Verification and Learning theory

Guy Offer[1]

March 21, 2001

[1]This work was partially supported by GIF, the German Israeli research fund, under contract I 0403-001.06/95 .

This work was carried out under the supervision of
Prof. Eli Shamir.

Acknowledgements

I could not find words strong enough in order to thank prof. Eli Shamir for his support and guidance along this thesis. For giving me the background of computational learning theory and research in general, and for devoting so much of his time and energy for this work.

I would also like to thank Dr. Orna Kupferman for going over the texts and providing me with help and guidance regarding the verification aspects of this work.

Last, but not least, I would like to thank my parents Yehudith and Michael Offer, my brothers Paz and Chen and my grandmas Ilse and Paula for their support.

Contents

1 Introduction **5**

2 Related work **5**

2.1 A Sublinear bipartiteness tester - work by Oded Goldreich and Dana Ron 6
2.1.1 The graph G representation and definition of distances 6
2.1.2 The Algorithm 6
2.1.3 The Markov Chain $M_{l_1}^{l_2}(H)$ 8
2.1.4 The proof of the algorithm 8
2.1.5 Conclusions 10
2.1.6 Remark 10
2.2 Learning to Reason - work by Roni Khardon and Dan Roth 10
2.2.1 Introduction 10
2.2.2 Definitions 10
2.2.3 The relation between L2R and L2C 11
2.2.4 Combining learning and reasoning 12
2.2.5 Learning to Reason via Model Based Reasoning 12
2.2.6 **Learning to Reason $CNF \cap DNF$ without reasoning** 12
2.2.7 Learning to Reason DNF without learning to classify 13
2.2.8 Conclusions 14
2.3 Automatic Abstraction Techniques for Propositional μ-calculus Model Checking . . . 14
2.3.1 Introduction 14
2.3.2 Basic definitions and verification of μ-calculus formulas 15
2.3.3 Conservative Abstraction 16
2.3.4 An Automatic Abstraction and Refinement Algorithm 17
2.3.5 Conclusions 19
2.4 Related work - and the connection to this work 19

3 Using Clustering to Order OBDD Variables. **21**

3.1 Ordered Binary Decision Diagrams, OBDDs 21
3.1.1 Introduction to OBDDs 21
3.1.2 The representation 21
3.1.3 OBDD - an illustrated example. 22
3.1.4 Advantages of OBDD representation 24
3.1.5 Logic operators on OBDDs - APPLY and RESTRICT 25
3.1.6 OBDD and verification 26
3.1.7 OBDD size and the variables ordering problem 26
3.1.8 Parity OBDDs 29
3.2 The Clustering of Variables 29
3.2.1 Introduction and intuition 29
3.2.2 Clustering 31
3.2.3 Pairwise Clustering 31
3.2.4 A Randomized Algorithm for Pairwise Clustering 32
3.3 Finding good order for the variables of OBDDs by clustering 33
3.3.1 Introduction 33
3.3.2 Examples of functions for which the OBDD size can be reduced by change of variables clustering 35

3.4 The Algorithm . . . 39
3.4.1 The setting . . . 39
3.4.2 Description of the algorithm . . . 39
3.4.3 Improving existing variable order using pairwise-sensitivity . . . 39
3.4.4 Working with multiple valued OBDDs . . . 40
3.4.5 Using the correlations between f variables and f value . . . 40
3.5 The 3 variables case . . . 40
3.5.1 3 variables correlation . . . 40
3.5.2 Example - 3 variables correlation . . . 41
3.5.3 Correlations formulation for the 3 variables case . . . 43
3.5.4 Using the clustering algorithm for hyper-graphs . . . 44
3.5.5 The algorithm for the 3 variables case . . . 45

4 Using PAC learning for verification of LTL properties. 46
4.1 Introduction . . . 46
4.1.1 Formal Checking . . . 46
4.1.2 Temporal logics and formal checking . . . 46
4.1.3 LTL model checking . . . 50
4.2 Introduction, the learning theory . . . 51
4.2.1 The Probably Approximately Correct (PAC) model . . . 51
4.2.2 Learning and definition of learning; efficient learning . . . 51
4.3 Algorithm for checking LTL properties by sampling . . . 51
4.3.1 How to check whether a sampled computation satisfies the LTL property ψ . 53
4.4 Using Bayesian inference to estimate sampling size . . . 55
4.4.1 Bayesian inference . . . 55
4.4.2 Description of the verification scheme . . . 56
4.4.3 The verification model . . . 56
4.5 Computing the probability function. . . . 56

1 Introduction

In this work I have tried to combine two emerging fields of research in computer science: verification and computational learning.

The combination of these two fields might seems awkward in some way, since the formal checking algorithms provide their users with one hundred percents of assurance that the properties that they verified holds in the system verified while computational learning algorithms usually permits an error in their results. There are two main justifications for using computational learning methods for verification.

1. **States explosion problem** The formal checking algorithms require space that is exponential in the number of variables of the model (and linear in the states of the model). This requirement is in many cases too much for systems with more states. eliminates

2. **Efficiency** The computational learning methods don't require checking all the states. As was suggested in papers by Dana Ron and Oded Goldreich, in order to check whether a certain model or a graph satisfies a given property, we can check only fractions of the whole model/graph.

Two main methods to combine verification and learning are presented in this work. However, there is important difference in the type of combination done in each of them. While in the first combination (using clustering to build OBDDs) we don't apply any approximation on the verification results and use the learning only to achieve compact data structure for the verification process, in the second method (using sampling with LTL verification) we approximate the results of the verification and we permit the verification algorithm to make errors. This work deals mainly with the first method. As for the second method, presented in chapter 4, we should note here that the ideas there are preliminary and are not fully justified.

Research on the connection between property checking and learning theory has been done in various areas. In the following section I survey three distinct work that deal with different aspects of the connection between property checking and learning.

2 Related work

Studies that deal with property checking and learning can be categorized into some major categories:

The first category researches deals with the problem of probabilistic checking properties in graphs. In these kind of problems we would like to decide whether a graph has a certain property. I'll describe below a paper by Oded Goldreich and Dana Ron [13] where they check bipartiteness of bounded degree graphs. During the verification process we would like to verify whether certain property holds in a program. The formal checking algorithms verify this by checking whether the Kripke structure of the program fulfills that property. Since Kripke structure can be abstracted as a graph, we would actually like to know whether a certain graph property holds in a certain graph. Some problematic aspect of this method with verification is that it is not always trivial to formulate a CTL or LTL temporal logics properties using the simple graph properties.

Other category of researches deals with probabilistic reasoning. In these researches methods were suggested for reasoning using computational learning. I'll describe below paper by Roni Khardon and Dan Roth [17], that suggest a new framework for reasoning using learning. The field of reasoning is useful for the understanding of verification since also in the problem of verification we are given a model, which is the program and a query, which is the property, and we are interested

whether the query can be reasoned from the model or equivalently whether the property holds in the program.

2.1 A Sublinear bipartiteness tester - work by Oded Goldreich and Dana Ron

The paper by Oded Goldreich and Dana Ron [13] presents an algorithm to test bipartiteness of a bounded degree graph.

The bipartiteness tester searches for a counter example for the bipartiteness of the graph checked. It searches an odd-length cycle in the graph. Such a cycle is an evidence that the graph is not bipartite. If an odd-length cycle is found, then the tester answer that the graph is not bipartite, otherwise the tester answer that the graph is bipartite. It is obvious that if the graph is bipartite, no odd-length cycle exists and the algorithm always right. An analysis in the paper proves that if the graph G is ϵ-far from being bipartite, that means that there are more then ϵdN violating edges in any partitioning of the graph, the graph is rejected with probability of at least $\frac{2}{3}$.

The paper presents an analysis of the algorithm that shows that if the graph is accepted with probability of at least $\frac{1}{3}$ then G is ϵ-close to being bipartite. This means that if G is ϵ-far from being bipartite, then G is rejected with probability of at least $\frac{2}{3}$.

2.1.1 The graph G representation and definition of distances

The bipartiteness tester is designed for bounded degree graphs. The algorithm works on a graph that is represented by its incidence-lists. That is, an N-vertex graph of degree bound d is represented by a function from $\{1, 2, ..., N\} \times \{1, 2, ..., d\}$ to $\{0, 1, 2, ..., N\}$. This means that the tester may make queries of the form "who is the ith neighbor of v" (and the answer may be a vertex or 0 indicating that v has less than i neighbors). In this model, the distance between N-vertex graphs of degree bound d is defined as the fraction of vertex-pairs on which they disagree over the total of dN pairs in the domain of the function.

2.1.2 The Algorithm

The algorithm has an oracle access to G. It can perform walks on the graph starting from vertices of its choice. The algorithm performs random walks on G: At each step, if the degree of the current vertex is d', then the walk remains at v with probability $1 - \frac{d'}{2d} \geq \frac{1}{2}$, for each of its neighbors u the walk traverses to u with probability $\frac{1}{2d}$.

Theorem 1 *The algorithm Test-Bipartite constitutes a tester for bipartiteness with complexity $poly((logN)/\epsilon) \cdot \sqrt{N})$.*

- *If G is bipartite then the algorithm always accepts.*
- *If G is ϵ-far from being bipartite then the algorithm rejects with probability at least $\frac{2}{3}$. Furthermore, whenever the algorithm rejects a graph it outputs certificate to the non bipartiteness of the graph in form of an odd-length cycle of length $poly(\epsilon^{-1} logN)$.*

The algorithm:

1. Repeat $T = \Theta(\frac{1}{\epsilon})$ times:
   (a) Uniformly select s in V.

(b) If odd-cycle(s) return found then reject.

2. In case the algorithm did not reject in any one of the iterations, it accepts.

The procedure odd-cycle(s)

1. Let $K = poly((logN)/\epsilon) \cdot \sqrt{N})$, and $L = poly((logN)/\epsilon)$;
2. Perform K random walks starting from s, each of length L;
3. If some vertex v is reached (from s) both on a prefix of a random walk corresponding to an even-length path and on a walk-prefix corresponding to an odd-length path then return $found$. Otherwise, return $not - found$.

Given the graph $G = (V, E)$, we can define a probability function $p_s(v)$ as the probability that a random walk of length L that starts at s will end at v. When this function behaves like the stationary distribution and is bounded from below and above, the analysis can become more simple. Lets assume that for any $v \in V$ $\frac{1}{2N} \leq p_s(v) \leq 2N$. If this is the case, we define (for a given $s, v\ inV$): p_v^0, The probability that a random walk starting at s and ending at v corresponds to an even length path and p_v^1 the probability that a random walk of length L starting at v and ending at s corresponds to an odd-length path. We know that for every $v \in V$, $\frac{1}{2N} \leq p_s(v) = p_v^0 + p_v^1 \leq \frac{2}{N}$. There are two cases regarding the sum $\sum_{v \in V} p_v^0 \cdot p_v^1$. If the sum is relatively "small" (smaller than $c \cdot \frac{\epsilon}{N}$ for a constant $c < 1$), We define the partition (V_0, V_1), where $V_0 = \{v| : p_v^0 \geq p_v^1\}$ and $V_1 = \{v|p_v^1 \geq p_v^0\}$. Consider a vertex $v \in V_0$. By definition of V_0 and our assumption, $p_v^0 \geq \frac{1}{4N}$. Assume v has neighbors in V_0. Then for each such neighbor u, $p_u^0 \geq \frac{11}{4N}$ as well. However, since there is a probability of $\frac{1}{2d}$ of taking a transition from u to v in walks on G, we can infer that each neighbor u contributes $\Omega(\frac{1}{2d} \cdot \frac{1}{4N})$ to the probability p_v^1. Thus if there are many violating edges with respect to (V_0, V_1), then the sum $\sum_{v \in V} p_v^0 \cdot p_v^1$ is large, contradicting our case hypothesis. In the second case, $\sum_{v \in V} p_v^0 \cdot p_v^1 \geq c \cdot \frac{\epsilon}{N}$. Here we consider the matrix of random variables $x_{i,j}$ that are 0 if the i and the j walks both ends at the same vertex v with different parity. The expected value of each of these variables $x_{i,j} = \sum_{v \in V} 2 \cdot p_v^0 \cdot p_v^1$. Therefore, since there are $K^2 = \Omega(N)$ (since the number of walks K is polynomial in $\sqrt{N}$) such variables, the expected value of their sum is greater than 1. Although these variables are not pairwise independent, a bound is calculated on the probability that their sum is 0.

The problem is that not for all graphs our assumption on the probability function $p_s(v)$ holds. The assumption that this function behave like the approximately like the stationary distribution is not true in most cases. However, it can be shown every graph that is accepted in probability that is greater than $\frac{1}{3}$, can be partitioned into subgraphs that for each of them there exists a vertex s_i and such probability function p_{s_i} that is bounded from below and above by appropriate boundaries that allow us to use the same technique as above (checking $\sum_{v \in V} p_v^0 \cdot p_v^1$). In addition it is shown that each of these partitions has small cut with the rest of the graph, such that each of them can be partitioned independently.

The analysis goes as follows: In the iteration i, a vertex s_i is chosen and for this vertex a set S_{s_i} is defined in such way that the probability p_{s_i} on S_{s_i} will be bounded from above and below. It is shown that such set S_{s_i} exists. It is then shown that the same procedure can applied on $V \setminus \cup_{j<i} S_{s_j}$. In order to make the analysis on G after some of the vertices were removed from it a markov chain is defined. This is to simulate walks on G only on the part that was not removed in the previous iterations. For each of these sets it is shown that the set has a small cut with the rest of G and that the set can be partitioned without many violations. Below are the main lemmas that are given in the paper.

2.1.3 The Markov Chain $M_{l_1}^{l_2}(H)$

As was explained in the sketch of the proof of the algorithm, the partition of G is done in iterations where in each iteration i an additional set of vertices $S_{s_i} \subseteq H = G \setminus \cup_{j<i} S_{s_j}$ is removed from H and being partied. Our problem is how to treat the probabilistic transition function in H. Some of the vertices from G are absent in H and we should adapt the graph for this. Therefore, we define the markov chain $M_{l_1}^{l_2}(H)$. Through this markov chain the probabilistic transition function over H is defined. For any given pair of length, l_1 and l_2, we define a Markov Chain $M_{l_1}^{l_2}(H)$. $M_{l_1}^{l_2}(H)$ captures random walks of length at most $l_1 \cdot l_2$ in G that do not exit H for (sub)walks of length l_2 or more. The states of the chain consist of the vertices of H and some additional auxiliary states. For vertices that do not have neighbors outside of H, the transition probabilities in $M_{l_1}^{l_2}(H)$ are exactly as in walks on G. However, for vertices v that have neighbors outside of H there are two modifications: (1) For each vertex u, the transition probability from v to u ,denoted $q_{u,v}$, is the probability of a walk (in G) starting from v and ending at u after less than l_2 steps (without passing through any other vertex in H). Thus, walks of length less than l_2 out of H (and in particular the walk $v - u$ in the case $(u, v) \in E$, are contracted into single transitions. (2) There is an auxiliary path of length l_1 emitting from v. The transition probability from v to the first auxiliary vertex on the path equals the probability that a walk starting from v exits H and does not return in less than l_2 steps. From the last auxiliary path there are transitions to vertices in H with the corresponding conditional probabilities of reaching them after such a walk. The markov chain $M_{l_1}^{l_2}$ therefore simulates a walks on H without going through vertices that were already removed for partition.

2.1.4 The proof of the algorithm

For the first step in the proof, it is shown that in every stage in the partition of the vertices left in H (after all the sets S_i that were already partitioned were removed), there exists a vertex $s \in H$ that is both good and useful. It is assumed that H contains at least

Lemma 1 *Let H be a subgraph of G, and l_1 and l_2 be integers. The probability that a walk in $M_{l_1}^{l_2}(H)$ starting from a uniformly chosen vertex of H enters an auxiliary path after at most l_1 steps , is at most $\frac{2l_1}{l_2} \cdot \frac{|G|}{|H|}$.*

Definition 1 *We say that a vertex s is useful with respect to $M_{l_1}^{l_2}(H)$ if the probability that a walk in $M_{l_1}^{l_2}(H)$ starting from s enters an auxiliary path after at most l_1 steps, is at most $\frac{4l_1}{l_2} \cdot \frac{|G|}{|H|}$.*

Corollary 1 *Let H be a subgraph of G, and l_1 and l_2 be integers. Then at least half of the vertices s in H are useful with respect to $M_{l_1}^{l_2}(H)$.*

After a vertex that is both good and useful were chosen, it is necessary to show that there exists a set S_s such that S_s can be both partitioned without many violations and has a small cut with the rest of G. We can assume that H contains no more than $\frac{\epsilon}{4}N$ vertices (if it contains less, they can be partitioned arbitrarily). The following lemma is used to prove that there exists a set S_s such that S has a small cut with the rest of G and the probability to reach a certain vertex in S from s (the source vertex) is similar to the stationary (and thus we can apply the analysis that treats $p_s(v)$ as a stationary distribution):

Lemma 2 *Let H be a subgraph of G with at least $\frac{\epsilon}{4}N$ vertices, and let $l_1 = \Theta((\frac{log(N/\epsilon)}{\epsilon^3})^3)$, $l_2 = \Theta(\frac{l_1}{\epsilon^2})$, and $F = O(\frac{1}{\epsilon})$. Then for every vertex s that is useful with respect to $M_{l_1}^{l_2}(H)$, there exists a subset of vertices S in H an integer t, $l_1/2 \leq t$*
$leql_1$, and a value $\beta = \Omega(\frac{\epsilon^2}{log(N/\epsilon)})$, such that:

1. *The number of edges between S and the rest of H is at most $\frac{\epsilon}{2} \cdot d \cdot |S|$.*

2. *For every $v \in S$, $\sqrt{\frac{1}{|S|} \cdot \frac{\beta}{|H|}} \leq q_{s,v}(t) \leq F \cdot \frac{1}{|S|} \cdot \frac{\beta}{|H|}$.*

After the set S_s was chosen, and it was shown that S has a small cut with the rest of G and a stationary-like probability function $p_s(v)$ (for the probability to reach a vertex v from s on a walk of length $l_1 \cdot l_2$, it is needed to show that S_s can be partitioned without many violations. As it was shown in the beginning, there are two cases here regarding the sum $\sum_{v \in S} q_{s,v}^0(t) \cdot q_{s,v}^1(t)$. If this sum is relatively small (smaller then $\frac{\epsilon}{c} \cdot |S| \cdot \alpha^2$ for a constant c and α as in the lemma), then there exists a partition of S_s with small number of violating edges. This is shown in the following lemma:

Lemma 3 *Let H be a subgraph of G, s a vertex in H, S a subset of vertices in H and l_1 and l_2 integers. Assume that for some $\alpha > 0$ and $t = \Omega(log(\frac{1}{\alpha}))$ the following holds in $M_{l_1}^{l_2}(H)$:*

1. *For every $v \in S$, $q_{s,v}(t) \geq \alpha$*

2. *$\sum_{v \in S} q_{s,v}^0(t) \cdot q_{s,v}^1(t) < \frac{\epsilon}{c} \cdot |S| \cdot \alpha^2$ for some constant c.*

Let (S_0, S_1) be a partition of S, where $S_0 = \{v | q_{s,v}^0(t) \geq q_{s,v}^1(t)\}$. Then the number of violating edges in G with respect to (S_0, S_1) is at most $2^5 \cdot \frac{\epsilon}{c} \cdot d|S|$.

The other case, is that the sum $\sum_{v \in S} q_{s,v}^0(t) \cdot q_{s,v}^1(t)$ is relatively large. In this case it is shown that the probability to found an odd cycle in $O(\frac{F}{\epsilon \cdot \alpha \sqrt{|S|}})$ is at least 0.99. This is shown by the following lemma:

Lemma 4 *Let H be a subgraph of G, s a vertex in H, S a subset of vertices in H and l_1 and l_2 integers. Assume that for some α, $F > 0$ and $t < l_1$, the following holds in $M_{l_1}^{l_2}(H)$:*

1. *For every $v \in S$, $\alpha \leq q_{s,v}(t) \leq F \cdot \alpha$.*

2. *$\sum_{v \in S} q_{s,v}^0(t) \cdot q_{s,v}^1(t) \geq \frac{\epsilon}{c} \cdot |S| \cdot \alpha^2$ for some constant c.*

Then with probability at least 0.99, if we perform $O(\frac{F}{\epsilon \cdot \alpha \cdot \sqrt{|S|}})$ random walks of length t starting from s in $M_{l_1}^{l_2}(H)$ then for some vertex v we shall end at v both on a walk corresponding to an even-length path and on a walk corresponding to an odd-length path.

And from here the following corollary can be derived:

Corollary 2 *Let H be a subgraph of G and $S, s, l_1, l_2, t, \alpha$ and F as in the previous lemma. Then with probability at least 0.99, if we perform $O(\frac{F}{\epsilon \cdot \alpha \cdot \sqrt{|S|}})$ random walks of length $l_1 \cdot l_2$ starting from s in G then for some vertex v in S we shall reach v both on a prefix of a walk that corresponds to an even-length path and on a prefix that corresponds to an odd-length path.*

2.1.5 Conclusions

In the article by Oded Goldreich and Dana Ron, they present a simple algorithm to test the bipartiteness of a graph. In order to check whether the graph is bipartite, an odd cycle is searched. Such a cycle is an evidence that the graph is not bipartite. If the graph is bipartite, no odd-length cycle exists and the algorithm always right. It is proved that if the graph G is ϵ-far from being bipartite, that means that there are more then ϵdN violating edges in any partitioning of the graph, the graph is rejected with probability of at least $\frac{2}{3}$.

2.1.6 Remark

Bipartite checking here is somewhat not typical because it relies on old cycle search. A more typical checkers studied by them relies on randomic choosing a small subgraph and testing whether it has the property.

