delta(s, i)$ for all possible $i \in I$. This operation can be interpreted as finding all the states that can be reached from s in one step. For a state $s_j \in \delta(s, i)$, push it onto the stack of the states to be processed if s_j has not been reached previously. Once the current state has been processed, a new current state is selected from the top of the stack. This procedure is continued until the stack is empty.

If there are several initial states, the procedure of the first initial state is the same as the one described above. For the remaining initial states, the set of reached states is initialized to the result of processing the previous initial state, so that only states that were not reached from previous initial states are considered.

A simple modification to the depth-first traversal by replacing the stack in the depth-first traversal with a queue leads to a breadth-first traversal. In the original implementation of the algorithms, the FSM was represented explicitly by adjacency lists. For concurrent systems with small numbers of processes, the number of states was usually small, and the approach was quite practical. However, the number of states in a FSM increases exponentially with the number of parallel components in the system that is modeled by the FSM, resulting in a so-called *state explosion problem*. This restricts the application of explicit state enumeration approaches to small circuits. Of the different techniques to solve this state explosion problem, one that provided a major breakthrough was that of manipulating states symbolically with *Binary Decision Diagrams*. The utilization of Binary Decision Diagrams facilitates the manipulation of Boolean operations. Since the domain of such operations is a set of states rather than an individual state, the space requirement of the algorithm is not determined by the state space, but by the underlying Boolean functions. We will recall the data structure of Binary Decision Diagrams in the next section.

2.3 Binary Decision Diagrams

Binary decision diagrams (BDDs) are graph representations of Boolean functions. Bryant [3, 24] introduced the BDD in its current norm, although the basic concepts was introduced by Akers [1]. BDDs have several useful properties. First, many common functions have small BDDs. Second, efficient algorithms exist for all the usual Boolean operations as well as other useful operations. Finally, a BDD is a canonical representation for a Boolean function once the ordering of the variables is fixed. In this chapter, we briefly review the data structure of BDDs and the algorithms performed on BDDs.

2.3.1 Data Structure

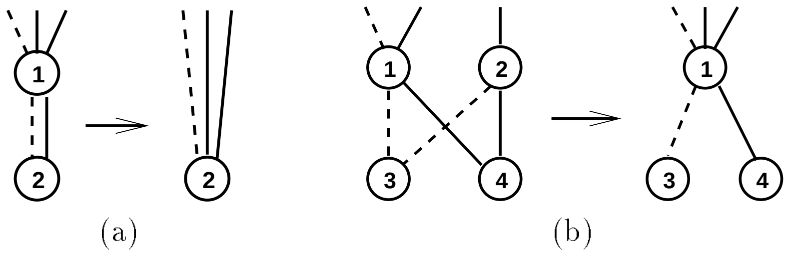


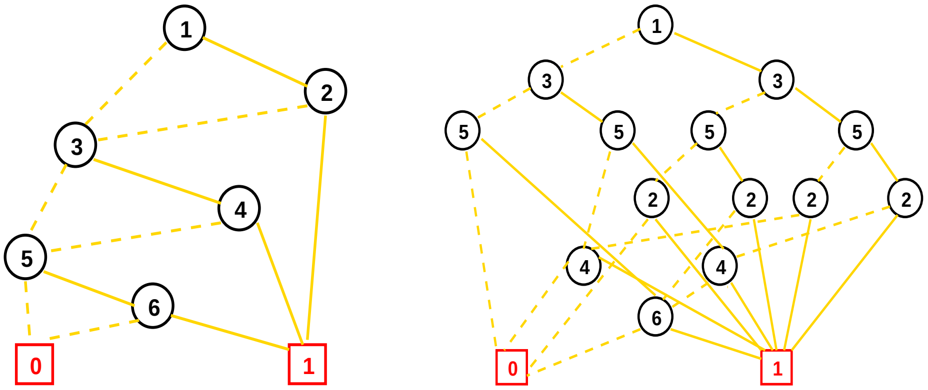
Figure 2.2 Rules for reducing a binary tree:
(a)Deleting node 1. (b)Merging nodes 1 and 2.

A BDD is a directed acyclic graph that consists of terminal and nonterminal nodes. There are only two terminal nodes, which are called the *0-terminal* and the *1-terminal*. Every non-terminal node has two outgoing edges, which are called the *0-edge* and the *1-edge*. In this dissertation, the 0-edges are drawn by dotted lines and the 1-edges are drawn by solid lines. Conceptually, we can build the BDD for a Boolean function as follows. First, construct a decision tree for the function, which obeys the requirements that no variable appear more than once along any path from root to leaf, and the variables always appear in the same order. Next, as shown in Figure 2.2, apply the following two rules as much as possible.

1. *Deletion rule:* Eliminate all the redundant nodes, which are those whose two edges point to the same node.

2. *Merging rule:* Share all the equivalent sub-graphs.

In practice, the decision trees are never built and BDDs are always manipulated in the reduced form.



(a) (b)

Figure 2.3 BDDs representing $x_1x_2 + x_3x_4 + x_5x_6$ with different ordering: (a)$x_1 < x_2 < x_3 < x_4 < x_5 < x_6$. (b)$x_1 < x_3 < x_5 < x_2 < x_4 < x_6$.

There is a strict total order placed on the occurrence of variables as one traverses BDDs from root to leaf. BDDs with fixed variable order are canonical, which means that BDDs represent a function uniquely. A canonical form is desirable because it makes equivalence tests easy. To see the impact of variable orders on the size of BDDs, consider $f = x_1x_2 + x_3x_4 + x_5x_6$ with variable order $x_1 < x_2 < x_3 < x_4 < x_5 < x_6$. The BDD representing f with this particular variable order is shown in (a) of Figure 2.3. The numbers in circles are the variable indices and values in the square represent the 0-terminal and the 1-terminal. Now let's consider a different variable order $x_1 < x_3 < x_5 < x_2 < x_4 < x_6$. The resulting BDD, shown in (b) of Figure 2.3, is considerably much bigger. Generalizing f to the function $x_1x_2 + ... + x_{2k-1}x_{2k}$ with $2k$ variables, the size of the BDD is $2k+2$ with variable order $x_1 < x_2 < ... < x_{2k-1} < x_{2k}$ and 2^{k+1} with variable order $x_1 < x_{k+1} < x_2 < x_{k+2} < ... < x_k < x_{2k}$. Therefore, the choice of variable order can make the difference between a linear size BDD and an exponential one.

We can get a better intuition on why variable orders affect the size of BDDs by checking the two BDDs in Figure 2.3. In the first ordering, one product term is considered at a time. After examining the first two variables, it is known whether the first product term $x_1 \cdot x_2$ is 0 or 1. If the result is 1, we are finished. Otherwise we continue without caring about the value of x_1 or x_2. Since there is not much to remember, the BDD has very few nodes. In the second ordering, all product terms are processed in parallel. None of the product term values is known if the first three variables are all 1's. In the worst case, the value of x_1 has to be remembered until x_2 is met. The value of x_3 has to be remembered until x_4 is met. The value of x_5 has to be remembered until x_6 is met.

Although good choices of variable order reduce the size of BDD representing certain Boolean function dramatically, some functions cannot be represented efficiently with BDD regardless of the variable ordering. For example, Bryant [3] shows that for any variable ordering of $A = (a_1, ..., a_k)$ and $B = (b_1, ..., b_k)$, at least one of the $2k$ functions representing the integer product of $A \cdot B$ requires a BDD of size $2^{k/8}$, which indicates the exponential complexity of these functions.

2.3.2 Fundamental Algorithms

The usual way of building new BDDs for a given Boolean function is to combine existing BDDs. At the starting point, we create a BDD for a single variable. Such a BDD has only one nonterminal node and the two terminal nodes. Then we are interested in an algorithm that, given BDDs for f and g, builds the BDD for $f \circ g$, where $\circ$ is a binary operator such as AND or OR. The key idea for efficient implementation of the algorithm is the expansion theorem:

$$f \circ g = (\neg x \cdot (f|_{x \leftarrow 0} \circ g|_{x \leftarrow 0})) + (x \cdot (f|_{x \leftarrow 1} \circ g|_{x \leftarrow 1})),$$

where x is the lowest indexed variable in f and g. It creates a new node with the variable x and two sub-graphs generated by $(f|_{x \leftarrow 0} \circ g|_{x \leftarrow 0})$ and $(f|_{x \leftarrow 1} \circ g|_{x \leftarrow 1})$. The operation $f|_{x \leftarrow 0}$ or $f|_{x \leftarrow 1}$ is called *restriction*. If f does not depend on x, f itself is returned. Otherwise the function pointed to by the 0-edge or 1-edge is returned.

The approach based on the expansion theorem can be improved by considering the ternary operator, *if-then-else*, defined as follows:

$$t \rightarrow f, g = t \cdot f + \neg t \cdot g,$$

where t, f, g are arbitrary Boolean expressions. By using the following two rules, any Boolean expression f can be represented by an equivalent expression built entirely from the *if-then-else* operator:

1. if f contains no variables, it is either equivalent to 0 or 1;

2. $f = x \to f|_{x\leftarrow 1}, f|_{x\leftarrow 0}$, where x is a variable of f. Since $f|_{x\leftarrow 0}$ and $f|_{x\leftarrow 1}$ both contain one less variable than f, we can recursively use the Shannon expansion until rule 1 can be applied.

Another property of the ternary operator is that all two-argument operators can be expressed in terms of *if-then-else*, as shown in table 2.1. We can get a recursive algorithm for the ternary operator *if-then-else* based on the following equation, where x is the Boolean variable with the lowest index of t, f, g.

index	expression	*if-then-else* form
1	0	0
2	$f \cdot g$	$f \to g, 0$
3	$f \cdot \neg g$	$f \to \neg g, 0$
4	f	f
5	$\neg f \cdot g$	$f \to 0, g$
6	g	g
7	$f \oplus g$	$f \to \neg g, g$
8	$f + g$	$f \to 1, g$
9	$\neg(f + g)$	$f \to 0, \neg g$
10	$\neg(f \oplus g)$	$f \to g, \neg g$
11	$\neg g$	$g \to 0, 1$
12	$f + \neg g$	$f \to 1, \neg g$
13	$\neg f$	$f \to 0, 1$
14	$\neg f + g$	$f \to g, 1$
15	$\neg(f \cdot g)$	$f \to \neg g, 1$
16	1	1

Table 2.1 Two-argument operators expressed in *if-then-else* form.

$$t \to f, g = x \to (t|_{x\leftarrow 1} \to f|_{x\leftarrow 1}, g|_{x\leftarrow 1}), (t|_{x\leftarrow 0} \to f|_{x\leftarrow 0}, g|_{x\leftarrow 0}).$$

The terminal cases are $1 \to f, g$; $0 \to g, f$; $f \to 1, 0$; $g \to f, f$. All of them are equivalent to f.

The other operations which we require for the reachability computation are quantification over Boolean variables and substitution of variable names. They can be handled by the operations introduced above. The evaluation of existential quantification $\exists v[f]$, where v is a Boolean variable and f is a formula, can be done by two restriction operations and one binary operation:

$$\exists v[f] = f|_{v\leftarrow 0} \vee f|_{v\leftarrow 1}.$$

Similarly, the universal quantification $\forall v[f]$ is equivalent to $f|_{v\leftarrow 0} \wedge f|_{v\leftarrow 1}$. The substitution of a variable u for a variable v in a formula f, denoted by $f|_{v\leftarrow u}$, can be accomplished by using existential quantification:

$$f|_{v\leftarrow u} = \exists v[(v \leftrightarrow u) \wedge f].$$

2.4 Symbolic Methods

The approach that uses BDDs in the model checking algorithm is called *symbolic model checking* [22]. In symbolic model checking, a system is modeled as a state transition relation that is represented implicitly by BDDs. In the following subsection, we discuss how to compute the set of states reachable from a set efficiently. Then in subsection 2.4.2 we provide an algorithm for reachability computation by using symbolic methods.

2.4.1 Image Computation

The evaluation of states reachable in one step from another set of states is called *image computation*. Similarly, the evaluation of states that can reach another set of states in one step called *preimage computation*. The set of states that are processed is represented by their characteristic functions.

> **Definition 2.1** Let $f : B^n \to B^m$ be a multiple-output Boolean function. The image of f, denoted by $Img(f)$, is defined by:
>
> $$Img(f) = \{v \in B^m : \exists x \in B^n, v \in f(x)\}.$$

According to the definition, if v belongs to the image of f, then there is at least one input x such that $v \in f(x)$. We may want to constrain the inputs that can be applied to f and thus restrict the output that f can produce, so we introduce the following definition.

> **Definition 2.2** Let $f : B^n \to B^m$ be a multiple-output Boolean function and let C be a subset of B^n. The image of f constrained by C, denoted by $Img(f, C)$, is defined by:
>
> $$Img(f, C) = \{v \in B^m : \exists x \in C, v \in f(x)\}.$$

C is called the *constraint set* and the evaluation of $Img(f, C)$ is called *constrained image computation*. In the traversal of a FSM, the constraint set C represents a set of current states and $Img(f, C)$ is the set of states that can be reached in one step from C. We illustrate how to use BDDs to perform image computations by a simple example. Figure 2.4 shows a digraph $D = (V, E)$, where $|V| = 8$ and $|E| = 10$. Since there are eight vertices, we need three binary bits to represent each vertex indexed

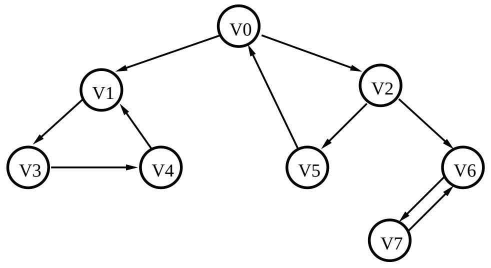


Figure 2.4 A digraph $D = (V, E)$ with $|V| = 8$ and $|E| = 10$.

from 000 to 111. Let Boolean variable x_0 represent the most significant bit and x_2 represent the least significant bit, the edge from v_0 with index 000 to v_1 with index 001 can be represented by a transition relation shown below, in which $t_0(v_0, v_1) = 1$ if and only if $(v_0, v_1) \in E$.

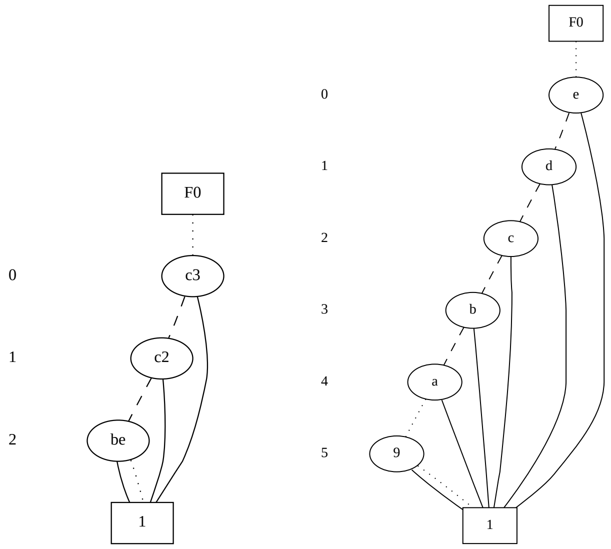
$$t_0(v_0, v_1) = \neg x_0 \cdot \neg x_1 \cdot \neg x_2 \cdot \neg x'_0 \cdot \neg x'_1 \cdot x'_2$$

We need to rename the Boolean variables($x_i \rightarrow x'_i$) for v_1 to indicate that the source vertex and destination vertex are in different name spaces in the transition relation. Otherwise we get $t_0 \equiv 0$. Similarly, we get the transition relations t_1,t_2...t_9 for the remaining edges. The Boolean function for the edge relation, which consists of all of the directed edges in the digraph, is the disjunction of the transition relations for each individual edge:

$$T(S, S') = \Sigma_{i=0}^{9} t_i.$$

A BDD representing $V_0 = \{v_0\}$ is shown in (a) of Figure 2.5 and a BDD representing $t_0(v_0, v_1)$ is shown in (b). The BDDs are generated by the software package CUDD [26]. The label of each node is the unique part of the node address. All nodes on the same level correspond to the same variable, whose index is shown at the left of the diagram. Solid lines and dashed lines indicate 1-edges and 0-edges respectively. Dotted lines indicate special 0-edges with attribution complement, which enable constant time for performing negation. Given BDDs for each transition relation t_i, it is not difficult to get the BDD, shown in Figure 2.6, for the transition relation $T(S, S')$ by using the algorithm introduced in Section 2.3. For $v_i, v_j \in V$, $T(v_i, v_j) = 1$ if and only if $(v_i, v_j) \in E$.

Let S_0 be a set of states that is represented by a BDD $S_0(X)$. We wish to compute a BDD $S_1(X)$ that represents the set S_1 that can be reached from S_0 in one step via the transition relation T. S_1 can be obtained by



(a) (b)

Figure 2.5 BDDs representing (a)vertex v_0, and (b) edge (v_0, v_1) in D.

$$S_1 = S_0 \cup \{s' | \exists s (s \in S_0 \wedge (s, s') \in T)\}.$$

We can compute the BDD $S_1(X)$ by performing the logical operations corresponding to the above expression:

$$S_1(X') = S_0(X') \vee \exists_{x \in X} (S_0(X) \cdot T(X, X')),$$

where the existential quantifier notation above indicates the existential quantification of all variables x in X. Figure 2.7 shows the algorithm for computing $\exists_{x \in X}(S(X) \cdot T(X, X'))$ proposed by Burch *et al.* [4]. Like many other BDD algorithms, *RelProd* uses a result cache. The keys for the cache are the pairs (S, T). If an entry (S, T, R) exists, R will be returned as the result of the previous call to *RelProd*(S, T). $Or(R_0, R_1)$ returns the binary operation $R_0 \vee R_1$, and $IfThenElse(z, R_0, R_1)$ returns the result of $(z \wedge R_1) \vee (\neg z \wedge R_0)$. In general, a BDD representing the states reachable in at most k steps from S_0 can be computated by:

$$S_k(X') = S_{k-1}(X') \vee \exists_{x \in X}(S_{k-1}(X) \cdot T(X, X')).$$

Note that S_j is a superset of S_i if $0 \leq i < j$. Since the total number of states is finite, there exists a number k such that $S_{k+1} = S_k$, and therefore for every $k' > k$, $S_{k'} = S_k$ will hold. The computation for the reachable states can be viewed as finding the *least fixed point*. Fixed points are a very important concept in symbolic model checking, and will be discussed in the following subsection.

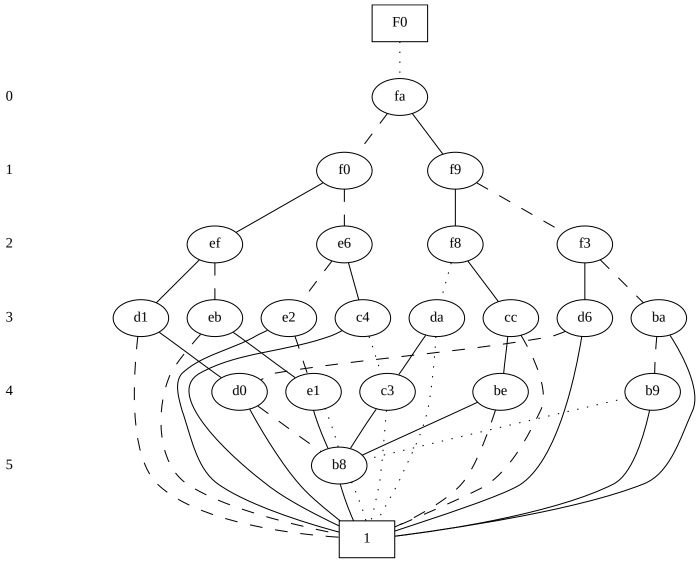


Figure 2.6 The BDD representing the transition relation for digraph D.

2.4.2 Fixed Points

Symbolic breadth-first traversal on finite set can be characterized as a fixed point computation. We first describe the characterization of fixed points, then show how to employ the properties of fixed points to compute reachable states.

Let $K = (S, R, L)$ be an arbitrary finite Kripke structure. The set $\mathcal{P}(\mathcal{S})$ denotes all subsets of S. We can identify every logical formula with an element of $\mathcal{P}(\mathcal{S})$. The logical constant *true* is identified with S while *false* is identified with the empty set. A function that maps $\mathcal{P}(\mathcal{S})$ to $\mathcal{P}(\mathcal{S})$ is called a *predicate transformer*. Let $\tau : \mathcal{P}(\mathcal{S}) \to \mathcal{P}(\mathcal{S})$ be such a function. Functions are written as $\lambda y.f$, where f is a Boolean formula, and y is a variable.

> **Definition 2.3** A fixed point of a function τ is any logical formula p such that $\tau(p) = p$.

For example, $x \wedge y$ is the fixed point of $\tau = \lambda y.(x \wedge y)$ since $\tau(x \wedge y) = x \wedge (x \wedge y) = x \wedge y$.

Definition 2.4 [8]

1. τ is monotonic provided that $p \subseteq q$ implies $\tau(p) \subseteq \tau(q)$;
2. τ is $\cup$-continuous provided that $p_1 \subseteq p_2 \subseteq \ldots$ implies $\tau(\cup_i p_i) = \cup_i \tau(p_i)$;

```
FUNCTION RelProd(S,T: BDD): BDD
IF $S$ = false $\vee$ $g$ = false
   RETURN false
ELSE IF $S$ = true $\wedge$ $g$ = true
   RETURN true
ELSE IF (S,T,R) is in the result cache
   RETURN R
ELSE
   let $x$ be the top variable of $S$
   let $y$ be the top variable of $T$
   let $z$ be the top variable of $x$ and $y$
   $R_0$ := RelProd($S|_{z=0}$, $T|_{z=0}$)
   $R_1$ := RelProd($S|_{z=1}$, $T|_{z=1}$)
   IF $z$ is in the current state space
      R:= Or($R_0$,$R_1$)
   ELSE
      R:= IfThenElse($z$,$R_0$,$R_1$)
   ENDIF
   Insert(S,T,R) in the result cache
   RETURN R
ENDIF
```

Figure 2.7 Pseudocode for image computation.

3. τ is $\cap$-continuous provided that $p_1 \subseteq p_2 \subseteq \ldots$ implies $\tau(\cap_i p_i) = \cap_i \tau(p_i)$;

One property of monotonic functions is they have a *least fixed point*(lpf) and a *greatest fixed point*(gpf), which are defined as follows:

> **Definition 2.5** [27] The intersection of all the fixed points of a monotonic function is called the least fixed point. Similarly, the union of all the fixed points of a monotonic function is called the greatest fixed point. The least and greatest fixed points of $\lambda y.f$ are denoted $\mu y.f$ and $\nu y.f$ respectively.

Clarke *et al.* [8] proved the following lemmas about fixed points:

> **Lemma 2.1** If S is finite and τ is monotonic, then τ is also $\cup$-continuous and $\cap$-continuous.

Lemma 2.2 If τ is monotonic, then for every i, $\tau^i(false) \subseteq \tau^{i+1}(false)$ and $\tau^i(true) \supseteq \tau^{i+1}(true)$.

Lemma 2.3 If τ is monotonic and S is finite, then there is an integer i_0 such that for every $j \geq i_0$, $\tau^j(false) = \tau^{i_0}(false)$. Similarly, there is some j_0 such that for every $j \geq j_0$, $\tau^j(true) = \tau^{j_0}(true)$.

Lemma 2.4 If τ is monotonic and S if finite, then there is an integer i_0 such that $\mu y.\tau(y) = \tau^{i_0}(false)$. Similarly, there is some j_0 such that $\nu y.\tau(y) = \tau^{j_0}(true)$.

Now we are able to implement the reachability algorithm as shown in Figure 2.8 by using BDDs as its data structure. The algorithm is based on breadth-first search because this method allows us to deal naturally with multiple states simultaneously. When multiple states are processed simultaneously in breadth-first manner, the algorithm is called *symbolic breadth-first traversal.* The number of iterations performed is called the *sequential depth.*

```
PROCEDURE traverse($\delta, S_0$)
1. reached_set := $S_0$;
2. current_set := $S_0$;
3. REPEAT
4.     reached_one_step := $Img(\delta, current\_set)$;
5.     reached_new := reached_one_step - reached_set;
6.     current_set := reached_new;
7.     reached := reached $\cup$ reached_new
8. UNTIL(reached_new = NULL)
```

Figure 2.8 Pseudocode for symbolic breath-first traversal.

Note that the *difference* of sets on line 5 and the *union* of sets on line 7 of Figure 2.8 are actually performed by manipulating the corresponding characteristic functions. The meaning of the assignment *current_set* := *reached_new* on line 6 is obvious: the newly reached states will be considered as the current states at the next iteration. However, the newly reached states is not the only choice for the current states. The *current_set* can be any set S_i such that

$$reached\_new \subseteq S_i \subseteq reached\_set \cup reached\_new. \tag{2.1}$$

If we choose *current_set* to be a superset of *reached_new*, then some states will be checked more than once. In other words, some states will be put into the queue more than once. In a pure breath first search algorithm, choosing a superset would be a bad idea. However, we are manipulating characteristic functions via BDDs instead of individual states. The size of BDDs and the complexity of the operations on them are only loosely related to the number of states. We will show in the next chapter that it is not rare for a BDD representing more states to be smaller. Coudert, Berthet and Madre [9] introduced an algorithm to choose a set S_1 with small BDD representation such that $S_2 - S_0 \subseteq S_1 \subseteq S_2$. The algorithm takes two BDDs $S(X)$ and $C(X)$ as input and outputs a BDD $S'(X)$ satisfying two requirements. The first one is $S(X) \wedge C(X) = S'(X) \wedge C(X)$. We may interpret the requirement as both $S(X)$ and $S'(X)$ agree for the states in $C(X)$. The second requirement is that the size of the BDD $S'(X)$ is smaller or equal to that of $S(X)$. Intuitively, these requirements can be fulfilled by adding or removing the states not in C. In our case, we may use the complement of *reached_set* in Equation 2.1 as the care set. Note that this improvement doesn't reduce any memory usage because all of the BDDs in the original reachability algorithm are still computed. The potential advantage is that smaller BDDs are used in image computations, which reduces the computation time.

Chapter 3

Random Digraphs

Using BDDs in model checking is efficient only in a sense of heuristic. The only claim that is made for BDDs is that they perform well for certain classes of Boolean functions. Some Boolean functions have small BDD representation, while others do not. Therefore, in order to evaluate the symbolic methods in model checking, we need empirical results showing the performance of the methods. In this dissertation, we evaluate reachability computation, the key problem of model checking, by experiments.

We can view designs as a digraph. The state space of a design corresponds to the vertex space, and there is a directed edge from u to v in the digraph if and only if there is a transition relation from u to v in the design. Designs in different areas have different patterns. In order to fully understand BDD behaviors in reachability computation, we need to build our experiments on general models that will not favor designs in any particular area. Therefore we choose random digraphs as our underground model. In the rest of this chapter, we first describe different models for random graphs and digraphs, then introduce relevant theorems on the model that we choose for our work.

3.1 Models

Let $V = \{v_1, ..., v_n\}$ be a finite set of vertices, and let E be a subset of unordered pairs of vertices. A graph G consists of an ordered pair (V, E) of sets of vertices and edges. The number n is called the *order* of G, and $|E|$ is the *size* of G.

Palmer [23] introduced three different probability models for random graphs. These include Models A, B and C that are defined as follows.

> **Definition 3.1** Model A: For each positive integer n and each number p with $0 < p < 1$, the sample space Ω consists of all labeled graphs G of order n. If G has q edges, the probability of G is given by $P(G) = p^q(1-p)^{\binom{n}{2}-q}$.

There are $\binom{n}{2}$ edge slots in a graph with order n. If we select each individual edge independently with probability p, then the probability of a certain graph with q edges is $P(G)$. For this reason, p is called the probability of an edge.

> **Definition 3.2** Model B: For each positive integer n and integer q between 0 and $\binom{n}{2}$, the sample space Ω consists of all labeled graphs G of order n and size q. The probability of G is given by
>
> $$P(G) = \binom{\binom{n}{2}}{q}^{-1}.$$

Model B has a clear meaning. There are $\binom{\binom{n}{2}}{q}$ different ways to pick out q edges if there are $\binom{n}{2}$ possible edges. Therefore, the probability of a particular graph with q edges, if chosen uniformly from the sample space, is $\binom{\binom{n}{2}}{q}^{-1}$. The third model is for digraphs.

> **Definition 3.3** Model C: For each positive integer n and r with $0 < r < 1$, the sample space Ω consists of all labeled graphs G of order n in which each vertex has out-degree r. Each digraph D is given the probability $P(D) = \binom{n-1}{r}^{-n}$.

Given a vertex v, there are $\binom{n-1}{r}$ possibilities to choose r adjacent vertices of v. If we go through all the n vertices, then there are $\binom{n-1}{r}^{n}$ digraphs in the sample space. Therefore, the probability of a particular digraph with fixed out-degree r is $\binom{n-1}{r}^{-n}$, provided the digraph is chosen uniformly.

The three probability models for random graphs introduced above are good candidates for our work. However, none of them are feasible. For the first two models, orientationless edges make them incompatible with the nature of transition relation in designs. As to Model C, uniform out-degree makes it too narrow to be a general model. The model we choose, Model D, is very similar to Model A, except that the edges have orientation.

> **Definition 3.4** [18] Model D: For each positive integer n and each number p with $0 < p < 1$, the sample space Ω consists of all labeled digraphs $D_{n,p}$ of order n. If $D_{n,p}$ has q edges, the probability of $D_{n,p}$ is given by $P(D_{n,p}) = p^q(1-p)^{n(n-1)-q}$.

Note that the difference between $P(D_{n,p})$ and $P(G)$ in Model A reflects the fact that there are twice as many edges in Model D than in Model A. Some important results based on Model D are described in the next section.

3.2 Relevant Theorems

Because all the experiments in this dissertation are performed on random digraphs drawn from $D_{n,p}$, we provide some important theorems on the random digraphs in this section. All the results are asymptotic in the sense that they show that certain events hold almost certainty in the limit as the digraph order n tends to infinity, with p varying as a function of n.

Let $X(r)$ be the set of vertices reachable from vertex r in a digraph and $Y(r)$ be the set of vertices that can reach r. Since a vertex is reachable and can reach itself, $r \in X(r) \cap Y(r)$. When np remains equal to a constant $d > 1$ as n tends to infinity, $D_{n,p}$ is written as $D_{n,d/n}$ and d is considered as the *density* of digraphs. The set of vertices reachable from r is called *large* if the size of $X(r)$ is about Θn, where Θ is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. $X(r)$ is small if $|X(r)|$ is less than $B \log n$, where B is a constant. The following theorem demonstrates an important gap phenomenon: $X(r)$ is likely to be either very large or very small in $D_{n,d/n}$.

> **Theorem 3.1** [18] Let d be a constant greater than 1. Let a be a positive constant. Let B be a constant greater than $(a+1)d(d-1)^{-2}$. Let $w(n)$ be a nondecreasing unbounded function. Let Θ be the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. Let D be drawn from $D_{n,d/n}$. Then for sufficiently large n, $Pr[|X(r)| \notin ([0, B \ln n] \cup [\Theta n - w(n)\sqrt{n}, \Theta n + w(n)\sqrt{n}])] < n^{-a}$. The probability that $|X(r)|$ lies in the interval $[\Theta n - w(n)\sqrt{n}, \Theta n + w(n)\sqrt{n}]$ tends to Θ as n tends to infinity.

The following theorem shows that the probability that $X(r)$ is small depends only on d. Also, the expected size of $X(r)$ when $X(r)$ is small tends to be a constant that depends only on d, too.

> **Theorem 3.2** [18] Let D be drawn from $D_{n,d/n}$ where $d > 1$. Then, as n tends to infinity:
>
> 1. The probability that $X(r)$ is small tends to $1 - \Theta$, where Θ is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$.
> 2. The expected size of $|X(r)|$, given that $X(r)$ is small, tends to $\frac{1}{1-d(1-\Theta)}$.

The following theorem says that given $\ln n$ random vertices, the probability that all of them have small reachability goes to zero as n tends to infinity.

> **Theorem 3.3** [25] Let $A(u)$ represent the event that $X(u)$ is small in D. Let $v_1, v_2, ..., v_k$ be k distinct vertices in D, where $k = \ln n$. Then $Pr[A(v_1) \wedge A(v_2)...A(v_k)]$ goes to zero as n tends to infinity.

The next theorem states the upper bound on the number of vertices reachable from r when $X(r)$ is small.

> **Theorem 3.4** [25] With probability tending to 1, $|X(r)|$ in a digraph D drawn from $D_{n,d/n}$ will be bounded above by $\ln \ln n$, given that $X(r)$ is small.

The following two theorems show that there are few vertices that have a short path to a particular vertex or are within a short distance from the particular vertex.

> **Theorem 3.5** [25] Let $Z(r)$ be the number of vertices that have a path to r of length $\ln \ln n$ or less in D. Then $Pr[Z(r) > n^{1/m}]$ tends to zero as n tends to infinity, for any constant $m > 0$.

> **Theorem 3.6** [25] Let $Z(r)$ be the number of vertices that have a path from r of length $\ln \ln n$ or less in D. Then $Pr[Z(r) > n^{1/m}]$ tends to zero as n tends to infinity, for any constant $m > 0$.

Let uRv mean that vertex v is reachable from u. The following theorem tells us that v is very likely reachable from u given that both $X(u)$ and $Y(v)$ are large.

> **Theorem 3.7** [18] Let D be drawn from $D_{n,d/n}$, where $d > 1$. Then with probability tending to 1, the following statement holds for all u and v: if $X(u)$ is large and $Y(v)$ is large, then uRv.

The next theorem tells us the number of vertices in a digraph that have a large forward/backward reachability. As n goes to infinity, the fraction of such vertices tends to Θ, which depends only on d.

> **Theorem 3.8** [18] Let LARGEOUT $=\{u|X(u)$ is large $\}$, LARGEIN $=\{v|Y(v)$ is large $\}$ and LARGE $=\{u|X(u)$ is large and $Y(u)$ is large $\}$. Let $w(n)$ be a nondecreasing unbounded function. Then with probability tending to 1, $|LARGEOUT - \Theta n| < w(n)\sqrt{n \log n}$, $|LARGEIN - \Theta n| < w(n)\sqrt{n \log n}$ and $|LARGE - \Theta^2 n| < w(n)\sqrt{n \log n}$.

The following theorem depicts a picture of typical random digraphs. It shows that a digraph drawn from $D_{n,d/n}$ is very likely to have exactly one giant strong component, which is the *LARGE* set in Theorem 3.8. Furthermore, the size of the giant strong component is $\Theta^2 n$.

> **Theorem 3.9** [18] Let $w(n)$ be a nondecreasing unbounded function. Let d be a constant greater than 1. There is a constant A such that, with probability tending to 1, a digraph drawn from $D_{n,d/n}$ has exactly one giant strong component with more than $A \log n$ vertices, and the number of vertices in that strong component differs from $\Theta^2 n$ by at most $w(n)\sqrt{n \log n}$.

Combining Theorems 3.8 and 3.9, we see that apart from the giant strong component, there are about $(\Theta - \Theta^2)n$ vertices that can reach the strong component. Similarly, there are about $(\Theta - \Theta^2)n$ vertices outside the strong component that are reachable from the strong component. Notice that these two sets are disjoint, otherwise some vertices would be in the giant strong component. Thus, there are about $n - 2(\Theta - \Theta^2)n - (\Theta^2)n = (1-\Theta)^2 n$ vertices can neither reach nor be reachable from the strong component. The following theorem states that nearly all the vertices of a digraph drawn from $D_{n,d/n}$ lie either in the giant strong component or in a strong component of size 1. In particular, almost certainly, all the cycles in the graph lie in the giant strong component.

> **Theorem 3.10** [18] Let d be a constant greater than 1. Let Θ be the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. In digraph D, call a vertex exceptional if it lies in a strong component of size greater than 1 that is not the unique giant strong component. Let the random variable s be the number of exceptional vertices in a digraph drawn from $D_{n,d/n}$. The expected value of s is bounded above by a function of n that converges to the positive constant $\frac{2d^2(1-\Theta)^2}{1-d(1-\Theta)} - \frac{d^2(1-\Theta)^4}{1-d(1-\Theta)^2}$.

The next theorem states the upper bound of the number of exceptional vertices.

> **Theorem 3.11** [25] Let the random variable s be the number of exceptional vertices in a digraph drawn from $D_{n,d/n}$. Then with probability tending to 1, s is bounded above by $\ln \ln n$.

The strict implementation of digraphs from $D_{n,d/n}$ is as follows: for each of the $n(n-1)$ edge slots between two vertices (u, v), decide whether to have an edge from u to v according to the probability d/n. However, as n becomes large, this procedure gets very time consuming. Instead we fix the number of edges to be the expected number dn. The procedure becomes choosing dn different edges from $n(n-1)$ candidates, and multiple edges are not allowed. Our experiments in the following chapters demonstrate that our approach provides a very good approximation to Model D.

Chapter 4

Size of Binary Decision Diagrams

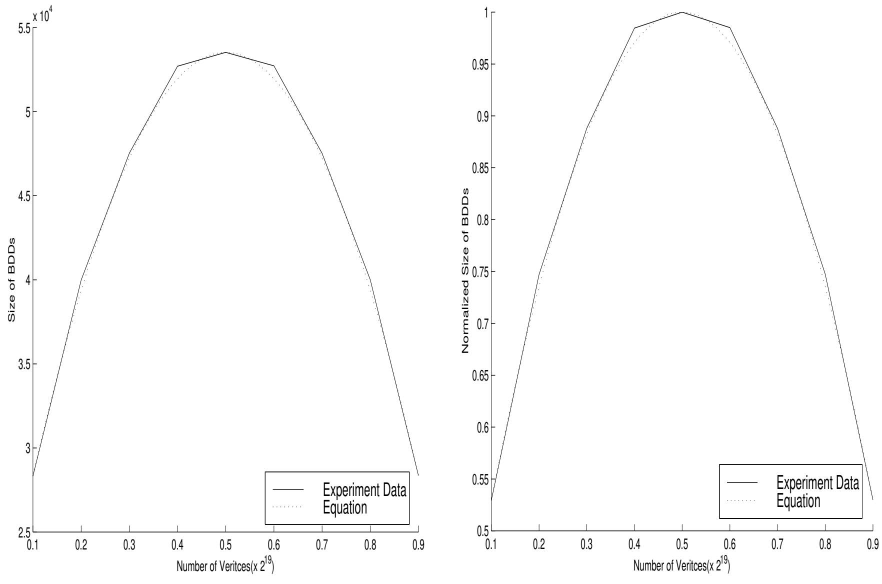
As described in Section 2.3, BDDs provide a concise representation for Boolean formulae and are used widely in different fields. However, few theories have been developed about the size of BDDs. This chapter provides an approach to predict the expected size of BDDs.

4.1 Relative Size of BDDs

Instead of dealing with a Boolean formula directly, we can identify every Boolean formula $f(x_1, ..., x_k)$ with a subset of the state space $S = \{s_0, ..., s_{2^k-1}\}$. Each state in S is indexed with a Boolean vector $\{0,1\}^k$. Given a subset P of S, we think of P representing a Boolean formula p, where the indices of the states in P make p true, and indices of the states not in P make p false. For example, $P = \{000, 001\}$ represents the Boolean formula $p = x_1'x_2'$. The empty set is identified with the Boolean constant 0, and S is identified with 1. Suppose we identify sets P and Q with Boolean formulae p and q respectively, then we can identify $P \cap Q$ with $p \wedge q$, $P \cup Q$ with $p \vee q$ and $S - P$ with $\neg p$. It is not difficult to show that this representation satisfies the axioms of Boolean algebra.

The primary goal of this section is to explore how the size of BDDs changes when they are representing sets with different numbers of states. The approach for doing the experiments is as follows. We use a BDD to represent a set of states chosen from a certain state space with $n = 2^k$ elements, where k is an integer between 4 and 19. The number of states in each set is a fraction that ranges from 10% to 90% with a 10% interval of the total state space, and the states in the set are chosen uniformly from the state space. The sizes of the BDDs are sampled and averaged over all experiments with the same fraction of a certain state space.

Figure 4.1(a) shows the average size of the BDDs representing different numbers of states chosen uniformly from a state space with 2^{19} elements. The x-axis in the figure displays the fraction of the states being represented, and the y-axis shows the size of



(a) (b)

Figure 4.1 Size of BDDs that represent sets of vertices.

the BDDs. The solid line graphs the experimental data. Each datum is averaged from 10 individual experiments. It is observed that the size of the BDDs is symmetrical regarding the percentage of states the BDDs are representing. For example, the size of the BDDs representing $0.9 \cdot 2^{19}$ states is the same as the size of the BDDs representing $0.1 \cdot 2^{19}$ states. This is because we need only switch the two terminal nodes in BDD to represent the complement set of a set currently being represented. The BDDs with largest size are those representing 50% of the total states. The dotted line, which tries to approximate the solid line, depicts the following equation:

$$y_{19} = 53520 - 157450 * (x - 0.5)^2, \qquad (4.1)$$

where x denotes the fraction of states being represented, and y_{19} denotes the expected size of a BDD used to represent a set with $x \cdot 2^{19}$ states.

If we consider the size of the BDDs representing $0.5 \cdot 2^{19}$ states as 1, then the relative size of the BDDs is $0 < \omega \leq 1$. For example, the relative size of the BDDs used to represent $0.1 \cdot 2^{19}$ states is about 0.55. Figure 4.1(b) graphs the relative size of the BDDs of (a). The x-axis in (b) has the same meaning as that in (a), while the y-axis shows the size relative to that of the BDDs representing 50% of total states.

The solid line is obtained by dividing the experimental data by 53520, which is the size of, the largest BDDs. The dotted line depicts the equation that is obtained by dividing Equation 4.1 by 53520, which is shown as follows:

$$w_{19} = 1 - 2.942 * (x - 0.5)^2, \tag{4.2}$$

where w_{19} denotes the expected relative size of BDDs, and x has the same meaning as in Equation 4.1.

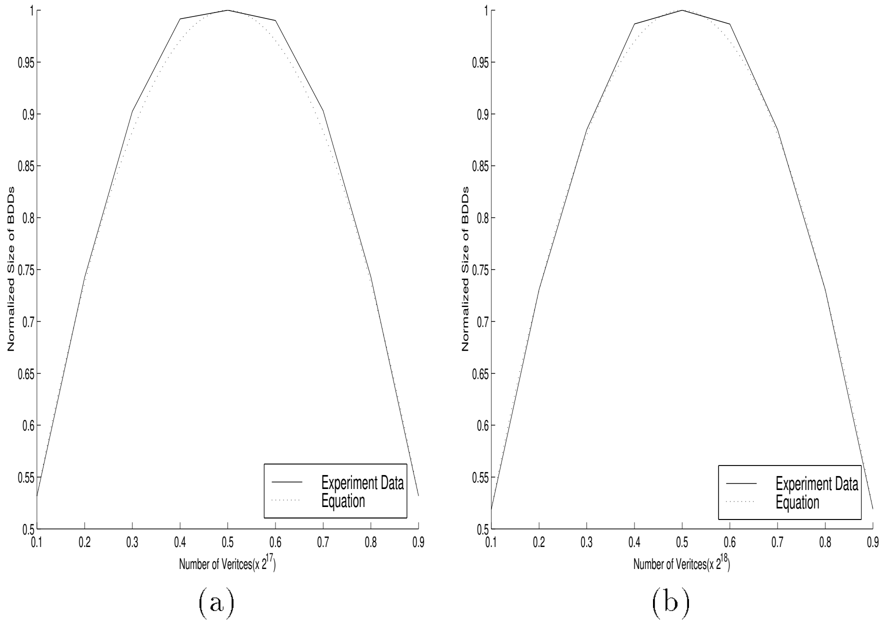


Figure 4.2 Relative sizes of BDDs.

The same phenomenon happens in different state spaces. The solid lines in Figure 4.2 graph the relative sizes of the BDDs obtained from experiments performed on a state space with 2^{17} or 2^{18} elements. The dotted line in (a), which approximates the experimental data on a state space with $n = 2^{17}$ elements, depicts the following equation:

$$w_{17} = 1 - 2.927 * (x - 0.5)^2. \tag{4.3}$$

The dotted line in (b), which approximates the experimental data on a state space with $n = 2^{18}$ elements, depicts the equation:

$$w_{18} = 1 - 3.007 * (x - 0.5)^2. \tag{4.4}$$

Based on Equations 4.2, 4.3 and 4.4, we come to the conclusion that the relative size of BDDs, as shown in Equation 4.5, does not depend on the size of a state space:

$$w = 1 - 3(x - 0.5)^2. \tag{4.5}$$

Note that the conclusions about the relative size of BDDs are based on the assumption that the states are uniformly chosen from the state space. Other methods may result in totally different sizes of BDDs. For example, we may generate a set with 50% of the states from a state space of size 2^k in the following way. First we randomly choose a Boolean variable x_i from $\{x_1, ..., x_k\}$ which are used to denote the state space. Then we fix x_i to be a Boolean constant c and ignore the values of the other variables. In other words, the characteristic function is $x_i = c$. The BDD representing this function has only one internal node regardless of the size of state space. Thus, we obtain a very concise BDD although it is representing 50% of the state space.

4.2 Absolute Size of BDDs

In order to obtain the size of BDDs it is necessary to recall the structure of BDDs. Suppose we use a complete binary decision tree B_k with root s to represent a Boolean function $f(x_1, ..., x_k)$. The nodes on level i are labeled x_i. The node v with label x_i is reached from s via all paths $(c_1, ..., c_k)$ that travel from the root to v. The decision tree rooted at v represents the sub-function $f'(x_i, ..., x_k) = f|_{x_1 \leftarrow c_1, ..., x_{i-1} \leftarrow c_{i-1}}$. We can reduce the binary decision tree to get a BDD equivalent to B_k by first merging all 0-leaves and all 1-leaves, then doing bottom-up induction by the deleting and merging rules shown in Section 2.3.

We partition the BDD into two halves such that the upper half contains the nodes whose labels are between x_1 and x_{i-1}, and the lower half contains the nodes whose labels are between x_i and x_k. It is obvious that the upper part contains at most $2^i - 1$ nodes. As to the lower half, it is clear that for different nodes v and u, the corresponding sub-functions rooted at the two nodes are different because of the deleting and merging rules. Since there are at most $2^{2^{k-i+1}}$ different sub-functions with $k - i + 1$ variables, there are at most $2^{2^{k-i+1}}$ nodes in the lower half. Therefore, the upper bound of the size of BDDs with k Boolean variables is $min\{2^i + 2^{2^{k-i+1}} | i = 1, ..., k\}$. Based on this observation, Liaw and Lin [20] state an upper bound on the size of BDDs in the following theorem:

Theorem 4.1 Let WR_k be the worst case BDD-size of k-variable Boolean functions. Then the complexity of WR_k is $O(2^k/k)$. More precisely,

1. WR_k is less than $(2^k/k)(2+\epsilon)$ for all sufficiently large k, where ϵ is an arbitrarily small positive number.
2. WR_k is less than or equal to $(2^k/k)(3.125)$ for all k.