2.2 Learning to Reason - work by Roni Khardon and Dan Roth

2.2.1 Introduction

In a work by Roni Khardon and Dan Roth [17], a new framework the study of reasoning is developed. Reasoning is abstracted as a deduction task of determining whether a query α, assumed to capture the situation at hand, is implied from knowledge based system KB, or whether $KB \models \alpha$. In the paper the framework for reasoning using sampling is developed. A definitions for algorithms that are learning to reason are presented. Both for exact learning and Probably Approximately Correct (PAC) learning. In order to overcome on the accuracy limitations of the learning algorithms, the term of a fair query is defined. Fair queries are queries for which the probability that we will reason correctly is greater than $1 - \delta$. The approach to reasoning by samples presented in the paper is different from other approaches in that it sees learning as integral part of the reasoning process. Reasoning from samples might use two approaches: The first one is to learn the world and to reach a hypothesis about the world first and then to reason from this hypothesis. The other approach that is presented in this paper is to combine learning and reasoning into one integral process. The advantage making learning and reasoning an integral process is that in some cases either learning or reasoning in separate might be intractable while reasoning from samples is tractable.

The authors give a Learning to Reason algorithms for classes of propositional languages for which there are no efficient reasoning algorithms, when represented as a traditional (formula based) knowledge base. This is an example when the task of reasoning from the hypothesis (the knowledge base) is not possible. The authors also exhibit a Learning to Reason algorithm for a class of propositional languages that is not known to be learnable in the traditional sense. This is an example when the learning of the concept (versus reasoning from the hypothesis) is not possible.

2.2.2 Definitions

Let $W \in \mathcal{F}$ be a Boolean function that describes the world exactly. Let $\mathcal{Q}$ be the class of queries considered, α be some Boolean function (a query) and let D be some fixed but arbitrary and unknown probability over the instance space $\{0,1\}^n$. We assume that D governs the occurrences of instances in the world:

Definition 1 *The query α is called legal if $\alpha \in \mathcal{Q}$.*

Definition 2 *The query α is called (W, ϵ)-fair if either $W \subseteq \alpha$ or $Prob_D[W \backslash \alpha] > \epsilon$.*

The intuition here is that if $Prob_D[W \backslash \alpha]$ is very small, we consider the query α not fair, and we allow the algorithm to make error. We will similar definition in our verification using learning schemes. The sampling approach used for learning to reason is one time sampling. We sample only once at the beginning and then we check all the queries against this sample. This is in order to bound all the sampling into the "grace period" of the reasoning algorithm. This is because in some cases we want the reasoning algorithm no to make period after the grace period. If we had sampled a sample for each query, we couldn't have bound the grace period to the first part of the algorithm.

Definition 3 *An algorithm A is an exact reasoning algorithm for the reasoning problem $(\mathcal{F}, \mathcal{Q})$, if for all $f \in \mathcal{F}$ and for all $\alpha \in \mathcal{Q}$, when A is presented with input (f, α), A runs in time polynomial in n and the size of f and α, and answers "yes" if and only if $f \models \alpha$.*

Definition 4 *An algorithm A is a Probably Approximately Correct Learn to Reason algorithm for the reasoning problem $(\mathcal{F}, \mathcal{Q})$, if there exists a polynomial $p(,,)$ such that for all $f \in \mathcal{F}$, for any probability distribution D, on input ϵ, δ, and given access to $I(f)$, A runs in time $p(n, 1/\epsilon, 1/\delta)$, and then with probability at least $1 - \delta$, when presented with any (f, ϵ)-fair query $\alpha \in \mathcal{Q}$, A runs in time $p(n, 1/\epsilon, 1/\delta)$, does not access $I(f)$, and answers "yes" if and only if $f \models \alpha$.*

The definition of learning to reason here require two phases of the learning algorithms (both exact and PAC): learning phase and answering phase. Both phases should have polynomial complexity in $n, 1/\epsilon, 1/\delta$. Of course, in the answering phase the algorithm is not allowed to use access interface to the world $I(f)$. The paper define also a counterexample oracle, similar to the counterexample oracle that we use also in learning. In the counterexample oracle we are provided with counterexample if we are wrong.

Definition 5 *A Reasoning Query Oracle for a function f and a query language $\mathcal{Q}$, denoted $RQ(f, \mathcal{Q})$, is an oracle that when accessed performs the following protocol with a learning agent A.*

1. *The oracle picks an arbitrary query $\alpha \in \mathcal{Q}$ and returns it to A.*
2. *The agent A answers "yes" or "no" according to its belief with regard to the truth of the statement $f \models \alpha$.*
3. *If A's answer is correct then the oracle says "correct". If the answer is wrong the oracle answers "wrong" and in case $f \not\models \alpha$ it also supplies a counterexample (i.e. $x \in f \setminus \alpha$).*

2.2.3 The relation between L2R and L2C

Here it is checked whether given an algorithm for learning to reason, is it necessarily the case that there is an algorithm that can Learn to Classify. The following theorem is proved: Let DISJ be the class of all disjunctions over n variables.

Theorem 1 *If there is an Exact-L2R algorithm for the reasoning problem ($\mathcal{F}$,DISJ) then there is an Exact-L2C algorithm for the class $\mathcal{F}$.*

The theorem is proved by making a reduction of any classification problem of a class $\mathcal{F}$, into a exact learning to reason problem (DISJ,$\mathcal{F}$).

2.2.4 Combining learning and reasoning

In this section the combination of learning and reasoning is considered. The output of the existing learning to classify algorithm is used for learning to reason. It is shown that this can be done successfully on some of the cases, but is limited to the particular type of learning algorithms.

Definition 6 *An algorithm that PAC Learns to Classify $\mathcal{F}$ is said to learn $f \in \mathcal{F}$ from below if, when learning f, the algorithm never makes mistakes on instances outside of f. (I.e., if h is the hypothesis the the algorithm keeps then it satisfies $h \subseteq f$.)*

The following theorem is then proved:

Theorem 2 *Let A be A PAC-Learn to Classify algorithm for the function class $\mathcal{F}$ and assume that A uses the class of representations $\mathcal{H}$ as its hypotheses. Then, if A learns $\mathcal{F}$ from below, and there is an exact reasoning algorithm B for the reasoning problem $(\mathcal{H}, \mathcal{Q})$, then there is a PAC-Learn to Reason algorithm C for the reasoning problem $(\mathcal{F}, \mathcal{Q})$.*

It is important to note that we are limited here to learning algorithms that learn $\mathcal{F}$ from below. The advantage of this attitude that we are not limited to example oracle (like we are required in the sampling approach).

2.2.5 Learning to Reason via Model Based Reasoning

In this section the authors present their main technical results. They show two examples of propositional languages classes for which their concept of learning to reason may be utilized effectively.

2.2.6 Learning to Reason $CNF \cap DNF$ without reasoning

For the propositional language class $CNF \cap DNF$ there is no efficient reasoning algorithms, when represented as a traditional (formula-based) knowledge base. So the straightforward combination of learning a function from this class and then reasoning from the hypothesis selected would not work if we represent the hypothesis using the formula based knowledge base. Instead, the algorithm that the authors use constructs a knowledge based representation that allows for efficient reasoning. The authors use the monotone theory developed by Bshouty [7]. [1]

In [7] an exact algorithm that learns to classify the class of boolean functions $CNF \cap DNF$ using equivalence queries is presented. One of the byproducts of this algorithm is constructing the set of all minimal models with respect to a basis B of f, Γ_f^B. [2] The minimal models set Γ_f^B is used as knowledge base using which the reasoning is made. This set can serve us only for a restricted type of queries. The algorithm presented in [7] builds this set as a byproduct of an exact learning to classify algorithm that uses an equivalence queries oracle. The following theorem shows that learning to reason can be done effectively for the function class $DNF \cap CNF$ using restricted type of queries.

[1] Some of the theorems in this section are based on the monotone theory. The monotone theory itself is beyond the scope of this paper and the theorems here are cited from [17] only to provide an example of a function class for which there exists an efficient algorithm for Learning to Reason while there is not known reasoning algorithm when represented in the traditional formula based knowledge based. More details on the monotone theory can be found in [17] and [7]

[2] For more thorough details on the monotone theory, on bases of boolean function classes and the minimal models set, see [17] and [7].

Theorem 3 *There is an Exact-Learn to Reason algorithm,that uses an Equivalence Query oracle and a Membership Query Oracle for the reasoning problem* $(CNF \cap DNF, \mathcal{Q})$*, where* $\mathcal{Q}$ *is any class of relevant and common queries.*

1. **Relevant queries**

   Definition 7 *Let* B *be a monotone basis for the Boolean function* f*. A class* $\mathcal{Q}$ *of boolean functions is relevant to* f *if* B *is also a monotone basis for all the functions in* $\mathcal{Q}$*.*

   Using the following theorem, results from studies on model based representation of knowledge [16]:

   Theorem 4 *Let* $f, \alpha \in \mathcal{F}$ *and let* B *be a basis for* $\mathcal{F}$*. Then* $f \models \alpha$ *if and only if for every* $u \in \Gamma_f^B, \alpha(u) = 1$*.*

   Since the algorithm in [7] produces as a byproduct the set of Γ_B^f we can check for any relevant query α whether $f \models \alpha$.

2. **Common queries**

   Definition 8 *A class* $\mathcal{Q}$ *of Boolean functions is common if* $\mathcal{Q}$ *has a fixed polynomial size monotone basis.*

   We use the following theorem ([16]):

   Theorem 5 *Let* $f \in \mathcal{F}, \alpha \in \mathcal{G}$ *and let* B *be a basis for* $\mathcal{G}$*. Then* $f \models \alpha$ *if and only if for every* $u \in \Gamma_f^B, \alpha(u) = 1$*.*

   Since we know that our queries class is common, we know that the size of the monotone basis B is fixed and polynomial. In this case, given the basis B, the algorithm in [7] provides us as a byproduct with the set of minimal models of f with respect to B, provided that f has a small DNF. Now we can use the theorem for the reasoning.

The size of Γ_f^B is shown in the paper to smaller then $|B| \cdot |DNF(f)|$. In this section it is shown that sometimes even when reasoning from the hypothesis is not possible, we can learn a compact size knowledge based and use it for reasoning.

2.2.7 Learning to Reason DNF without learning to classify

In this section an algorithm for Learning to Reason is presented for the class of polynomial size DNF boolean functions. The results here are significant because there is no known algorithm that Learns to Classify this class. It is important to note that an algorithm for Learning to Classify DNF was recently developed by Jackson [14], but this algorithm is limited to the assumption that the distribution over the examples is uniform. While PAC learning require that the algorithm works for any distribution on the examples. Two algorithms are presented in the paper and I'll bring here only one of them. The algorithms presented in the paper are for Exact Learning to reason and PAC Learning to Reason. I'll describe here the Exact Learning to Reason algorithm. It is important to note that this algorithm approximate the world with its knowledge base and still exhibits an Exact Learning to Reason behavior. The algorithm also uses the monotone theory ([7]).

The algorithm interacts with the reasoning query oracle RQ and collects set of models $\Gamma = \cup_{b \in B} \Gamma_b$. If the algorithm makes a mistake (a mistake of the algorithm occurs only when it answers that for a query α, $f \models \alpha$, while there is x for which $f(x) = 1$ and $\alpha(x) = 0$. If the algorithm makes a mistake, it uses the counter examples oracle to add the missing model to Γ.

1. $\forall b \in B$, *initialize* $\Gamma_b \leftarrow 0$.
2. $\alpha \leftarrow RQ(f, \mathcal{Q})$
3. *Answer* $f \models \alpha$ *by performing model-based test on* $\Gamma = \cup_{b \in B} \Gamma_b$.
4. *If "wrong" then*
5. *let* x *be the counterexample received from* $RQ(f, \mathcal{Q})$
6. $\forall b \in B$ *such that* $x \notin \mathcal{M}_b(\Gamma_b)$
7. $\Gamma_b \leftarrow \Gamma_b \cup$ *Find-min-model(x,b)*
8. *Goto 2*

Procedure: Find-min-model(x,b)

1. *If* $\exists y \in \lfloor x \rfloor_b$ *such that* $f(y) = 1$
2. $x \leftarrow y$*: Goto 1*
3. *else*
4. *Return(x)*

Important feature of the algorithm here is that every mistake that the algorithm do, provides him with counterexample that helps it to improve its future answers by expanding its models set Γ_b.

2.2.8 Conclusions

The paper Learning to Reason [17] provides suggestion for framework for Learning to Reason. It defines oracles used for Learning to Reason the type of Learning to Reason algorithms and the term of fair query, a query type that is more easy to answer. The main technical results brought in the paper are two examples of propositional languages classes for which the the concept of learning and reasoning in one process can be utilized effectively as against to the other approach of learning the world and then making the reasoning on the hypothesis chosen by the learning algorithm.

2.3 Automatic Abstraction Techniques for Propositional μ-calculus Model Checking

2.3.1 Introduction

In this section I'll summarize work done by Abelardo Pardo and Gary D. Hachtel. Their work [21] presents a paradigm for approximating formal verification for the μ-calculus temporal logic. The motivation for establishing this paradigm is that many times the memory space limits of the system are reached and the verification fails. In order to deal with this they present a paradigm for gradual verification of the system. Given a μ-calculus formula ψ and a system M, the algorithm can answer that $M \models \psi$ or that $M \not\models \psi$ or that the memory space limits were reached. The algorithm approximates the set of states that fulfills the property ψ, $Sat(\psi)$ and outputs a subset of this set $\widehat{Sat(\psi)} \subseteq Sat(\psi)$. The computation is done using the tree of the formula ψ, where each

vertex in the tree represents one of the operators (or subformulas) of ψ. In each vertex we will try to compute the set of states that fulfills the subformula in this vertex. If we see that the memory space limits of the system are reached, we will compute only a subset of this set. Upon reaching the root of the tree, we compute the approximate set of states that fulfills ψ, $\widehat{Sat(\psi)} \subseteq Sat(\psi)$. If the set of initial states I fulfills $I \subseteq \widehat{Sat(\psi)}$, then we know for sure that the system fulfills the property ψ and the algorithm exits. However, if $I \nsubseteq \widehat{Sat(\psi)}$, then this might be because $\widehat{Sat(\psi)}$ is only approximation (subset) of $Sat(\psi)$. In this case a refinement procedure is called and its goal is to include $I \setminus \widehat{Sat(\psi)}$ in the new approximation of $Sat(\psi)$. If we succeed then the system fulfills ψ and the algorithm exit. If we fail again then $M \not\models \psi$ or that the memory space limits were reached.

In the following sections, I'll describe the paradigm in more detail:

2.3.2 Basic definitions and verification of μ-calculus formulas

Definition 1 *A Kripke structure is a tuple $M = (S, R, A, \lambda)$ in which S is a set of states, $R \subseteq S \times S$ is a transition relation, A is a set of atomic propositions, and $\lambda : A \rightarrow 2^S$ is a labeling function that returns the set of states in S labeled with a given atomic proposition.*

Definition 2 *Given a Kripke structure $M = (S, R, A, \lambda)$ and a set of states $S_1 \subseteq S$, we define the image function $Img(R, S_2)$ as the set of states S_1 such that $\forall s \in S_1, \exists s' \in S_2, (s', s) \in R$. We also define the reverse image function $Pre(R, S_1)$ as the set of states S_2 such that $\forall s \in S_2, \exists s' \in S_1, (s, s') \in R$.*

Definition 3 *Given the set of variables X and a state space S, we define an* environment *as a function $e : X \rightarrow 2^S$.*

We use the environment e, when we want to calculate the assignment operator $\mu x.\psi$. In this scenario, the value of $e^i(x) = Sat(\psi, e^{i-1})$.

In the paper the algorithm for the computation of $Sat(v, e)$ at each vertex v is presented. For each v this computation is done in induction on the main operator in the subformula at the vertex v. For v_0, the expression is ψ. I'll bring here only the computation of the operator $Eval\mu x(v, e)$. For the rest of the operators, $\neg, \wedge$ and X, this computation is trivial and is done in induction on the sons of v and the same e.

```
funct Evalμx(v, e)
   x := ReadValue(v);
   result := ∅;
   do
      e(x) := result;
      result := Sat(v1, e);
   while (result ≠ e(x)) od;
   return result;
end
```

Given a Kripke model M, the set of all states that satisfies the formula ψ is $Sat(\psi, e\bot)$, where $e\bot$ is the environment that maps every variable to the empty set.

Definition 4 *We define the labeled operational graph of the formula $\psi \in L\mu$ as $G = (V, E, P)$. V is a set of vertices. Each vertex is of the type $\{and, not, EX, \mu x\} \cup \{p \in A\} \cup \{x \in X\}$, where A is the set of atoms and X is the set of variables. Each vertex represents subformula of ψ. $(v_1, v_2) \in E$ if v_2 is a direct sub-formula of v_1. We will denote by $top_\psi \in V$ the vertex representing ψ. Every vertex v is labeled with a parity which is the number of vertices of type not that are traversed in the path from v to top_ψ excluding v itself. P is a function $P : V \to \{+, -\}$ such that $P(v) = +$ if v has odd parity and $P(v) = -$ if v has even parity.*

The most acute problem that we with during model checking is the states explosion problem. For a system with n variables, the model checking algorithm should handle sets with size that is exponential in n. The Kripke model that represents the transition relation between these states, should represent transition relation R that is also of size exponential in n. In order to make the model checking practical and efficient and OBDD representation is used. [3] OBDD is a canonical representation for boolean functions. The sets of states are represented by their characteristic boolean function. The relation R is represented by the boolean function $R(v_1, v_2)$ that is true iff $\langle v_1 v_2 \rangle \in R$. A formal verification algorithm that is using OBDD representation is called symbolic checking algorithm. In some cases, an OBDD representation might be made significantly more compact by taking a subset or a superset of the set represented by the OBDD. We will use extensively this property of the OBDD during the algorithm.

2.3.3 Conservative Abstraction

Abstraction is a paradigm that is used in order to overcome the problem of states explosion in verification. Instead of verifying the original system M, another system, that approximate M, but has fewer states is verified. Since we represent all the states sets using OBDDs, the complexity depends on the representation of the states sets using OBDD rather then in the actual sizes of the states sets.

Definition 5 *Given a Kripke structure M, we say the function $\widehat{Sat}$ provides a conservative interpretation if and only if $\forall \psi \in L\mu, \widehat{Sat}(\psi, e\bot) \subseteq Sat(\psi, e\bot)$.*

Therefore, if the set of initial states of the Kripke model M is I and our verification algorithm provides us with a conservative interpretation $\widehat{Sat}(\psi, e\bot)$, if $I \subseteq \widehat{Sat}(\psi, e\bot)$, we know for sure that $M \models \psi$. However, if $I \not\subseteq \widehat{Sat}(\psi, e\bot)$, we cannot make any conclusions on whether ψ holds in M. In this case, the approximation is too conservative to make any conclusions and we should apply a refinement on the approximation.

In order to get a conservative approximation of $Sat(v, e\bot)$ at a certain vertex v, we should check which type of approximation we should make on any of v children, or formally on any v' such that $(v, v') \in E$. Since v represents a subformula of ψ and any of its children represents a direct sub-formula of v, the type of approximation required for any of v children depends on the main operator in v. If v holds any of the monotone operators such as $\wedge$,μ or EX, then for any of its children v', $Sat(v', e\bot)$ should also be approximated by a conservative approximation. However if the main operator in v is $\neg$, then if we want to get a conservative approximation at v, we need to approximate $Sat(v', e)$ by $\widehat{Sat}(v', e)$ such that $\widehat{Sat}(v', e) \supseteq Sat(v', e)$. The type of approximation (conservative or a superset) in a vertex v depends on the parity of the vertex or the number of negations that are traversed from v to v_0 excluding v itself.

There are a lemma and a theorem in the article that describe the connection between the approximations computed at neighboring vertices:

[3] OBDD will be discussed more thoroughly later in the introduction to the work on ordering OBDD variables.

Lemma 1 *Let us consider an operational graph $G = (V, E, P)$ and two vertices v_1, v_2 such that $(v_1, v_2) \in E$. Let us assume that in the computation of $Sat(v_1, e)$ the evaluation of v_2 has been approximated by $\widehat{Sat}(v_2, e)$. If v_1 is of type not and $\widehat{Sat}(v_2, e)$ is a superset (subset), then the computed $Sat(v_1, e)$ is a subset (superset) of the exact result. If v_1 is of type and, EX or μx and $\widehat{Sat}(v_2, e)$ is a superset (subset), then $Sat(v_1, e)$ also is a (subset) of the exact result.*

Theorem 1 *Let us consider a Kripke structure M, a formula $\psi \in L\mu$, and its labeled operational graph $G = (V, E, P)$. Any function $\widehat{Sat}$ such that for every vertex $v \in V$*

$$Sat(v, e) \subseteq \widehat{Sat}(v, e) if P(v) = +$$

$$Sat(v, e) \supseteq \widehat{Sat}(v, e) if P(v) = -$$

provides a conservative interpretation of ψ in M.

Using these definitions, we can make the approximations at any stage in the computation of $Sat(\psi, e)$. The advantage in this paradigm is that the approximations can be done locally on each vertex. For example: If we are in an and ($\wedge$) vertex v, and the parity of the vertex is $+$ (odd parity). We know that also the parity of our children is odd (since the vertex is and and its parity is odd). Therefore, we know that the approximation in each of the children is approximation from above. If the children are v_1 and v_2, $\widehat{Sat}(v_1, e) \supseteq Sat(v_1, e)$ and the same for v_2. Therefore, if we are lack of memory space for computing the operator and on the two operands, we know that we can propagate one of the operands ($\widehat{Sat}(v_1, e)$ or $\widehat{Sat}(v_2, e)$) and provide thus a valid approximation from above for $Sat(v, e)$.

2.3.4 An Automatic Abstraction and Refinement Algorithm

In this section an algorithm is presented for verification of the system. The algorithm works in two phases: approximation phase and refinement phase. In the approximation phase, for each of the operators an approximation is computed according to the parity of the vertex. If the vertex is of odd parity, then a superset of the exact result is computed. If the vertex is of even parity, a subset of the exact result is computed. After the conservative approximation for the root, ψ, vertex was computed it is checked whether $I \in \widehat{Sat}(\phi, e)$. If yes the algorithm can decide that ϕ holds. If no, then the algorithm enter to the second phase. It tries to refine the approximation with the set of goals $I \setminus \widehat{Sat}(\phi, e)$. The refinement process can end with one of the three results: ϕ can be proved true or false, or the algorithm can exit due to lack of resources. At each of the vertices we can make approximation according to the parity of the vertex. Any kind of approximation is legal as long as it provides conservative approximation for vertex with even parity and over approximation for vertex with odd parity. In general the size of a set represented by OBDD can be reduced by adding/removing items from the set. If the system resources are going to be exhausted and we are at a vertex v, we can remove/add items to the set at v or to its children according to the parity of v. In addition the following approximations are suggested for each type of the operators:

- **Negation** $\neg$ No approximation is needed, since in order to compute the exact result in constant time operation.
- **And** $\wedge$ In order to achieve conservative approximation, we can remove items from either of the sets or from the result. In order to achieve over approximation, we can add items to one of the sets or return one of the sets.

- **Fix point** μx in order to achieve conservative approximation, we can stop the computation of the fix point $\emptyset = \mu_0 \subseteq \mu_1 \subseteq ... \subseteq \mu_i \subseteq \mu_\infty$. Any subset μ_i is legal subset of the exact result. If we want to achieve a superset approximation we must compute the exact result.

- **EX** A computation method that approximate the EX operator is based on manipulating the transition relation as a conjunction of relational blocks.

The refinement algorithm: At any vertex we are given a set of goals, a set of vertices that we want to include (exclude) in the conservative (over) approximation. At the first stage, we are given a vertex v and a set of goals f_v. We compute which set of goals f_{v_i} to propagate to the children.

```
funct RefineVertex(v, f_v):boolean
  if (f_v = ∅) then return TRUE;
  if (Sat_v is an approximation) then
    (Sat'_v, f'_v) := RefineApproximation(Sat_v, f_v);
    if (f'_v = ∅) return TRUE
    f_v := f'_v;
  endif
  Sort Sub-formulas;
  foreach (Sub-formula v_i) do
    f_{v_i} := PropogateGoalSet(v, v_i)
    result_i = RefineVertex(v_i, f_{v_i})
  od
  if (RefinementInSubFormulas(result)) then
    Sat'_v := ReEvaluate(v);
    if ((P(v) = +) ∧ (f_v ∩ Sat'_v = ∅)) return TRUE
    if ((P(v) = −) ∧ (f_v ⊆ Sat'_v)) return TRUE;
  endif
  return FALSE
end
```

The approximation at a vertex v may be a result of two possible sources of approximations: (1) at the operands of v or v children. (2) an approximation that was made during the calculation of the operator in v. If an approximation was done during the calculation of v, the operation is calculated again ($RefineApproximation$). This time, the calculation is done in such way that f_v will be included in the result. If the new result include f_v ($f'_v = \emptyset$), then the procedure return TRUE. If it fails, then for each of the children v_i the set f_{v_i} is propagated. This set include the goals for the computation at v_i. For each of the operators we choose the set f_{v_i} as follows:

- **Negation** $\neg$ $f_{v_1} = f_v$

- **Conjunction** $\wedge$ If $P(v) = +$ then $f_{v_1} = f_v$ and $f_{v_2} = Sat'_{v_1} \cap f_v$. If $P(v) = -$ then $f_{v_1} = f_v$ and $f_{v_2} = f_v$.

- **EX** $f_{v_1} = Img(R, f_v)$ (in order to include f_v in $Sat(v, e)$, we need to include $Img(R, f_v)$, the successors of f_v at $Sat(v_1, e)$.

- **Fixed point vertex** μx $f_{v_1} = f_v$

- **Variable(x) formula** In this vertex the refinement can be done at the vertex containing the fix point operator.
- **Atom(p)** If we reach a vertex with an atom, this vertex (leaf) cannot be refined since it contain the exact result of $Sat(v, e)$. Therefore, if we reach an atom vertex with a set of goals we exit with FALSE. In this case $M \not\models \psi$.

After the algorithm refines all the subformulas of v it checks whether v should be refined again. The refinement is done again in a way to produce a conservative or over approximation as needed by v parity.

The correctness of the refinement and reevaluation algorithm is proved using the following proposition:

Proposition 1 *Given a vertex v with sub-formula v_1 (and v_2 when applicable), for every sub-formula v_i, $RefineVertex(v_i, f_{v_i}) = TRUE \Rightarrow RefineVertex(v, F_v) = TRUE$.*

Theorem 2 *For a given vertex v, an approximation Sat_v, and a set f_v, the algorithm returns $TRUE$ if the new approximation Sat'_v satisfies:*

$$Sat'_v \subseteq Sat_v \setminus f_v \text{ if } P(v) = +$$
$$Sat_v \cup f_v \subseteq Sat'_v \text{ if } P(v) = -$$

and FALSE otherwise.

Using this theorem we see that if the algorithm return with $TRUE$, then we can conclude that $M \models \psi$. This is since $I \subseteq Sat_v \cup f_v \subseteq Sat'_v$, and Sat'_v is the new approximation returned from the algorithm. If the algorithm returns $FALSE$ we have two options: it is might be because $M \not\models \psi$ (f_v could not be included in Sat'_v) or that the system resources were exhausted.

2.3.5 Conclusions

The paradigm presented in the paper by Pardo and Hachtel provides a general setting for approximated verification of $\mu - calculus$ properties. This paradigm enable local approximation for each type of the operators of the $\mu - calulus$. The algorithm also presents a technique to gradually refine the approximate results. The paradigm enables using different approximation technique as long as we keep that it would be conservative/over approximations according to the parity of the vertices.

2.4 Related work - and the connection to this work

The three papers just described are relevant in different aspects to the problem of using learning methods in verification. The sublinear bipartiteness tester by D. Ron and O. Goldreich [13] checks general properties of a graph using sampling. The automatic abstraction techniques by A. Pardo and G. Hachtel [21] approximates the set of states that satisfy a $\mu - calculus$ query in order to overcome the memory limits. The learning to reason framework by D. Roth and R. Khardon [17] presents approach for reasoning by sampling. I'll discuss below each of these work in the specific context of this work: using learning methods in verification.

The problem of testing properties in graphs has many common features with the problem of verification. In each we have a graph (in verification this is the Kripke model) and we should answer whether a specific property holds in this graph. There are testing algorithms for several graph properties such as clique, bipartiteness or connectivity. However, since there is no natural

way to express CTL or LTL expressions using these graph properties, these algorithms are not very helpful for verification.

Moreover, the Kripke model, the graph that is used to represent the system in verification is a *colored* graph unlike the graphs that we are dealing with when testing the basic graph properties.

Still, there are two important features in this paper that are relevant for my work:

1. **Learning by sampling subgraphs** One of the paradigms that is used for testing graph properties, and is also used in this paper is sampling a set subgraphs from the graph, checking a certain property in the subgraphs and then deducting to the graph. This method, as I'll show later seems quite useful also for program testing and can be used for testing CTL properties.

2. **Considering the distance between graphs** In the bipartiteness tester work, graphs are rejected in probability of $\frac{2}{3}$, if they are *ϵ-far from being bipartite.* The paper therefore distinguish between graphs that does not satisfy the property of bipartiteness but are close to being bipartite and graphs that are far from being bipartite. In my work I'm also making a distinction between systems that close and far from satisfying certain LTL property. The actual definition of a graph being ϵ-far or ϵ-close to another graph will be presented soon.

The paper Learning to Reason [17] provides suggestion for framework for Learning to Reason. Verification and reasoning are very similar tasks. In both tasks we are given a model (in the verification setting, this is the Kripke model) and we would like to check a given properties on the model. However, in the verification setting the properties and the model are of temporal type and not static. The implications of the Learning to Reason paper on my work are in the following aspects:

1. **Learning and Reasoning in one process** In my work I'm also applying learning and reasoning in one process. Also in the case of formal verification the task of learning the system seems much more complicated then reasoning the LTL queries on the system.

2. **One time sampling approach** I'm using the one time sampling approach when testing LTL properties. The one time sampling approach can actually PAC learn any class of function. Its drawback is however that it does not use its reasoning mistakes. Another feature of this approach is that the approximate nature of PAC learning is expressed in the type of queries that we might ask, the fair queries and not in the answers of the algorithms. In probability of $1 - \delta$ the answers are correct.

The paradigm presented in the paper by Pardo and Hachtel provides a general setting for approximated verification of $\mu - calculus$ properties. The algorithm deals with the memory limits problems that arise when verifying large systems. It does so by approximating the set of states that satisfy the μ-calculus property. It is important to observe, that while the methods described above, the bipartiteness tester and the learning to reason approach, are not exact and their results are given in respect to an error ϵ and confidence δ, the algorithm of Pardo and Hachtel is exact. In case it cannot approximates the set of states that satisfy the μ-calculus property and arrive at a conclusion with the available space, it return "no space left" message.

In my work, the two approaches are used. In the OBDD work the task is two reach a compact representation of the data-structures used by the symbolic verification algorithms. Even if the order reached is not good, this does not affect on the results of the verification. In the LTL properties checking work, the algorithm answer with respect to an error ϵ and confidence δ.

3 Using Clustering to Order OBDD Variables.

3.1 Ordered Binary Decision Diagrams, OBDDs

3.1.1 Introduction to OBDDs

Ordered Binary Decision Diagrams, OBDDs for short, were introduced by Bryant [4] as a tool for symbolic representation of boolean functions. OBDDs are actually a compact representation of a boolean functions that is based on the binary decision diagrams. The main usage of OBDDs is to represent boolean functions or more specificly finite boolean domains. Many tasks in digital system design, combinatorial optimization, mathematical logic, and artificial intelligence can be formulated in terms of operations over small, finite domains. For example: in some of the verification algorithms, we are given a set of nodes (the finite domain) and the transition function in a graph and we need to find the set of neighbors of these nodes. We are actually required to perform an operation (find the neighbors) over a finite domain (the set of nodes). The complexity of such operations might be dependent on the size of the representation of the domain. OBDDs can be used as a compact representation of binary domains (domains of binary vectors) and binary functions, and thus save the amount of resources required to perform the operation.

3.1.2 The representation

The OBDD representation put restrictions on the BDD Binary Decision Diagram representation introduced by Lee [18]. BDDs can be used for abstract representation of data. The BDD represent a boolean function $f(x_1, x_2, ..., x_n)$ using a directed acyclic graph with a single root. Each nonterminal vertex v in the graph is labeled by one of the variables of the function, we denote by $var(v) \in \{x_1, x_2, ..., x_n\}$, and has two outgoing arcs $low(v)$ and $high(v)$. Each one of the terminals is assigned to the value 0 or 1. Given a binary function $f(x_1, x_2, ..., x_n)$ represented as BDD, we can compute its value for a given substitution $\langle x_1, x_2, ..., x_n \rangle$ by following the path from the root of the OBDD to one of the terminals. In every vertex v we choose $low(v)$ if the variable $var(v)$ assigned to 0 by the substitution and $high(v)$ if $var(v)$ assigned to 1. The value of f is 1 if we arrive to a terminal with the value 1 and 0 if we arrive at terminal with the value 0. It is important to note that the BDD's variables can be ordered by different order along any of the BDD's branches.

OBDD is defined by putting some restrictions on the BDD representation. First, we impose a total order on the variables of the function. We require that a nonterminal vertex v can be a descendent of a nonterminal vertex u only if $var(v) < var(u)$. We apply on the resulted graph the following three rules:

1. **Remove duplicate terminals.** Remove all the duplicate 1 or 0 terminals at the bottom of the BDD. Merge all these terminals into one terminal. (one 1 terminal and one 0 terminal).

2. **Remove duplicate nonterminals.** If nonterminals v and w have $var(v) = var(w)$, $low(v) = low(w)$ and $high(v) = high(w)$, eliminate one of the vertices and redirect all the edges that point on it to point on the other vertices.

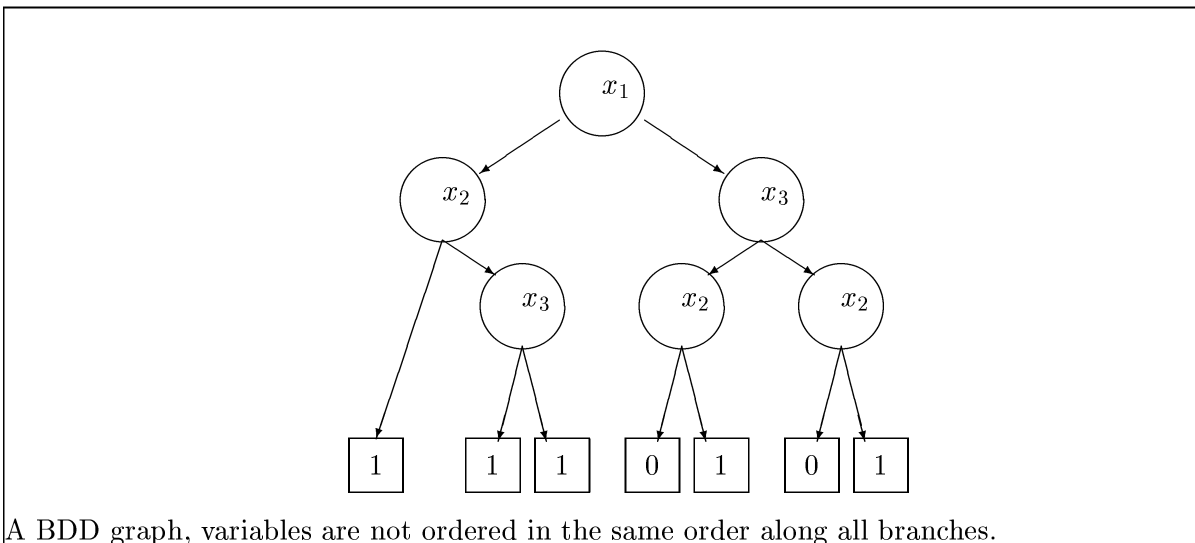
3. **Remove Redundant tests.** If a nonterminal v has $low(v) = high(v)$ then eliminate v and redirect all the edges that point on v to point on $low(v)$.

Starting with a BDD we can apply repeatedly these operations. An OBDD is canonical when neither of these operations can still be applied. As different from the BDD, in each of the branches in the OBDD graph all the variables are ordered according to the same order. However, not all the variables appear in each of the branches.

3.1.3 OBDD - an illustrated example.

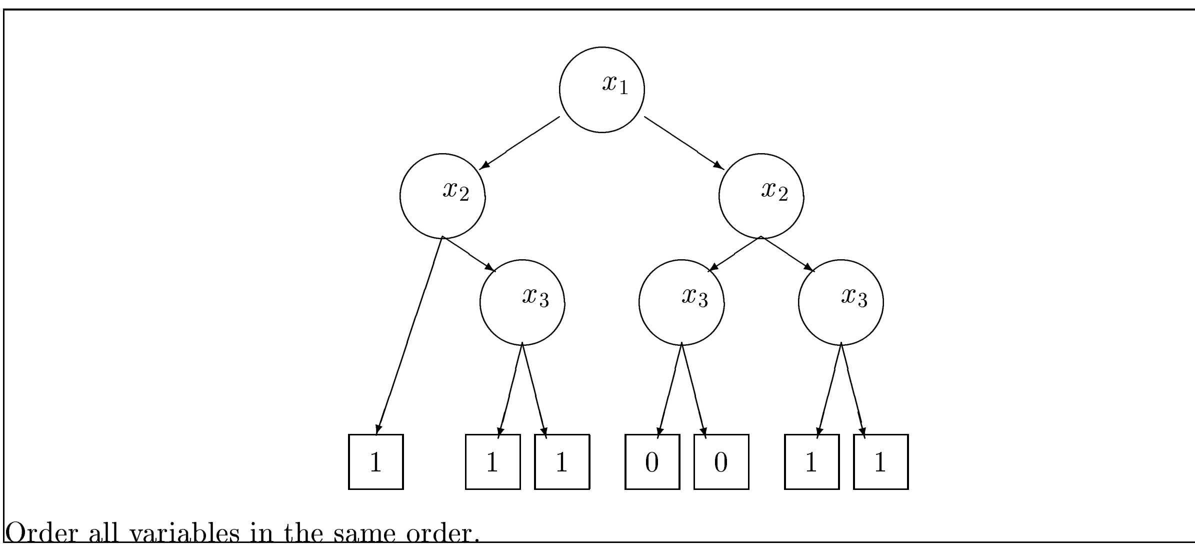
I'll start with an example BDD, then apply each of these rules until I'll get an OBDD:

First we have a BDD. In each branch of the BDD graph the variables can be ordered by different order. In each of the illustrations, the right son represents $high(v)$ and the left son represents $low(v)$.



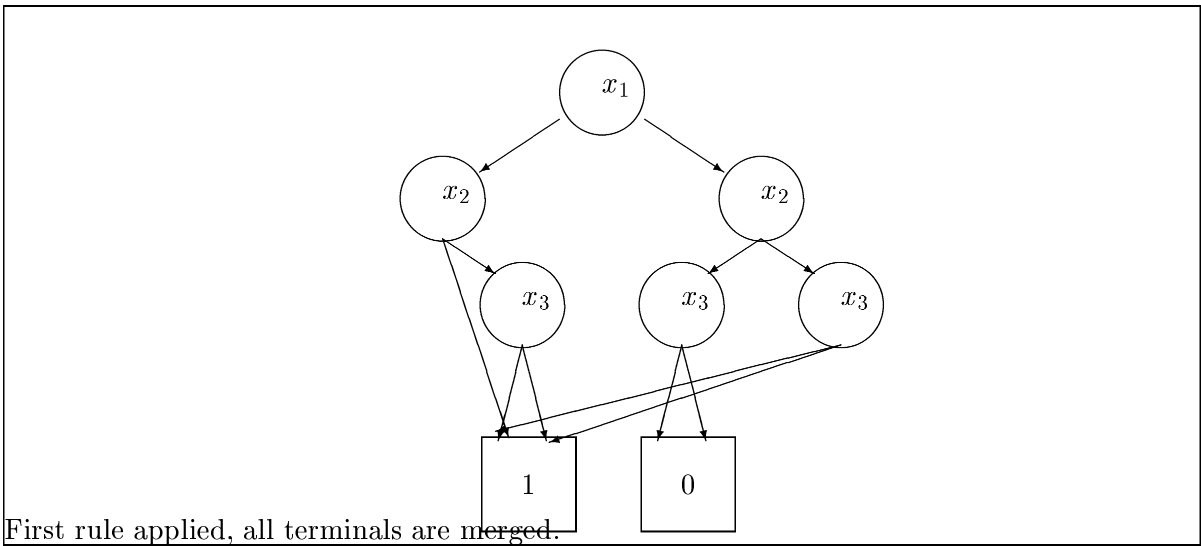
A BDD graph, variables are not ordered in the same order along all branches.

We now order all the variables according to the same order. In the right part of the graph x_2 is now ordered before x_3. The edges from x_3 to the leafs are changed so the function will not changed.



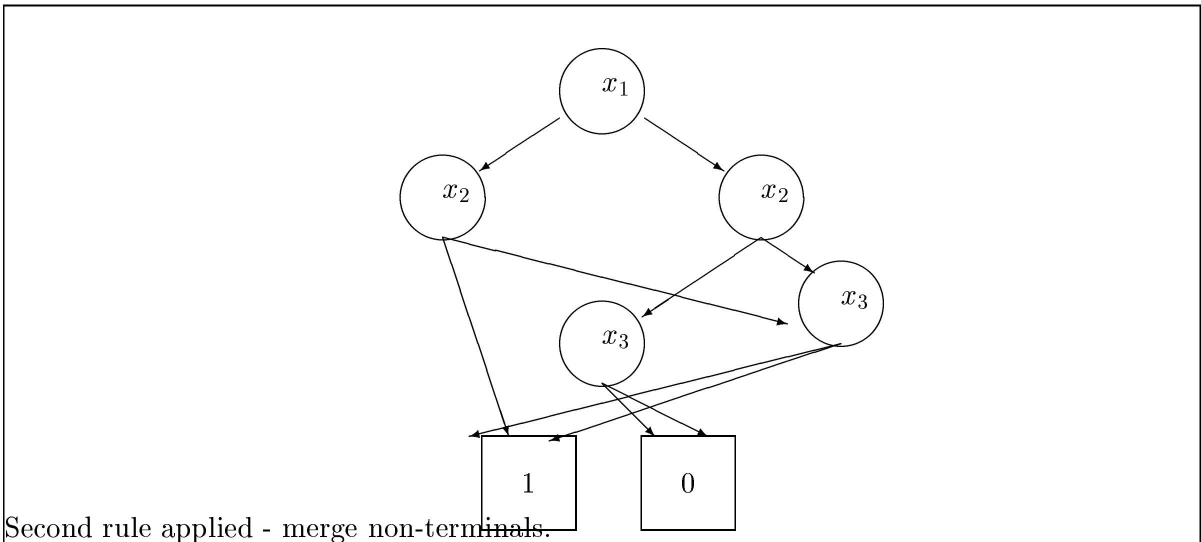
Order all variables in the same order.

Now we apply the first rule on the graph. All the terminals of are now merged into one terminal of the same value.