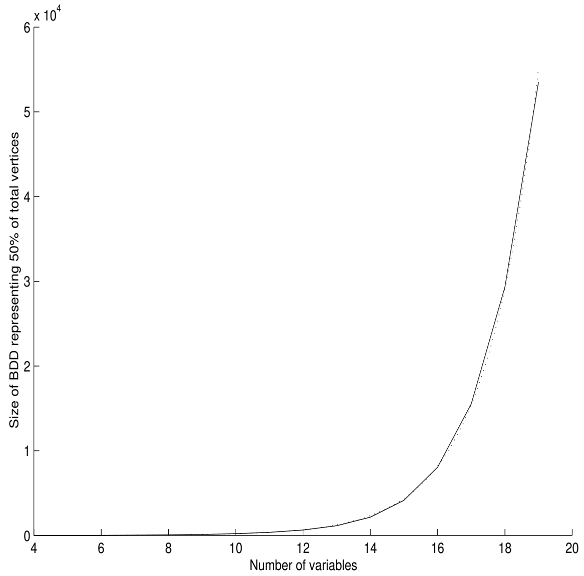


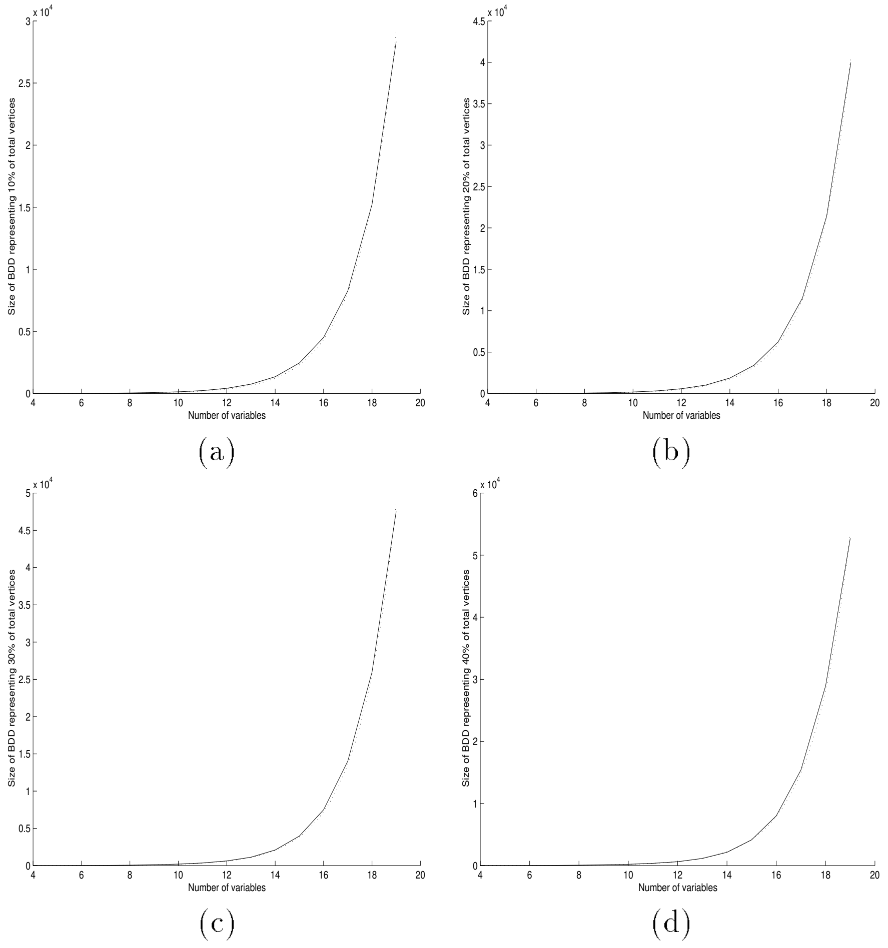
Figure 4.3 Size of BDDs in the worst case.

As we know from the last section, the size of BDDs reaches the worst case when the BDDs are representing 50% of a state space. Figure 4.3 graphs the size of BDDs in such a case. The solid line indicates the data obtained from experiments performed on state spaces with sizes ranging from 2^4 to 2^{19}. In each individual experiment, we randomly chose half of the total states and used BDDs to represent the set. Each datum on the solid line is averaged from 10 individual experiments. The dotted line depicts the function that arises from Theorem 4.1:

$$y = 2^{x+1}/x. \tag{4.6}$$

As observed from Figure 4.3, the dotted line matches the solid line. We can draw two conclusions. First, representing half of the total states by BDDs is the worst case.

Considering a Boolean function $f(x_1, ..., x_k)$, there are 2^k possible truth assignments $f(x_1, ..., x_k)|_{x_1 \leftarrow a_1, ..., x_k \leftarrow a_k}$. When f is satisfied by half of the assignments, the BDD used to represent f will have the worst size. Second, the upper bound $(2^k/k)(2+\epsilon)$ for BDD-size is tight.



(a) (b) (c) (d)

Figure 4.4 Size of the BDDs representing certain percentages of the total states: (a)10%. (b) 20%. (c)30%. (d)40%.

Now we consider the size of BDDs in the general case. Equation 4.5 claims that the relative size of BDDs depends only on the fraction of states they are representing. We also know the size of BDDs in the worst case. Therefore, we can predict the size based on the percentage of states a BDD is representing and the number of Boolean variables being used. Combining Equations 4.5 and 4.6, we get the following equation:

$$y = \frac{2^{x+1}}{x} \cdot (1 - 3(\beta - 0.5)^2), \tag{4.7}$$

where y denotes the size of BDDs, x is the number of Boolean variables and β shows the fraction of states that are represented in the state space.

Figure 4.4 graphs the size of BDDs from experiments and Equation 4.7. The x-axes display the number of Boolean variables used for representing the state space. In other words, the size of state space is 2^x. The y-axes display the size of BDDs. The solid line in (a) displays the experimental data obtained from BDDs that represent 10% of the states from the state space. Similarly, the solid lines in (b), (c) and (d) show the experimental data obtained from BDDs that represent 20%, 30% and 40% states respectively. The results for 50% states are shown in Figure 4.3, while the results for percentages higher than 50% need not be shown because of symmetry. The dotted lines depict Equation 4.7.

4.3 Size of BDDs Representing Digraphs

If we want to know the size of BDDs that are related to digraphs, the questions will be: what is the average size of the BDDs that are used to represent n vertices? and what is the average size of the BDDs that are used to represent e edges?

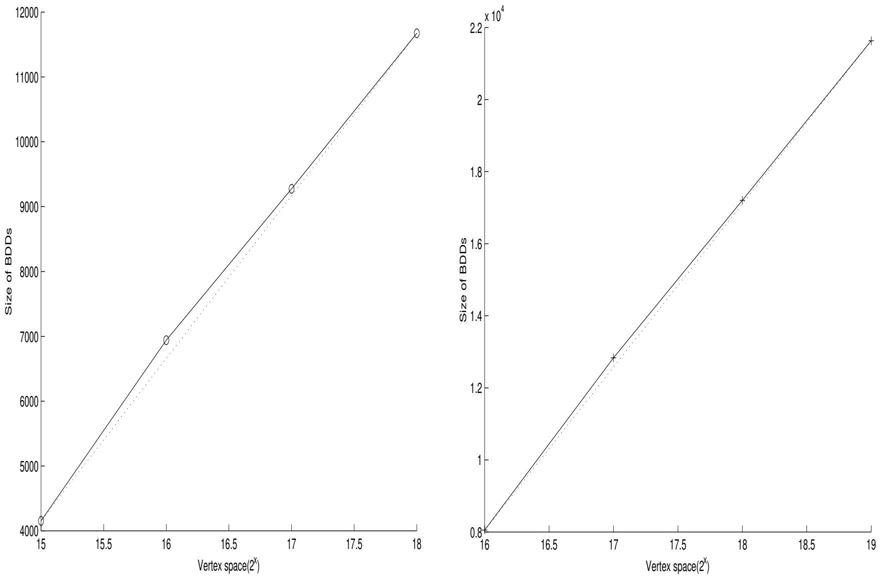
Actually the first question is very similar to asking about the size of BDDs representing a certain set of states if we cast the vertex space to the state space S. However, the size of the state space in the last two sections was always a power of 2. Since the number of total vertices in a digraph is often not a power of 2, the size of a vertex space may not equal the number of vertices in a digraph. Given a digraph $D(V, E)$, we need to use $u = \lceil \log |V| \rceil$ variables to represent all of the vertices, where $\lceil a \rceil$ equals the smallest integer that is larger than or equal to a. Consequently, the size of a vertex space is $|V'| = 2^u$. Equation 4.7 is still valid for predicting the size of BDDs used to represent v vertices, with the following format:

$$y = \frac{2|V'|}{\log |V'|} \cdot (1 - 3(v/|V'| - 0.5)^2), \tag{4.8}$$

where $|V'|$ is the size of the state space, and v is the number of vertices that need to be represented by BDDs.

It is observed from Equation 4.8 that the size of BDDs is affected by two factors: the size of the vertex space, and the number of vertices that need representing. When BDDs are used to represent a set of vertices from a digraph, we always choose the size of a vertex space to be the smallest power of 2 that is larger or equal to the number of total vertices. An interesting question is whether it is possible to get a smaller BDD representation if we use a larger vertex space. (In other words, we could use more Boolean variables than necessary in the BDDs.) For example, we need to represent $V/2$ vertices from a digraph $D(V, E)$ by a BDD. Equation 4.8 states that the size of the BDD will be around $2|V'|/\log |V'|$, which is the worst case. If we increase the vertex space by 2 times, then the BDD representation won't be in the worst case.

Figure 4.5 shows experimental data for the question raised above. The X-axis displays the size of the vertex space, and the Y-axis displays the size of the BDDs. The number of vertices being represented by BDDs in (a) is fixed to be 2^{14}. The experiments start with a vertex space of size 2^{15}. Then the size is doubled in each of the following experiments until it reaches 2^{18}. Therefore, the percentage of the vertex space that we are representing becomes smaller: 50%, 25%, 12.5% and 6.25% respectively in the four groups of experiments in (a). The number of vertices being

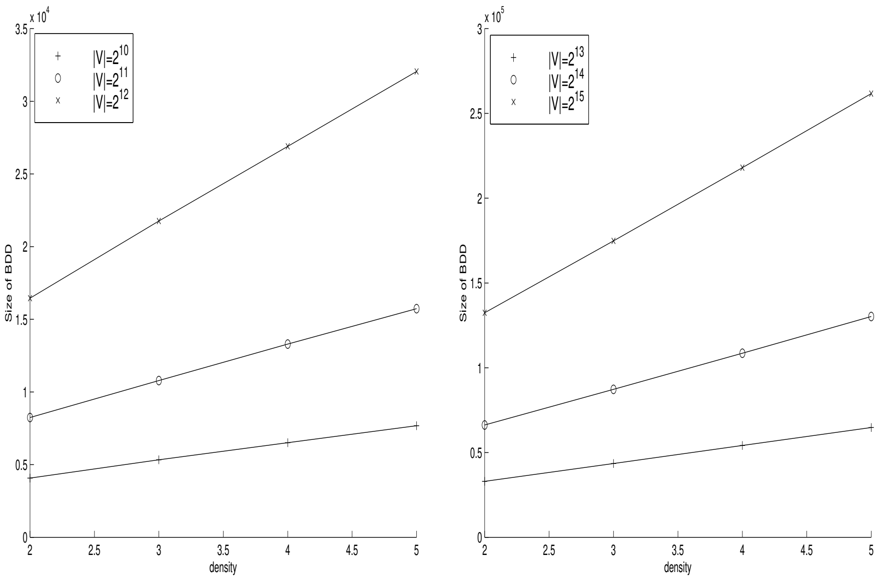


(a) (b)

Figure 4.5 Size of the BDDs representing fixed number of vertices: (a)16384. (b)32768.

represented by BDDs in (b) is fixed to be 2^{15}, and the size of a vertex space ranges from 2^{16} to 2^{19}. An obvious observation from Figure 4.5 is that increasing the size of the vertex space does not help reduce the size of the BDDs. This can also be concluded from Equation 4.8. Suppose we want to represent $A/2$ vertices from a vertex space of size A; then the size of the corresponding BDD is $2A/\log A$. Note that the smallest vertex space size larger that A is $2A$ because the size is always a power of 2. Now if we increase the size of the vertex space to $2A$, the size of the BDD is $4A/\log 2A \cdot (1 - 0.0625\alpha)$. Let α be 3, which is based on the experimental data. The new BDD has size around $3.25A/\log 2A$, which is larger than $2A/\log A$ when A is not trivial.

In the rest of this section we consider representing the edge relation for a digraph $D(V, E)$. Let f_{e_i} be the Boolean function representing $e_i \in E$. Then $F_E = \bigvee_i f_{e_i}$ is the Boolean function representing the whole edge relation. Such representation is called *monolithic* because the edge relation is represented by a single BDD. Assume we need k Boolean variables to represent the vertex space of D. In order to represent the edge relations, we need two sets of Boolean variables to be active at the same time. In this case, we need $2k$ Boolean variables to represent the edge relation.



(a) (b)

Figure 4.6 Size of the BDDs representing edges (I).

Consider a digraph with a vertex space represented by Boolean variables $(x_1, ..., x_k)$. Since we need two sets of the Boolean variables, the size of edge space is 2^{2k}. Because our digraphs are drawn from $D_{n,d/n}$, it is natural to ask about how the size of the BDDs changes with the digraph density that represents the ratio of the number of edges over the digraph order. In the experiments, we randomly generate digraphs with 2^k vertices, where k is an integer between 10 and 15. We choose the density d to be an integer between 2 and 5. Then we use BDDs to represent edge relations in each digraph and average the size of the BDDs over ten individual experiments that are performed on digraphs with same order and density. Figure 4.6 shows the results. The x-axes display the density and the y-axes display the size of the BDDs that represent the edge relations. The figure is divided into two parts (a) and (b). Each sub-figure contains three solid lines. Each solid line represents a group of experiments performed on digraphs with the same orders. The three solid lines in (a), from bottom to top, represent experimental data from digraphs with 2^{10}, 2^{11} and 2^{12} vertices respectively. The three solid lines in (b) represent experimental data from digraphs with 2^{13}, 2^{14} and 2^{15} vertices respectively from bottom to top. The dotted lines show the size of BDDs if they were growing linearly with the number of edges. It is observed that the size of the BDDs is related to both the order and density of digraphs. When the digraph order is fixed, we can see that the size of BDDs grows almost linearly in the number of edges that they represent.

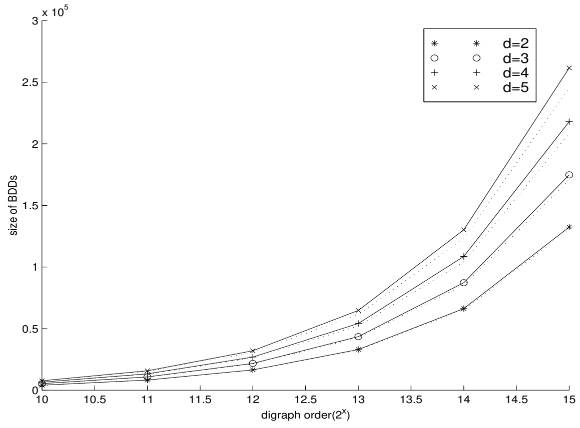


Figure 4.7 Size of BDDs representing edges (II).

Figure 4.7 graphs the data in a different way. While the y-axis still shows the size of the BDDs, the x-axis shows the number of Boolean variables needed to represent the vertex space, *i.e.*, the digraph order is 2^x. Each solid line in the figure represents experimental data obtained from digraphs with a fixed density. The densities for the four lines are 2, 3, 4 and 5 respectively from bottom to top. The dotted lines show the size of BDDs if they were growing linearly with the number of edges. It is observed that the dotted lines are below the corresponding solid lines. The phenomenon seems contradictory to that in Figure 4.6. Actually, the difference comes from the number of active variables. In Figure 4.6, all the BDDs on the same line have the same number of active variables. In Figure 4.7, when the value on x increases by 1, the number of edges is doubled. At the same time, the number of active variables used to represent those edges is increased by 2. Since there are 2 more levels in the BDDs, the size of BDDs grows faster that the number of edges does.

Figure 4.8 shows another way to view the relation between the size of BDDs with the number of edges. The x-axis displays the digraph densities, and the y-axis shows the ratio of the size of BDDs over the number of edges. There are three lines in the

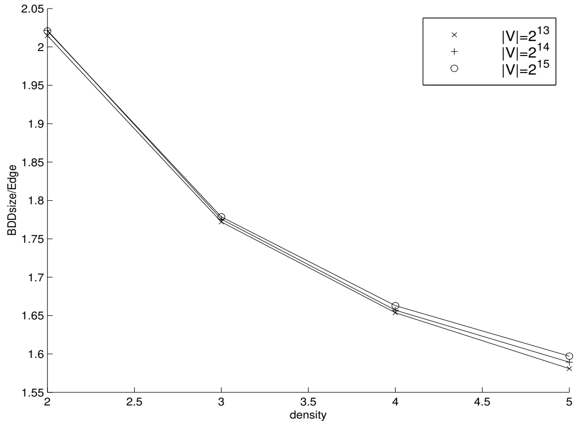


Figure 4.8 Ratio of the size of BDDs over the number of edges.

figure. The line with "x" is obtained from digraphs with 2^{13} vertices; the line with "+" is obtained from digraphs with 2^{14} vertices and the one with "o" is obtained from digraphs with 2^{15} vertices. The three lines are very close and ratios are between 1.55and 2.05. From the previous section, we know that the worst case BDD-size is $2^{x+1}/x$ when BDDs represent 2^{x-1} states. Therefore, the ratio of the BDD-size over the number of states is less or equal to $4/x$. When the number of states is not trivial, We can see that the BDDs representing transition relations are less efficient than those that represent random state sets. Since the BDDs that represent the transition relation are larger and less efficient than the BDDs that represent state sets, BDDs that represent transition relations are the bottle neck in reachability computation.

Chapter 5

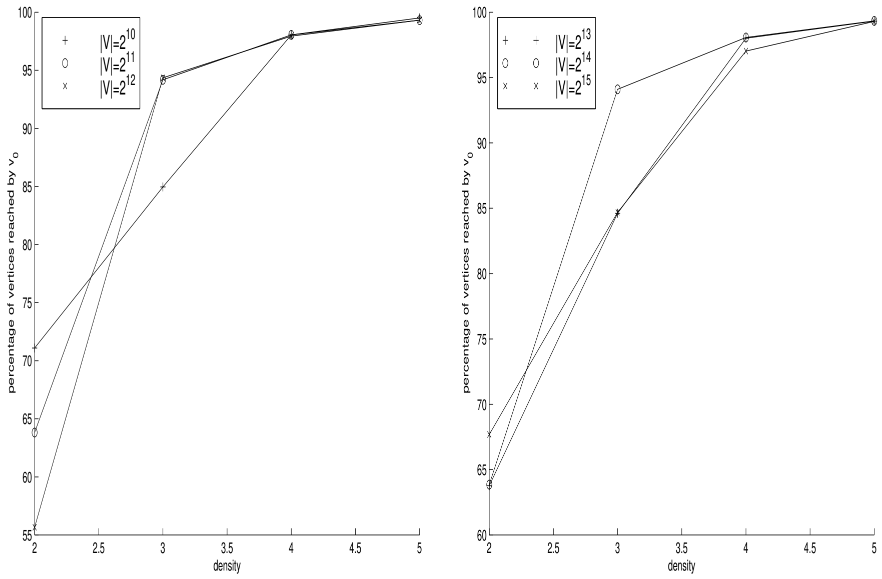
Reachability Computations

Roughly speaking, model checking is performed as an exhaustive reachability analysis. Given a state transition graph, some frequently asked questions are as follows. How many states can be reached from an initial state? How many iterations are needed to reach the fixed point? During the computation, how do the sizes of the BDDs change and how many states are reached at each iteration? The primary goal of this chapter is to explore these questions by doing experiments performed on random digraphs drawn from $D_{n,d/n}$.

5.1 Number of Reachable Vertices

In this section, we analyze the number of vertices which can be reached from an initial vertex. For a digraph $D(V, E)$ drawn from $D_{n,d/n}$, the density d plays the key role. The set of experiments are designed to explore how the density affects the percentage of vertices that can be reached. We generated random digraphs with order n between 2^{10} and 2^{15} and choose a density d between 2 and 5. For each (order, density) pair, we did 10 experiments and averaged the data we obtained. In each individual experiment, we chose an initial vertex v_0 and tested the number of vertices that are reachable from v_0. The results from the experiments are shown graphically in Figure 5.1. Data in (a) are based on digraphs with orders 2^{10}, 2^{11} and 2^{12}; data in (b) are based on digraphs with orders 2^{13}, 2^{14} and 2^{15}. It is interesting to observe that a fairly large fraction of the vertex space can be reached from one initial vertex in such sparse digraphs. For example, more than half of the total vertices can be reached when $d = 2$ in all the experiments; when $d = 5$, almost all of the vertices can be reached. The percentage of reachable vertices is affected only by the density and is independent of the order of the digraphs.

However, the percentage of vertices that can be reached is not uniform across all experiments. The experimental data display that there are two distinct cases. Starting from one initial vertex, the number of reachable vertices is either very small

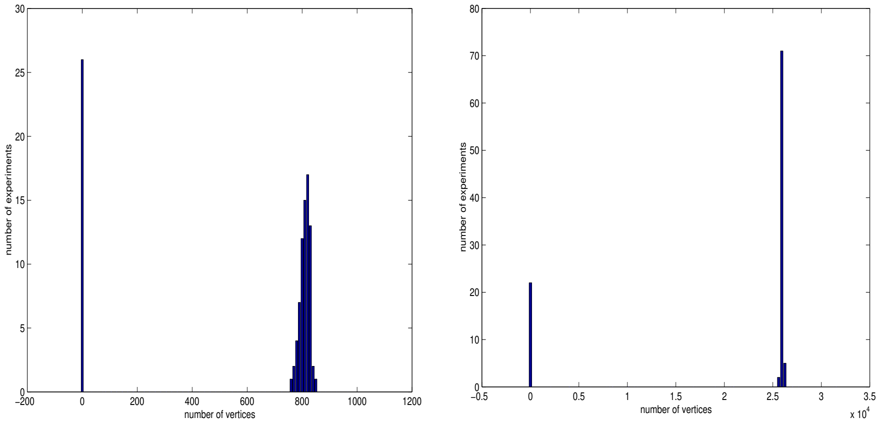


(a) (b)

Figure 5.1 Percentage of vertices reachable from v_0.

or very large. It is misleading to combine the data from the two different cases. We divided the experiments into seven groups. Experiments in each group are performed on digraphs with same density but different orders. For the first group of experiments, we set the density $d = 2$ and let the order be 2^k, where k is an integer between 10 and 15. Figure 5.2 graphs the experimental data. The x-axis displays the number of vertices reachable from an initial vertex, and the y-axis shows the number of experiments. Each of (a) and (b) contains 100 individual experiments. Data in (a) are based digraphs with 2^{10} vertices and (b) are based on digraphs with 2^{15} vertices. Figure 5.2 shows clearly that only very few vertices can be reached in some experiments, while a fairly large amount can be reached in other experiments. In the case of (a), less than 10 vertices are reachable from v_0 in about 26 experiments and more than 700 vertices are reachable in the remaining experiments. There is no experiment in between. (b) shows the same pattern with about 23 experiments reaching very few vertices and 67 experiments reaching much more vertices.