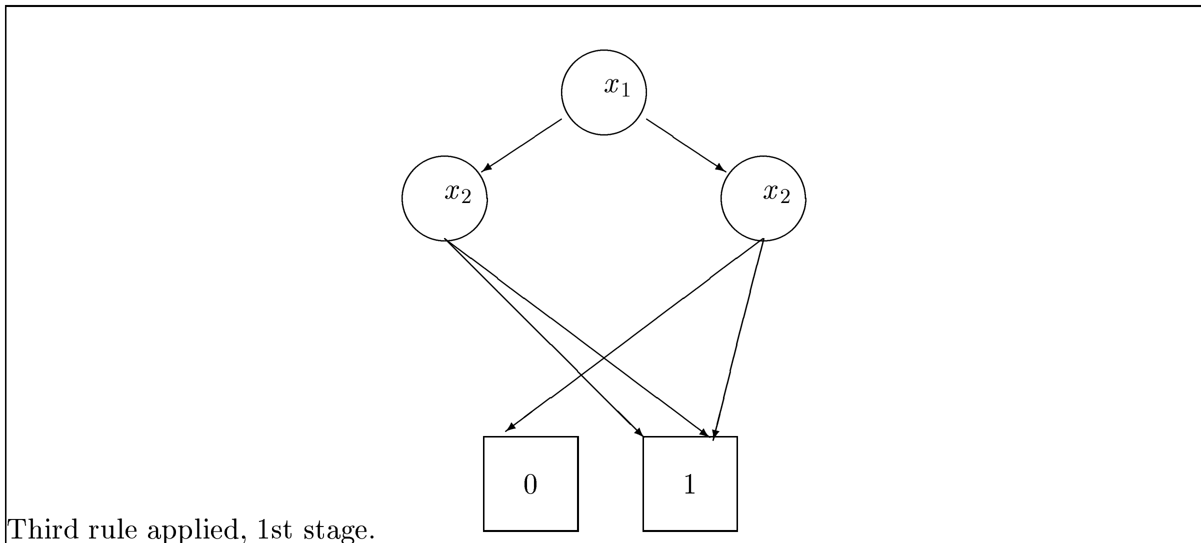
First rule applied, all terminals are merged.

Now the second rule is applied. All the non-terminals are merged. Every pair of non terminals v_1 and v_2, such that $var(v_1) = var(v_2)$ and $low(v_1) = low(v_2)$ and $high(v_1) = high(v_2)$ is merged into one non-terminal. Here we can merge the two non-terminals that holds x_3 (the two in the extremes, not the one in the middle). These two non-terminals contain the same variable ($var(v_1) = var(v_2) = x_3$) and have $high(v_1) = high(v_2) = 1$ and $low(v_1) = low(v_2) = 1$. We merge them both into one non-terminal.



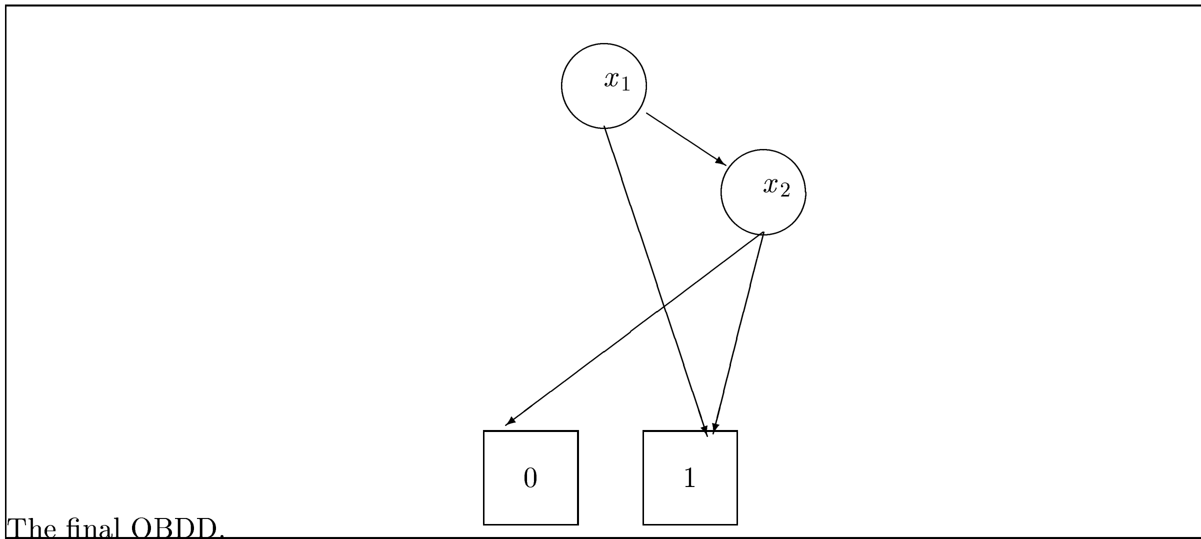
Second rule applied - merge non-terminals.

Now the third rule is applied. In the first stage, the non-terminals that contain x_3 are removed. This is because these non-terminals are redundant tests. They have $low(v) = high(v)$ and therefore they are not necessary.



Third rule applied, 1st stage.

We now apply the third rule second time. This time we omit the left non-terminal that contains x_2. It can be seen that this test is redundant since $low(v) = high(v) = 1$. This completes the operations we should apply on the BDD in order to get an OBDD.



The final OBDD.

3.1.4 Advantages of OBDD representation

The OBDD representation is much smaller than DNF and CNF for many important functions. It is also a canonical representation this means that two boolean functions are equivalent iff they have isomorphic OBDD representation. This feature, make it simple to apply some important operations on boolean functions represented using OBDD. Some of this operations are:

- Checking whether a function f is satisfiable. Due to the canonity, in order to check this we

should only check if the OBDD representing f is different from the OBDD that contains only the 0 terminal.

- Checking whether two functions are equivalent. As was previously noted, in order to check this we should only check if their OBDD representations are isomorphic.
- The APPLY operation - Given two boolean functions f, g and a logic operator $\langle op \rangle$, to compute $f\langle op \rangle g$. This operation, that is very useful in many algorithms that work with boolean function has an efficient implementation when f and g are represented using OBDDs.
- The RESTRICT operation - Given boolean function f and a variable x_i, to compute $f\mid_{x_i \leftarrow 0}$ and $f\mid_{x_i \leftarrow 1}$. This is also useful algorithm when working with boolean functions. As the APPLY operation, it can also be implemented using OBDDs. Short description of the RESTRICT and APPLY algorithms will be given in the next section.

Supporting these four operations efficiently, makes the OBDD representation very useful in different algorithms and mainly in verification algorithms that should work on domains or boolean functions represented using OBDDs. In the next section, a short description of the RESTRICT and APPLY algorithms will be given.

3.1.5 Logic operators on OBDDs - APPLY and RESTRICT

The RESTRICT and APPLY operations, described in the previous section, are two important operations on boolean functions that have an efficient implementation using OBDD.

The RESTRICT operation receives a boolean function f and a variable x_i and return the function $f\mid_{x_i \leftarrow 0}$ or $f\mid_{x_i \leftarrow 1}$ represented using OBDD. In order to restrict the variable x_i to a constant $k \in \{0, 1\}$ we should just redirect all the edges that point to a vertex v with $var(v) = x_i$ to $high(v)$ (if $k = 1$) or $low(v)$ (if $k = 0$). We then receive an OBDD that represents $f\mid_{x_i \leftarrow k}$.

Given boolean functions $f(x)$ and $g(x)$, we might be interested in the function $f(x)\langle op \rangle g(x)$ where $\langle op \rangle$ is $\vee, \wedge$ or other boolean operator. The operation that receives two boolean functions, $f(x)$ and $g(x)$, and a logical operator $\langle op \rangle$ and return the boolean function $f(x)\langle op \rangle g(x)$ is called APPLY. One of the advantages of the OBDD representation is the existence of a polynomial graph algorithm that computes the APPLY operation and returns the result represented by an OBDD graph. Given two OBDDs which represent two boolean functions, $f(x)$ and $g(x)$, using the same variables order and a logical operator $\langle op \rangle$ the APPLY graph algorithm would return as a result the OBDD of $f(x)\langle op \rangle g(x)$ arranged by the same variables order.

The algorithm works recursively and the recursive step is based on Shannon expansion: Given boolean functions f and g on a set of boolean variables, then for any boolean variable x: $f\langle op \rangle g = \neg x \cdot (f\mid_{x \leftarrow 0} \langle op \rangle g\mid_{x \leftarrow 0}) + x \cdot (f\mid_{x \leftarrow 1} \langle op \rangle g\mid_{x \leftarrow 1})$ This expression holds for any of the function variables. We use this expression to compute the APPLY operation as follows: When we are given two OBDDs with the roots v and v' and an operator $\langle op \rangle$ we would go by the following rules in recursion:

1. If v and v' are both terminals, $f\langle op \rangle g' = v\langle op \rangle v'$. The value of $v\langle op \rangle v'$ is calculated according to the truth table of the operator $\langle op \rangle$. This is the condition that stop the recursion.
2. If v or v' are terminals (assume that v' is terminal), $f\langle op \rangle f' \in \{f, \neg f, 0, 1\}$ depends on the value of $\langle op \rangle$ and v'.

3. If v and v' are non-terminals, then both v and v' hold one of the variables x_i and x_j accordingly. Assume that $i < j$, then according to Shannon expansion: $f\langle op\rangle g = (\neg x_i \wedge (f\ |_{x_i \leftarrow 0} \langle op\rangle g)) \vee (x_i \wedge (f\ |_{x_i \leftarrow 1} \langle op\rangle g))$. Thus the problem transformed to $APPLY(f\ |_{x_i \leftarrow 0}, g, op)$ and $APPLY(f\ |_{x_i \leftarrow 1}, g, op)$ such that the index of the root of $f\ |_{x_i \leftarrow 0}$ and $f\ |_{x_i \leftarrow 1}$ is greater than i.

4. If v and v' are both non-terminals, and hold the same variable x_i, then according to Shannon expansion: $= (\neg x_i \wedge (f\ |_{x_i \leftarrow 0} \langle op\rangle g\ |_{x_i \leftarrow 0})) \vee (x_i \wedge (f\ |_{x_i \leftarrow 1} \langle op\rangle g\ |_{x_i \leftarrow 1}))$

3.1.6 OBDD and verification

One of the tasks for which OBDDs have been extensively used, is symbolic checking. Checking of systems with large number of states involves the problem of the space needed for the representation of the finite state machine (FSM) or in the context of verification, the Kripke model. Using OBDDs, one can represent the transition relation, R, of the FSM as a boolean function represented by an OBDD. The set of the final states F, as well as the set of initial states W_0, can also be represented using their characteristic boolean function represented by OBDD. Since the main problem of verification algorithms are the states explosion problem and the large amount of memory space required for the representation of the Kripke model, the introduction of OBDDs and their usage in symbolic checking made the verification of many systems feasible.

3.1.7 OBDD size and the variables ordering problem

The size of the OBDDs is defined as the number of vertex in the OBDD and is dependent in the ordering of the variables along the levels of the OBDD. Consider for example (taken from Bryant [6]), the function $a_1b_1 + a_2b_2 + ... + a_nb_n$. Organizing the variables in the order $a_1 < b_1 < a_2 < b_2 < ... < a_n < b_n$ yields an OBDD with $2n$ nonterminal vertex - one for each variable. Ordering the variables in the order : $a_1 < a_2 < a_3 < ... < a_n < b_1 < b_2 < ... < b_n$ yields an OBDD with $2(2^n - 1)$ nonterminal vertex. For large values of n, the difference between the linear growth of the first ordering versus the exponential ordering of the 2nd ordering is crucial. The function $a_1b_1 + a_2b_2 + ... + a_nb_n$ is obviously sensitive to the ordering of the variables in the OBDD. Its size given an order on the variables ranges between linear and exponential growth. Different boolean functions have different sensitivity to the order of the variables inside the OBDD.

Does a good ordering of the OBDD's variables always matter? Can it always reduce significantly the size of the OBDD representation? Wegener [28] has shown that for almost all functions the sensitivity to the ordering, formally, the relation between the minimal OBDD size for an optimal variable ordering and the minimal OBDD size for the worst variable ordering is bounded by $1+\epsilon(n)$ where $\epsilon(n)$ converges exponentially fast to 0. In the paper, the two reduction rules for reducing a binary tree in order to get an OBDD are checked separately :

1. **deletion rule.** If two edges leaving some node v lead to the same node w, the node v can be deleted, i.e. all edges leading to v may lead directly to w.

2. **merging rule.** If two nodes v and w have the same label, the same 0-successor and the same 1-successor, v and w can be merged.

First, it is shown that the deletion rule has a restricted influence on the final size and that actually the merging rule is the crucial one. The quotient between the size of the OBDD when not using the deletion rule (using only the merging rule) to the size of the OBDD when using the

deletion rule was calculated. Let us refer by quasi-reduced OBDD to the to an ordered BDD (BDD with the variables ordered by the same order along all branches) for which only the merging rule was applied. The reduced OBDD is the usual OBDD (two rules applied). The following theorem was proven for the deletion rule:

Theorem 1 *Let $Q^*(f)$ be the maximal quotient of the size of the quasi-reduced OBDD and the size of the reduced OBDD for f with the same variable ordering where the maximum is taken over all variable orderings. Then $Q^*(f) \leq 1 + O(2^{-n/3}n)$ for all but a fraction of $O(2^{-n/3+\delta n})$ of all Boolean function where δ is an arbitrary positive constant.*

In order to understand the intuition behind this theorem we look at the quasi-reduced OBDD (OBDD before the deletion rule). For each level in the graph, we will let X_i be the number of nodes that are deleted by the deletion rule and we then define $E(X_i)$ to be the mean of X_i. On the upper part of the quasi-reduced OBDD, $level \leq n - \log n$, $E(X_i) < n^{-1}$. In order for a node v to be deleted by the deletion rule, it is required that its two sons $low(v)$ and $high(v)$ would represent the same function. The probability for that is very small at the upper size of the quasi-reduced OBDD since each of the subfunctions represented by $high(v)$ and $low(v)$ has many arguments and therefore the probability that they agree on the value for each substitution is small. On the middle part of the OBDD, $n - \log n < level \leq n - \log n + \log 3$, and therefore $E(X_i) \leq 2^{2n/3+1}n^{-1}$. This number is very small in relation to the number of nodes in the quasi-reduced OBDD in these levels. In the lower part of the OBDD, $level > n - \log n + \log 3$, the number of nodes in the quasi-reduced OBDD is $N_i \leq 2^{2n/3}$. Since the number of nodes in the quasi-reduced OBDD is small, there is almost no effect for the deletion rule in these levels. The reason for the low number of nodes in this level is that the probability of two functions to be the same on these levels is much higher (since these boolean functions are having only $\log n - \log 3$ variables).

Then, it is shown that only for a small fraction of the set of all boolean functions there is importance to the order of the OBDD variables. This is done using three theorems:

Theorem 2 *Let the variable ordering $x_1, ..., x_n$ be fixed. The expected size of the quasi-reduced OBDD for a random Boolean function is equal to :*

$$S(n) := \sum_{0 \leq i \leq n-1} 2^{2^{n-i}} (1 - (1 - 2^{-2^{n-i}})^{2^i})$$

Theorem 3 *The fraction of Boolean functions whose reduced OBDD size with respect to the variable ordering $x_1, ..., x_n$ differs more than $O(2^{2n/3})$ from $S(n)$ is bounded by $O(2^{-n/3})$. The weak Shannon effect holds for Boolean functions and reduced OBDDs with fixed ordering of the variables.*

Theorem 4 *The fraction of Boolean functions whose optimal OBDD size differs more than $O(n2^{2n/3})$ from $S(n)$ is bounded by $O(2^{-n/3+\delta n})$ for arbitrary constants $\delta > 0$. The weak Shannon effect holds for Boolean functions and OBDDs. The fraction of Boolean functions whose sensitivity is larger than $1 + O(n^2 2^{-n/3})$ is bounded by $O(2^{-n/3+\delta n})$ for $\delta > 0$.*

Theorem 2 claims that $S(n)$ is the expected of the quasi-reduced OBDD for a *fixed variables order*. Since it was previously proved that the merging rule is strong enough, it is enough to show that this for the quasi-reduced OBDD. Theorem 3 claims that for almost all functions and for all variables orderings the size of the OBDD is close to $S(n)$. This shows again that the merging rule is strong enough. Theorem 4 shows that for almost all functions the size of the optimal OBDD (with the best variable ordering) is close to $S(n)$. The conclusion from these 3 theorems, is that

for almost all functions the size of the optimal OBDD is very close to the size of the OBDD with any other variable ordering.

One of the functions that has been specificly shown to be insensitive to the ordering of the variables in the OBDD, is the multiplication function. It has been shown by Bryant [5], that for any ordering of the variables the OBDD of that function has an exponential in the number of variables. The MULT function is defined as the boolean function that accepts a pair of n-bit integers and computes the middle bit in their product. In a paper by S.Ponzio [22] it was proven that any read-once branching program that computes MULT has size at least $2^{\sqrt[3]{n}/4}$. Read once branching program is a less restrictive model than the OBDD since it does not require all the variables to appear in the same order along all the branches. Since this arrangement of the OBDD variables the size of the OBDD that represents the bits of the integers multiplication operation is exponential in the number of variables.

Wegener indeed showed that for almost all boolean functions the size of the OBDD representation is exponential in the number of variables, no matter which variable ordering is used. However, it is known that for many important boolean functions such as the integer addition, parity ([5]) and many more the size of the OBDD representation is dependent in the order of variables. This subset of the set of all boolean functions, although very small, contain many important and interesting functions.

In this work we won't focus in the question of how useful are OBDDs for the representation of functions. The extensive usage of OBDDS, especially in the field of verification and symbolic checking, presents strong evidence that they are. Historically, OBDDs made many of the verification schemes feasible due to the compact representation that they achieved for the Kripke model. Given a boolean function f, we assume that we can reduce the OBDD representation of f if we use appropriate variable ordering. We don't try to find the optimal variable ordering, just an approximated ordering that will yield small OBDD representation.

Even if we knew on a specific boolean function that it can be represented by an compact OBDD representation using an optimal variables order, finding such optimal ordering is infeasible. According to a paper by Beate Bollig and Ingo Wegener [3], given an OBDD G representing a boolean function f and a size bound s, answering whether there exist an OBDD G^* (respecting an arbitrary variable ordering) representing f with at most s nodes is a problem in NP-complete. In the paper, a polynomial time reduction is presented from the NP-complete problem Optimal Linear Arrangement (that is in NP-complete by [10]).

We assume that $RP \neq NP$, and therefore that a language that is in NP-complete cannot be identified using a randomized polynomial algorithm.

Another result that encourages us to look for a heuristic for the good ordering is presented in a paper by Detlef Sieling, The Nonapproximabilty of OBDD Minimization [24] (also [25]). In this paper it is shown that for each constant $c > 1$ there is no polynomial time approximation algorithm with performance ratio c for the variable ordering problem unless $P = NP$. As the author claims, this result justifies, also from the theoretical point of view, to use heuristics for the variable ordering problem.

We therefore don't look for the optimal variables ordering but, as was previously stated, to a heuristic to find a good ordering. We base our algorithm on clustering of the function variables. The next section will provide description clustering in general and the specific clustering method that we use.

3.1.8 Parity OBDDs

In this section we discuss briefly a particular type of OBDD introduced by Gergov and Meinel [12], parity OBDDs.

Parity ordered decision diagrams (OBDDs) were introduced by Gergov and Meinel [12] and further developed by Waack [27]. Following is a brief description of parity OBDDs taken from [23]: The structure of a parity OBDD depends on the ordering of the variables represented by a permutation π of $\{1, 2, ..., n\}$. For a given permutation π, a parity π-OBDD over variables $x_1, x_2, ..., x_n$ means directed acyclic graph with at most one source and at most one sink satisfying the following: Every nonsink node is labeled by a variable x_i for $i \in \{1, 2, ..., n\}$ and every edge is labeled by 0 or 1 or both. Moreover, it is required that if an edge leads from a node labeled by x_i to a node labeled by x_j, then $\pi(i) < \pi(j)$. Let an assignment $a = (a_1, a_2, ..., a_n)$ of the variables be given. An edge starting in a node labeled by x_i is called consistent with the assignment, if the set of its labels contains a_i. A path from the source to the sink is called consistent with the assignment,if all its edges are consistent with it. The assignment is accepted, if the number of paths from the source to the sink consistent with the assignment is odd. In particular, if the graph is empty then no assignment is accepted and if the source coincides with the sink the all assignments are accepted. There is a simple algorithm that decides whether an assignment is accepted or not, which works in time linear in the number of edges of the graph. It should be pointed out that the trivial algorithm to decide whether an assignment is accepted by a regular OBDD works in time linear in the number of variables. The number of edges in parity OBDD might be exponential in the number of variables which makes this operation much simpler in OBDD.

For parity OBDDs a strong result proved by Savicky [23] shows that a random ordering almost not better than the worst variable ordering. Formally: for every $\epsilon > 0$ there is a number $c > 0$ such that the following holds. If a Boolean function f of n variables is such that a random ordering of the variables yields a parity OBDD for f of size at most s with probability at least ϵ, where $s \geq n$, then every ordering of the variables yields a parity OBDD for f of size at most s^c.

OBDDs are used in applications as a data structures for representing the Boolean functions, since there are efficient algorithms for several required operations with the functions, if they are represented by OBDDs [4]. These algorithms are the RESTRICT and APPLY introduced before, as was noted before the verification algorithms extensively use these algorithms. Analogous algorithms exist also for the parity OBDDs, however some of them include Gaussian elimination and hence, these are less efficient than corresponding algorithms for BODDs see A paper by M. Lobbing, D. Sieling and I. Wegener, Parity OBDDs cannot be handled efficiently enough [19].

For the following reason, we focused in the OBDDs rather in parity OBDDs. However, the same intuition and heuristic that is presented in this work might also work for parity OBDDs. This is because that the reduction rules applied on each of these graphs are very similar.

3.2 The Clustering of Variables

3.2.1 Introduction and intuition

Given a boolean function f, we would like to find an heuristic that would help us to order f variables such that the resulting OBDD would be of a relatively small size. For any such function f we can ask two questions regarding the ordering of the OBDD. The first question is whether any ordering of f's variables might have a positive effect on the OBDD size? As we discussed before, for most of the boolean functions most of the orderings will not have a dramatic effect over the OBDD size (Wegener, [28]). The second question is assuming that a certain ordering will have a positive effect

over the OBDD size, what is that order? It should be stressed again that since the problem of finding the optimal order is in NP-complete (Wegener and Bollig, [3]), we will be satisfied with an heuristic for a good order of the variables.

In order to answer the first question, we will look for a criteria that observes whether the function f is sensitive (in its OBDD representation size) to different orderings of the variables. Such a criteria might be the presence of many short minterms in the 1 domain of f and many short maxterms in the 0 domain of f. Such short maxterms and minterms can yield a short path of variables from the root of the OBDD to the 0 or 1 terminals. In order to observe if f has many minterms in its 1 domain or many maxterms in its 0 domain, we will make a check over any pair of f's variables. We will reach a decision as to whether this pair included in a short minterm or short maxterm by sampling. The check is performed as follows: for any possible substitution to the pair of variables, $\{\langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$, we will check whether f is sensitive to it or to what degree the value of f influenced by the substitution to the variables pair. We will check this by sampling the 0 or 1 domains of f and checking whether any of these 4 substitutions is more frequent than the others in the domain. We are actually not interested in the specific substitution that is most frequent but only in whether such a substitution exists. Therefore, we can just compare the sum frequencies of any 2 substitutions to the sum of the other two and check whether these sums are almost equal or that one tuple of substitutions is more frequent in the domain than the other tuple. This will be done by splitting the 4 possible substitutions into 3 possible splits: $\{\{\langle 0,0\rangle, \langle 0,1\rangle\}, \{\langle 1,0\rangle, \langle 1,1\rangle\}\}, \{\{\langle 0,0\rangle, \langle 1,0\rangle\}, \{\langle 0,1\rangle, \langle 1,1\rangle\}\}\{\{\langle 0,0\rangle, \langle 1,1\rangle\}, \{\langle 1,0\rangle, \langle 0,1\rangle\}\}$ and then checking whether the sum of frequencies of 2 substitutions in any of these splits is considerably larger than the sum of frequencies of the other two in that split. If we find a split in which such a considerate difference in the frequencies of the tuples exists, we will conclude that f is quite sensitive to the pair of variables examined. In order to compare f sensitivity to any of the possible pairs of variables, we assign to any pair of f's variables the largest difference in the frequencies over any of the 3 possible splits. We will call this measure the *pairwise-sensitivity* of f to this pair of variables. This can be seen as a natural extension to the traditional concept of sensitivity of a boolean function to any of its variables. After measuring the pairwise-sensitivity to all the possible pairs of variables, we can guess which of the possibles pairs is included in short minterms or short maxterms. We will guess that a pair of variables for which f is more sensitive to, is probably in short minterm or short maxterm and therefore has a considerate impact on f value. We are now motivated to put the pairs of variables for which f is most sensitive close in the OBDD ordering. The problem with producing an ordering that is based on this sensitivity measure alone is that we have an heuristic only regarding pairs of variables while we would like to know on the relation on larger sets of variables (which triples of variables should be placed together? which quadruples and 5th should be placed together? and so on..). Extending the above check to larger sets of variables $(3, 4, ...n)$ will involve exponential time complexity since we will have to go over all the possible subsets of f variables and measure the frequency of all the substitutions of values into them. Therefore, in order to produce information on how to order larger sets of variables, we will use clustering. We will choose a clustering algorithm that uses the transitive closure of the proximity measure (we mean by this that we are looking for a clustering algorithm that given two tuples of points that are close (x, y) and (y, z) will produce clustering in which x and z will be also close). The clustering process will save us the need to make the sensitivity check for larger sets of variables. Later, in order to improve our method, we will only extend the sensitivity check to triples of variables, but for larger subsets of variables we will use the clustering algorithm. For the clustering algorithm we choose the Randomized Algorithm for Pairwise clustering by Yoram Gdalyahu, Daphna Weinshall and Michael Werman [11]. We will therefore apply the clustering

algorithm on the pairwise-sensitivity measure that we get for f, and then use the clusters as an heuristic as to which variables should be placed close together in the ordering. The algorithm of Gdalyahu, Weinshall and Werman such as an algorithm by Blatt, Wiseman and Domany [2] works on pairwise data rather than vectorial data and therefore suitable for our needs.

We give here more intuition to our method. If f is highly sensitive to variables x and y, it is obvious that we should place them together. If we place them far along the OBDD ordering, we will have to check many variables (those that are ordered between them) that in most cases will have less effect on f value (given a certain substitution to x and y). Another way to look on it is that if x and y are placed far from each other, the OBDD will have to "remember" the value of x until reaching to y, and thus increasing its size.

A function f for which the pairwise sensitivity of all pairs is equal, or almost equal is expected to be insensitive (in its OBDD representation size) to different orderings of the variables. Thus, we make answers to the two questions presented before. IN order to check the sensitivity of f OBDD representation size to different ordering, we'll check if the pairwise-sensitivity of f variables is similar. In order to provide heuristics for good ordering will provide the clusters resulting from applying the clustering algorithm on f's pairwise-sensitivity matrix of all the variables tuples. We will produce ordering in which variables from similar clusters will be placed closely.

It should be noticed, that the method described here is useful also in the cases when all the minterms/maxterms are short. In such cases, even small differences in the length of the minterms/maxterms will be expressed by different sensitivity measures for the variables in these minterms/maxterms. This is because that any difference in the length of short minterms/maxterms has more influence over the sensitivity.

It is important to note here that we didn't deal with the questions as to what should be the order among the clusters. We think that a possible heuristic might be to put first the clusters with the higher average sensitivity. These clusters has the greatest impact over f's value.

3.2.2 Clustering

Clustering is the process of classifying objects into subsets that have meaning in the context of particular problem. In our context, we cluster variables into groups of variables that are correlated or proximated in order to produce an arrangement of the OBDD variables that will lead to compact representation of a given boolean function. Clustering is a form of unsupervised learning. Unlike supervised learning, where we are given an instructor, an oracle, that tags examples and checks our hypothesis, in unsupservised learning we are only given rare data and we should analyze them without the help of an oracle. Some description of the clustering problem can be found in Algorithms for Clustering Data, by Anil K. Jain and Richard C. Dubas, [15]. They define clustering as a type of classification imposed on a finite set of objects. The relationship between objects is represented in a proximity matrix in which rows and columns correspond to objects. This matrix, in our case, is the correlations matrix of f variables. Unless a meaningful measure of distance, or proximity, between pairs of objects has been established, no meaningful cluster analysis is possible. The proximity matrix is the one and only input to a clustering algorithm.

3.2.3 Pairwise Clustering

Clustering algorithms can be categorized into two main categories:

- Clustering algorithms that works on a set of vectors. Each vector represents one of the objects in the set.

- Clustering algorithms that get as input a matrix that represents the distance or proximity between any pair of objects.

The second category of clustering algorithms is called pairwise clustering algorithms. Since the information that we have on the set of f's variables is the correlation between any pair of variables, the category that we should consider to cluster the variables is pairwise clustering algorithms category. There are several familiar strategies for pairwise clustering:

- **Using threshold**
  To use this strategy, one should choose a threshold ϕ and define a graph $G = \langle V, E \rangle$ based on the proximities matrix. The graph is defined as follows: $V =$ the set of objects and $E = \{(x_1, x_2) | M(x_1, x_2) \geq \phi\}$. This means that every points for which the proximity(distance) M is greater(smaller) than the threshold is joined together with an edge. The clusters defined using this strategy are the connected components in the resulted graph.

- **Bind the closest objects together**
  When using this strategy, we start with N clusters for the N points. Then we start to join the the closest clusters into one. We should then define the proximity between the resulted cluster and the rest of the clusters. We continue this way until the proximity between the closest clusters is lower than some threshold. The algorithms that use this strategy are differ in the way that they define the proximity between clusters. Some take the minimal/maximal proximity between objects in each of the clusters.

- **Minimal cut**
  Given a non-directed graph $G = \langle V, E \rangle$ and a capacity function $c : E \to \Re$ a cut in G is defined as the partition of V into two disjoint subsets $A, B \subseteq V, A \cap B = \emptyset$. The capacity of a cut is defined as $c(A, B) = \sum_{e \in (A,B) \cap E} c(e)$. The minimal cut in a graph is the graph with the minimal capacity. The minimal cut clustering algorithm use the notion of minimal cut for clustering.

The Randomized Algorithm for Pairwise Clustering use a combination of these strategies to choose randomally the minimal r-cut from all the r-cuts in the set to cluster. The algorithm is presented in the next section.

3.2.4 A Randomized Algorithm for Pairwise Clustering

The Randomized Algorithm for Pairwise Clustering was designed by Yoram Gdalyahu, Daphna Weinshall and Michael Werman [11]. The algorithm uses combination of all the above strategies. It can be seen as a probabilistic generalization of the minimal cut strategy. It generalize the minimal cuts strategy in two manners. It chooses the minimal cut in a probabilistic manner using a distribution that is assigned to each cut. The probability is reversely proportional to the cut's capacity. This way cuts with small capacities are more probable to be chosen. Another generalization is that also r-cuts, cuts with more that two sides, are considered. In the first stage of the algorithm we assign to each pair of objects, x_i, x_j a probability of the two objects to be in the same r-cut. This probability is denoted by $p^r_{i,j}$ and is computed as a function of the proximity $M(x_i, x_j)$ between the objects. The probability is defined for any r between 1 and N. The probability assigned to $p^r_{i,j}$ by the algorithm is found to decay fast enough with the capacity. This means that $p^r_{i,j}$ is dominated by the minimal cuts. The probability $p^r_{i,j}$ of two objects x_i and x_j to be in the same r-cut is computed using the contraction algorithm as follows:

- Select edge (i,j) with probability proportional to $w_{i,j}$.
- Replace nodes i and j by a single node $\{ij\}$.
- Let the set of edges incident on $\{ij\}$ be the union of the sets of edges incident on i and j, but remove self loops formed by edges originally connecting i to j.

We estimate $p^r_{i,j}$ by repeating the contraction algorithm M times. In each iteration there exists a single r at which the edge between points $i-j$ is added and the points are merged. We denote by r_m the level r which joins i and j at the m-th iteration ($m = 1..M$). The median r' of the sequence $\{r_1, r_2, ..., r_M\}$ is the r for which : $p^{r'}_{i,j} = 0.5$. After computing for any i,j the r for which $p^r_{i,j} = 0.5$ we can find the clusters for any r. Given r, the clusters for that r are the connected components that are established using the edges for which there exists $r' \leq r$ such that $p^{r'}_{i,j} = 0.5$. As it is shown, for any r chosen we get another clustering. For lower r values the clustering accepted is more delicate while for higher r values there are less clusters. As we can see, the clustering that are accepted are hierarchical. These means that if two nodes are in the same cluster for a certain r, the will be at the same cluster for any $r' \geq r$.

3.3 Finding good order for the variables of OBDDs by clustering

3.3.1 Introduction

In the discussion at the introduction we saw that given a boolean function, the order of the variables might have impact on the size of the OBDD. It was shown that in some cases, for some function classes (for example: any bit of the integer addition), changing the order of the variables can reduce the OBDD size from exponential to linear size (in the number of variables). It has been shown that the problem of finding good order for the OBDD variables is NP-complete [3]. In this work, a different strategy is taken. Instead of looking for a good *ordering* of the function variables, we are looking for a good *clustering* of the variables. We try to look for a subsets of variables that should be placed closely by the variables arrangement in order to impose compact OBDD representation. We are not considering the exact order of the variables, just clustering the variables into groups that should be placed close by such order. The distance between pair of variables is defined as follows. Given an arrangement of variables $\langle x_1, x_2, ..., x_n \rangle$, the distance between the variables x_i and x_j is defined by $|i-j|$.

The main intuition behind this strategy is that since we are calculating a boolean function f represented using the OBDD, we want that any pair of variables that are very dependent when calculating f to be close, while we agree that variables that are independent when calculating f will be far. We find the measure of dependency between f's variables by implicitly testing f for various inputs. We treat f as a black box, we are not interested in the way f is implemented just in the values of f for various inputs. From the learning theory perspective, our main difficulty with ordering the variables is that even *checking* the OBDD size for a given ordering is hard and require that we build the complete representation of f. We can look at the function $F : S_n \rightarrow Z$ that tells us for any permutation of f variables π what is the size of the OBDD. Our target is to find π that minimize $F(\pi)$. Unlike some other maximizations/minimizations problems that are approximated using computational learning methods and are based on sampling values of the target functions for some inputs, here even the calculation of F for a given permutation π is hard (if we might given a bad permutation π that yields exponential representation even if f has a compact representation with the right ordering). We therefore should use our ability to sample the values of f, the original boolean function that is easy to calculate and therefore sampling it is in-expensive.

In the previous section (on clustering of variables), we described our basic method of measuring the tuples of variables to which f is most sensitive. We can see such variables as variables that are dependent in each other, since that given a value of f (1 or 0), one variable determine the value of the other variable (since as a tuple they have strong effect on the value of f). Therefore, another way to look on our method is that we place dependent variables together in the OBDD order.

Some justification for the connection between the dependency between variables and the size of the OBDD representation we can find in papers by Berman [1] and McMillan [20]. In a paper where they tried to find upper bounds for classes of boolean functions based on the structural properties of their logic network realizations. Given a network consisting of m logic blocks: each block may have inputs, outputs and wires to other blocks. We define linear ordering of the blocks from 1 to m such that the block that produces the main output of the network is placed at the end. We define two measures: a *forward cross section* at a block i is the number of output wires of all blocks b_j with $j < i$ that are inputs of some block b_k with $i \leq k$. Define the *forward cross section* w_f of the network as the maximum forward cross section for blocks $1 \leq i \leq m$. Define the *reverse cross section* at block i as the number of output wires from block b_j, with $i < j$ to input of block b_k with $k \leq i$. Finally, let the *reverse cross section* w_r of the network be the maximum reverse cross section for blocks $1 \leq i \leq m$. Using these measures it was shown in [20] that there is an OBDD representation of the function with $n2^{w_f^{w_r}}$. It was also shown that finding an arrangement with low cross section yields good ordering of the function variables. Although this finding only tells us on the upper bound of the OBDD representation given w_f and w_r, it still promise us that as smaller these measures are, as smaller will be the size of a certain OBDD representation (maybe not the smallest) for the given function. Our target is therefore, to find a network of f that minimize w_f and w_r. Finding such network given a boolean function seems quite hard task, and therefore we'll try to achieve more moderate target: only to find subsets of variables that would probably be arranged together in such a network. Since w_f and w_r represents the flow of data along the computation network, it seems reasonable that given two blocks, the more variables in each of these blocks are independent the less input and output wires that we should set between these blocks and thus the smaller degrees of w_f and w_r that we get.

We can therefore see our task to arrange f's variables in a blocks of such computation network in a way that will bring to a minimum the degree of dependency between the blocks. Such arrangement will potentially bring to a minimum the degrees of w_f and w_r and thus will lead to a small upper bound on the size of the OBDD representation. It is important to note here some limitations of these strategy:

- As it can be seen by the upper bound, it is exponentially more affected by w_r then by w_f. While our strategy treat these two measures symmetrically.
- As it was noted before, our strategy based on a theorem that promise a certain upper bound while we have no evidence that this upper bound is tight.
- It is not clear what is the connection between the statistical dependencies between f's variables and the measures of w_r and w_f in a network that keeps the dependent variables in the same clusters.

In the next section we'll present some examples of simple boolean functions for which we'll show how the size of the OBDD representation can be reduced when variables pairs to which f is sensitive are clustered together.

3.3.2 Examples of functions for which the OBDD size can be reduced by change of variables clustering

We present here three examples of functions for which the OBDD size can be reduced by ordering differently the function variables and the good order can be found by clustering the function variables. For each function, we'll measure the sensitivity of f to any pair of variables, and we'll show that when we cluster the sensitive variables together the size of the OBDD representation is minimized.

1. **Example 1, The equivalence function** Consider the equivalence boolean function $f(x_1, x_2, ..., x_n)$ defined on X^n as follows: For $a, b \in Z^{n/2}$

$$f(a,b) = \begin{cases} 0 & a \neq b \\ 1 & a = b \end{cases}$$

The size of the OBDD graph for $f(x)$ is strongly dependent in the variable ordering. For the "natural" variable ordering: $\langle a_1, a_2, a_3, ..., a_{n/2}, b_1, b_2, b_3, ..., b_{n/2} \rangle$, the size of the OBDD graph is exponential in n. We prove this by observing that in the $n/2$ level of the graph there are at least $2^{n/2}$ nodes. Assuming, by way of contradiction, that there are less than $2^{n/2}$ nodes. Then there are two different inputs $x_1, x_2 \in Z^{n/2}, x_1 \neq x_2$ that "travel" from the root to the same node at the $n/2$ level. Therefore, $f(x_1, x) = f(x_2, x), \forall x \in Z^{n/2}$ Then, $1 = f(x_1, x_1) = f(x_2, x_1)$. In contradiction for the assumption that $x_1 \neq x_2$. Intuitively, the reason for the exponential size of the OBDD is that f has to "remember" the values of each of the first $n/2$ variables in order to compare them with the last n/2 variables.

We'll try to cluster $f(x)$ variables and to group the pairs of variables to which f is sensitive together. We'll do that by checking the sensitivity of f to any pair of variables $x_i, x_j, 1 \leq i, j \leq n$ in the domain $f^{-1}(1)$. [4] The pairwise-sensitivity will be calculated as follows: for every pair of variables x_i and x_j the sensitivity, $C(i,j)$, will be calculated by:

$$C(i,j) = \left| \frac{\sum_{x \in f^{-1}(1)} x_i \overline{\oplus} x_j - \sum_{x \in f^{-1}(1)} x_i \oplus x_j}{\mid f^{-1}(1) \mid} \right|$$

It is very important to note, that here we checked the sensitivity in relation to the split : $\{\{\langle 0,0 \rangle, \langle 1,1 \rangle\}, \{\langle 0,1 \rangle, \langle 1,0 \rangle\}\}$. In the general method, we measure the sensitivity using all the 3 possible splists and chose the split that produce the highest sensitivity.

The domain $f^{-1}(1)$ contain all the vectors $x \in Z^n, x = \langle w, w \rangle, w \in Z^{n/2}$. It should be clear that for any pair of variables

For $i = j, 1 \leq i \leq m$, $x_i = x_j$ for every $x \epsilon f^{-1}(1)$.. For $i \neq j, 1 \leq i \leq m$, for half of the inputs $x_i = x_j$ and half $x_i \neq x_j$ The sensitivity is simple here and is 1 for any $i = j$ and 0 for whenever $i \neq j$. Which variables order would be consistent with the closeness of the variables by their sensitivity? Any ordering that places a_i near b_i for any $1 \leq i \leq m$. We can observe that the size of the OBDD graph by any of these ordering is n. We can also observe that the order of the OBDD variables does not affect here as the closeness of a_i and b_i (the size is the same as long as a_i and b_i are consequent. In the next example we observe a case where there is positive sensitivity of f to any pair of variables.

[4] We sometimes treat f as a n variables boolean function $f(x_1, ..., x_n)$ as f really is and sometimes denote $f(x,y)$ where $x, y \in Z^{n/2}$ for convenience reasons.

2. **Example 2**

   I'll show here another example of a function where a good ordering of the function variables can be found by clustering of the function variables into clusters of sensitive variables.

   Now we look at the boolean function $f(a_1, ..., a_n, b_1, ..., b_n) = a_1b_1 + a_2b_2 + ... + a_nb_n$. The function f was introduced by Bryant in [6] as an example of a function of which the OBDD representation size is strongly dependent in the variables ordering. For one variables ordering (namely, $a_1 < b_1 < a_2 < b_2 < ... < a_n < b_n$) the size of the OBDD representation is $2n$ while for another variables ordering $(a_1, a_2, ..., a_n, b_1, b_2, ..., b_n)$ the size of the OBDD representation is $2(2^n - 1)$. I'll show here how the good ordering can be found by clustering variables to which f is pairwise-sensitive together. We will measure the sensitivity of any pair of variables in the domain of substitutions that satisfy f, $S = f^{-1}(1) \subseteq Z^n$, We define $C(i,j)$ to be the pairwise-sensitivity of any pair of variables i and j:

$$C(i,j) = |\frac{\sum_{x \epsilon S} x_i \overline{\oplus} x_j - \sum_{x \epsilon S} x_i \oplus x_j}{| S |}|$$

   Again, the sensitivity here is measured according to the split of substitutions: $\{\{\langle 0,0\rangle, \langle 1,1\rangle\}, \{\langle 0,1\rangle, \langle 1,0\rangle\}\}$.

   We now calculate the value of $C(i,j)$ for this function:

   We first calculate the value of $C(i,j)$ when i and j are in the same x_ix_j term. We will denote such x_i and x_j by x_k and $x_{n/2+k}$ for $1 \leq k \leq 2/n$ We actually have to calculate for how many x vectors that satisfy $f(x) = 1$ x_k and $x_{n/2+k}$ agree, and for how many vectors they disagree. We define X_l as the number of possible substitutions of size l that satisfy f_l (f of size l). For x_k and $x_{n/2+k}$, $1 \leq k \leq n/2$:

$$\sum_{x \in S} x_k \overline{\oplus} x_{n/2+k} = |\{x \in S | x_k = 1 \wedge x_{n/2+k} = 1\}| + |\{x \in S | x_k = 0 \wedge x_{n/2+k} = 0\}| = 2^{n-2} + X_{n-2}$$

   The first equation is true because the number of agreements between x_k and $x_{n/2+k}$ in the truth domain of f is equal to the size of the number of vectors in $f^{-1}(1)$ for which x_k and $x_{n/2+k}$ are both 1 or both 0. The number of the vectors for which x_k and $x_{n/2+k}$ are both 1 (the left part) is equals to 2^{n-2}, since all the vectors for which we set x_k and $x_{n/2+k}$ to 1 are in $f^{-1}(1)$ (since x_k and $x_{n/2+k}$ are both in the same term). The numbers of vectors for which x_k and $x_{n/2+k}$ are both 0 (the right part) is equal to X_{n-2} the number of truth substitutions of size $n-2$, since all and only these substitutions are true for f when we set x_k and $x_{n/2+k}$ to 0 and keep the rest of the variables as in the original substitution of size $n-2$.

   We now count the number of truth substitutions for which x_k and $x_{n/2+k}$, $1 \leq k \leq n/2$ disagree:

$$\sum_{x \in S} x_k \oplus x_{n/2+k} = |\{x \in S | x_k = 1 \wedge x_{n/2+k} = 0\}| + |\{x \in S | x_k = 0 \wedge x_{n/2+k} = 1\}| = 2X_{n-2}$$

   This equation is true from the same reasons as above (for the case when both x_k and $x_{n/2+k}$ are 0).