The densities in other groups range from 1.2 to 4. Figure 5.3 graphs results from two different groups. Both (a) and (b) are performed on digraphs with 2^{15} vertices, while the densities are 1.2 and 4 respectively. In (a), about 34 experiments have a



(a) (b)

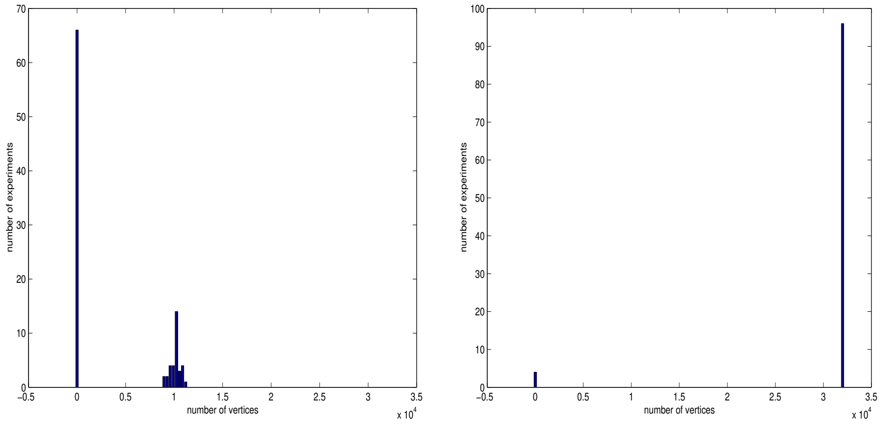
Figure 5.2 Distribution on the number of reachable vertices(I): (a) $|V| = 2^{10}$, $d = 2$. (b)$|V| = 2^{15}$, $d = 2$.

large reachable set; while in (b) about 94 experiments have a large reachable set. Also, the sizes of large sets, which are about 10^4 and $3.2 \cdot 10^4$ in (a) and (b) respectively, are different. However, the number of vertices reached when the reachable set is small does not change.

The results of the experiment agree with the theorems in Section 3.2. Theorem 3.1 demonstrates the important gap phenomenon: given a digraph drawn from $D_{n,d/n}$, the number of vertices reached from an initial vertex is very likely to be either large or small. The probability for the reached set to be large tends to Θ as n tends to infinity, where Θ is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. Theorem 3.2 shows that the probability that the reached set is small depends only on the density. Also, the expected size of the small reached set tends to be a constant that depends only on density, with value $1/1 - d(1 - \Theta)$.

Let $X(v_0)$ be the set of vertices reachable from the initial vertex v_0. Let $Pr(X(v_0))_{sm}$ be the probability for $X(v_0)$ to be small. Let $|X(v_0)|_{sm}$ be the expected size of $X(v_0)$ when it is small. Let $|X(v_0)|_{15}$ be the expected size of $X(v_0)$ when it is large and the digraph order is 2^{15}. Table 5.1 shows the results computed from the relevant theorems in Section 3.2.

Table 5.2 summarizes the data we obtained from our experiments performed on digraphs with a fixed density of 2 but different orders. In this table, $Frac(X'(v_0))_{sm}$ stands for the fraction of experiments that have small reachable set. $|X'(v_0)|_{sm}$ stands



(a) (b)

Figure 5.3 Distribution on the number of reachable vertices(II): (a)$|V| = 2^{15}$, $d = 1.2$. (b)$|V| = 2^{15}$, $d = 4$.

density	4	3	2	1.8	1.6	1.4	1.2
Θ	0.980	0.940	0.799	0.732	0.642	0.512	0.311
$Pr(X(v_0))_{sm}$	0.020	0.060	0.201	0.268	0.358	0.488	0.689
$\|X(v_0)\|_{sm}$	1.09	1.22	1.67	1.93	2.34	3.16	5.77
$\|X(v_0)\|_{15}$	32113	30802	26182	23986	21037	16777	10191

Table 5.1 Results computed from relevant theorems.

for the average number of vertices reached from v_0, provided $X'(v_0)$ is small. $|X'(v_0)|_{lg}$ stands for the average number of vertices reached from v_0 when $X'(v_0)$ is large. σ_{lg} is the standard deviation of $|X'(0)|$ when $|X'(0)|$ is large. Note from Table 5.1, $Pr(X(v_0))_{sm} = 0.201$ and $|X(0)|_{sm} = 1.67$ when $d = 2$.

Table 5.3 lists the data obtained from experiments performed on digraphs with order 2^{15} and different densities ranging from 1.2 to 4. The last rows in both Table 5.2 and 5.3 show that the range of the sizes of large reachable sets is related to density. For example, the standard deviation is only 25.5 when $d = 4$, while it becomes 3451.7 when $d = 1.2$.

$\lvert V\rvert$	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
$Frac(X'(v_0))_{sm}$	0.26	0.27	0.26	0.24	0.20	0.22
$\lvert X'(v_0)\rvert_{sm}$	1.69	1.56	1.54	1.67	1.15	1.64
$\lvert X'(v_0)\rvert_{lg}$	815	1632	3267	6526	13066	26098
σ_{lg}	18.2	21.0	33.0	47.8	71.8	92.8

Table 5.2 Results obtained from experiments when $d = 2$.

density	4	3	2	1.8	1.6	1.4	1.2
$Frac(X'(v_0))_{sm}$	0.04	0.03	0.22	0.28	0.42	0.48	0.66
$\lvert X'(v_0)\rvert_{sm}$	1.00	1.00	1.64	1.82	2.50	3.67	8.30
$\lvert X'(v_0)\rvert_{lg}$	32117	30817	26098	23984	21005	16821	10281
σ_{lg}	25.5	47.5	92.8	125.5	156.5	2297.7	3451.7

Table 5.3 Results obtained from experiments when $|V| = 2^{15}$.

5.2 Number of Iterations

When verification engineers use model-checking tools to verify a design, all they can do is to submit the task and wait for the results. No one knows how long the procedure will take because most of the algorithms used in model checking are heuristic. Since model checking is reduced to a reachability computation, it would be very helpful to have some knowledge about the number of iterations needed in the reachability computation.

From the last section, we know that the set reachable from an initial vertex is either very small or very large. The expected size of the small reachable set tends to be a constant that depends only on density. When the density is greater than or equal to 2, the expected size is less than 2. Therefore, it usually takes just one iteration to reach the fixed point when the reached set is small. For this reason, we are only interested in the cases where the reached set is large.

Let v_0 be the initial vertex in a digraph $D(V, E)$ drawn from $D_{n,d/n}$. Let $S_0(V)$ be the BDD representing the initial set $\{v_0\}$. The BDD representing the set of reachable vertices from v_0 in one step can be computed by:

$$S_1(V') = S_0(V') \vee \exists_{v \in V}(S_0(V) \wedge E(V, V')).$$

In general, the vertices reachable in at most $k + 1$ steps are represented by a BDD as:

$$S_{k+1}(V') = S_k(V') \vee \exists_{v \in V}(S_k(V) \wedge E(V, V')).$$

The existential quantifier notation above indicates the existential quantification of all variables v in V. Since the total number of vertices in digraphs is finite and the set represented by $S_i(V)$ is the subset of that represented by $S_{i+1}(V)$, at some point we must have $S_{k+1} = S_k$. The set S_k is the vertices reachable from initial set S_0 and k is the number of iterations.

Chapter 3 states that the expected number of vertices reachable from an initial vertex is Θn, where Θ is a constant that depends only on density d. If at each iteration, the newly reached vertices are distinct from each other and have not been reached before, then d times as many vertices are reached as the previous step. Therefore, we have $1 + d + d^2 + ... + d^m = \Theta n$, where m is the number of iterations in such case. It is easy to get that $m = -1 + \log_d(\Theta n(d-1)+1)$. Obviously, this number is much smaller than the actual data. However, inspired by the formula, we may assume the equation for computing the number of iterations has the format $\log_y \Theta n$. Figure 5.4 depicts the value of y for different experiments. The experiments are performed on digraphs with order between 2^{10} and 2^{15}, and density between 2 and 5. The dotted lines show the experimental data, which are calculated as follows. Let M be the number of reachable vertices and $iter$ be the number of iterations. Note that M and $iter$ are averaged from 15 experiments with same order and density. Then $y = M^{1/iter}$. Each of the six dotted line represents experimental data obtained from digraphs with fixed orders ranges from 2^{10} to 2^{15}. It is observed that y grows linearly with density and does not depend on digraph orders. The solid line depicts an equation that approximates the experimental data. The equation is as follows:

$$y = 0.405d + 0.683, \tag{5.1}$$

where d denotes the density.

Based on Equation 5.1, we can calculate the expected number of iterations needed to reach the fixed point from an initial vertex in a digraph drawn from $D_{n,d/n}$ by the following equation:

$$iter = \log_{0.405d+0.683} \Theta n, \tag{5.2}$$

where d is the density and Θ is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. Figure 5.5 shows both the experimental data and the result from Equation 5.2. The solid lines with "+", "x" or "o" depict the data from experiments. The orders of the digraphs used in the set of experiments are 2^{10}, 2^{12}, 2^{15}, and the density is an integer between 2 and 5. Each datum in Figure 5.5 is averaged from 15 individual

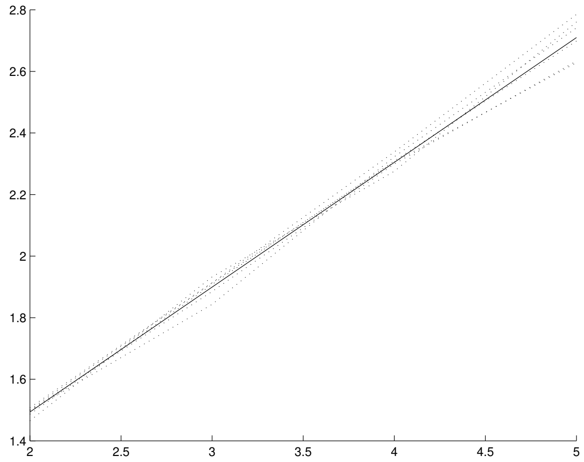


Figure 5.4 Approximation of y.

experiments. When the density is fixed, the bigger the order, the more the iterations. This can be explained by the difference in the number of vertices that are reachable from the initial vertex. For example, when the density is fixed to be 2, about 815 vertices are reachable from v_0 when $|V| = 2^{10}$, about 3267 are reachable when $|V| = 2^{12}$, and about 26098 are reachable when $|V| = 2^{15}$. Since the expected out-degree of each vertex is fixed, the more vertices that can be reached, the more iterations are needed. When the order is fixed, the bigger the density, the fewer the iterations. Although more vertices are reachable with larger density, more vertices can be reached at each step, which results in overall fewer iterations. Each of the dotted lines in Figure 5.5 links four points calculated from Equation 5.2 when densities are 2, 3, 4 and 5. It is observed that Equation 5.2 offers a good approximation on the expected number of iterations when the reachable sets are large.

5.3 Size of BDDs at Each Iteration

The goal of this section is to explore the number of vertices reached at each iteration and the size of BDDs that are used to represent them.

Chapter 4 states that the size of BDDs is the largest if they represent half of the total vertices. Since more than half of the total vertices can be reached in most of the experiments when the reachable set is large, we expect the BDDs representing

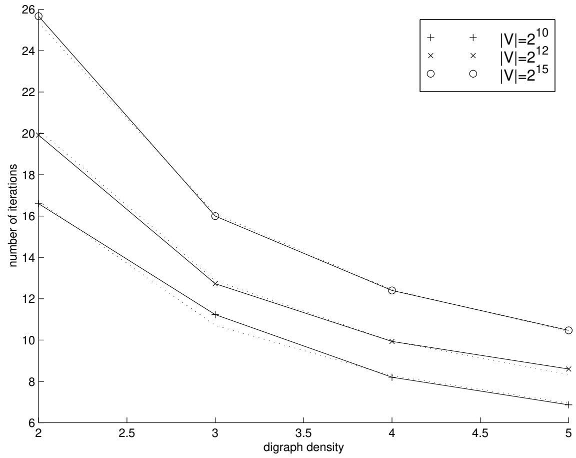
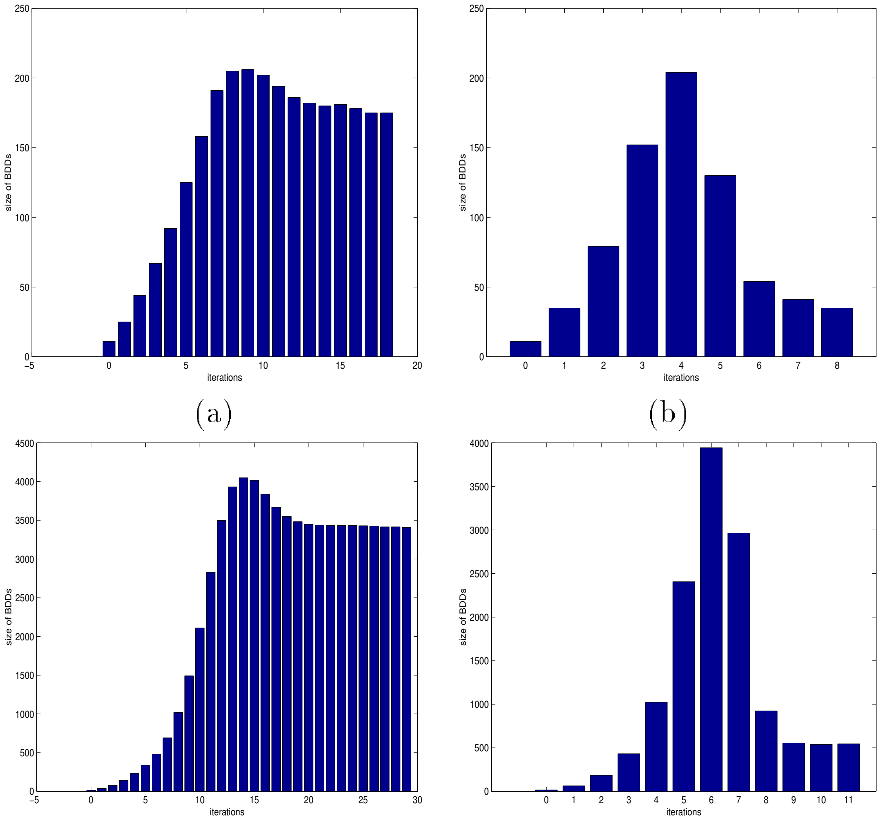


Figure 5.5 Number of iterations when reachable sets are large.

vertices reached at each iteration to become larger at each subsequent iteration until at some point, they become smaller.

We did 6 groups of experiments. The experiments in a group have the same density. Each group is further divided into 4 sets where each set contains 15 individual experiments performed on digraphs with same order. Figure 5.6 graphs part of the results obtained from the experiments. The x-axes in Figure 5.6 show the index of iterations and the y-axes show the size of the BDDs. The digraph order in (a) and (b) is 2^{10}; in (c) and (d) it is 2^{15}. The density in (a) and (c) is 2; in (b) and (d) it is 5. For all the sub-figures, the size of BDDs increases at first and then decreases. Correspondingly, we graph the number of vertices reached by v_0 in Figure 5.7. The x-axes still show the index of iterations, while the y-axes show the number of vertices reached at each iteration.

Comparing the sub-figures with different densities in Figure 5.6, we notice that the final size of the BDDs is much smaller in experiments with larger density. For example, the final size of the BDD in (c) is about 3400 and in (d) it is about 500. This is explained by checking the corresponding sub-figures (c) and (d) in Figure 5.7. The numbers of reached vertices are about 26000 and 32500 for (c) and (d) respectively. Since the digraph order is 32768, both of the experiments in (c) and (d) reaches more than half of the total vertices. Chapter 4 states that the size of the BDDs will be largest if they represent half of the total vertices; beyond that number, the more vertices a BDD is representing, the smaller the BDD. When we compare the sub-figures with different orders in Figure 5.6, it is observed that more iterations are needed and the size of the BDDs is larger at each iteration when the order is larger.

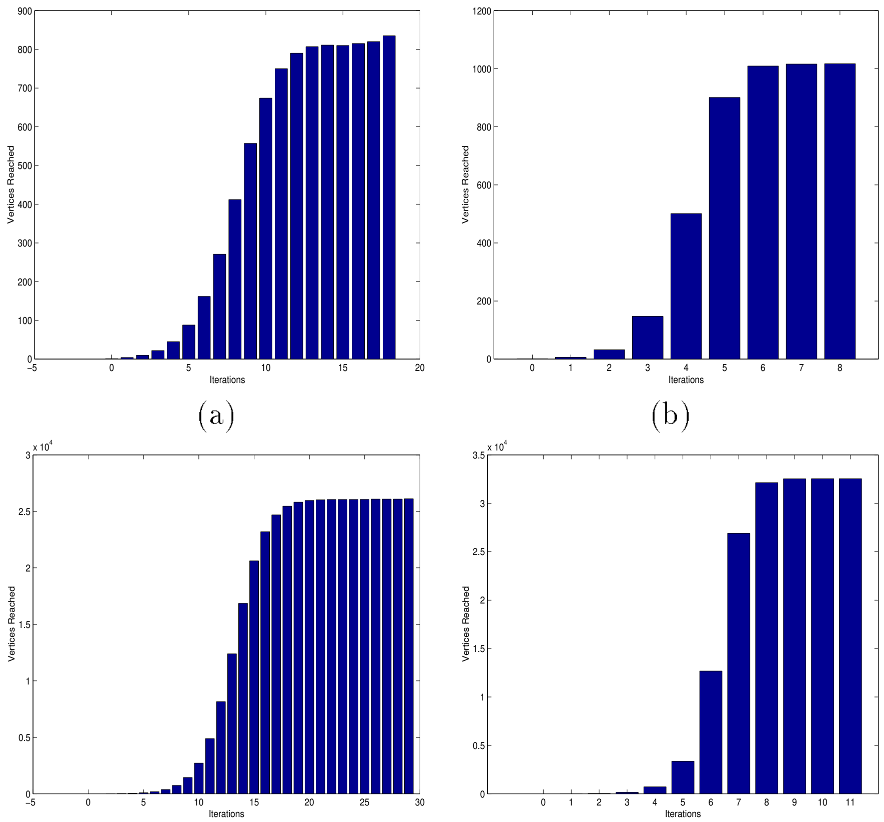


(a) (b)

(c) (d)

Figure 5.6 Size of BDDs at each iteration: (a)$|V| = 2^{10}, d = 2$. (b)$|V| = 2^{10}, d = 5$. (c)$|V| = 2^{15}, d = 2$. (d)$|V| = 2^{15}, d = 5$.

We notice in Figure 5.7 that the number of vertices reached at each iteration increases very quickly for the first two-thirds of iterations and changes very slowly for the last one-third. This observation is useful to approximate reachability analysis. Even with the best algorithms, there remain many designs that are too difficult to handle. In such case, it may be possible to perform an approximate reachability analysis. That is, computing a subset of all reachable states instead of doing an exhaustive search. A reachability computation produces a sequence of sets of states that eventually converges to the set of all reachable states. The elements of the sequence provide increasingly accurate approximations to the exact solution. If we cannot



(a) (b)

(c) (d)

Figure 5.7 Number of vertices reached at each iteration: (a)$|V| = 2^{10}, d = 2$. (b)$|V| = 2^{10}, d = 5$. (c)$|V| = 2^{15}, d = 2$. (d)$|V| = 2^{15}, d = 5$.

complete reachability computation, we can always stop when we run out of memory or time, and claim that we have obtained an approximation to the solution. This is called *partial verification*. The challenge lies in devising algorithms that produce good approximations with limited resources. If time is critical, the observation from Figure 5.7 provides a simple solution: we may provide a threshold. If the difference of reached vertices in two successive iterations is less than the threshold, the following iterations are unlikely to reach a significant number of new vertices. Therefore, we may stop the computation as soon as the threshold is met.

Chapter 6

Model Checking Safety Properties

In model checking, properties fall into two categories. One category is called safety properties, and the other one, liveness properties. Informally, a property that should be satisfied by all the states is called a safety property. A property that should be satisfied eventually along some path is called a liveness property. Consider a property P that stipulates that a bad state does not exist. If a bad state exists in an infinite sequence σ, then it must appear after some finite prefix. In other words, if $\sigma \not\models P$, there is some prefix of σ for which no extension to an infinite sequence will satisfy P. Alpern and Schneider[2] give the following formal definition of a safety property P:

$$(\forall \sigma : \sigma \in S^{\omega} : \sigma \models \Leftrightarrow P(\forall i : 0 \leq i : (\exists \beta : \beta \in S^{\omega} : \sigma_0..\sigma_i \beta \models P))), \tag{6.1}$$

where S^{ω} is the set of infinite sequences of states and s_j is an individual state. Alpern and Schneider[2] also give following the formal definition of a liveness property P:

$$\forall \alpha : \alpha \in S^{*} : (\exists \beta : \beta \in S^{\omega} : \alpha\beta \models P)), \tag{6.2}$$

where S^{*} is the set of finite sequences of states.

The goal of this chapter is to explore the behavior of reachability computation for safety properties, and in the next chapter we will discuss algorithms used for dealing with liveness properties. A natural model for safety properties is to specify some vertices in a digraph $D(V, E)$ as bad vertices, and then determine whether they are reachable from an initial set. There are three methods. First, we may start from the initial set and do a forward reachability computation to check if some state in the bad set can be reached. Second, we may start from the bad set and do a backward reachability computation to check if the initial set can be reached. Third, we may use a combination of the forward and backward methods by starting from both the initial set and the bad set to check whether their frontiers can meet each other.

This chapter compares the three methods and discusses the following questions: if some bad vertices scatter over a digraph $D(V, E)$, what is the probability that they can be reached? How many vertices are reached during the search for bad vertices? How many iterations are needed to reach some bad vertices or a fixed point?

6.1 Are Bad States Reachable?

A safety property is a desired property over the entire state space. A state is *bad* if it violates a safety property. In model checking, designs fail if bad states can be reached from the initial set. However, if bad states are not reachable from the initial set, then the design is correct. In the set of experiments in this section, we discuss how likely it is that the bad states can be reached from an initial vertex.

Our method for doing the experiments is as follows. Given a digraph $D(V, E)$ drawn from $D_{n,d/n}$, we randomly choose an initial set V_0 and a bad set V_B. We assume that the vertices in the bad set do not satisfy a given safety property. Then starting from V_0, we do the forward computation to check whether any vertices in V_B can be reached. Note that there is no difference of doing forward, backward or combined computation in this case. The size of V_0 we choose is either 1, 2, 4 or 8, and the size of V_B is either 1, 2, 4 or 8 times the size of V_0. Since the digraphs drawn from $D_{n,d/n}$ are symmetric, the probability for l vertices to be reachable from k vertices is the same as that for k vertices to be reachable from l vertices. Therefore, we do not need to do experiments where V_B is larger than V_0. The digraph orders we choose are between 2^{11} and 2^{15}, and the densities are between 2 and 5.

Figure 6.1 shows part of the experimental results. The x-axis displays the size of the bad set V_B, and the y-axis represents the percentage of experiments in which V_B is reachable from V_0. There are three lines in the figure. The dotted line shows the data from experiments with 4 initial vertices, the line with "o" is obtained from experiments with 2 initial vertices and the line "+" is obtained from experiments with 1 initial vertex. Each datum on the lines is averaged from 100 individual experiments performed on digraphs with 2^{15} vertices and density 2. It is observed that when $|V_0| = 1$ and $|V_B| = 1$, V_B is reachable from V_0 in a fairly large number of experiments (68%). The data obtained from experiments with more than 2^2 initial vertices are not shown because the percentage of experiments where V_B is reachable from V_0 is 100%, same as that shown by the dotted line in Figure 6.1. Thus, we come to the conclusion that the existence of even a very small number of bad states will make it very likely that one of bad states is reachable.

6.2 Cost of Finding the Bad Set

When there exist bad states, we hope they are not reachable. However, we can see from the last section that bad states are very likely to be reachable if the number of

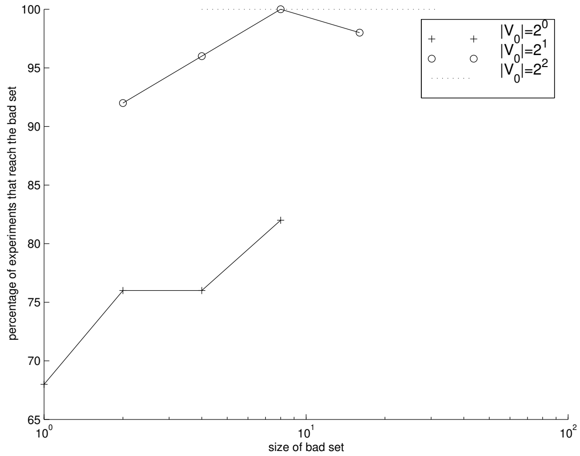


Figure 6.1 Percentage of experiments in which bad set is reachable.

them is not trivial. The next best thing is to find the bad states as early as possible. The primary goals of the set of experiments in this section is to explore which one of the forward, backward and combined search has the least cost to find a bad state. One obvious choice is to compare the three methods by the number of iterations. However, this does not work. The number of iterations used to reach v_j from v_i by using forward search is exactly the same as that to reach v_i from v_j by using backward search. Similarly, there is no difference by starting from both sides. Therefore, we need to use another way to do the comparison. Since all three methods need the same number of iterations, we may examine the amount of work at each iteration. This can be done by checking how many "innocent" vertices are reached before any bad vertices can be reached. The less vertices are searched, the less cost of a certain method.

Figure 6.2 shows the percentage of vertices reached in a digraph during the procedure of searching bad vertices. The experiments are performed on digraphs with one initial vertex, 2^{13} vertices and density 2. The x-axis shows the size of bad set, which is one of 1, 2, 4 or 8; the y-axis shows the percentage of vertices that is reached. There are three lines in the figure. The top line with "+" is obtained from experiments that use backward search; the middle one with "o" represents the forward search and the bottom one with "x" represents the combined search. Each data in the figure is averaged from 100 individual experiments.

Comparing the two lines with "+" and "o" in Figure 6.2, we observe that the forward search reaches a smaller number of vertices than the backward search when the number of bad states is more than 1. For example, when there are 8 bad vertices, less than 20% of vertices are reached in forward search, and more than 40% of vertices are reached in backward search. The difference of the number of reached vertices is not from the methods themselves, but from the different sizes of the initial and bad set. When both the initial and bad set have only one vertex in our experiments, the percentages of vertices reached in digraphs by forward and backward search are the same. When the size of the bad set increases, the forward search reaches bad vertices sooner because there are more bad vertices; this reduces the number of reached vertices. As to the backward search, two facts need to be considered when the size of the bad set increases: the first is that more bad vertices cause the backward search to reach more vertices at each iteration because it starts from a larger bad set; the second is that more bad vertices cause the computation to have fewer iterations. Therefore, we should make a decision that depends on the relative sizes of the initial set and the bad set: choose the forward approach if the initial set is smaller; otherwise, choose the backward approach.

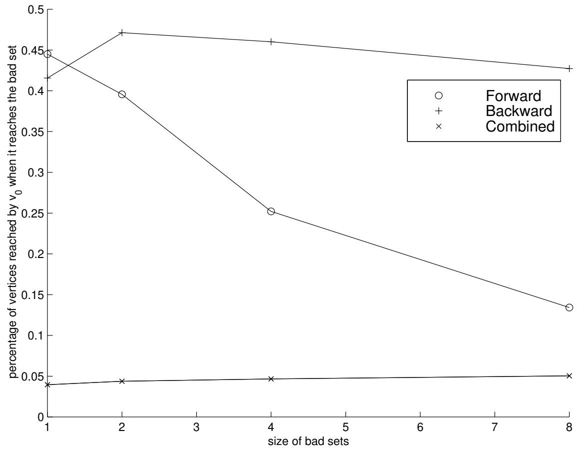


Figure 6.2 Percentage of vertices reached.

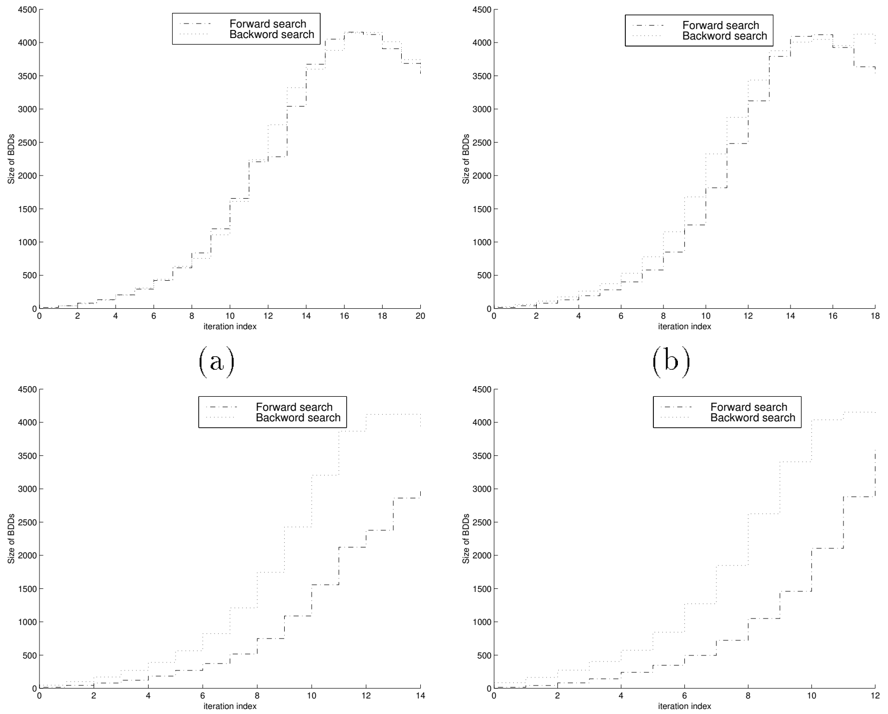
The line with "x" in Figure 6.2 shows the percentage of vertices reached in the procedure of searching bad vertices by using the combined forward and backward search. In the combined approach, an iteration consists of two steps. One is the computation of the forward frontier and the other is the computation of the backward frontier. Compared with the data from the forward and backward search, the combined method reaches many fewer vertices. In the forward search, around 15% to 45% of the vertices are reached and in the backward search around 42% to 47% of the vertices are reached, while in the combined method, only about 4% to 5% of the vertices are reached. The reason that the combined search reaches fewer vertices is that it makes the backward frontier and forward frontier meet each before the frontiers become very large.

6.3 Size of the BDDs at Each Iteration

The primary goal of this section is to check the size of the BDDs in the three search methods. At each iteration of forward search, backward search or combined search, a new frontier is computed and the newly reached vertices are added to the set of already reached vertices. BDDs are needed to keep track of the reached vertices at each iteration.

Figure 6.3 shows part of the experimental data on BDD size. Since we know how the size of the BDDs change when the bad set is not reachable from Chapter 5, here we are only interested in the case when the bad set is reachable. The data in Figure 6.3 are obtained from experiments performed on digraphs with density 2 and 2^{15} vertices. The size of the initial set in all the sub-figures is 1, while the sizes of the bad set are 1, 2, 4, and 8 in (a), (b), (c), and (d) respectively. The x-axis displays the iteration index and the y-axis shows the size of the BDDs that represent the reached vertices. The dashed and dotted lines show the data from forward and backward search, respectively.

First we can see that the number of iterations in the forward and backward search is the same. Also, fewer iterations are needed when the size of the bad set increases. Actually, the number of iterations is the length of the shortest path between any two vertices in the initial set and the bad set. The more vertices in either set, the shorter this path is likely to be. It is observed from (a) that the size of the BDDs at each iteration is almost the same at each iteration for the two methods, while the size of the BDDs is larger from backward search than from forward search in (b), (c) and (d). Also, the difference between the two methods gets bigger as the size of the bad set increases.



(a) (b)

(c) (d)

Figure 6.3 Size of BDDs. (a)$|I| = 1, |B| = 1$. (b)$|I| = 1, |B| = 2$. (c)$|I| = 1, |B| = 4$. (d)$|I| = 1, |B| = 8$.

A simplified forward and backward search using different initial and bad set sizes is shown in Figure 6.4. Starting from the only initial vertex v, the forward search stops when one of the bad vertices b is met. Starting from both of the bad vertices b and b', the backward search stops when v is met by any of the bad vertices. Some vertices are reachable by both b and b'. If the frontier of b reaches v first, those vertices reachable from b' but not reachable from b contribute to the larger size of BDDs. Therefore, the more initial or bad vertices, the larger the BDDs that are needed for forward or backward search, respectively.

As for the combined search, two sets need to be represented by BDDs: a set reached by forward search and the other reached by backward search. Notice that these sets are disjoint, otherwise the computation can stop because we know that the bad set is reachable from the initial set. Since an iteration in combined search consists of the computation of the forward frontier and the computation of the backward

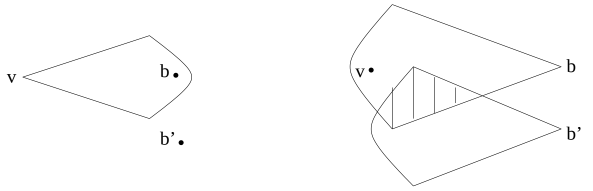


Figure 6.4 Forward and backward search.

frontier, the memory requirement for combined search at each iteration is the sum of the sizes of the BDDs for forward search and backward search at each corresponding iteration, and the number of iterations needed for combined search is half the number needed for forward or backward search. It is easy to observe from Figure 6.3 that the combined approach requires the least memory because the sizes of the BDDs grow more than linearly in the number of iterations. Therefore, we should avoid pure forward search or pure backward search in checking safety properties.

Chapter 7

Model Checking Liveness Properties

In model checking, the verification of liveness properties is reduced to the graph problem of finding a *bad* cycle, a reachable cycle that is not contained in one or more designated sets of states [7, 10, 19, 21]. Therefore, there are two steps. The first step is to detect whether there are bad cycles, and the second is to check whether the bad cycles are reachable from the initial sets. Since most of the second step is covered in the previous chapters, we concentrate on the problem of cycle detection in this chapter.

In explicit-state model checking that uses linked list to represent transition relations, cycle detection may be done in time linear in the size of the program state space, by using depth-first traversal to find the strongly connected components of a graph [19]. However, as mentioned in Chapter 3, the depth-first traversal is incompatible with the routines used to efficiently manipulate BDDs. We present in this chapter two existing bad cycle detection algorithms, Emerson-Lei (EL) [28, 29] and Catch-them-young (CTY) [14], which are used in symbolic model checking and compare their efficiency. We realize some inefficiency in CTY and improve the algorithm. We call the improved one CTY$^+$. We also introduce a new algorithm One-way-catch-them-young (OWCTY) which shows superior performance over the others.

Before our discussion, we introduce some concepts and denotations which are used throughout the whole chapter. In a state space S, a designated set of states $\mathcal{C}$ is a *cycle set* if cycles are allowed to exist within $\mathcal{C}$. $\mathcal{F}$ is called a *fair set*, where $\mathcal{F} = S - \mathcal{C}$. A cycle C is *good* if and only if there exists a cycle-set $\mathcal{C}$, such that for every $s \in C$, $s \in \mathcal{C}$. In other words, a cycle is *bad* if and only if it passes through the intersection of all fair sets. A state s is *bad* if it is on a bad cycle.

7.1 Description of the Emerson-Lei Algorithm

Before Emerson and Lei [12] introduced a formula that expressed a fairness constraint, the bad cycle detection in symbolic model checking was based on transitive closure described as follows. Let $T(x,y)$ be a transition relation. Let $R(x,y)$ be the transitive closure of the transition relation $T(x,y)$. $R(x,y)$ is equal to the least fixed point of F, $lfp(F)$, where F is the following function:

$$F(c)(x,y) = T(x,y) + \exists z(c(x,z) \cdot c(z,y)). \tag{7.1}$$

$R(x,y) = 1$ if and only if y is reachable from x. Then $R(x,y) \cdot R(y,x)$ defines the set of cycles involving both x and y. Let $\mathcal{F}_1, ..., \mathcal{F}_n$ be the fair-sets in a state space S. For each $\mathcal{F}_i$, a set $NC_{\mathcal{F}_i}$ is computed as follows:

$$NC_{\mathcal{F}_i}(x) = \exists y(R(x,y) \cdot R(y,x) \cdot \mathcal{F}_i(y)).$$

A state $s \in NC_{\mathcal{F}_i}$ if and only if there is a cycle C involving s that passes through $\mathcal{F}_i$.

> **Theorem 7.1** [28, 29] There is no bad cycles if and only if NC is empty, where $NC = \cap_{1 \le i \le n} NC_{\mathcal{F}_i}$.

The algorithm based on transitive closure requires only a linear number of steps to find bad cycles, following a logarithmic number of BDD manipulation iterations to compute the transitive closure of a transition relation. The most computationally expensive step in the algorithm is the computation of the transitive closure. It relies on the fixed point computation of Equation 7.1, which requires three sets of simultaneously active state variables. The intensive memory requirement makes it infeasible in real applications [16].

The EL algorithm avoids the computation of the transitive closure $R(x,y)$ of the transition relation $T(x,y)$. Instead of trying to find the set NC, the EL algorithm computes another set NC_μ. Touati, Brayton and Kurshan show that emptiness of NC is equivalent to the emptiness of NC_μ by the following theorem:

> **Theorem 7.2** [29] NC_μ is the set of states from which NC can be reached.

The evaluation of NC_μ requires a double nesting of fixed point computations: it is obtained as the greatest fixed point of the following function F:

$$F(c)(x) = c(x) \wedge \bigwedge_{i=1}^{n} \exists y(T(x,y) \wedge lfp(G_i^c)(y)), \tag{7.2}$$

then for a given set c, G_i^c is defined as follows:

$$G_i^c(d)(x) = c(x) \wedge (\mathcal{F}_i(x) \vee \exists y(T(x,y) \wedge d(y)). \tag{7.3}$$

It can be observed that each least fixed point computation involves a number of iterations that are linear in the diameter of the state space, and the number of times the least fixed point computation is executed is linear in the size of the state space. Therefore, the EL algorithm has time complexity that is quadratic in the size of underlying state space. Although the transitive-closure algorithm requires only the time linear in the size of the state space to find bad cycles after the closure of transition relation is built, the EL algorithm advantageously trades the quadratic increase in algorithmic steps for a decrease in the number of steps required to manipulate the associated BDDs, resulting in fewer steps overall.

7.2 Description of the Catch-Them-Young Algorithm and Its Improvement

Hardin *et al.* [14] proposed a new heuristic, CTY, to look for bad cycles. We call a cycle bad *relative to* a fair set $\mathcal{F}_i$ if the cycle passes through $\mathcal{F}_i$. The central idea of the new heuristic is to use a combined forward and backward reachability computation to find out if there is a cycle-set $\mathcal{C}_i = S - \mathcal{F}_i$ that contains all the bad cycles relative to $\{\mathcal{F}_1, ..., \mathcal{F}_{i-1}, \mathcal{F}_i, ..., \mathcal{F}_n\}$. If that is the case for some i, then there is no bad cycle.

Let $B^\#$ be the set of states through which some bad cycle potentially may pass. The algorithm continuously refines $B^\#$ until $B^\#$ becomes a fixed point, which is B. Let

$$^*\mathcal{F}_i = \{x |\text{ there is a path from state } x \text{ to some state in } \mathcal{F}_i\},$$
$$\mathcal{F}_i^* = \{x |\text{ there is a state in } \mathcal{F}_i \text{ from which there is a path to } x\}$$

The sets $^*\mathcal{F}_i$ and $\mathcal{F}_i^*$ can be computed using standard backward and forward reachability methods described in Chapter 2.

Lemma 7.1 [14] Let $\mathcal{F}_i^\# = {}^*\mathcal{F}_i \cap \mathcal{F}_i^*$, $\mathcal{F}^\# = \cap_{i=1}^n \mathcal{F}_i^\#$. Then $\mathcal{F}^\#$ contains all the bad states.

Note that $\mathcal{F}^\#$ is not the set of bad states because there may be some states in $\mathcal{F}^\#$ that are not on any bad cycle. The first approximation to the set of bad states is given as $B^\# = \mathcal{F}^\#$. In the algorithm, $B^\#$ is continuously refined until either $B^\#$ is covered by some cycle-set, at which point we can stop since no bad cycle can exist, or the approximation reaches a fixed point where no more states can be thrown out of $B^\#$. The CTY algorithm is based on the observations of Lemma 7.1 and 7.2, both of which help to identify early on some states that do not belong to the set of bad states.

> **Lemma 7.2** [14] If $s \in B$ then there must be a predecessor s' of s in the state reachability graph such that $s' \in B$. Similarly, if $s \in B$ then there must be a successor s' of s in the state reachability graph such that $s' \in B$.

From the lemma above, we may identify those states that have none of their predecessors or none of their successors in $B^{\#}$ to be good states. Hence, the next step is to change the characteristic function of $B^{\#}$ to a new one as follows:

$$B^{\#}(x) := B^{\#}(x) \wedge (\exists y(T(x,y) \wedge B^{\#}(y))) \wedge (\exists y(T(y,x) \wedge B^{\#}(y))). \tag{7.4}$$

We keep deleting the states without either predecessors or successors with the formula above until a fixed point is reached. At this point, $B^{\#}$ contains all the bad states and possibly some good states. We continue the whole procedure by first removing those states in $\mathcal{F}_i$ that are not bad states. This is accomplished by $\mathcal{F}_i = \mathcal{F}_i \cap B^{\#}$. Then we again compute $\mathcal{F}^{\#}$ by using the modified $\mathcal{F}_i$.

We notice that $B^{\#}$ is the approximation set which contains all bad states. In other words, those states outside $B^{\#}$ are definitely not bad states. Therefore, we do not need to consider the states outside $B^{\#}$ when we search for bad states in the reachability computation. This idea leads us to evaluate the following two sets instead of $^*\mathcal{F}_i$ and $\mathcal{F}_i^*$:

$^*\mathcal{G}_i = \{x|$ there is a path contained in $B^{\#}$ from state x to some state in $\mathcal{F}_i\}$,
$\mathcal{G}_i^* = \{x|$ there is a path contained in $B^{\#}$ from some state in $\mathcal{F}_i$ to state $x\}$.

Let $\mathcal{G}_i^{\#} = {}^*\mathcal{G}_i \cap \mathcal{G}_i^*$, $\mathcal{G}^{\#} = \cap_{i=1}^{n} \mathcal{G}_i^{\#}$. It is obvious that $\mathcal{G}^{\#} \subseteq B^{\#}$. $\mathcal{G}^{\#}$ captures all the states inside $B^{\#}$ that may be on bad cycles. Therefore, $\mathcal{G}^{\#}$ still contains all the bad states. Also, because $\mathcal{G}^{\#} \subseteq \mathcal{F}^{\#}$, we expect that CTY$^+$, which manipulates $\mathcal{G}^{\#}$ instead of $\mathcal{F}^{\#}$, performs better.

7.3 Implementation of the Emerson-Lei Algorithm and the Catch-Them-Young Algorithm

This section provides the pseudocode for the algorithms EL and CTY. We choose the number of fair sets to be one, which enables us to examine the fundamental differences between the algorithms without going into too many unnecessary details.

For bad cycle detection based on EL, we need to compute the greatest fixed point of F and the least fixed point of G^c. F and G^c are defined as follows, derived from

Equations 7.2 and 7.3:

$$F(c)(x) = c(x) \wedge \exists y(T(x,y) \wedge lfp(G^c)(y)), \tag{7.5}$$

$$G^c(d)(x) = c(x) \wedge (\mathcal{F}(x) \vee \exists y(T(x,y) \wedge d(y)). \tag{7.6}$$

Since F and G^c are formally monotonic and the state space is finite, in the light of Lemma 2.4, there is an integer $i(c)$ such that $lfpG^c(d) = (G^c)^{i(c)}(d)(false)$. Similarly, there is some j such that $gfpF(c) = F^j(c)(true)$. The implementation shown in Figure 7.1 is based on this observation.

```
PROCEDURE gfpF
BEGIN
1.  newC := TRUE;
2.  REPEAT
3.  BEGIN
4.      oldC := newC;
5.      newC := oldC $\wedge\exists y(T(x,y)\wedge$ lfpG(oldC)(y));
6.  UNTIL (newC = oldC);
7.  RETURN oldC
END

FUNCTION lfpG(c)
BEGIN
1.  newD := FALSE;
2.  REPEAT
3.      oldD := newD;
4.      newD := $c \wedge (\mathcal{F} \vee \exists y(T(x,y) \wedge oldD(y)))$
5.  UNTIL (newD = oldD);
6.  RETURN oldD
END
```

Figure 7.1 Pseudocode for EL.

Next we give an example to show how the EL algorithm works. Figure 7.2 is a digraph $D(V,E)$ with fair-set $\mathcal{F} = \{4,5,6\}$ as denoted by the dashed box. Notice that there is a bad cycle $1 \rightarrow 4 \rightarrow 2 \rightarrow 1$. We trace the sets denoted by c and d to follow the computation. At the beginning, c contains all the vertices and d is empty. Table 7.1 lists the set d each time the computation reaches line 6 in FUNCTION

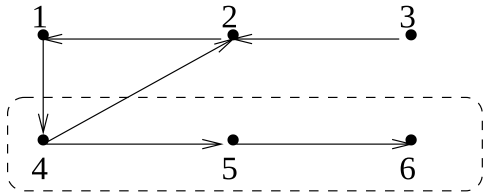


Figure 7.2 A digraph showing how different algorithms work for checking liveness properties.

lfpF(c) and the set c each time the computation reaches line 6 in PROCEDURE *gfpF*. When the set c is not changed from the previous iteration, we obtain the set NC_μ, which equals $\{1,2,3,4\}$. NC_μ is the set of vertices that can reach the bad cycle $1 \to 4 \to 2 \to 1$.

external iteration	d	c
1	{1,2,3,4,5,6}	{1,2,3,4,5}
2	{1,2,3,4,5}	{1,2,3,4}
3	{1,2,3,4}	{1,2,3,4}

Table 7.1 Evaluation procedure for EL.

Next we implement the CTY algorithm. If we denote by $f, f^*, {}^*f, f^\#$, and b the characteristic functions of $\mathcal{F}, \mathcal{F}^*, {}^*\mathcal{F}, \mathcal{F}^\#$, and $B^\#$ respectively, $f^\#$ can be computed as follows:

```
$f := f \wedge b;$
$f^\# := f^* \wedge {}^*f$
```

Let b^+ and ${}^+b$ denote the characteristic functions of the set of successors of states in $B^\#$, and the set of predecessors of states in $B^\#$ respectively. The next approximation $B^\#$ can be computed as follows:

```
$b := f^\#;$
WHILE $b$ changes
BEGIN
    $b := b \wedge b^+;$
    $b := b \wedge^+ b$
END
```

The complete algorithm is listed in Figure 7.3.

```
PROCEDURE CTY
BEGIN
1.  $b := true$;
2.  $f := f \wedge b$;
    $f^{\#} := f^* \wedge {}^*f$;
3.  $b := f^{\#}$;
    WHILE $b$ changes
    BEGIN
        $b := b \wedge b^+$;
        $b := b \wedge^+ b$
    END
4.  IF $b$ is changed in step 3
        go to step 2
    ELSE
        go to step 5;
5.  IF $b$ = NULL then
        there is no bad cycle
    ELSE
        $b$ contains bad cycle;
END
```

Figure 7.3 Pseudo code for CTY.