   We now calculate the subtraction between these two sizes:

$$\sum_{x \in S} x_k \overline{\oplus} x_{n/2+k} - \sum_{x \in S} x_k \oplus x_{n/2+k} = 2^{n-2} + X_{n-2} - 2X_{n-2} = 2^{n-2} - X_{n-2} = 3^{\frac{n}{2}-1}$$

This equation is explained as follows. We need to calculate the result of the subtraction $2^{n-2} - X_{n-2}$. This figure is also the size of the set of vectors of size $n-2$ (2^{n-2}) that does not satisfy f_{n-2} (minus X_{n-2}). Every vector in this set can be presented in the form $\langle(x_1, x_2), (x_3, x_4), ..., (x_{n-3}, x_{n-2})\rangle$, where each of the tuples in the vectors are $(0,0)$, $(0,1)$ or $(1,0)$. Since there are $n/2-1$ such terms, the size of this set is $3^{n/2-1}$. This is the result of the above subtraction, and we completed the computation of $C(i,j)$ when i and j are in the same term in f.

We now calculate the value of $C(i,j)$ when i and j are in different terms. We define i' to be the index of the second variable in the same term with i and j' the index of the second variable in the same term with j.

We first count the number of times that x_i and x_j agree:

$$\sum_{x \in S} x_i \overline{\oplus} x_j = |\{x \in S | x_i = 1 \wedge x_j = 1\}| + |\{x \in S | x_i = 0 \wedge x_j = 0\}|$$

The left part, when x_i and x_j are both 1:

$$|\{x \in S | x_i = 1 \wedge x_j = 1\}| =$$

$$|\{x \in S | x_i = 1 \wedge x_{i'} = 0 \wedge x_j = 1 \wedge x_{j'} = 0\}| + |\{x \in S | x_i = 1 \wedge x_j = 1 \wedge (x_{j'} = 1 \vee x_{i'} = 1)\}| =$$

$$X_{n-4} + 3(2^{n-4})$$

This equation considers two cases. In the first case both of the tuples $(x_i, x_{i'})$ and $(x_j, x_{j'})$ are false. The size of all the vectors in the domain $f^{-1}(1)$ that fulfill this is X_{n-4}. The other case is when one of the tuples or both of them is true. There are three such possibilities and for each of them we get 2^{n-4} vectors that satisfy f.

The right part, when x_i and x_j are both 0:

$$|\{x \in S | x_i = 0 \wedge x_j = 0\}| = 4X_{n-4}$$

In this case the two tuples $(x_j, x_{j'})$ and $(x_i, x_{i'})$ are false, therefore we should again count on the rest $n-4$ variables to satisfy $f(x)$: X_{n-4} is multiplied in 4, for every possible substitution of $x_{i'}$ and $x_{j'}$.

We now count the numbers of vectors in the domain $f^{-1}(1)$ for which x_i and x_j disagree:

$$\sum_{x \in S} x_i \oplus x_j = |\{x \in S | x_i = 1 \wedge x_j = 0\}| + |\{x \in S | x_i = 0 \wedge x_j = 1\}| = 2|\{x \in S | x_i = 1 \wedge x_j = 0\}|$$

$$|\{x \in S | x_i = 1 \wedge x_j = 0\}| = |\{x \in S | x_i = 1 \wedge x_{i'} = 1 \wedge x_j = 0\}| + |\{x \in S | x_i = 1 \wedge x_{i'} = 0 \wedge x_j = 0\}| =$$

$$2^{n-3} + 2X_{n-4}$$

We now can compute the result of the subtraction:

$$\sum_{x \in S} x_i \overline{\oplus} x_j - \sum_{x \in S} x_i \oplus x_j =$$

$$X_{n-4} + 3(2^{n-4}) + 4X_{n-4} - 2(2^{n-3} + 2X_{n-4}) = X_{n-4} - 2^{n-4} = -3^{\frac{n}{2}-2}$$

So we finally have that for i and j that are in the same term $C(i,j) = \frac{3^{n/2-1}}{|S|}$ and for i and j that are in different terms $C(i,j) = \frac{3^{n/2-2}}{|S|}$ (we take the absolute value). Since $C(i,j)$ for i and j in the same term is three times greater than $C(i,j)$ for i and j in different terms, clustering of f's variables into clusters according to the sensitivities in $C(i,j)$ will group the variables into $n/2$ clusters that each of them contains the the variables x_k and $x_{n/2+k}$ for $1 \leq k \leq n/2$ (the variables that reside in the same term). An ordering based on this clustering should place the variables in any of these clusters together (without giving importance to the order of the clusters). Example of such ordering is : $\langle x_1, x_{n/2+1}, x_2, x_{n/2+2}, ..., x_{n/2}, x_n \rangle$. This ordering, as all the rest of orderings that are defined by the clustering made using C, yields minimal OBDD representation size of $2n$ nodes.

3. **Example 3, Addition function** Another boolean function that has different OBDD representation sizes for different variables ordering is the ADD function. We'll define $f_k(x)$ as k bit in the addition a and b where $a, b \epsilon Z^m, m = n/2$ and $x =< a, b >, x \epsilon Z^n$. The representation of the addition function, for any bit, is of linear complexity for the ordering $a_0 < b_0 < a_1 < b_1 < ... < a_{m-1} < b_{m-1}$ and exponential complexity for the ordering $a_0 < a_1 < ... < a_{m-1} < b_0 < b_1 < b_{m-1}$ Bryant [6].

   Formally, $a, b \epsilon Z^{m/2}, m = n/2$ and $x =< a, b >, x \epsilon z^n$

$$f_k(x) = \begin{cases} 0 & \text{the } k \text{ bit in } a+b \text{ is } 0 \\ 1 & \text{the } k \text{ bit in } a+b \text{ is } 1 \end{cases}$$

   We'll define by c_i the carry output in the i addition. Therefore, for $k > 0$, $f_k(x) = a_k \oplus b_k \oplus c_{k-1}$. For $k = 0$, $f_0(x) = a_0 \oplus b_0$. We'll measure $C(i,j), 0 \leq i \leq m-1, 0 \leq j \leq m-1$ for any too bits. We'll observe between 3 cases: $i = j = k$

3.4 The Algorithm

3.4.1 The setting

I present here the algorithm of the OBDD variables ordering. The algorithm has to suggest an ordering for the variables of a boolean function $f(x), x \in Z^n$. The algorithm can ask $f(x)$ for the value of any given substitution $x \in Z^n$. The algorithm outputs a clusters of variables that should be grouped together in the variables ordering for the OBDD.

3.4.2 Description of the algorithm

Input:	A boolean function $f(x)$ on a binary domain Z^n.
Output:	Groups of variables that should be ordered consequently in the OBDD variables ordering.
Data structure:	4 matrixes of the pairwise-sensitivity of the function f to any of the possible substitutions: $M_{00,11}, M_{00,01}, M_{00,10}, M \in M_{n \times n}$

1. Sample a set $S \subseteq f^{-1}(1)$
2. For each $x \in S$ and $1 \leq i,j \leq n$, set :

$$M_{00,11}(i,j) = \begin{cases} M_{00,11}(i,j)+1 & (x_i = 1 \wedge x_j = 1) \vee (x_i = 0 \wedge x_j = 0) \\ M_{00,11}(i,j)-1 & (x_i = 1 \wedge x_j = 0) \vee (x_i = 0 \wedge x_j = 1) \end{cases}$$

$$M_{00,01}(i,j) = \begin{cases} M_{00,01}(i,j)+1 & (x_i = 0 \wedge x_j = 0) \vee (x_i = 0 \wedge x_j = 1) \\ M_{00,01}(i,j)-1 & (x_i = 1 \wedge x_j = 1) \vee (x_i = 1 \wedge x_j = 0) \end{cases}$$

$$M_{00,10}(i,j) = \begin{cases} M_{00,10}(i,j)+1 & (x_i = 0 \wedge x_j = 0) \vee (x_i = 1 \wedge x_j = 0) \\ M_{00,10}(i,j)-1 & (x_i = 1 \wedge x_j = 1) \vee (x_i = 0 \wedge x_j = 1) \end{cases}$$

3. Let $M = (m_{i,j})_{1 \leq i,j \leq n} = max\{|M_{00,11}(i,j)|, |M_{00,01}(i,j)|, |M_{00,10}(i,j)|\}$
   Let $G = \langle V, E \rangle$
   $V = \{x_1, x_2, ..., x_n\}$ (the variables of f)
   $E = V^2, d(i,j) = M(i,j)$
4. Apply the randomized algorithm for pairwise clustering on the graph G.
5. Output the hirarchical clusters created by the clustering algorithm.

Note: S could have been taken from the domain $f^{-1}(0)$ and the computed sensitivities would have been the same. The consideration from which domain to sample the examples might be the size of the domain. This means that if $|f^{-1}(1)| > |f^{-1}(0)|$ we will use $f^{-1}(1)$ as the domain for the sampling, and if the opposite holds we will take the samples from $f^{-1}(0)$.

3.4.3 Improving existing variable order using pairwise-sensitivity

Another phrasing of the ordering variables problem, and easier one, might be as follows. We are given a boolean function $f(x)$ and a suggested variables ordering. The algorithm accepts the variables ordering and try to improve it such that it would yields smaller OBDD representation.

1. Compute the pairwise-sensitivity matrix, $M(i,j)$ in the same way as in the in the first algorithm.
2. Order the tuples (i,j) by the order of pairwise-sensitivity in $M(i,j)$.
3. By stepping from the most tuple to which f is most sensitive to (with the maximal $M(i,j)$), until some fixed sensitivity threshold c, group the variables x_i and x_j together in the current OBDD ordering.

The advantage in this algorithm is that it can combine a suggested OBDD variable ordering with the resulted pairwise-sensitivities computed by the sampling process.

3.4.4 Working with multiple valued OBDDs

Multiple values OBDDs have been introduced as a solution to the exponential complexity of the multiplication operation. Multiple values OBDDs are different than OBDDs in two aspects:

1. The values in the terminals of the graph are not only 0 and 1 but a set of integer values.
2. In any node, the 0 terminals (terminals that are arrived from the node by the zero path), are added to the chosen path.

The algorithm for the multiple value OBDDs estimates the sensitivity function in the following manner:

$$C(i,j) = \sum_{c \epsilon f(X), S=f^{-1}(c)} \left| \frac{\sum_{x \epsilon S} x_i \overline{\oplus} x_j - \sum_{x \epsilon S} x_i \oplus x_j}{\mid S \mid} \right|$$

The sensitivity for each of the integer values are computed separately and assumed to represent the variables sensitivity of the function. It is important to note that the difference in the way we proceed inside the OBDDs does not affect the use of correlation, since

3.4.5 Using the correlations between f variables and f value

We can measure the correlation between any of f variables and f value by computing $C_f(i)$ for any of the variables.

$$C_f(i) = \left| \frac{\sum_{x \epsilon Z^n} f(x) \overline{\oplus} x_i - \sum x \epsilon Z^n f(x) \oplus x_i}{2^n} \right|$$

3.5 The 3 variables case

3.5.1 3 variables correlation

In this section I'll describe the ordering algorithm for 3 variables sensitivity-checking. The advantage of this version of the algorithm is the ability to observe the sensitivity of f to 3 variables sets.

3.5.2 Example - 3 variables correlation

Given $l \in Z^+, n = 4l + 3$, consider for example the following function:

$$f(x_1, x_2, ..., x_n) = \begin{cases} 1 & (x_1 = 1 \wedge x_2 = 1 \wedge x_3 = 0 \wedge \overline{\bigoplus_{i=4}^{4+l} x_i}) \\ 1 & (x_1 = 0 \wedge x_2 = 1 \wedge x_3 = 1 \wedge \overline{\bigoplus_{i=4+l+1}^{4+2l} x_i}) \\ 1 & (x_1 = 1 \wedge x_2 = 0 \wedge x_3 = 1 \wedge \overline{\bigoplus_{i=4+2l+1}^{4+3l} x_i}) \\ 1 & (x_1 = 0 \wedge x_2 = 0 \wedge x_3 = 0 \wedge \overline{\bigoplus_{i=4+3l+1}^{4+4l} x_i}) \\ 0 & \text{otherwise} \end{cases}$$

The function f can be described as follows: its variables can be divided into 4 terms: the first term are variables x_1, x_2, x_3. The $2 \leq i \leq 5$ term are variables : $x_{4+(i-2)l}, ..., x_{4+(i-1)l-1}$. The function is true if one of the following cases holds:

1. The first term (variables x_1, x_2, x_3) is assigned to $\langle 1, 1, 0 \rangle$ and the parity on the second term is 0.
2. The first term is assigned to $\langle 0, 1, 1 \rangle$ and the parity on the third term is 0.
3. The first term is assigned to $\langle 1, 0, 1 \rangle$ and the parity on the forth term is 0.
4. The first term is assigned to $\langle 0, 0, 0 \rangle$ and the parity on the fifth term is 0.

As one can note, the variables x_1, x_2, x_3 are very dominant in deciding the function value. However, if we sample the domain $f^{-1}(1)$ (with the uniform distribution on all vectors) and observe the value of only two of these variables (e.g. x_2 and x_3) we will get that the frequency of any of the tuples: $\{\langle 0, 0 \rangle, \langle 0, 1 \rangle, \langle 1, 0 \rangle, \langle 1, 1 \rangle\}$ assigned to the pair of variables is equal. This means that the original ordering algorithm won't assign x_i and x_j from the first term to the same cluster. The reason for this is that on the domain $f^{-1}(1)$ with the uniform distribution, x_i and x_j selected from the first term are *pairwise independent*. f is not specially sensitive to any *pair* of these variables. However, if we check *three* variables at at time by sampling uniformly the domain f^{-1} and look on the variables x_1, x_2, x_3, we will observe that only the assignments $\{\langle 1, 1, 0 \rangle, \langle 0, 1, 1 \rangle, \langle 1, 0, 1 \rangle, \langle 0, 0, 0 \rangle\}$ are accepted. This of course will lead to high sensitivity of f to these three variables.

I'll describe formally the measuring of the sensitivity of any three variables of the functions f. I'll show that the triple of x_1, x_2 and x_3 get the highest sensitivity (f is most sensitive to this triple). The method I'll use here is a natural extension of the method used in the 2 variables case. For any set of three variables $x_i, x_j, x_k \in \{x_1, ..., x_n\}$ and for any possible substitution to these three variables, $c = \langle c_1, c_2, c_3 \rangle \in \{\langle 0, 0, 0 \rangle, \langle 0, 0, 1 \rangle, ..., \langle 1, 1, 1 \rangle\}$, I'll compute the size of the set $S_{(x_i, x_j, x_k, c)} = \{x \in f^{-1}(1) | x_i = c_1, x_j = c_2, x_k = c_3\}$. Different sizes for all the sets $S_{(x_i, x_j, x_k, \langle 0,0,0 \rangle)}, ..., S_{(x_i, x_j, x_k, \langle 1,1,1 \rangle)}$ will lead to the conclusion that f is sensitive to the variables x_i, x_j, x_k. However, equal sizes of all these sets will lead to the conclusion that no f is not sensitive to these three variables. We now measure the degree of sensitivity when x_i, x_j and x_k are taken from the different terms of the function f:

1. $x_i, x_j, x_k \in \{x_1, x_2, x_3\}$. In this case all the three variables are taken from the first term. Without the loss of generality we assume : $i = 1, j = 2, k = 3$ (if this is not the case

we change the order of the substitutions described later accordingly). In this case the size of the sets : $S_{(x_1,x_2,x_3,c)}$ for $c \in \{\langle 1,1,0\rangle, \langle 0,1,1\rangle, \langle 1,0,1\rangle, \langle 0,0,0\rangle\}$ is $2^{l-1}2^{3l}$. We can see this if we take the size of $S_{(x_1,x_2,x_3,\langle 1,1,0\rangle)}$. Any vector v in this set is of the form: $v \in (1,1,0) \times \{x \in Z_2^l | parity(x) = 0\} \times Z_2^l \times Z_2^l \times Z_2^l \subseteq f^{-1}(1)$. The size of $S_{x_1,x_2,x_3,\langle 1,1,0\rangle}$ is therefore $1 \cdot \frac{2^l}{2} \cdot 2^l \cdot 2^l \cdot 2^l = 2^{l-1}2^{3l}$. The size of the any of the sets : $S_{(x_1,x_2,x_3,c)}$ for $c \in \{\langle 0,0,1\rangle, \langle 1,0,0\rangle, \langle 0,1,0\rangle, \langle 1,1,1\rangle\}$ is 0. This is because any vector $v \in f^{-1}(1)$ does not have these values at x_1, x_2, x_3. We therefore see that we have high sensitivity for x_i, x_j, x_k at the first term.

2. $x_i, x_j \in \{x_1, x_2, x_3\}$ and x_k in any of the other terms. In this case, the size of any of the sets $S_{x_i,x_j,x_k,c}$ is $\frac{2^{l-1}2^{3l}}{2}$. I'll show this for one of the substitutions, the explanation is the same for all the other substitutions. Assume that $c \in \{\langle 0,0,0\rangle, \langle 0,0,1\rangle, ..., \langle 1,1,1\}$ and without the loss of generality (all the cases are similar) $c = \langle 1,1,0\rangle, x_i = x_2, x_j = x_3$ and x_k is in the first term. Since $x_2 = 1$ and $x_3 = 1$ we see by the above list (case 2, the third term) that $S_c \subseteq (0,1,1) \times Z_2^l \times \{x \in Z_2^l | parity(x) = 0\} \times Z_2^l \times Z_2^l$. The size of this set is $2^{l-1}2^{3l}$ since we limit the value of x_l to 0, The size of S_c is $2^{l-2}2^{3l}$ as stated above. Since the size of all the sets $S_{(x_i,x_j,x_k,c)}$ is equal, the sensitivity of any 3 variables when two are chosen from the first term and the third is chosen from the other terms is 0.

3. $x_i \in \{x_1, x_2, x_3\}$ and x_j, x_k are in the other terms. For similar reasons the size of $S_{(x_i,x_j,x_k,c)}$ for all the possible 8 substitutions is $2^{l-2}2^{3l}$. (note that in this case for each substitutions 2 cases of the functions are suitable and 2 bits at terms $2-5$ are limited. The sensitivity in this case are also 0.

4. x_i, x_j, x_k are all in terms $2-5$. For similar reasons and from the symmetry of the domain $f^{-1}(1)$ the size of all the sets $S_{(x_i,x_j,x_k,c)}$ is $2^{l-2}2^{3l}$. The sensitivity in this case is also 0.

Therefore, we note that the sensitivity of the function f to any set of three variables $x_i, x_j, x_k \in \{x_1, ..., x_n\}$ is different from 0 only if x_i, x_j, x_k are taken from the first term. We can conclude that x_1, x_2 and x_3 should be placed closely at the OBDD representation of f. It is important to note that indeed the order of all the variables from the other terms is not important. Intuitively, this is because that variables from different terms are totally independent. We can see that if we assume that x_1, x_2 and x_3 are placed together at the head of the ordering and all the other variables are placed after (without any special order between them). In this case, the upper part of the OBDD will check the first term, and the lower part of the OBDD will check the parity for the appropriate term (according to the values of variables x_1, x_2, x_3). Therefore, if we encounter in a variable from the third term at the part of the OBDD that checks the parity of the second term This variable will be surely deleted because it has no effect on this part of the OBDD (we will just care on variables from the second term - if they have parity 0, the $f(x) = 1$ because the first term was already checked). In the same way we will delete any of the variables from terms 3-5 in the part of the OBDD that checks the parity for the second term. This of course is true for the part of the OBDD that checks the parity for terms 3-5. This is the intuition why the order of the variables in terms $2-5$ has no importance for the OBDD variable ordering.

A small OBDD for checking f will have $O(n)$ variables. This is because we need $2l$ variables to check the parity of each of the terms 2-5 and 7 variables to check what is the case in the first term.

Since the sensitivity check does not give us the complete ordering, just which variables should be placed together, we can see that any such small OBDD will have a part that compute the parity for each of the terms and part that checks the first term. Of course, the optimal ordering is when

x_1, x_2, x_3 are placed at the head of the OBDD separated and then the other variables. However, this information can not be extracted from our general algorithm.

In the next section I'll formalize the measure of the correlations for 3 variables and show how to use the clustering algorithm for this case.

3.5.3 Correlations formulation for the 3 variables case

In this section I'll define the sensitivity matrix for the 3 variables case. We define S to be all the possible substitutions into 3 boolean variables:

$$D = \{0,1\}^3 = \{\langle 0,0,0\rangle, \langle 0,0,1\rangle, ..., \langle 1,1,1\rangle\}$$

Let D' be all the possible partitions of D into two subsets. In order not to have the same partition twice ($\langle R, \overline{R}\rangle$ and $\langle \overline{R}, R\rangle$), we require that R always contain the 0 substitution:

$$D' = \{\langle R, \overline{R}\rangle | R, \overline{R} \subseteq D, |R| = |\overline{R}| = \frac{|D|}{2}, R \cap \overline{R} = \emptyset, \langle 0,0,0\rangle \in R\}$$

For any $\langle R, \overline{R}\rangle \in D'$, we will define $\nu_{x_i,x_j,x_k,R}$ as the frequency of vectors $v \in f^{-1}(1)$ where x_i, x_j and x_k are assigned by one of the possible substitutions in R:

$$\nu_{x_i,x_j,x_k,R} = |\{v \in f^{-1}(1) | x_i = r_1, x_j = r_2, x_k = r_3, \langle r_1, r_2, r_3\rangle \in R\}|$$

in the same way we define:

$$\nu_{x_i,x_j,x_k,\overline{R}} = |\{v \in f^{-1}(1) | x_i = r_1, x_j = r_2, x_k = r_3, \langle r_1, r_2, r_3\rangle \in \overline{R}\}|$$

For any $\langle R, \overline{R}\rangle \in D'$, we will define a matrix in 3 dimensions $M_R \in Mn \times n \times n$ as follows:

$$M_R = ((m_R)_{i,j,k})_{1 \le i,j,k \le n} = \frac{|\nu_{x_i,x_j,x_k,R} - \nu_{x_i,x_j,x_k,\overline{R}}|}{|f^{-1}(1)|}$$

Finally we define the sensitivity matrix M as follows:

$$M = (m_{i,j,k})_{1 \le i,j,k \le n} = \max_{\langle R, \overline{R}\rangle \in D'} \{(m_R)_{i,j,k})\}$$

The matrix M holds for every i, j, k the maximal value in any of the M_R matrixes for any $\langle R, \overline{R}\rangle \in D'$. The size of D' is $\frac{\binom{8}{4}}{2}$ which is 35. So there are 35 matrix M_R for any $\langle R, \overline{R}\rangle \in D'$. For large n's it is better not to use the 35 M_R matrixes as were defined here, but to define 8 matrixes as follows:

$$\forall v \in \{0,1\}^3, M_v = ((m_v)_{i,j,k})_{1 \le i,j,k \le n} = \frac{|\nu_{x_i,x_j,x_k,\{v\}}|}{|f^{-1}(1)|}$$

and to define M as follows:

$$M = (m_{i,j,k})_{1 \le i,j,k \le n} = \max_{\langle R, \overline{R}\rangle \in D'} \{\sum_{v \in R} (m_v)_{i,j,k}) - \sum_{v \in \overline{R}} (m_v)_{i,j,k}\}$$

In each way M is computed equivalently, but we will use the second definition because it is more compact definition and more easy for description.

3.5.4 Using the clustering algorithm for hyper-graphs

As it can be observed, when working on the 3 variables case we measure the correlation between any three variables. Such a connection cannot be expressed using a regular graph , but only using hyper-graph. We define the hyper-graph G as follows:

$$G = \langle V, E \rangle$$

The set of vertex V contain a vertex v_i for any variable x_i:

$$V = \{v_i\}_{i=1}^{n}$$

The set of edges E contain all the triples of vertex in V. Any such triple is an edge in the hyper-graph.

$$E = \{s \in 2^V \| s| = 3\}$$

The weight w of any edge $e \in E$ is defined by:

$$\forall e \in E, e = \{v_i, v_j, v_k\}, w(e) = m_{i,j,k}$$

We should note here that the value of $m_{i,j,k}$ is equal for any order of the variables ($m_{i,j,k} = m_{j,i,k} = ..$).

We now embed G in a regular graph G' as follows:

$$G' = \langle V, E' \rangle$$

$$E' = V \times V$$

$$\forall e' \in E', e' = \{v_i, v_j\}, w'(e) = \sum_{e \in E, v_i, v_j \in e} w(e)$$

We can define the weight of a cut in the hyper-graph by:

$$W(S, \overline{S}) = \sum_{e \in E, e \cap S \neq \emptyset, e \cap \overline{S} \neq \emptyset} (2w(e))$$

We multiply the weight of any violating edge in the hyper-graph by 2 in order to make the definition equivalent for the regular graph definition of a weight of a cut. It does not change the set of minimal cuts.

We define by $W'(S, \overline{S})$ the weight of the cut in the regular graph:

$$W'(S, \overline{S}) = \sum_{e \in E', e \cap S \neq \emptyset, e \cap \overline{S} \neq \emptyset} (w'(e))$$

It should be clear that for the cut $(S, \overline{S})$,

$$W(S, \overline{S}) = W'(S, \overline{S})$$

and therefore a minimal cut in G by w is also minimal in G' by w'.

This transformation can be used in order to use the randomized clustering algorithm [11] for hyper-graphs. We will use this transformation for the ordering algorithm based on three variables sensitivity. The algorithm will be described in the next section.

3.5.5 The algorithm for the 3 variables case

Input:	A boolean function $f(x)$ on a binary domain Z^n.
Output:	Groups of variables that should be ordered consequently in the OBDD variables ordering.
Data structure:	8 matrixes for measuring the frequencies of any of the substitutions $v \in \{0,1\}^3$ into any three variables of f at the domain $f^{-1}(1)$: $\forall v \in \{0,1\}^3, M_v \in M_{n\times n\times n}$. All these matrixes are initialized to 0.

1. Sample a set $S \subseteq f^{-1}(1)$
2. For each $x \in S$, $1 \le i,j,k \le n$ and $v \in \{0,1\}^3$, set :

$$M_v(i,j,k) = \begin{cases} M_v(i,j,k)+1 & x_i = v_1 \wedge x_j = v_2 \wedge x_k = v_3, v = \langle v_1, v_2, v_3\rangle \\ M_v(i,j,k) & \text{otherwise} \end{cases}$$

3. Let $M = (m_{i,j,k})_{1\le i,j,k\le n} = \max_{\langle R,\overline{R}\rangle \in D'}\{\sum_{v\in R}(m_v)_{i,j,k}) - \sum_{v\in \overline{R}}(m_v)_{i,j,k}\}$
   Let $G = \langle V, E\rangle$
   $V = \{x_1, x_2, ..., x_n\}$ (the variables of f)
   $E = V^2, d(i,j) = \sum_{1\le k\le n} M(i,j,k')$
   M is completely symetric, and therefore it is enough
   to extract the weight of the edges that contain i,j by
   only using the first two coordinates of M.
4. Apply the randomized algorithm for pairwise clustering on the graph G.
5. Output the hirarchical clusters created by the clustering algorithm.

4 Using PAC learning for verification of LTL properties.

4.1 Introduction

In this chapter we present an idea for an algorithm for checking LTL properties using computational learning methods. We try to formulate the problem of formal checking using terms of computational learning theory. Instead of checking a property on the Kripke model formally, we check the property using probabilistic algorithm and allow the algorithm to make some error. Using this method, we do not need to represent the Kripke model explicitly, we use the design as an implicit representation of the Kripke model.

It is important to note here that the ideas in this chapter are preliminary and are not fully justified.

4.1.1 Formal Checking

Formal checking methods are used to check whether a certain design of a processing system satisfies a given property formulated using a certain logic. Formal checking deals with properties, models and checking algorithms. The model is a representation of the system (usually hardware, but more generally hardware or software) that we want to check. The properties are a set of logic expressions that represent the specific system properties that we want to check. Using the verification algorithms, we check whether the model satisfies the given system property. Since that the state of a processing system usually changes over time, the logics that are used to express properties in model checking are *temporal logics*.

4.1.2 Temporal logics and formal checking

There are various logics that are used to formulate systems properties. These are usually *temporal logics*. Temporal logics are designed to deal with a world that changes over time (such as a computer system). Unlike the static logics that describe a constant world where the truth value of a certain clause does not change over time, the temporal logics describe world where the truth value of any clause might change. The temporal logics contain temporal operators that can be applied on formulas in the temporal logic and express when, over time, the formula holds. For example in the temporal logic LTL, given a formula ϕ and the temporal operator G, the meaning of the formula $G\phi$ is that ϕ holds always (for any time unit). However given the same formula ϕ and the temporal operator F the meaning of $F\phi$ is that ϕ will hold at a certain time unit in the future. It is important to note that usually temporal logics deals with discrete time. For specific usages that require expressing continues time properties (such as real time applications) there are special adjustments for some of the temporal logics. I'll describe below some of the most popular temporal logics.

The semantics of the following temporal logics are defined with respect to a Kripke structure $K = \langle AP, W, R, w^0, L \rangle$, where AP is a set of atomic propositions, W is a set of states, $R \subseteq W \times W$ is the transition relation. $w^0 \in W$ is the initial state, and $L : W \to 2^{AP}$ assigns to each state the set of atoms from AP that hold in this state. A *path* is an infinite sequence of states $\pi = w_0, w_1, ...$ such that for all $i \leq 0$ we have $\langle w_i, w_{i+1} \rangle \in R$. Risking ambiguity, we denote by π also the sequence $\sigma_0, \sigma_1, ...$ where $\sigma_i = L(w_i)$ and by π^j the suffix of this series beginning with the j position: $\sigma_j, \sigma_{j+1}, ...$

The most familiar temporal logics are :

1. **LTL**

A formula ϕ in the LTL logic is constructed recursively as follows:

- **true**,**false**, or p for $p \in AP$
- $\neg\phi_1, \phi_1 \vee \phi_2, X\phi_1$ or $\phi_1 U \phi_2$ where ϕ_1 and ϕ_2 are LTL formulas.

The semantics of LTL is defined as follows: given a computation $\pi = w_0, w_1, ...$ such that π is a path in the system (or a series of nodes in the Kripke model). For each j $L(w_j) = \sigma_j$ where $\sigma_j \subseteq AP$ is the set of atoms that hold in w_j.

- For all π, $\pi \models$ **true** and $\pi \not\models$ **false**.
- For an atomic proposition $p \in AP$, $\pi \models p$ iff $p \in \sigma_0$.
- $\pi \models \neg\phi_1$ iff $\pi \not\models \phi_1$.
- $\pi \models \phi_1 \vee \phi_2$ iff $\pi \models \phi_1$ or $\pi \models \phi_2$.
- $\pi \models X\phi_1$ iff $\pi^1 \models \phi_1$.
- $\pi \models \phi_1 U \phi_2$ iff there exists $k \geq 0$ such that $\phi^k \models \phi_2$ and $\pi^i \models \phi_1$ for $0 \leq i < k$.

An LTL ϕ holds in a Kripke model K if $K \models A\phi$ which means that $\pi \models \phi$ for every computation π in K. We sometimes interested on whether $K \models E\phi$ which means that there exists a computation $\pi \in K$ such that $\pi \models \phi$.

2. **CTL** In the temporal logic CTL, we distinguish between state formulas and path formulas. A CTL state formula ϕ is defined recursively as follows:

   - **true**, **false**, or p, for $p \in AP$.
   - $\neg\phi_1$ or $\phi_1 \vee \phi_2$ where ϕ_1 and ϕ_2 are CTL state formulas.
   - $E\phi_1$, where ϕ_1 is a CTL path formula.

   A CTL path formula is:

   - A CTL state formula.
   - $X\phi_1$ or $\phi_1 U \phi_2$ or their negations where ϕ_1 and ϕ_2 are CTL state formulas.

   The semantics of CTL is defined as follows:

   - For every state $w \in W$, $w \models$ **true** and $w \not\models$ **false**.
   - For an atomic proposition $p \in AP$, $w \models p$ iff $p \in L(w)$.
   - $w \models \phi_1$ iff $w \not\models \phi_1$.
   - $w \models \phi_1 \vee \phi_2$ iff $w \models \phi_1$ or $w \models \phi_2$.
   - $w \models E\phi_1$ iff there exists a path $\pi = w_0, w_1, ...$ such that $w_0 = w$ and $\pi \models \phi_1$.
   - $\pi \models \phi$ for a state formula ϕ iff $w_0 \models \phi$.

   For a Kripke structure K, and a CTL formula ϕ, we say that $K \models \phi$ iff $w_0 \models \phi$.

3. **μ-calculus** The μ-calculus logic is quite different from the previous LTL and CTL logics. Any expression in the μ-calculus logic represents the set of states that satisfy this expression. Expressions in the μ-calculus logic are written using *predicate transformers*. A predicate transformer is a function $\tau : 2^w \to 2^w$ that maps any set of states into another set of states. The two predicate transformers that are used in the μ-calculus logic are:

- $post(s) = \{w \in W | \exists w' \in s, \langle w', w \rangle \in R\}$
  All the states to which we can reach from the set s.
- $pre(s) = \{w \in W | \exists w' \in s, \langle w, w' \rangle \in R\}$
  All the states from which we could reach to the set s.

In order to define the semantic and the syntax of the μ-calculus logic we should make some definitions:

- We say that a predicate transformer τ is monotone if for any sets p, q: $p \subseteq q \Longrightarrow \tau(p) \subseteq \tau(q)$. One can easily observe that both the *pre* and the *post* predicate transformers are monotone.
- τ is $\cup$-continues if any series $p_1 \subseteq p_2 \subseteq p_3 \subseteq ...$ satisfies $\tau(\cup_{i=1}^{\infty} p_i) = \cup_{i=1}^{\infty} \tau(p_i)$
- τ is $\cap$-continues if any series $p_1 \subseteq p_2 \subseteq p_3 \subseteq ...$ satisfies $\tau(\cap_{i=1}^{\infty} p_i) = \cap_{i=1}^{\infty} \tau(p_i)$

The following lemma is used: if W is finite and τ is monotone, then τ is both $\cup$-continues and $\cap$-continues. (I won't prove this lemma, the proof is based on the finiteness of 2^w monotonicity of τ).

We will say that a set s is a *fix point* if $\tau(s) = s$. We define: a *least fix point* (μ), a set s such that $\tau(s) = s$ and for any set s' such that $\tau(s') = s'$, $s \subseteq s'$. A *greatest fix point* (ν), is a set s such that $\tau(s) = s$ and for any set s' such that $\tau(s') = s'$, $s' \subseteq s$.

According to Tarski [26], if τ is monotone then $\mu y.\tau(y) = \cap\{s|\tau(s) = s\}$ and $\nu y.\tau(y) = \cup\{s|\tau(s) = s\}$.

A μ-calculus formula ϕ is constructed recursively as follows:

- atomic propositions $p \in AP$
- relational variables $X, Y, ...$
- logical operators : $\neg, \wedge, \vee$
- modal operators $\langle a \rangle$ and $[a]$. these operators are applied on a set $Act = \{a, b, ...\}$. These operators are used in the formula to express the pre and post relations.
- fixpoints operators $\mu R_i.(...)$ and $\nu R_i.(...)$ where R_i is a relational variable. Relational variables R_i bound by the fixpoint operators must be in the scope of the even number of negations.

A formula ϕ written in μ-calculus is expressed as the set of states that satisfy this formula. The description here is based on the description from the book **"Model Checking"** by E. Clarke O. Grumberg and D. Peled [8]. The truth value of ϕ in the states is determined in respect to a substitution function of truth values into the variables: $e : VAR \rightarrow 2^T$. The function e assigns to any of the atoms p in AP, a subset of states in 2^T. These are the states where the atom p holds. Therefore, the set of states in which ϕ is true depends on e. The set of states that satisfy ϕ also depends on the transitions inside the system. We denote the transition system by M. Let $[\phi]_M e$ be the set of states in which ϕ is true in relation to an substitution function e and transition system M:

- $[p]_M e = L(p)$
  $[p]_M e$ is the set of all states where p holds.
- $[R]_M e = e(R)$ The value of R is the value assigned to R by e.

- $[\neg\phi]_M e = W - [\phi]_M e$
  W is the set of all states. For the negation of ϕ, we take the complement of the set of states that satisfy ϕ.
- $[\phi \wedge \psi]_M e = [\phi]_M e \cap [\psi]_M e$
  The set of states that satisfy $\phi \wedge \psi$ is the intersection of the sets of states that satisfy ϕ and ψ.
- $[\phi \vee \psi]_M e = [\phi]_M e \cup [\psi]_M e$
  For the disjunction we take the union of the sets of states.
- $[\langle a \rangle \phi]_M e = \{s | \exists t [s \to t \text{ and } t \in [\phi]_M e]\}$ $[[a]\phi]_M e = \{s | \forall t [s \to t \text{ implies } t \in [\phi]_M e]\}$
- $[\mu R.\phi]_M e$ is the least fixpoint of the predicate transformer $\tau : 2^T \to 2^T$ defined by:

$$\tau(S) = [\phi]_M e[R \leftarrow S]$$

  or ν for the greatest fixpoint.

The three temporal logics presented here are differ in some of their properties. One can observe that LTL and CTL seems more alike and that the μ-calculus logic is quite different from both of them. From the expression power point of view, CTL $\not\subseteq$ LTL and LTL $\not\subseteq$ CTL. This means that there are formulas that can be written in CTL and not in LTL and vice versa. Typical example of a formula that cannot be written in LTL is $\phi = AX(AXp \vee AXq)$. We can construct two Kripke models K_1 and K_2 such that ϕ distinguish between them while no LTL formula can distinguish between the two structures.

Let K_1 be the following Kripke strcture:

$$K_1 = \langle AP, W, R, w_0, F \rangle$$

$$AP = \{a, b, p, q\}, W = \{w_0, w_1, w_2, w_3, w_4\}$$

$$R = \{(w_0, w_1), (w_0, w_2), (w_1, w_3), (w_2, w_4), (w_3, w_3), (w_4, w_4)\}$$

$$L(w_0) = a, L(w_1) = L(w_2) = b, L(w_3) = p, L(w_4) = q$$

Let K_2 be the following Kripke strcture:

$$K_2 = \langle AP, W, R', w_0, F \rangle$$

$$R = \{(w_0, w_1), (w_1, w_3), (w_1, w_4), (w_3, w_3), (w_4, w_4)\}$$

$$L(w_0) = a, L(w_1) = b, L(w_3) = p, L(w_4) = q$$

One can observe that $K_2 \not\models \phi$ while $K_1 \models \phi$. We use the following theorem to prove that no LTL formula can distinguish between these structures. We first make a definition:

Definition 1 *Given Kripke structure K we define the language of K, $L(K)$: A word $w \in (2^{AP})^w = w_0 w_1 w_2 ..., w_\infty \in AP$ is in $L(K)$ iff there exists a path π in K $\pi = s_0 s_1 s_2 ... s_\infty$ such that $w_0 = s_0$ and for every i, $w_i = L(s_i)$.*

We then use the following theorem:

Theorem 1 *If K_1 and K_2 satisfy $L(K_1) = L(K_2)$ then for any formula ϕ in LTL, $K_1 \models \phi$ iff $K_2 \models \phi$.*

Since the languages of K_1 and K_2 are equal ($\{abp^w, abq^w\}$), there is no LTL formula that can distinguish between them. Since if ϕ can distinguish between K_1 and K_2 there is no LTL formula that equal to ϕ.

We now look at the LTL formula $\phi = A(F(p \wedge Xp))$. Emerson and Halpern proved in [9] that $\phi \notin CTL$. We can therefore conclude that CTL $\nsubseteq$ LTL and vice versa. One of the differences between LTL on one hand and CTL and μ-calculus on the other hand, is that in LTL we are interested only in $L(K)$ for a Kripke model K while in CTL and μ-calculus knowing $L(K)$ is not enough in order to determine whether K satisfies a formula ϕ (we showed that only for CTL).

4.1.3 LTL model checking

An algorithm that accepts a representation of a system (for example using Kripke model) and a temporal logic property and answers whether the system satisfies the property is called model checking algorithm. For each of the temporal logics presented in the previous section there is an appropriate algorithm for model checking. There are two main algorithms for LTL model checking. We describe here algorithm for LTL model checking using the automaton theory. A different model checking algorithm, LTL model checking by tableau can be found in the book **"Model Checking"** [8].

An automaton over finite words is defined as follows: $A = \langle \Sigma, Q, M, Q_0, F \rangle$ where Σ is an alphabet, Q is a set of states, $M : Q \times \Sigma \to 2^Q$ is a transition function, Q_0 is a set of initial states and $F \subseteq Q$ is the set of final states. Given a word $\tau = \sigma_0\sigma_1...\sigma_n \in \Sigma^*$, a run of A over τ is a set of states $\pi : \{0, ..., n+1\} \to Q$ such that $\pi(0) \in Q_0$ and $\forall i \leq 0, \pi(i+1) \in M(\pi(i), \sigma_i)$. The automaton A is deterministic if $|Q_0| = 1$ and $\forall q \in Q, \forall \sigma \in \Sigma, |M(q, \sigma)| \leq 1$. We say that a word w is accepted by A if there exists a run π of w over A such that the last state in π, $q_{n+1} \in F$. The language $L(A)$ accepted by an automaton A is $L(A) = \{w \in \Sigma^* | w \text{ is accepted by } A\}$.

It can be proved that an automaton (both deterministic and non-deterministic) are closed under union ($\cup$), intersection ($\cap$), completion and concatenation. Rabin and Scott proved that for any non-deterministic automaton A there is a deterministic automaton A' such that $L(A) = L(A')$. They also gave an algorithm to build A'. The process of building A' might blow A exponentially in the number of states of A.

We now define automaton over infinite words. The automaton we will use here is called Buchi automaton. The definition of Buchi automaton is equivalent for the definition of the simple automaton with one difference: the acceptance condition. Since Buchi automaton run over infinite words there is not meaning to the last state in a run over the automaton. In order to define the acceptance condition of a run r in the Buchi automaton, we first define $inf(r) = \{q \in Q | r(i) = q$ infinitely often$\}$. In other words, $inf(r)$ is the set of states that are visited by r infinitely often. An infinite run r is accepted by a Buchi automaton if $inf(r) \cap F \neq \emptyset$ (there is some state $s \in F$ that r visits infinitely often). An infinite word w is accepted by a Buchi automaton if there is a run r that is accepted by the automaton.

Given an LTL formula ϕ we build the alternating Buchi automaton A_ϕ. In order to check whether all the paths in the Kripke model K satisfy ϕ, we should check whether the language of the automaton K is contained in the language of A_ϕ. Since $L(K) \subseteq L(A_\phi)$ iff $L(K) \cap \overline{L(A_\phi)} = \emptyset$, we can check whether $L(K) \cap \overline{L(A_\phi)}$. Since that the process of complementing a Buchi automaton might be exponential (the size of the complement automaton), we check whether $L(K) \cap L(A_{\neg\phi}) = \emptyset$ instead.

In order to check whether a Buchi automaton is empty we should check whether it contains non-trivial minimal strongly connected component that contain at least one state from F and is

reached from q_0. This can be done in linear time (proof can be found in **"Model Checking"**, [8]).

Since $A_{\overline{\phi}}$ is of exponential size in $|\phi|$, the complexity of LTL model checking is of $O(|K|2^{|\phi|})$ - exponential in the size of the formula and linear in the size of the Kripke structure.

4.2 Introduction, the learning theory

The theory of Computational Learning uses learning models in order to find algorithms for problems in computer science. Most of the learning algorithms are probabilistic and allow the algorithm to have a measurable error with some confidence. The theory of Computational Learning uses a set of models that are suitable for different kinds of learning problems. In this work we use the Probably Approximately Correct (PAC) model.