We use Figure 7.2 again to show how CTY works. Initially at step 2 of Figure 7.3, $f = \{4, 5, 6\}$. Therefore, $f^* = \{1, 2, 4, 5, 6\}$ and ${}^*f = \{1, 2, 3, 4, 5, 6\}$, which makes the value of b at step 3 be $\{1, 2, 4, 5, 6\}$. At step 3, those vertices in b without successors or predecessors inside b are removed from the set until b is empty or we reach a fixed point. At the beginning of step 4, $b = \{1, 2, 4\}$. Since b is changed from its initial set $\{1, 2, 4, 5, 6\}$, a second external iteration is needed. This time at step 2, $f = f \wedge b = \{4\}$, $f^* = \{1, 2, 4, 5, 6\}$ and ${}^*f = \{1, 2, 3, 4\}$. So $b := f^* \wedge {}^*f = \{1, 2, 4\}$ at the beginning of step 3. Since all vertices in b have successors and predecessors, b remains unchanged after step 3. Therefore, only two external iterations are needed to find the bad cycle $\{1, 2, 4\}$ in Figure 7.2 by CTY.

If we denote by g^*, ${}^*g$ and $g^{\#}$ the characteristic functions of $\mathcal{G}^*$, ${}^*\mathcal{G}$ and $\mathcal{G}^{\#}$ respectively, the pseudocode for CTY$^+$ is the same as in Figure 7.3, except that the second line in step 2 and the first line in step 3 should be replaced by:

```
$g^{\#}$ := $g^{*} \wedge {}^{*}g$;
$b$ := $g^{\#}$;
```

We use Figure 7.2 again to show how CTY^{+} works. When the execution reaches step 2 of Figure 7.3 for the first time, $g = \{4, 5, 6\}$. Since b is assigned *true* at step 1, we get $g^{*} = \{1, 2, 4, 5, 6\}$ and ${}^{*}g = \{1, 2, 3, 4, 5, 6\}$, which makes the value of b at step 3 be $\{1, 2, 4, 5, 6\}$. At step 3, those vertices in b without successors or predecessors inside b are removed from the set until b is empty or reaches a fixed point. At the beginning of step 4, $b = \{1, 2, 4\}$. Since b is changed from the previous set $\{1, 2, 4, 5, 6\}$ at step 2, a second external iteration is needed. This time at step 2, $b = \{1, 2, 4\}$ and $f := f \wedge b = \{4\}$. Because $g^{*} = \{1, 2, 4\}$ and ${}^{*}g = \{1, 2, 4\}$, $b := g^{*} \wedge {}^{*}g = \{1, 2, 4\}$ at the beginning of step 3. Since all vertices in b have successors and predecessors, b remains unchanged after step 3. Therefore, the execution jumps to step 5 and b contains the bad cycle.

7.4 Description and Implementation of the One-Way-Catch-Them-Young Algorithm

Based on the descriptions of EL and CTY, we can observe the differences in their computation patterns. At each external iteration, there is one fixed point computation in EL, while in CTY there are three: one for forward reachability, one for backward reachability and one for approximation of B. We expect that EL will involve more external iterations because it does less work in its internal iterations. A fundamental difference in the computation method is that CTY involves both image and preimage computations, while EL uses only preimage computations.

We introduce a new heuristic that falls in the middle of the two algorithms. The basic ideas are rooted from CTY, which refines $B^{\#}$ until $B^{\#}$ becomes a empty set or fixed point. However, like EL, it involves only preimage computations. No image computation is ever used in the whole computation. We call the new approach One-way-catch-them-young (OWCTY) because it is CTY with only one direction. First we define a set ${}^{*}\mathcal{H}_i$ that is similar to ${}^{*}\mathcal{F}_i$ and ${}^{*}\mathcal{G}_i$:

${}^{*}\mathcal{H}_i = \{x|$ there is a path contained in $B^{\#}$ from state x to some state in $\mathcal{F}_i\}$.

Note that the computation of ${}^{*}\mathcal{H}_i$ is same as ${}^{*}\mathcal{G}_i$, but not ${}^{*}\mathcal{F}_i$. Let $\mathcal{H}_i^{\#} = {}^{*}\mathcal{H}_i$, $\mathcal{H}^{\#} = \cap_{i=1}^{n}\mathcal{H}_i^{\#}$. It is obvious that $\mathcal{G}^{\#} \subseteq \mathcal{H}^{\#}$. Thus, $\mathcal{H}^{\#}$ contains all the bad states. Let

$^*\mathcal{H} = \{x|$ there is a path contained in $B^\#$ from state x to some state in $\mathcal{H}\}$.

If we denote by $^*h$ and $h^\#$ the characteristic functions of $^*\mathcal{H}$ and $\mathcal{H}^\#$ respectively, the pseudocode shown in Figure 7.4 for OWCTY is similar to Figure 7.3.

```
PROCEDURE OWCTY
BEGIN
1.  $b$ := true;
2.  $f := f \wedge b$;
    $h^\# := {}^*h$;
3.  $b := f^\#$;
    WHILE $b$ changes
    BEGIN
        $b := b \wedge b^+$;
        $b := b \wedge^+ b$
    END
4.  IF $b$ is changed in step 3
        go to step 2
    ELSE
        go to step 5;
5.  IF $b$ = NULL then
        there is no bad cycle
    ELSE
        $b$ contains bad cycle;
END
```

Figure 7.4 Pseudocode for OWCTY.

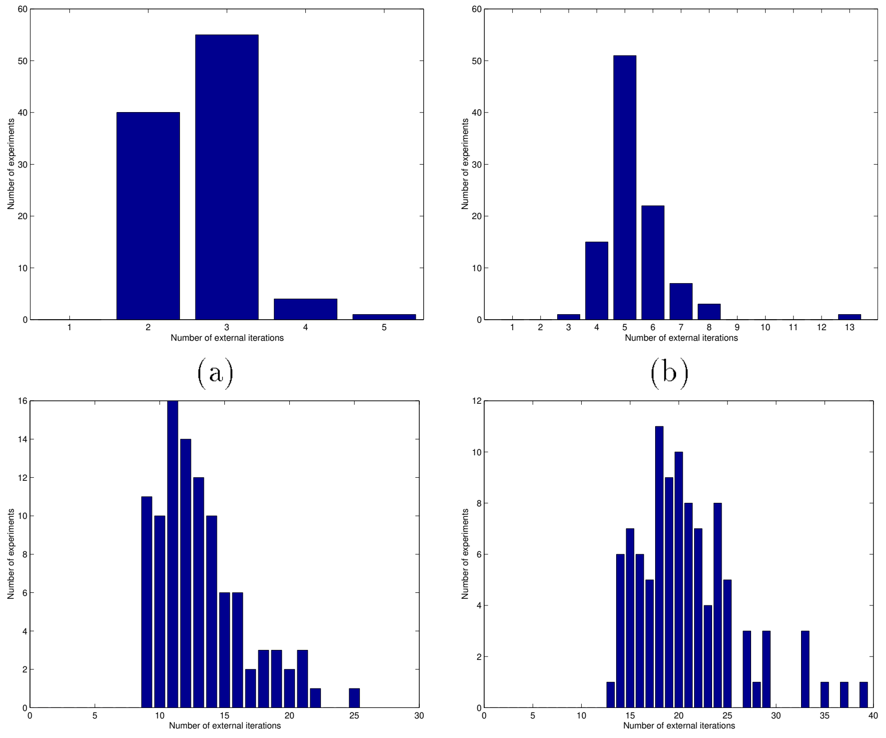
Again, we use Figure 7.2 to show how OWCTY works. When the excution reaches step 2 of Figure 7.4 for the first time, $h = \{4, 5, 6\}$. Since b is assigned *true* at step 1, we have $^*h = \{1, 2, 3, 4, 5, 6\}$, which makes the value of b at step 3 be $\{1, 2, 3, 4, 5, 6\}$. At step 3, those vertices in b without successors inside b are removed from the set until b is empty or reaches a fixed point. At the beginning of step 4, $b = \{1, 2, 3, 4\}$. Since b is changed from the previous set $\{1, 2, 3, 4, 5, 6\}$ at step 2, a second external iteration is needed. This time at step 2, $b = \{1, 2, 3, 4\}$ and $f := f \wedge b = \{4\}$. Because $h^\# := {}^*h = \{1, 2, 3, 4\}$, $b := h^\# = \{1, 2, 3, 4\}$ at the beginning of step 3. Since all vertices in b now have successors inside b, b remains unchanged after step 3. Therefore, the execution jumps to step 5 and b contains the bad cycle.

7.5 Number of External Iterations

In the rest of this chapter, we do experiments to compare the EL, CTY, CTY$^+$ and OWCTY algorithms. As before, we perform all of the experiments on digraphs drawn from $D_{n,d/n}$. In each digraph $D(V, E)$, we designate a fair set. Any cycles that pass through the fair set are bad cycles. We choose the digraph order n to be 2^{12}, which is large enough to explore the behavior of the algorithms as well as to enable the experiments to be finished in a reasonable time. The densities are 1.2, 1.6, 2.0 or 2.4. Another important parameter is the size of the fair set. We let the number of vertices in the fair set to be $n \cdot 1\%, n \cdot 10\%, n \cdot 30\%, n \cdot 50\%, n \cdot 70\%$ or $n \cdot 90\%$, where n is the digraph order. We did 12 sets of experiments with parameters chosen from the combinations of different densities and sizes of fair sets. For each set of experiments, we did 100 individual experiments and analyzed the results. We divided the experiments into two groups. Let d denote the density and $|\mathcal{F}|$ denote the size of the fair set. The combinations of $(d, |\mathcal{F}|)$ in the first group are $(1.2, n \cdot 1\%), (1.2, n \cdot 10\%), (1.2, n \cdot 30\%), (1.2, n \cdot 50\%), (1.2, n \cdot 70\%)$, and $(1.2, n \cdot 90\%)$, which have the same density but different sizes of fair-sets. The experiments in this group enable us to examine the effect of $|\mathcal{F}|$ on the three algorithms. The combinations of $(d, |\mathcal{F}|)$ in the second group are $(1.2, n \cdot 90\%)(1.6, n \cdot 90\%)$, $(2.0, n \cdot 90\%)$ and $(2.4, n \cdot 90\%)$, which have same size of fair set but different densities. This enables us to examine the effect of densities on the algorithms.

In EL, the existence of bad cycles depends on the emptiness of NC_μ. NC_μ is obtained as the greatest fixed point of the function F shown in Equation 7.5. Evaluation of F requires the computation of the greatest fixed point of the set c. At each evaluation step of F, the computation of the least fixed point of G^c is needed. Thus, the computation of the greatest fixed point forms the external iterations and the computation of the least fixed point forms the internal iterations. As to CTY, CTY$^+$ and OWCTY, the pseudocode in Figure 7.3 and 7.4 shows that the computation pattern is 2, 3, 3,..., 3; 2, 3, 3,..., 3; ... ; 2, 3, 3,..., 3, 5; We count the number of step 2 appearing in those procedures as the number of external iterations.

Figure 7.5 shows graphically the distribution of the number of external iterations for EL. The x-axis shows integers indicating the number of external iterations; the y-axis displays the number of experiments that has the same number of external iterations as shown on the x-axis. The figure consists of four bar-graphs. Each bar-graph shows the results obtained from 100 individual experiments performed on



(a) (b)

(c) (d)

Figure 7.5 Distribution of the number of external iterations for EL with digraph order 2^{12} and density 1.2. (a)$|\mathcal{F}| = n \cdot 1\%$; (b)$|\mathcal{F}| = n \cdot 10\%$; (c)$|\mathcal{F}| = n \cdot 50\%$; (d)$|\mathcal{F}| = n \cdot 90\%$.

digraphs with $n = 2^{12}$ and $d = 1.2$. The sizes of fair sets for the four bar-graphs are different. The $|\mathcal{F}|$'s for (a), (b), (c) and (d) are $n \cdot 1\%$, $n \cdot 10\%$, $n \cdot 50\%$ and $n \cdot 90\%$ respectively.

All the 100 experiments in (a) of Figure 7.5 have fewer than 6 external iterations. As $|\mathcal{F}|$ increases, the number of external iterations grows. When $|\mathcal{F}| = n \cdot 90\%$ as shown in (d), 94 out of the 100 experiments have external iterations between 14 and 29, while the smallest number is 13 and largest is 39.

In order to explain the phenomenon that the number of external iterations increases with the size of fair set, we must understand what factors affect the number of external iterations in EL.

Definition 7.1 Given a digraph $D(V, E)$ and a set $K \subseteq V$, let $p = x_1x_2...x_k$ be the longest path such that for all i that $(1 \leq i < k)$,

$(x_{i-1}, x_i) \in E$, $x_i \in K$ and x_i is not on any strongly connected components. The length of p is called the straight diameter of K and represented by $sd(K)$. x_k is called the tail of p.

For a digraph $D(V, E)$ with a fair set $\mathcal{F}$, the straight diameter of $\mathcal{F}$ determines the number of external iterations in EL when there is no bad cycle.

Theorem 7.3 The number of external iterations of EL in a digraph $D(V, E)$ with a fair set $\mathcal{F}$ is $sd(\mathcal{F})$, provided there is no bad cycle.

Proof: The result returned by the computation of the least fixed point of G^c is a set $K(c)$ of vertices that can reach $c \wedge \mathcal{F}$. Then at line 5 of PROCEDURE *gfpF*, a new set c' is obtained by removing the vertices in $K(c)$ that do not have successors inside c. Then c is replaced by c' and passed to FUNCTION *lpfG* to continue the computation. With initial value *true*, c functions as a care set. In other words, we do not need to consider the vertices outside c at any time. Since there are no bad cycles, this procedure continues until $c \wedge \mathcal{F} = \emptyset$. Also, the length of the longest path p inside $\mathcal{F}$ is the straight diameter of $\mathcal{F}$ because no vertex on p can be in a strongly connected component.

Suppose the number of external iterations is smaller than $sd(\mathcal{F})$, then at least two vertices from p are deleted at some iteration. However, all vertices except the tail of the remaining p have successors at any step. Therefore, at most one vertex from p is deleted at each iteration. Now suppose the number of external iterations is larger than $sd(\mathcal{F})$. This means that none of the vertices from p is deleted at some iteration. That is, all vertices on the remaining p have successors. Because there are no cycles, there must exist a path longer than p and this path shares some vertices with p. This conclusion contradicts our assumption that p is the longest path. Thus, the number of external iterations is $sd(\mathcal{F})$, which is the number of vertices on p. $\square$

The case where there exist bad cycles is more complicated. The vertices in a digraph can be divided into two parts: those that fall into a strongly connected component and those not on any strongly connected component.

Definition 7.2 For a strongly connected component C in a digraph $D(V, E)$, the set $C' = \{x | x$ is reachable from C but cannot reach any strongly connected components$\}$ is called the *peripheral set* of C. Correspondingly, a peripheral path of C is a path in $D(V, E)$ such that all vertices on the path are in C'.

Notice that $C \cap C' = \emptyset$. Since $D(V, E)$ is finite, the number and length of peripheral paths of every strongly connected components are finite. We define the peripheral length of C with respect to a set K as follows:

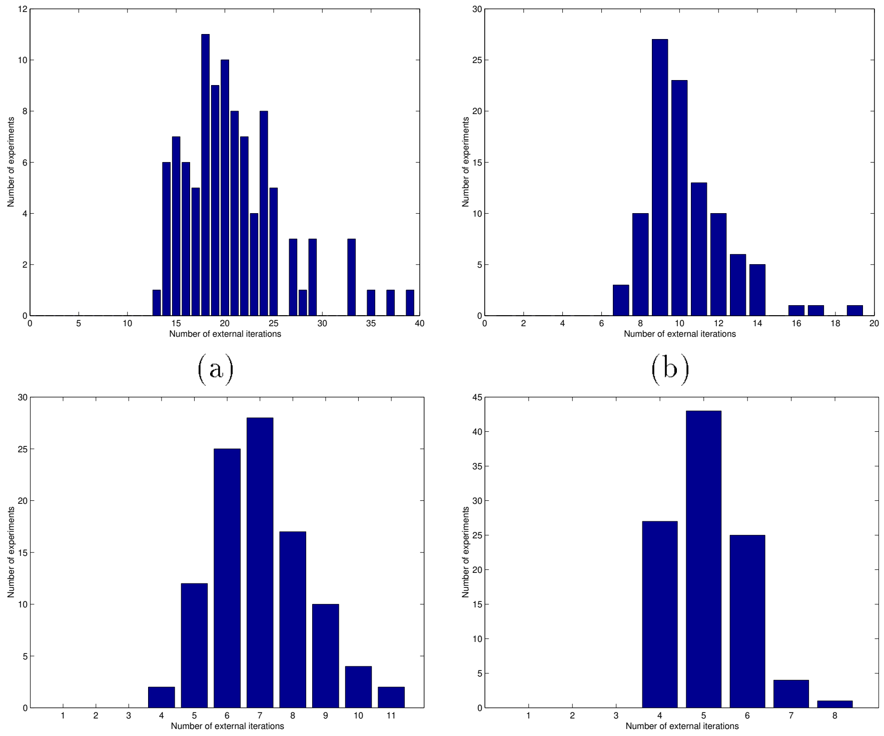
> **Definition 7.3** Let $p_1, p_2, ..., p_k$ be peripheral paths of a strongly connected component C. The peripheral length of C with respect to a set K is
>
> $$pl(C, K) = MAX_{i=i}^{n}(|S_i|), \text{ where } S_i = p_i \cap K (1 \leq i \leq k).$$

The following theorem shows that the number of external iterations for a digraph $D(V, E)$ under EL is related to both the straight diameter and peripheral length of the fair set $\mathcal{F}$ in $D(V, E)$.

> **Theorem 7.4** Let $D(V, E)$ be a digraph with a fair set $\mathcal{F}$. Suppose there are k strongly connected components $C_1, C_2, ..., C_k$. Let $pl(\mathcal{F}) = MAX_{i=i}^{k} pl(C_i, \mathcal{F})$. The number of external iterations to detect bad cycles by using EL is MAX$(sd(\mathcal{F}), pl(\mathcal{F}))$.

Proof: If there is no bad cycle, then the number of external iteration is $sd(\mathcal{F})$ in the light of Theorem 7.3. If there are bad cycles, the computation should terminate if the current set c is same as the previous one. Consider a cycle C that passes through $\mathcal{F}$. We need $pl(C, \mathcal{F})$ iterations to remove the peripheral path of C with respect to $\mathcal{F}$. For all the cycles that pass through $\mathcal{F}$, $pl(\mathcal{F})$ external iterations are needed to remove all the peripheral paths with respect to $\mathcal{F}$. Since we also need to remove those vertices on the straight diameter of $\mathcal{F}$, MAX$(sd(\mathcal{F}), pl(\mathcal{F}))$ external iterations are needed.$\square$

It can be observed that the straight diameter and peripheral length of a set K is closely related to the size of K. Consider a digraph $D(V, E)$ drawn from $D_{n,d/n}$ and two sets $K_1, K_2 \subseteq V$ whose vertices are randomly chosen from V. Suppose $|K_1| < |K_2|$. Since the two sub-digraphs $D(K_1, E_{K_1})$ and $D(K_2, E_{K_2})$ have similar structures, it is very likely that $sd(K_1) \leq sd(K_2)$ and $pl(K_1) \leq pl(K_2)$. Consider the case $K_1 \subset K_2$, then $sd(K_1) \leq sd(K_2)$ and $pl(K_1) \leq pl(K_2)$ for sure. In our experiments, the experiments are performed on digraphs with the same order and density, but different sizes of fair sets. We may assume that the experiments are performed on the same digraph with independent fair sets. The larger the fair set, the more likely there will be more external iterations.



(a) (b) (c) (d)

Figure 7.6 Distribution of external iterations for EL with fixed digraph order and size of fair sets: (a)$d = 1.2$; (b)$d = 1.6$; (c)$d = 2.0$; (d)$d = 2.4$.

Figure 7.6 shows graphically the distribution of the number of external iterations for EL with fixed digraph order and size of fair sets. The x-axis and y-axis have the same meaning as in Figure 7.5. The figure consists of four bar-graphs. Each bar-graph shows the results from 100 individual experiments performed on digraphs with $n = 2^{12}$ and $|\mathcal{F}| = n \cdot 90\%$. The densities vary from one bar-graph to another. All the 100 experiments in (a) are performed on digraphs with density $d = 1.2$, while the densities for the (b), (c) and (d) are 1.6, 2.0 and 2.4 respectively. It can be observed from Figure 7.6 that the number of external iterations decreases as the density increases from 1.2 to 2.4.

Consider a digraph $D(V, E)$ with a fair set $\mathcal{F}$ and density d. EL proceeds by refining the set c. At line 1 in PROCEDURE *gfpF*, c is initialized to be *true*. After the line 5 in PROCEDURE *gfpF* is executed for the first time, c becomes the set of vertices that have successors in K, where K is the set of vertices that can reach the fair set $\mathcal{F}$. Let us use c_i to denote the set c after line 5 in PROCEDURE *gfpF* is executed for the ith time and K_i to denote the set returned by FUNCTION *lfpG(c_{i-1})*. It is obvious that $c_{i+1} \subseteq c_i$. The iteration is ended when $c_k = c_{k-1}$. Since the digraph is finite, n is finite. c_k is either the empty set or the set of vertices that can reach bad cycles. At the ith iteration, those vertices in K_i that do not have successors are removed. Given another digraph $D'(V, E')$ with density $d' > d$, then fewer vertices are removed at each iteration than those in digraph $D(V, E)$. This may lead to the conclusion that larger density result in more external iterations. However, Figure 7.6 shows that the number of iterations decreases as densities increase. The reason is that another factor needs to be considered here, *i.e.*, the size of the final set c_k. The following paragraph proves that the final set c_k gets larger if the digraph density becomes bigger, which may results in fewer external iterations.

Theorem 3.9 depicts a picture of typical random digraphs. It shows that a digraph drawn from $D_{n,d/n}$ is very likely to have exactly one giant strong component with size $\Theta^2 n$, where Θ is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. When $d > 1$, Θ increases as the density d becomes larger. Particularly, the values of Θ and the sizes of the giant strong component for the densities we choose in our experiments are shown in Table 7.2. Theorem 3.10 states that nearly all the vertices of a digraph drawn from $D_{n,c/n}$ lie either on the giant strong component G_1 or on a strong component of size 1. In particular, almost certainly, all the cycles in the graph lie on the giant strong component. Therefore, c_k consists of the vertices in $\mathcal{F}$ that are inside the giant component or can reach the component. Let G_2 be the set of vertices that can reach G_1. From Theorem 3.10 and 3.8, the size of G_2 is $(\Theta - \Theta^2)n$. $\mathcal{F}_n$ is very likely to be $(G_1 \cup G_2) \cap \mathcal{F}$. Since $G_1 \cap G_2 = \phi$ and the vertices in the fair-set are randomly chosen, $|\mathcal{F}_n| = |\mathcal{F}| \cdot \frac{|G_1|+|G_2|}{n} = |\mathcal{F}_n|\Theta$. Since Θ increases with digraph density, the final set $|\mathcal{F}_k|$ is bigger when digraph density gets bigger.

density	Θ	Θ^2	$\Theta^2 n$
1.2	0.311	0.097	397.3
1.6	0.642	0.412	1687.5
2.0	0.799	0.638	2613.2
2.4	0.879	0.772	3161.8

Table 7.2 Value of Θ.