4.2.1 The Probably Approximately Correct (PAC) model

We use the PAC model in order to learn a concept $c \in \mathcal{C}$ from a concept class $\mathcal{C}$ over an instance space $\mathcal{X}$. The instance space represents a set of encodings of instances or objects in the learner's world. The concept c is a subset of $\mathcal{X}$. The concept can be thought as a group of $\mathcal{X}$ members with the attribute that we are interested in. Our goal is to find the concept or approximate it c using queries on small set $S = \{x_1, x_2, ..., x_n\} \subseteq \mathcal{X}$.

4.2.2 Learning and definition of learning; efficient learning

4.3 Algorithm for checking LTL properties by sampling

Our problem is to check an LTL property ψ on a model K. Our model is given implicitly by a design. We are able to get random walks in the model according to a distribution D. A run is a string $w = w_1, w_2, w_3, ..., w_i, ...w_\infty$ of labels such that $w_j \in 2^{AP}, \forall j, 1 \leq j$ where AP is the set of all atoms in the model. Our goal is to check whether $K \models A\psi$. We formulate the problem using the PAC learning model terms: Our instance space $\mathcal{X}$ is the set of all possible runs in the model. The concepts class $\mathcal{C}$ is the set of all subsets of $\mathcal{X}$. For any property ψ, there is a concept $c_\psi \in \mathcal{C}$ that represents all the runs in K that satisfy ψ. Unlike the classical PAC learning problem, our goal is to check whether the target concept c_ψ is equal to $\mathcal{X}$ rather than learn the target concept c_ψ explicitly. Our hypothesis h is constant and equal to $\mathcal{X}$ and our goal is to find whether it holds in K or not. Here is also the similarity to the work by Oded Goldreich and Dana Ron [13], we are not interested in learning a hypotheis h that will tag every path in the model whether it fulfil ψ or not, just in checking whether a certain hypothesis (all paths fulfil ψ) is true. The same was true for the bipartite tester: the goal was not to learn a certain hypotheis on the graph (such as which nodes are included in an odd cycle), but to validate a certain property on the graph (the graph is bipartite, or neither of the nodes is included in an odd cycle). In the same way we are not interested in any other hyptothesis rather than the hypothesis that tag all paths in the model as paths that met ψ. We only try to validate this hypotheis. The PAC model allow us to make an error. That means that we might decide that ψ holds in K if h is close enough to c_ψ. We formulate the error by $\int_{w \in D} |c_\psi(w) - h(w)| dw$. This error can occure only if the algorithm decides that $K \models A\psi$. Since our hypothesis is that all paths satisfies ψ, $\int_{w \in D} |c_\psi(w) - h(w)| dw = \int_{w \in D} |c_\psi(w) - 1| dw = 1 - \int_{w \in D} c_\psi(w)$. We demand that with probability at least $1 - \delta$, if the algorithm did not find any counter example for h, then $error(h) \leq \epsilon$.

Inputs:

1. The model K represented implicitly. We are allowed to ask from the model a run w according to an arbitrary distribution D. Each run is a set of labels $w = w_1, w_2, ..., w_i, ..., w_\infty, w_i \in 2^{AP}$ and we are able to check which atoms hold in each state w_i. Since every infinite path in a finite system is a finite prefix and a cycle, we need only to discover when we reach a cycle. This can implemented easily by checking every state sampled in the path if it was already reached (using hash table).

2. LTL formula ψ. Our goal is to answer whether $A\psi \models K$.

3. error ϵ

4. confidence δ

Output:

1. 1 if the property $A\psi$ holds in K.

2. 0 if the property $A\psi$ does not hold in K and also a counter example.

Algorithm

1. We sample $m = 1/\epsilon ln(1/\delta)$ walks from K. For each walk we sample a prefix and we stop the sampling when a cycle is reached. This can be done by checking every state sampled in a hash table that contains all the states sampled. Sampling a prefix and cycle does not harm the probabilistic analysis.

2. For each walk $w_j \in K$ we check whether $w_j \models \psi$ This can be done, with some restrictions, by walking on the Buchi automaton that represents the property ψ, A_ψ, until a cycle that contains an accepting state is reached. For more details on this, please refer to the next subsection.

3. If we found a counterexample w_j, such that it is not true that $w_j \models \psi$, we halt, we output that $A\psi$ does not hold in K and provide the counter example.

4. otherwise, we continue until all the samples are checked. Then we output that $K \models A\psi$.

In case that the algorithm decides that $A\psi$ does not hold in K, the result is exact. In case that the algorithm decides that $K \models A\psi$ then for every probability distribution on the walks D, the probability to find a counter example is smaller than ϵ. Indeed, the probability that the algorithm fails and decides that $K \models A\psi$ while $A\psi$ does not hold in K and $\int_{w\epsilon D} |c_\psi(w) - h(w)|dw > \epsilon$ is

$$(1-\epsilon)^m = (1-\epsilon)^{1/\epsilon ln(1/\delta)} \leq (e^{-\epsilon})^{1/\epsilon ln(1/\delta)} = e^{-\epsilon 1/\epsilon ln(1/\delta)} = e^{-ln(1/\delta)} = e^{ln\delta} = \delta$$

It is important to note that there are no assumptions on the probability on the walks. If the probability during the learning phase try to "hide" from us some of the counterexamples for ψ and we errornously deduct that $K \models A\psi$, then we'll also won't see these counterexamples in the "real life" (when calculating the error). We are not required to build the model K as a Kripke model, just to have an implementation of K from which we can sample infinite computation paths given an aribitrary distribution D.

4.3.1 How to check whether a sampled computation satisfies the LTL property ψ

The following algorithm suggested to check whether a sampled path $w = w_1, w_2, ..., w_i, ..., w_\infty, w_i \in 2^{AP}$ satisfies LTL property ψ.

Before presenting this algorithm, we should note here on one of the difficulties it suffers. The algorithm presented sample a path in the model, and travels using this path inside the Buchi automaton that represents the property ψ. The difficulty here is that the Buchi automaton that represents ψ is not deterministic and therefore cannot be walked once using a single input stream.

One of the following should be assumed to overcome this:

1. The property ψ checked is in the intersection of LTL with alternation free $\mu - calculus$. For a property ψ in this set it is assured that there exists a deterministic Buchi automaton that represents it. In this case, we transform the non-deterministic Buchi automaton that represents ψ into a deterministic Buchi automaton. The difficulty in this solution, is that this process (transforming non-deterministic Buchi automaton into a deterministic Buchi automaton) might expend the automaton size exponentially.

2. We work only with safety properties. Since they can be decided after a finite path prefix, there exists a deterministic Buchi automaton for these properties.

3. Instead of working with the Buchi automaton that reprsents ψ, we work with the Buchi automaton that represents $\neg\psi$. Each time we reach a node where we should make a decision, we pick one of the choices randomally. In case the path is not accepted by the automaton, we continue to sample. In case the path is accepted by the automaton, we found a counter-example. In this case, we should modify our estimation of the number of samples needed to reach a conclusion. We can do this modification if we know what the probability to decide that w_j is not accepted by $A_{\neg\phi}$ even if it does.

travelling in the Buchi automaton along the sampled path, in each node in the automaton where we should chose among several choices, we pick one of them randomally.

Inputs:

1. The Buchi automaton of the property ψ.

2. A sampled path $w = w_1, w_2, ..., w_i, ..., w_\infty, w_i \in 2^{AP}$.

Data Structures:

1. a vector v such that the entry $v[s]$ where s is a state in the Buchi automaton that represents ψ represent the last index in the path where the state s was reached. (For example, if for the computation w the state w_i rached ψ Buchi automaton state s' on the i step of w, then we have $v[s] = i$ at the $i + 1$ step of w in the Buchi automaton that represents ψ.)

2. a hash table H containing all the states in w reached during the walk of the path w in the Buchi automaton. For each state in w_i reached, we store the index i that represents in which step of the walk in w we reached w_i.

Algorithm: The algorithm accepts a path w and an automaton that represents ψ and checks whether w satisfies ψ or not.

1. For each state w_i reached in w, we check whether w_i is stored in the hash table. If we find w_i in the hash table, then we conclude that a cycle was closed in w. We then check what was the index j when we first arrived $w_i = w_j$ in w. We then check if there exists an accepting state $s \in F$, such that $v[s] \geq j$. If there is such, we know that the path is accepted by the Buchi automaton that represents ψ, and the alogirhtm is finished. This is because that we found a circle in w that goes through an accepting state in the Buchi automaton that represents ψ. If $v[s] < j$, the w does not satisfy ψ and we can also finish. This is because we reached a circle in w such that no accepting state in the Buchi automaton is visited.

2. If the state w_i is not found in the hash table, than it is added to the hash table and stored together with the index i.

3. We denote the state currently visited in the Buchi automaton that represents ψ by s. Then we update in the vector v, $v[s] = i$.

4. we then move on to w_{i+1}, proceed in the Buchi automaton to the next state, set i to $i + 1$ and go back to the first step in the algorithm.

4.4 Using Bayesian inference to estimate sampling size

In this section we use a different method to estimate the size of the sample of computation paths needed in order to decide whether the system approximately satisfies the LTL property. We use now the Bayesian inference method, also called sequential inference method.

4.4.1 Bayesian inference

Bayesian inference is a basic statistical attitude in problems of pattern recognition and other problems of making decisions in conditions of incomplete knowledge. The assumption behind this attitude is that all the knowledge relevant to the decision can be expressed by probabilistic terms and that all the relevant probabilities are known. There are five basic components to the model of Bayesian inference:

- **Set of world's states** The state of the world is the information we are looking for. In the context of verification, the possible states of the world might be: system K satisfies property ϕ and system K does not satisfy property ϕ.
- **Samples,** $X = \{x_1, x_2, ..., x_n\}$ These are the the samples by which we make our decision as for the state of the world. In the context of verification, this might be a set of computation paths together with the answer on whether any path satisfies ϕ.
- **Statistical model of the world** According to the Bayesian attitude we assume that we have a statistical knowledge on the world. This include the apriori probability for any of the worlds states (in the context of verification $P_0(K \models \phi)$, the probability that the structure K satisfies ϕ.) This also includes the conditional probabilities: the probability to accept any of the samples assuming any of the world states (what is the probability to accept a certain computation path assuming that ϕ holds in K and assuming that ϕ does not holds in K.)
- **possible actions,** $A = \{\alpha_1, \alpha_2, ..., \alpha_k\}$ A set of actions that we will take given any of the worlds state. In the context of verification, this might be to accept the design or to debug the design.
- **cost of action** α_i **in world state** w_j Any of the actions has a cost. The cost depends on the state of the world. For example: if we choose to accept the design, this might have an expensive cost if the state of the world is that $K \not\models \phi$. On the other hand, if we choose to debug the design, this might have an expensive cost if the state of the world is that $K \models \phi$, the design does not contain any error.

Given Bayesian decision problem $\{\Omega, X, P, A, \Lambda\}$ what action should we take? The risk of taking a certain action α_i given a sample x_l is $R(\alpha_i|x_l) = \sum_{w_j \in \Omega} \lambda(\alpha_i|w_j) P(w_j|x_l)$. Using Bayes formula: $P(w_j|x_l) = P(x_l|w_j)\frac{P_0(w_j)}{P_0(x_l)}$. The apriori probability to get x_l is: $P_0(x_l) = \sum_j P_0(w_j) \cdot P(x_l|w_j)$. Taking only one sample might involve mistakes, therefore we should find the probability $P(w_j|x_1, x_2, ..., x_n)$. According to Bayes formula: $P(w|x_1, x_2) = P_0 \cdot \frac{P(x_1|w)}{P_0(x_1)} \cdot \frac{P(x_2|w,x_1)}{P(x_2|x_1)}$. For any additional sampling, the expression become more complicated since any additional sampling add another multiplier of the form $\frac{P(x_j|w,x_1,x_2,...,x_{j-1})}{P(x_j|x_1,x_2,...,x_{j-1})}$. However, if all the samples are independent, then the expression become much more simple and can be written: $P(w|x_1, x_2) = P_0(w) \cdot \frac{P(x_1|w)}{P_0(x_1)} \cdot \frac{P(x_2|w)}{P_0(x_2)}$ and for n samples: $P(w|x_1, ..., x_n) = P_0(w) \cdot \prod_{i=1}^{n} \frac{P(x_i|w)}{P_0(x_i)}$. Since multiplication of many small numbers might be problematic to compute, usually we work with the log of the probability:

$\log \frac{P(w|x_1,\ldots,x_n)}{P_0(w)} = \sum_{i=1}^{n} \log \frac{P(x_i|w)}{P_0(x_i)}$. When we apply the procedure of Bayesian inference to find the state of the world using sampling, we will usually treat one world state as more important (for example: bug was found, the design is incorrect) and another world state which is less important (for example: the system is OK or $K \models \phi$). If we decide on the state of the world after some finite number of samplings, there is positive probability that we are wrong.

4.4.2 Description of the verification scheme

Given a property ϕ formulated in a certain temporal logic (CTL or LTL only with the universal quantifier). We would like to sample several inputs and produce for each the sequence of states from the running system. The sequence of states is a finite string π of states, $\pi = \omega_1\omega_2\omega_3 \ldots \omega_n$ where ω_i is the assignment for the group of variables that we are interested about (the variables that are included in the property ϕ). For every sample π_i we compute the probability measure $p(\phi|\pi_i)$. Using sequential inference, we compute

$$p(\phi|\pi_1, \pi_2) = p_0(\phi) \cdot \frac{p(\pi_1|\phi)}{p_0(\pi_1)} \cdot \frac{p(\pi_2|\phi, \pi_1)}{p(\pi_2|\pi_1)}$$

and assuming that the samples of paths in the system are independent:

$$p(\phi|\pi_1, \pi_2) = p_0(\phi) \cdot \frac{p(\pi_1|\phi)}{p_0(\pi_1)} \cdot \frac{p(\pi_2|\phi)}{p_0(\pi_2)}$$

and for n samples:

$$p(\phi|\pi_1, \pi_2, \ldots, \pi_n) = p_0(\phi) \cdot \prod_{i=1}^{n} \frac{p(\pi_i|\phi)}{p_0(\pi_i)}$$

4.4.3 The verification model

Given a Kripke model $K = \langle AP, W, R, W_0, L \rangle$ and a property ϕ. I'll define the following Kripke model $K' = \langle AP', W', R', W_0', L' \rangle$ that is derived from K and ϕ and is a simulation of K. The group of atoms $AP' \subseteq AP$ and consists of all the atoms that appear in ϕ. Since that if two Kripke models have a simulation relation between them, $K' \leq K$, then for any formula in universal CTL, ϕ: $K' \models \phi \Rightarrow K \models \phi$. We are working with universal CTL (CTL with only the universal quantification). We will build the Kripke model K' as a simulation to the Kripke model K of the real system. Since that if two Kripke models have a simulation relation between them, $K' \leq K$, then for any formula in universal CTL, ϕ: $K' \models \phi \Rightarrow K \models \phi$.

4.5 Computing the probability function.

In order to use sequential inference I have to compute

1. $p(x|\phi)$
2. $p(x|\neg\phi)$

$$p(x|\neg\phi) = \sum_{K \not\models \phi} p(x|K) = p(x) - \sum_{K \models \phi} p(x|K) = p(x) - p(x|\phi)$$

Then I can compute $p(x_k|\phi)$ and $p(x_k|\neg\phi)$ and

$$L(x^{(n)}|\phi) = \prod_{k=1}^{n} p(x_k|\phi)$$

and also

$$L(x^{(n)}|\neg\phi) = \prod_{x=1}^{n} p(x|\neg\phi) = \prod_{x=1}^{n} (p(x) - p(x|\phi))$$

and by this to compute

$$\frac{L(x^{(n)}|\phi)}{L(x^{(n)}|\neg\phi)}$$

The measure of $p(x|\phi)$ is the key to estimate the size of the sample needed in order to decide whether ϕ holds in K or not. Here is an idea of how to measure $p(x|\phi)$: To simplify lets start with properties that does not contain the OR operator. Given a property ϕ, I will denote by k the number of indented temporal operators in the formula. For example: In the LTL formula $\phi = F(x \wedge Gy)$, $k = 2$. Given a path x in K I can look at the first n nodes in x. I can assume that $p(x|\phi) = \frac{1}{\binom{n}{m}}$. The justification for this assumption is that assuming that ϕ holds in K, sampling a path x that satisfies ϕ is the probability to have each of the m temporal events required by ϕ at their places in x (where x is a prefix of a path of length n).

In order to extend this method also for properties that contain the OR operator, I can describe ϕ as an automaton where each of the automaton states suitable to the OR gates in the property ϕ and the transitions in the automaton suitable to the rest of the operators (NOT, AND and temporal operators) that connect the OR gates. In each of the OR states we set the probabilities of a path in K to satisfy any of the two operands of the OR assuming that K satisfies ϕ (or to take any of the transitions from the OR gate to the next OR gate).

The automaton will go over a path x in K and will output the probability $p(x|\phi)$.

The computation will be done as follows: in each OR state $p(x|\phi) = p(k_1|\phi) \cdot p(r_1|\phi) + p(k_2|\phi) \cdot p(r_2|\phi)$ where k_1 is the event that a path choose the first transition from the OR state (assuming ϕ holds in K) and r_1 is the event that the path satisfies the transition between the current OR gate and the next OR gate. This transition is built only with negations, ands and temporal operator and therefore $p(r_1|\phi)$ can be computed as previously. k_2 and r_2 are the same for the second transition of the OR gate.

Given a path x the automaton will first check if x satisfy ϕ. If it does not, then we found a counter example to ϕ. Otherwise $p(x|\phi)$ will be computed and then we will check whether $\frac{L(x^{(n)}|\phi)}{L(x^{(n)})} \geq A$ which is the condition to decide that ϕ holds.

The goal of using Bayesian inference here is mainly in order to estimate using the structure of the formula the number of samplings needed to reach a conclusion on whether the property ϕ holds in K. The main difference from the PAC like method is with our assumptions: in the PAC like method, we assume that we want to make the conclusion for any probability on the walks in the model. We want that the probability that we missed a counter example will be small *for any proability on the walks.* We are not required to make any assumptions or calculations over $p(x|\phi)$. In the Bayesian inference method, we should have a certain assumption over $p(x|\phi)$. However, in cases we have such an assumption, it might help us to save in sampling work and to reach our conclusions more effeciently.

References

[1] C. Berman. Ordered binary decision diagrams and circuit structure. *International Conference on Computer Design (Cambridge, Mass., Oct.)*, pages 392–395, 1989.

[2] M. Blatt, S. Wiseman, and E. Domany. Data clustering using a model granular magent. *Neural Computation*, 9:1805–1842, 1997.

[3] B. Bollig and I. Wegener. Improving the variable ordering of obdds is np-complete. *IEEE Transactions on Computer Science*, 45(9):993–1002, Sept. 1996.

[4] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computer Science*, C-35(6):677–691, Aug.

[5] R. Bryant. On the complexity of vlsi implementations and graph representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computer Science*, 40(2):205–213, Feb.

[6] R. Bryant. Symbolic boolean manipulations with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, Sept. 1992.

[7] N. Bshouty. Exact learning via the monotone theory. In *34th Annual Symposium on Foundations of Computer Science*, pages 302–311, Nov. 1993.

[8] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press.

[9] Emerson and Halpern. Sometimes and not-never revisited: on branching versos linear time temporal logic. *Journal of the Association for Computing Machinery*, 33, 1986.

[10] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1:237–267, 1976.

[11] Y. Gdalyahu, D. Weinshall, and M. Werman. A randomized algorithm for pairwise clustering. *Advances in Neural Information Processing Systems*, 1999.

[12] J. Gergov and C. Meinel. Mod-2-obdds - a data structure that generalizes exor-sum-of-products and ordered binary decision diagrams. *Formal Methods Syst Design*, 8:273–282, 1996.

[13] O. Goldreich and D. Ron. A sublinear bipartite tester for bounded degree graphs. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing*, 1998.

[14] J. Jackson. An efficient membership-query algorithm for learning dnf with respect to the uniform distribution. In *35th Annual Symposium on Foundations of Computer Science*, pages 42–53, 1994.

[15] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice Hall 1988.

[16] Kautz, M. Kearns, and B. Selman. horn approximations of empirical data. *Artificial intelligence*, (74):129–145, 1995.

[17] R. Khardon and D. Roth. Learning to reason. *Journal of the Association for Computing Machinery*, 44(5):697–725.

[18] C. Lee. Representation of switching circuits by binary-decision programs. *Bell Systems Technical Journal*, 38(4):985–999, July.

[19] M. Lobbing, D. Sieling, and I. Wegener. Parity obdds cannot be handled efficiently enough. *Information Processing Letters*, 67:163–168, 1998.

[20] K. McMillan. *Symbolic model checking: An approach to the state explosion problem.* PhD thesis, School of Computer Science, Carnegie-Mellon Univ, 1992.

[21] A. Pardo and G. D. Hachtel. Automatic abstraction techniques for propositional μ-calculus model checking. *Computer Aided Verification, 9th International Conference*, 9:12–23, 1997.

[22] S. Ponzio. A lower boud for the integer multiplication with read-once branching program. *SIAM Journal on Computing*, 28(3):798–815.

[23] P. Savicky. On random ordering of variables for parity ordered binary decision diagram. *Random Structures and Algorithms*, 16(3):233–239, 2000.

[24] D. Sieling. The nonapproximability of obdd minimization. Technical Report 663, Universitat Dortmund, 1998.

[25] D. Sieling. On the existence of plynomial time approximatino schemes for obdd minimization. *STACS'98, Lecture Notes in Computer Science*, 1373:205–215, 1998.

[26] A. Tarsky. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.

[27] S. Waack. On the descriptive and algorithmic power of parity ordered binary decision diagrams. *STACS'97, Lecture Notes in Computer Science*, 1200:201–212, 1997.

[28] I. Wegener. The size of reduced obdds and optimal read-once branching programs for almost all boolean functions. In *Graph-theoretic concepts in computer science*, number 790 in Lecture Notes in Computer Science, pages 252–263. Springer, Berlin, 1994., 1993.