In this set of experiments, the digraph orders and the size of fair sets are fixed. As the density increases, $|G_1|$ and $|G_2|$ get larger. Therefore, if there are no bad cycles in a digraph $D(V, E)$ with cycle set C, that is, if c_k is empty, then more iterations are needed when the density increases because fewer vertices are deleted from $\mathcal{F}$ at each iteration. The case where no bad cycles exist is rare. Consider the case where $d = 1.2$ and $|\mathcal{F}| = n \cdot 1\%$. That is, the most likely case in our experiments that there is no bad cycle. Assume that there is only one strong component with $\Theta^2 n = 0.097 \cdot 2^{12}$ vertices, the probability that there is no bad cycle is $\binom{|\mathcal{F}|}{n-\Theta^2 n}/\binom{|\mathcal{F}|}{n} = 0.015$. Therefore, with very high probability all the experiments have bad cycles. When there do exist bad cycles, two factors need to be considered: the number of vertices removed at each iteration and the size of $\mathcal{F}$. The experiments show that larger fixed point of $\mathcal{F}$ overwhelmed deleting more vertices at each step. So larger density results in less external iterations in EL.

CTY, CTY$^+$ and OWCTY have fewer external iterations than EL. Table 7.3 lists the number of external iterations that are averaged from 100 experiments performed on digraphs with order $n = 2^{12}$ and density $d = 1.2$. The sizes of fair sets range from $n \cdot 1\%$ to $n \cdot 90\%$. For comparison, we also list the number of iterations for EL.

	$\lvert F\rvert = n \cdot 1\%$	$\lvert F\rvert = n \cdot 10\%$	$\lvert F\rvert = n \cdot 50\%$	$\lvert F\rvert = n \cdot 90\%$
CTY	2.18	2.41	2.09	2.00
CTY$^+$	2.18	2.41	2.09	2.00
OWCTY	2.17	2.37	2.07	2.00
EL	2.66	5.36	13.20	20.89

Table 7.3 Average number of external iterations on digraphs with order 2^{12}, density 1.2 and different sizes of fair sets.

Table 7.4 shows the number of external iterations averaged from 100 experiments performed on digraphs with order $n = 2^{12}$ and size of the fair sets $|\mathcal{F}| = n \cdot 90\%$, while the densities range from 1.2 to 2.4.

It can be observed from the two tables that the number of external iterations for CTY, CTY$^+$ and OWCTY is much less than that of EL. However, we cannot come to the conclusion that CTY and OWCTY are more efficient than EL. As discussed before, CTY and OWCTY do more work in inner loops. The number of image computations, which will be analyzed in section 7.7, offers a more fair comparison.

	$d = 1.2$	$d = 1.6$	$d = 2.0$	$d = 2.4$
CTY	2.00	2.00	2.00	2.00
CTY$^+$	2.00	2.00	2.00	2.00
OWCTY	2.00	2.00	2.00	2.00
EL	20.89	10.37	7.02	5.09

Table 7.4 Average number of external iterations on digraphs with order 2^{12}, size of the fair sets $2^{12} \cdot 90\%$ and different densities.

7.6 Size of Approximation Set

The common point among EL, CTY, CTY$^+$ and OWCTY is that all of them start with an approximation set(c for EL, b for CTY, CTY$^+$ and OWCTY) which includes all of the bad vertices, then proceed to refine the set until a fixed point or empty set is reached. In this section, we examine how the approximation sets change at each external iteration.

The EL algorithm is based on Equations 7.5 and 7.6 when there is one fair set. The two equations continuously refine the set c. The procedure of the refinement is follows:

$$c_0(x) = true;$$
$$c_{i+1}(x) = c_i(x) \wedge \exists y(T(x,y) \wedge lfp(G^{c_i})(y)).$$

Since the digraphs are finite, the procedure will stop after a finite number of iterations. Let k be that number, then either $c_k(x) = \emptyset$ or $c_{k+1}(x) = c_k(x)$. Figure 7.7 shows the number of vertices in the set c at each external iteration for EL. All six sub-figures are obtained from experiments performed on digraphs with fixed order $n = 2^{12}$. (a), (b) and (c) have the same density $d = 1.2$, but different sizes of fair sets ($|\mathcal{F}_a| = n \cdot 1\%, |\mathcal{F}_b| = n \cdot 10\%, |\mathcal{F}_c| = n \cdot 90\%$ respectively). (c), (d), (e) and (f) have the same size of fair sets $|\mathcal{F}| = n \cdot 90\%$, but different densities ($d_c = 1.2, d_d = 1.6, d_e = 2.0, d_f = 2.4$ respectively). The lines with "+" display the number of vertices in the set c at each external iteration with the iteration index shown on the x-axis. As shown in Figure 7.5, some experiments have fewer external iterations than others even when performed on digraphs with same order, density and size of fair sets, which means that the number of vertices in c is averaged from fewer experiments at higher index of external iterations. The lines with "o" denote how many experiments that the value in the line with "+" are averaged from. For example, 99 of the 100 experiments performed on digraphs with $n = 2^{12}, d = 1.2$ and $|\mathcal{F}| = n \cdot 10\%$ have fewer than nine external iterations. However, there is one experiment with 13 external iterations. So the number of vertices in c starting from the 9th external iteration is averaged from only one experiment in (b) of Figure 7.7, which is less reliable than the value averaged from more experiments.

It is observed from Figure 7.7 that the dotted line in (a) drops sharply from the 4th to the 5th iteration. Actually the number of vertices at the 5th iteration in (a) is 0. Since 0 cannot be drawn in the figure, we change it to 1. The value is 0 because there is one experiment left at the 5th iteration and the experiment happens not to have a bad cycle. In other words, the final set c is empty for that particular experiment. This is the only experiment in this chapter that does not have bad cycles. The reason that bad cycles exist in almost all the digraphs can be explained in the light of Theorems 3.8, 3.9 and 3.10. Theorems 3.8 and 3.9 state that in all likelihood there is one giant component $LARGE$ with size $\Theta^2 n$ in digraphs, where Θ is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. Theorem 3.10 states that almost certainly all the cycles in the graph lie in the giant strong component. Thus, bad cycles are very likely to exist in a digraph with a fair set $\mathcal{F}$ if $G \cap \mathcal{F} \neq \emptyset$. Let $k = |\mathcal{F}|$ and $m = \Theta^2 n$. The probability of a digraph without bad cycles is about $\binom{k}{n-m}/\binom{k}{n}$. Since n is very large, we cannot evaluate the formula directly. The following manipulation enables us to get the results:

$$\frac{\binom{k}{n-m}}{\binom{k}{n}} = \frac{(n-m)!k!(n-k)!}{k!(n-m-k)!n!} = \frac{\Pi_{i=0}^{m-1}(n-k-i)}{\Pi_{i=0}^{m-1}(n-i)} = \Pi_{i=0}^{m-1}\frac{n-k-i}{n-i}. \tag{7.7}$$

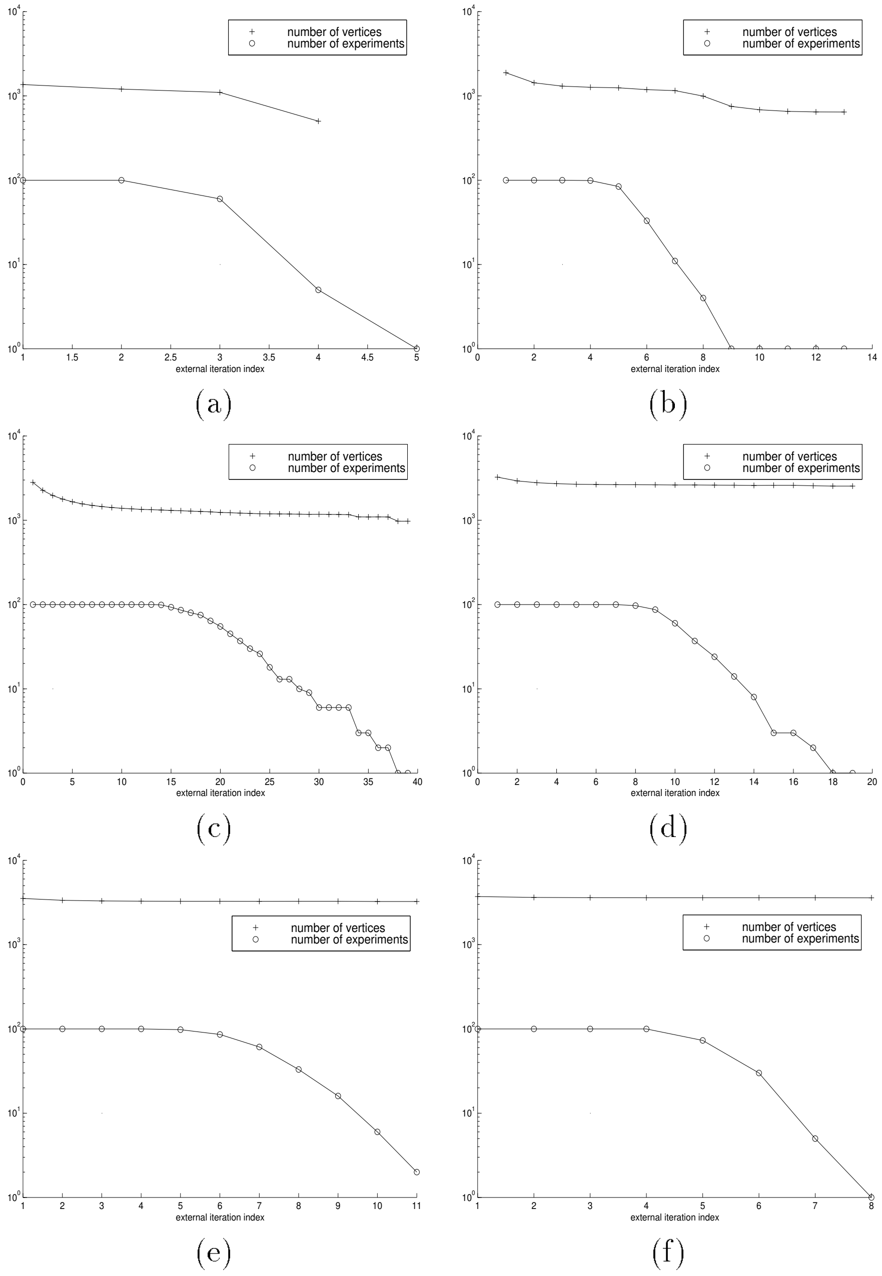
The probability for a digraph with $n = 2^{12}, d = 1.2$ and $|\mathcal{F}| = n \cdot 1\%$ not to have any bad cycles is 1.51%. And for digraphs with $n = 2^{12}, d = 1.2$ and $|\mathcal{F}| \geq n \cdot 10\%$, the probability of not having bad cycles drops below 10^{-7}. Therefore, bad cycles exist in almost all the experiments in this chapter.

It is also observed that the final sizes of c in (a), (b) and (c) are almost the same with values between 1000 and 1500, provided that there exist bad cycles. According to Equations 7.5 and 7.6, the final set c is actually NC_μ, which is the set of vertices on bad cycles or those that can reach bad cycles. Theorem 3.10 states that almost certainly all of the cycles in a digraph will lie in the giant strong component. Therefore, NC_μ is the set consisting of the giant component and the vertices that can reach the component. Based on Theorem 3.8, NC_μ is the set $LARGEIN$ that has size around Θn. For digraphs with $n = 2^{12}$ and $d = 1.2$, $|NC_\mu| = 1274$, which agrees with the experiments. Note that the size of NC_μ depends on the digraph order and density, but not on the size of the fair set. The experiments in (d), (e) and (f) are performed on digraphs with the same digraph order but different densities. Since the size of NC_μ is Θn, the final size of c in (d), (e) and (f) should be 2629, 3272 and 3600 respectively. Figure 7.7 shows that the experimental data agree with the calculated data pretty well.

Next we check the size of approximation sets for different CTY algorithms. In the light of Figure 7.3 and 7.4, the approximation set for CTY, CTY$^+$ and OWCTY is b. In the algorithms, the vertices without successors and/or predecessors in b are deleted at step 3 until b reaches a fixed point or becomes empty.

Figure 7.8 shows the number of vertices in the set b each time after the first line at step 3 of Figure 7.3 is executed. Among the six sub-figures, (a) and (b) show the result from CTY, (c) and (d) are from CTY$^+$, (e) and (f) are from OWCTY. All of the sub-figures are obtained from experiments performed on digraphs with fixed order $n = 2^{12}$ and density $d = 1.2$. The size of fair sets for (a), (c) and (e) is $n \cdot 10\%$, while for (b), (d) and (f) is $n \cdot 90\%$. As for the lines in Figure 7.7, the lines with "+" at each sub-figure in Figure 7.8 display the number of vertices in $|b|$ at the external iteration with the index shown on the x-axis, and the lines with "+" denote the number of experiments that the value in the upper line is averaged from.

It is observed from Figure 7.8 that the sizes of b are almost the same for CTY and CTY$^+$, and that the size of the final set b is about 400. The final set of b in CTY, if not empty, consists of two kinds of vertices: the vertices that are on bad cycles, and the vertices that are reachable from and can reach the bad cycles. Since the improvement in CTY$^+$ only tries to make b converge faster, the final set b for CTY$^+$ is the same as its counterpart in CTY. Theorem 3.10 states that almost certainly all of the bad cycles lie in the giant strong component. Since the giant strong component consists of all cycles and all the vertices in this component are reachable from and can reach each other, the final set of b is the giant strong component if bad cycles exist. The giant strong component is the set $LARGE$ in Theorem 3.9 with size $\Theta^2 n$, where Θ is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. When $n = 2^{12}$ and $d = 1.2$, $\Theta^2 n = 396$. From (a), (b), (c) and (d), we can see 396 is consistent with the data obtained from the experiments. (e) and (f) show that the size of the final set b in OWCTY is above 1000, which is much larger that those in CTY and CTY$^+$. This is because the final set in OWCTY consists of the vertices that are on bad cycles or can reach the bad cycles. Thus, it is very likely that the final b is the set $LARGEOUT$ in Theorem 3.8 with size around $\Theta n = 1273$.

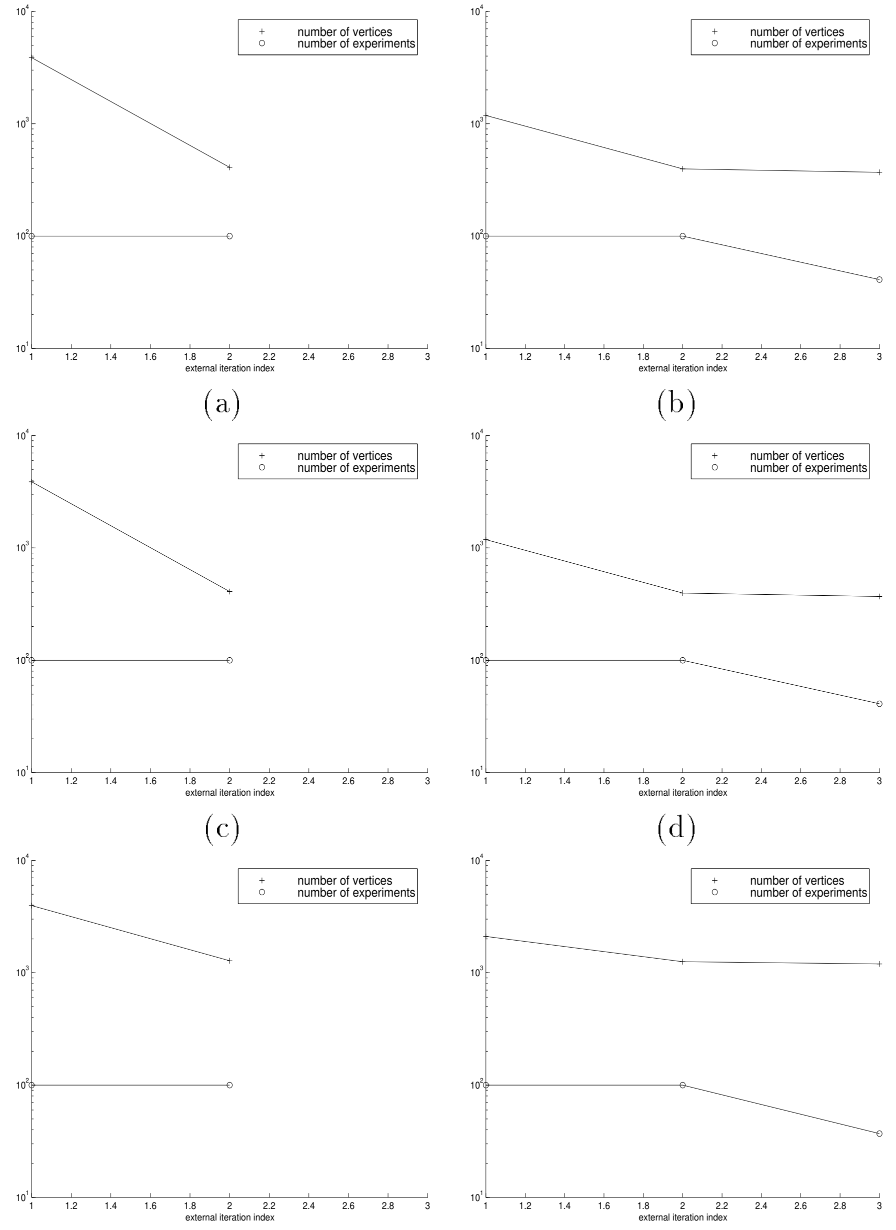


(a) (b)

(c) (d)

(e) (f)

Figure 7.7 The number of vertices in c for EL on digraphs with $|V| = 2^{12}$. (a)$d = 1.2, |\mathcal{F}| = n \cdot 1\%$; (b)$d = 1.2, |\mathcal{F}| = n \cdot 10\%$; (c)$d = 1.2, |\mathcal{F}| = n \cdot 90\%$; (d)$d = 1.6, |\mathcal{F}| = n \cdot 90\%$; (e)$d = 2.0, |\mathcal{F}| = n \cdot 90\%$; (f)$d = 2.4, |\mathcal{F}| = n \cdot 90\%$.



(a)

(b)

(c)

(d)

(e)

(f)

Figure 7.8 The number of vertices in b for CTY, CTY$^+$ and OWCTY on digraphs with $|V| = 2^{12}$ and $d = 1.2$. (a)CTY, $|\mathcal{F}| = n \cdot 10\%$; (b)CTY, $|\mathcal{F}| = n \cdot 90\%$; (c)CTY$^+$, $|\mathcal{F}| = n \cdot 10\%$; (d)CTY$^+$, $|\mathcal{F}| = n \cdot 90\%$; (e)OWCTY, $|\mathcal{F}| = n \cdot 10\%$; (f)OWCTY, $|\mathcal{F}| = n \cdot 90\%$.

7.7 Number of Image/Preimage Computations

The most computationally expensive operation of the EL, CTY, CTY$^+$ and OWCTY algorithms is the image/preimage computation. The cost of image/preimage computation depends on the density and order of the underlying digraphs. Since we perform the four algorithms on the same digraph at each comparison, the cost of a single image/preimage computation is similar. Thus, the number of image/preimage computations is a fair parameter on which to compare these algorithms. Because of the symmetry of random graphs, we will not distinguish images and preimages in the rest of this chapter.

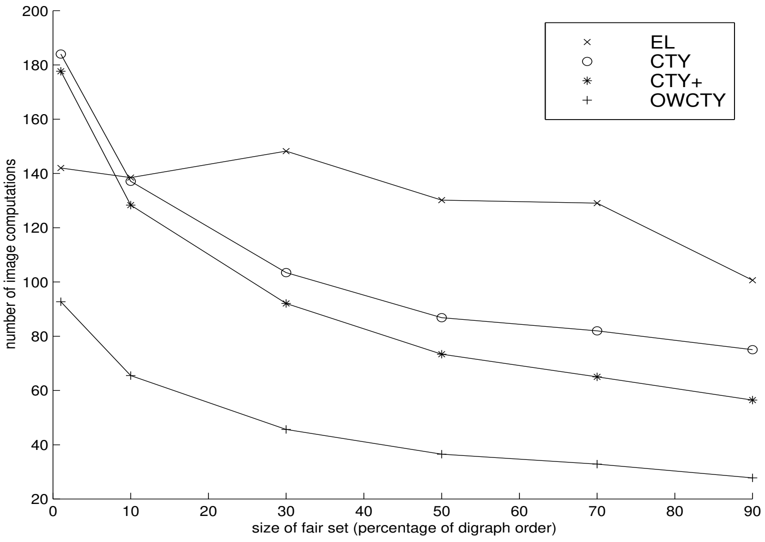


Figure 7.9 Number of image computations for EL, CTY, CTY$^+$ and OWCTY(I).

Figure 7.9 shows the number of image computations. Each datum is averaged from 100 experiments performed on digraphs with fixed order ($n = 2^{12}$) and density ($d = 1.2$). The size of a fair set is $n \cdot 1\%$, $n \cdot 10\%$, $n \cdot 30\%$, $n \cdot 50\%$, $n \cdot 70\%$ or $n \cdot 90\%$, which is displayed on the x-axis. The y-axis shows the number of image computations. The line with "x" displays the results for EL, the line with "o" is for CTY, the line with "*" is for CTY$^+$ and the line with "+" is for OWCTY, respectively.

As shown in Figure 7.1, image computations are needed in the least fixed point computation and the greatest fixed point computation in EL. Suppose there are k external iterations, and at the ith external iteration, x_i image computations are needed

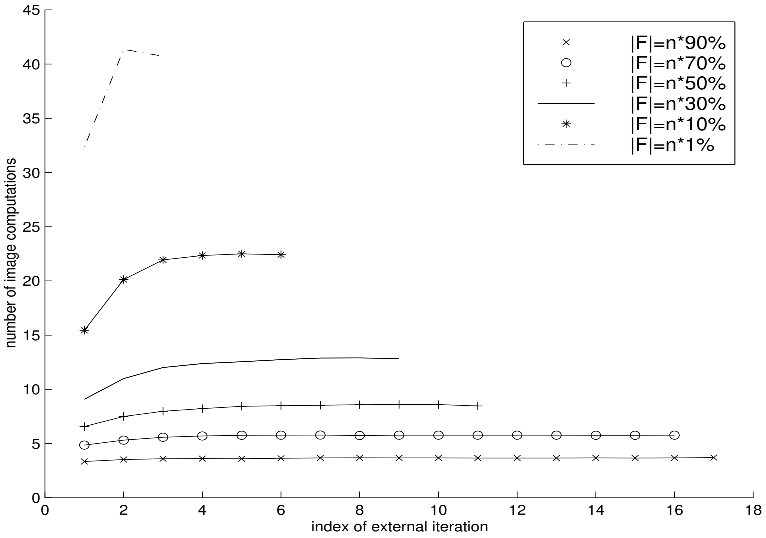


Figure 7.10 Number of image computations for EL at each external iteration.

to evaluate the least fixed point, then the total number of image computations is

$$\Sigma_{x=1}^{k}(x_i + 1). \tag{7.8}$$

We showed in Section 7.5 that the number of external iterations increases as the fair set becomes larger. In other words, there are more summation terms in Equation 7.8 when the fair set gets larger. Now we need to check the value of x_i. As shown in Figure 7.7, a smaller fair set results in a little smaller set c at the beginning of each external iteration. Therefore, in all likelihood, the initial set $c \wedge \mathcal{F}$ for the reachability computation is smaller when the fair set is smaller. So the number of iterations in the reachability computation, *i.e.*, the number of image computations for the least fixed point computation, is larger when the fair-set is smaller.

Figure 7.10 summarizes the number of image computations for EL in a more detailed fashion. The line with "x" shows the number of image computations when $|\mathcal{F}| = n \cdot 90\%$. At the first external iteration, there are about 3 image computations. This number is averaged from 100 experiments performed on digraphs with the same order, density and size of fair sets. As shown in (d) of Figure 7.5, some experiments have fewer external iterations than others, which means that the number of image computations is averaged from fewer experiments at the higher indices of external iterations. For reliability, we discarded the data averaged from less than 20 experiments. Similarly, the plain line and the dotted line are for experiments performed on digraphs with $|\mathcal{F}| = n \cdot 1\%$ and $|\mathcal{F}| = n \cdot 30\%$, the lines with "*", "+" and "o" are for $|\mathcal{F}| = n \cdot 10\%$, $|\mathcal{F}| = n \cdot 50\%$ and $|\mathcal{F}| = n \cdot 70\%$ respectively. It can be observed from Figure 7.10 that although more image computations in the least fixed point computation are needed at each external iteration, there are fewer external iterations when $|\mathcal{F}|$ is smaller. Since these two factors affect each other, the number of image computations does not change much with the size of fair sets.

Returning to Figure 7.9, the lines for CTY, CTY$^+$ and OWCTY show that the number of image computations becomes smaller as the size of the fair sets gets larger. Let us analyze the data from CTY first. There are two steps in Figure 7.3 that require image computations: the reachability computation in step 2 and the internal iterations in step 3. A breakdown of the image computations in CTY is shown in Table 7.5. The sizes of the fair sets are listed in the first column. The next two columns f^* and $^*f$ show the number of image computations used in forward and backward reachability computation respectively. The figures in the column $^+b + b^+$ are the number of image computations in step 3 of Figure 7.3. It can be observed from the algorithm that the times to compute ^+b are always the same as those of b^+; therefore half the figures in this column represent ^+b or b^+. Finally, the last column shows the total number of image computations in CTY, which is the sum of the numbers in previous columns on the same rows.

$\|\mathcal{F}\|$	f^*	$^*f$	$^+b + b^+$	total
$n \cdot 90\%$	13.93	13.70	47.42	75.05
$n \cdot 70\%$	16.83	16.37	48.80	82.00
$n \cdot 50\%$	19.30	19.50	48.04	86.84
$n \cdot 30\%$	27.26	26.59	49.60	103.45
$n \cdot 10\%$	46.70	46.01	44.42	137.13
$n \cdot 1\%$	76.11	76.35	31.56	184.02

Table 7.5 Breakdown on the number of image computations for CTY(I).

Some observations can be obtained from Table 7.5. First, the number of image computations in the column f^* is almost the same as that in column $^*f$. This is not surprising because of the symmetry in random digraphs. In other words, given an initial set, the number of image computations used to reach the fixed point by forward reachability is expected to be same as that by backward reachability in a digraph drawn from $D_{n,d/n}$. Second, the number of image computations increases in the reachability computations as the fair set gets smaller. The experiments that we did show that more iterations are needed to reach the fixed point when the initial set gets smaller in reachability computations. As shown in step 2 of Figure 7.3, the initial set is obtained by $f \cap b$. The initial set is smaller when $\mathcal{F}$ is smaller. Since the number of iterations equals the number of image computations in the reachability computation, more image computations in step 2 are needed when the fair set gets smaller. Third, the number of image computations in ${}^{+}b + b^{+}$ is not affected by the size of fair sets except in the case where $\mathcal{F}$ is very small($n \cdot 1\%$). Because the number increases in the two columns(f^{*}, ${}^{*}f$) and remains almost the same in column (${}^{+}b$+b^{+}), the number of overall image computations increases as the fair set becomes smaller.

Table 7.6 and 7.7 show the breakdown of image computations for CTY^{+} and OWCTY respectively. For the same reason as for CTY, more image computations overall are needed when the size of a fair set gets smaller.

$\|\mathcal{F}\|$	g^{*}	${}^{*}g$	${}^{+}b + b^{+}$	total
$n \cdot 90\%$	4.53	4.53	47.42	56.48
$n \cdot 70\%$	8.07	8.17	48.80	65.04
$n \cdot 50\%$	12.66	12.67	48.04	73.37
$n \cdot 30\%$	21.43	21.06	49.60	92.09
$n \cdot 10\%$	42.09	41.82	44.42	128.33
$n \cdot 1\%$	72.78	73.33	31.56	177.67

Table 7.6 Breakdown on the number of image computations for CTY^{+}(I).

Now let's compare the number of image computations between CTY and CTY^{+}. It is observed from Figure 7.9 that CTY needs more image computations than CTY^{+}. The difference between the two algorithms comes from the method in reachability computation. While CTY^{+} restricts the reachability computation within the set b, CTY does not. Therefore, more iterations are needed by the reachability computation in CTY, which results in more image computation than in CTY^{+}.

It is observed from Figure 7.9 that OWCTY performs better than CTY^{+} with fewer image computations. First, we compare the columns g^{*}, ${}^{*}g$ in Table 7.6 and the

$\|\mathcal{F}\|$	$^*h$	^+b	total
$n \cdot 90\%$	5.01	22.77	27.78
$n \cdot 70\%$	8.72	24.13	32.85
$n \cdot 50\%$	13.75	22.77	36.52
$n \cdot 30\%$	21.98	23.65	45.63
$n \cdot 10\%$	43.93	21.57	65.50
$n \cdot 1\%$	76.70	16.01	92.71

Table 7.7 Breakdown on the number of image computations for OWCTY(I).

column $^*h$ in Table 7.7. The numbers in column $^*h$ in Table 7.7 are almost the same as those in the corresponding column $^*g$ of Table 7.6, which means that the number of image computations saved in OWCTY on reachablity computation is almost the number shown in column g^* in Table 7.6. Next, we compare column $^+b + b^+$ in Table 7.6 to column ^+b in Table 7.7. It is observed that the numbers in the column ^+b are about half of that in column $^+b + b^+$. In other words, the number of image computations used to compute ^+b is almost the same for CTY$^+$ and OWCTY, in which there is no computation of b^+ in OWCTY. Therefore, not needing to compute g^* and b^+ makes OWCTY less computationally expensive. The reason lies in different results from the two algorithms. The meaning of the final set returned by CTY$^+$, if not empty, is the set of vertices that are reachable from and can reach the bad cycles. The set returned by OWCTY, if not empty, consists of those vertices that can reach the bad cycles. Therefore, the set returned by OWCTY is a super-set of the set returned by CTY$^+$.

Figure 7.11 shows the number of image computations with respect to the densities. Data are averaged from 100 experiments performed on digraphs with $n = 2^{12}$, $|\mathcal{F}| = n \cdot 90\%$. The x-axis shows the density. The y-axis shows the number of iterations. The four lines are for EL, CTY, CTY$^+$ and OWCTY respectively. It can be observed that density plays a very important role in all four algorithms. As the densities increase from 1.2 to 2.4, the number of image computation decreases sharply.

First we explain the phenomenon for EL. As shown in Equation 7.8, the number of image computations is $\Sigma_{x=1}^{k}(x_i + 1)$, where k is the number of external iterations and x_i is the number of image computations at the ith external iteration. As explained in the previous subsection, k becomes smaller as the density increases. The evaluation

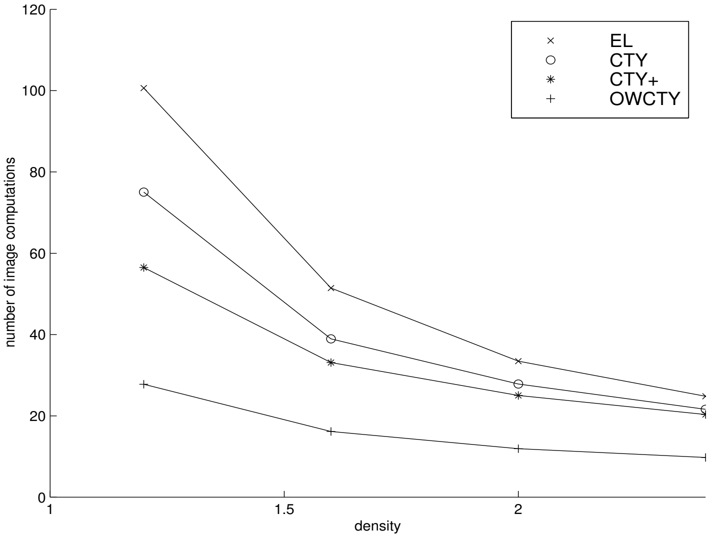


Figure 7.11 Number of image computations for EL, CTY, CTY$^+$ and OWCTY(II).

for the least fixed point in Equation 7.6 returns the set of vertices that can reach a certain set. In Chapter 5, we showed that the number of iterations required to compute the set of vertices reachable from a certain set decreases as the density becomes greater. Since digraphs drawn from $D_{n,d/n}$ are symmetrical, the number of iterations required to compute the set of vertices that can reach a certain set decreases as the density becomes greater. The only problem left is that the sizes of the initial sets in the inner loop are not the same for different densities. From Equation 7.6, the initial set for reachability computation is $c \wedge \mathcal{F}$. Since $|\mathcal{F}|$ is fixed regardless of density, we need to consider only $|c|$. Figure 7.5 shows the initial size of c for each reachability computation. It is observed that the greater the density, the bigger the initial set c. We know that a bigger initial set involves fewer iterations in reachability computation in the same random digraphs. Now that we have greater density and a bigger initial set, fewer iterations are needed in each least fixed point computation. That is, x_i gets smaller as density increases. Combined with the previous conclusion about k, the number of preimage computations drops sharply when the density increases in EL.

Next we explain the phenomenon for CTY. In Figure 7.3, steps 2 and 3 require image computations. The initial set for the reachability computation at step 2 is $f \wedge b$, where f denotes the set $\mathcal{F}$ and b is either *true* or the value returned from step 3. When step 2 is executed for the first time, the set denoted by $f \wedge b$ is the same, provided that $\mathcal{F}$ is fixed. When step 2 is executed beyond the first time, b is returned from step 3; the experimental data show that the size of b returned from step 3 is larger when the digraph density increases. Therefore, the reachability computation at step 2 has a larger initial set after step 2 is executed for the first time. It is clear that given fixed digraph order and density, larger initial sets involve fewer iterations in reachability computation. Also, given fixed digraph order and initial set, fewer iterations are needed in denser digraphs for computing reachable vertices. Therefore, fewer image computations are needed when the digraph density increases. The function of step 3 in Figure 7.3 is to remove those vertices without predecessors or successors in the set denoted by b. Although fewer vertices are removed at each iteration within step 3 when the digraph density is larger, the final size of b returned by step 3 is also larger. The overall effect is that fewer iterations within step 3 are needed when digraph density is greater, which results in fewer image computations at step 3. Thus, by using the CTY algorithm, fewer image computations are required when the digraph density increases.

density	$^*f$	f^*	$^+b + b^+$	total
1.2	13.93	13.70	47.42	75.05
1.6	7.70	7.75	23.48	38.93
2.0	5.99	5.83	16.02	27.84
2.4	5.15	5.06	11.40	21.61

Table 7.8 Breakdown on the number of image computations for CTY(II).

density	$^*f$	f^*	$^+b + b^+$	total
1.2	4.53	4.53	47.42	56.48
1.6	4.79	4.85	23.48	33.12
2.0	4.52	4.47	16.02	25.01
2.4	4.51	4.44	11.40	20.35

Table 7.9 Breakdown on the number of image computations for CTY$^+$(II).

For the same reasons as those in CTY, we can explain the effect of density on the number of image computations shown in Table 7.9 and 7.10 by using CTY$^+$ and OWCTY.

density	$^*f$	$^+b + b^+$	total
1.2	5.01	22.77	27.78
1.6	5.07	11.08	16.15
2.0	4.53	7.41	11.94
2.4	4.45	5.31	9.76

Table 7.10 Breakdown on the number of image computations for OWCTY(II).

Finally we tested the running time for each algorithm. The results are very similar to the number of image computations analyzed above. For example, Figure 7.12 shows the running time of experiments performed on digraphs with different fair sets. It is observed that the figure is like Figure 7.9. Since the running time does not provide as much insight on the algorithms as image computations, we do not need to write more here.

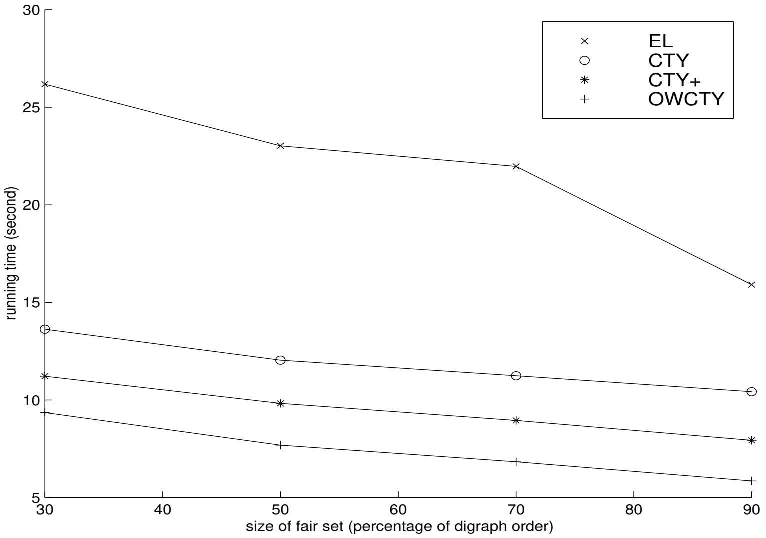


Figure 7.12 Running time for EL, CTY, CTY$^+$ and OWCTY.

Chapter 8

CONCLUSION

Model checking is currently the most practical approach in formal verification. However, the understanding of how algorithmic performance depends on the various attributes of BDDs is sorely lacking, which makes the performance optimization of verification algorithms difficult. This dissertation has provided a way to address this issue.

8.1 Contributions

The size of BDDs is a key issue in understanding the time and space requirements of model checking algorithms. Most existing research [20, 15, 31] in this area uses BDDs to represent uniformly distributed Boolean formulae, then tests the size of such BDDs. Unfortunately, this approach fails to provide enough information about the size of the BDDs. First, the approach cannot tell us in which cases a Boolean formula has the worst BDD representation. Second, given a Boolean formula, we cannot predict the expected size of the BDD that represents this formula. Third, the performance of BDDs in symbolic algorithms is still unknown.

This dissertation has provided a more practical approach to study the performance of BDDs. Instead of using BDDs to represent uniformly distributed Boolean formulae, we use BDDs to represent sets of states that are uniformly chosen from state space. Then we study the size of BDDs against the number of states being represented. Based on experiments, we provided the following empirical formula for calculating the expected size of BDDs:

$$y = 2^{x+1}/x \cdot (1 - \alpha * (\beta - 0.5)^2), \tag{8.1}$$

where y denotes the size of BDDs, x is the number of Boolean variables that are used to represent state space, α is a constant and β is the fraction of states in the state space being represented. Our experiments showed that the empirical formula offered good predictions, provided that the percentage of states being represented is not very small($< 10\%$) or very large($> 90\%$).

We also studied the performance of BDDs in symbolic algorithms on our random digraph model. Since model checking can be considered as the exhaustive search, we concentrated on symbolic reachability algorithms. We tested the size of BDDs and the number of vertices reached at each iteration during the computation. Our research confirmed Karp's [18] theories that either very few vertices or very many vertices are reachable from an initial vertex in random digraphs drawn from $D_{n,d/n}$. The probability for a vertex to reach very many vertices in a digraph with density d is Θ, which is the unique root in [0,1] of the equation $1 - x - e^{-dx} = 0$. The probability to reach very few vertices is $1 - \Theta$. Based on experiments, we were able to offer the following empirical formula to calculate the expected number of iterations needed to reach the fixed point from an initial vertex in a random digraph, provided that the initial vertex can reach very many vertices:

$$iter = \log_{0.405d+0.683} \Theta n,$$

where $iter$ denotes the number of iterations, d is the density and n is the order.

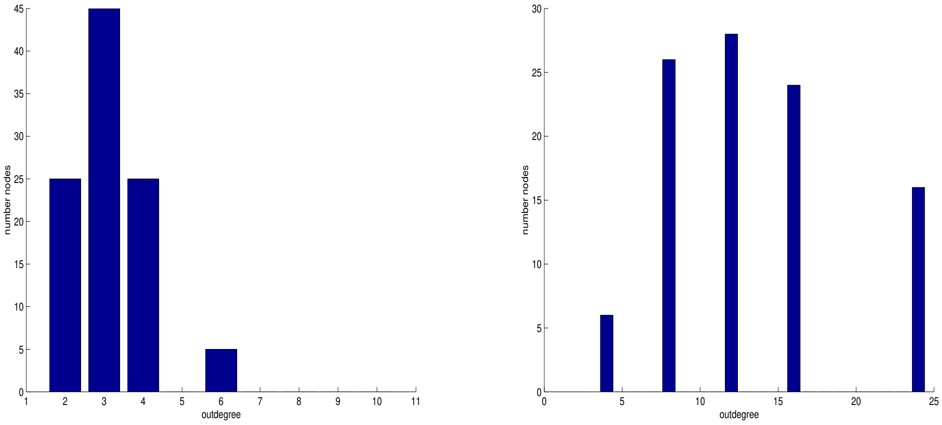
Two kinds of important properties in model checking are safety properties and liveness properties. We designed experiments to evaluate the performance of current symbolic algorithms being used to check these two types of properties. For checking safety properties, we compared different reachability algorithms by using forward search, backward search, and combined forward and backward search. Our research indicates that the combined approach works best in checking safety properties in the sense of memory requirements. Furthermore, the sizes of initial set and the bad set determine which one is better, forward search or backward search. Our research indicates that we should start from the initial set and use forward search if it is smaller than the bad set, otherwise we should start from the bad set and use backward search.

The current symbolic algorithms being used to check liveness properties are the Emerson-Lei(EL) and Catch-Them-Young(CTY) algorithms. We discussed the two algorithms in Chapter 7. Based on a better understanding of BDDs, we were able to design a new heuristic algorithm to check liveness properties. The new algorithm performs better than the two current algorithms.

8.2 Future Work

The advantage of BDDs over other representation of Boolean formulae comes from the fact that BDDs capture the regularity hidden in the states or transition relations being represented. In this dissertation, we used BDDs to represent sets of states randomly chosen from a state space, then proceeded to study the size of BDDs. This approach made our study of BDDs very general. However, we are not sure how well our results can be applied to the real designs in different fields.

We studied some designs issued with VIS. In the experiments, we chose 100 states from the state space, then checked the outdegree of each state. The (a) in Figure 8.1 is obtained from a benchmark named Eisenberg. We can see that the outdegrees ranges from 2 to 6, and the mean outdegree is 3. The (b) is obtained from a benchmark named Arbiter. The outdegrees of the 100 states that we chose from Arbiter ranges from 4 to 24, with the mean outdegree to be 12. The mean outdegrees can be considered as the densities in our experiments. We believe that designs in different fields have different degrees of regularity that can be captured by BDDs, and that the randomly chosen digraphs provide less regularity than most of the real designs.



(a)

(b)

Figure 8.1 Size of BDDs that represent sets of vertices.

We used random graphs drawn from $D_{n,d/n}$ to simulate transition relations. This approach enabled us to avoid the details of specification languages as well as to provide enough experiments. We chose the digraph orders between 2^{10} and 2^{20} because such digraphs allowed experiments to be finished in a reasonable time and were large enough to provide meaningful results. However, the digraphs are still very small compared to real designs. For example, the largest benchmark distributed with VIS has 90 state variables, which is equivalent to a digraph with 2^{45} vertices. Given more time, we would like to do experiments on larger digraphs.

In chapter 7, we designed a new algorithm OWCTY to check liveness properties. Based on the experiments performed on random digraphs, it outperforms the current EL and CTY algorithms. We hope to implement it in some verification tools to compare its efficiency against other algorithms on real designs.

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Trans. on Computers*, C-2(6):509–516, June 1978.

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

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

[4] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 13(4), April 1994.

[5] E. M. Clark and E. A. Emerson. Synthesis of synchronization skeletons for branching time temporal logic. *Lecture Notes in Computer Science*, 131, 1981.

[6] E. Clarke and J. Wing. Formal methods: state of the art and future directions. Technical report, CMU Computer Science, 1996.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, January 1986.

[8] E. M. Clarke, O. Grumberg, and D. E. Long. *Model checking.* M.I.T. Press, November 1999.

[9] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. *Lecture Notes in Computer Science*, 407, 1989.

[10] C. Courcoubetis, M. Y. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. *Formal Methods in System Design*, 1:275–288, 1992.

[11] E. A. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, 1991.

[12] E. A. Emerson and C. L. Lei. Efficient model checking in fragments of the propositional model mu-calculus. *Proceedings of LICS 1986*, pages 267–278, 1986.

[13] A. Gupta. Formal hardware verification methods: a survey. *Formal Methods in System Design*, 1:151–238, 1992.

[14] R. H. Hardin, R. P. Kurshan, S. K. Shukla, and M. Y. Vardi. A new heuristic for bad cycle detection using BDDs. In *Proc. Conf. on Computer-Aided verification (CAV'97)*, pages 268–278. Springer-Verlag. LNCS 1254, 1997.

[15] M. A. Heap and M. R. Mercer. Least upper bounds on OBDD sizes. *IEEE Transaction on Computers.*

[16] R. Hojati, H. J. Touati, R. K. Brayton, and R. P. Kurshan. Efficient ω-regular language containment. *Proceedings of CAV 93*, LNCS 663:396–409, 1993.

[17] G. E. Hughes and M. J. Creswell. *Introduction to Modal Logic.* Methuen. London., 1977.

[18] R. M. Karp. The transitive closure of a random digraph. *Random Structures and Algorithms*, 1(1), 1990.

[19] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes.* Princeton University Press, 1994.

[20] H. T. Liaw and C. S. Lin. On the OBDD-representation of general Boolean functions. *IEEE transaction on Computers*, 41(6):661–664, 1992.

[21] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proc. 12th ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.

[22] K. L. McMillan. *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[23] E. Palmer. *Graphical Evolution.* Willy Interscience, 1985.

[24] R.E.Bryant. Symbolic Boolean manipulation with ordered binary-decision digrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[25] S. Seshadri and J. F. Naughton. On the expected size of recursive datalog queries. *ACM*, 1991.

[26] F. Somenzi. CUDD: CU decision diagram package release 2.1.2. Dept. of Electrical and Computer Engineering, Univ. of Colorado at Boulder, April 1997.

[27] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309, 1955.

[28] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for omega-automata using BDD's. *Proceedings of The 1991 International Workshop on Formal Methods in VLSI Design*, January 1991.

[29] H. J. Touati, R. K. Brayton, and R. P. Kurshan. Testing language containment for ω-automata using BDD's. *Information and Computation*, 118(1):101–109, April 1995.

[30] M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *Proc. 1st Ann. Symp. on Logic in Comput. Sci.* IEEE Comp. Soc. Press, June 1986.

[31] I. Wegener. The size of reduces OBDDs and optimal read-once branching programs for almost all Boolean functions. *IEEE Transaction on Computers*, 43(11):1262–1269, 1994.