A heuristic for variable ordering in Hierarchical Set Decision Diagrams

AZAIS Lucas PINON Olivier

22 Avril 2014

Contents

Introduction

Model checking is about testing a particular model of a given system, comprehensively and automatically in order to check if it meets a given specification. This can be achieved by generating all the possible states of a system, known as the state space. This kind of testing is especially interesting in aeronautics, as it allows engineers to find all the possible flaws of a system. This method is best implemented using Computer Science, as model checking involves simple but numerous operations.

However, when the system increases in complexity, because of the combinatorial explosion, the quantity of data to deal with can reach overwhelming amounts. This is why alternative methods have been developed, and the one we have been working on is the Hierarchical Set Decisions Diagram, which consists in reducing the size of the representation of the state space by using symetries, patterns and by merging the parts that lead to the same results. We have based our work on Pr. Hamez's C++ Model-Checking library called '*libsdd*' [**libsdd**]. The idea was to find a pre-processing algorithm that would speed up the computing done by '*libsdd*'. An extremely efficient way to improve this computing is to find a good order for the variables used by the model checker, as it can reduce its computing time by orders of magnitude. The aim here is to design an algorithm that will find the optimal variable order, thus leading to the fastest computing time possible.

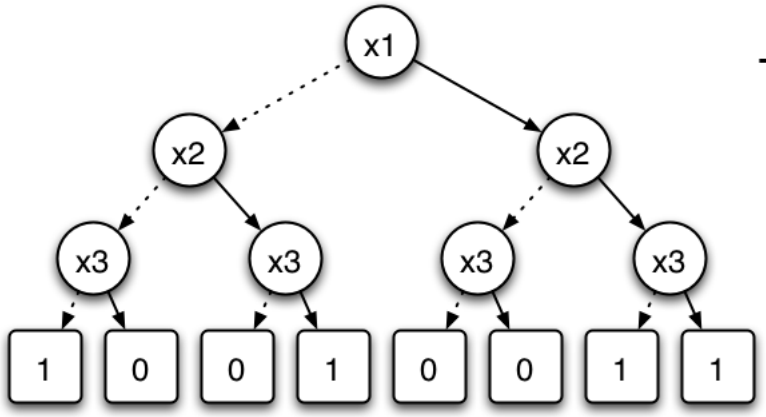
We will first introduce the main notions required to understand how a model checker and decision diagrams work. We then started our work by trying to adapt and implement an algorithm named FORCE, which was originally designed for Binary Decision Diagrams where variables can only be 0 or 1. Third, we ran tests to verify that the model checker was faster with FORCE than without, and we will present the results of these tests. Finally, we laid the basis of a brand new algorithm, based on much simpler matrices.

Chapter 1

Model Checking and Decision Diagrams

Various notions are required before even trying to explain the different techniques used to reduce computing time in a model checker.

First and foremost, a system, may it be a plane, a program or a spaceship, can be represented by a set of variables which can have a variety of values (infinite or not). Take one of these variables and freeze its value : you can choose another one, and compute the different values it can assume, knowing the value of the first one. If you do that for each possible value of every single variable of the system, a graph can be drawn and each path in the graph (a line from top to bottom) represent one possible state of the system.



x1	x2	x3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Figure 1.1: A simple BDD

In figure 1.1, you can see the comprehensive Binary Decision Diagram (BDD) of the function :

$$f(x_1, x_2, x_3) = \bar{x_1}\bar{x_2}\bar{x_3} + x_1x_2 + x_2x_3$$

However, this BDD can easily be reduced by noticing that if both x_1 and x_2 equal 1, then $f(x_1, x_2, x_3)$ will equal 1 as well, no matter what value x_3 assumes. If the path leading to the same value of f (0 or 1) are also joined, the same BDD will look like that :

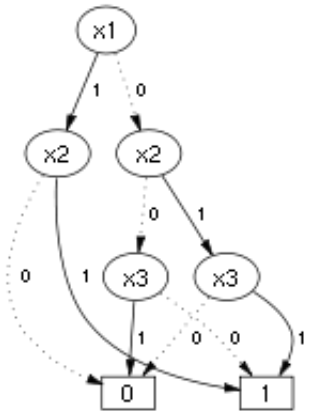


Figure 1.2: The same but reduced BDD

A noticable thing in this diagram is that if you change the order of the variables and switch x_3 and x_1, the size of the representation will increase greatly since you will not be able to keep track of the simplifications made above in figure 1.2. This phenomena can be highlighted in a second example, figure 1.3 where a rearrangement of the variables can lead to a much simpler BDD.

It is therefore readily understandable that a model checker will have a lot less trouble checking the rearranged diagram on the right, rather than the one on the left. It will thus be faster and will prove less prone to making mistakes. The diagram used in our algorithm, the Hierarchical Set Decision Diagram (SDD), is slightly more complex as it allows links between variables to be SDDs themselves. However, the principle remains the same and if a method is found to order BDDs, is can be adapted to SDDs.

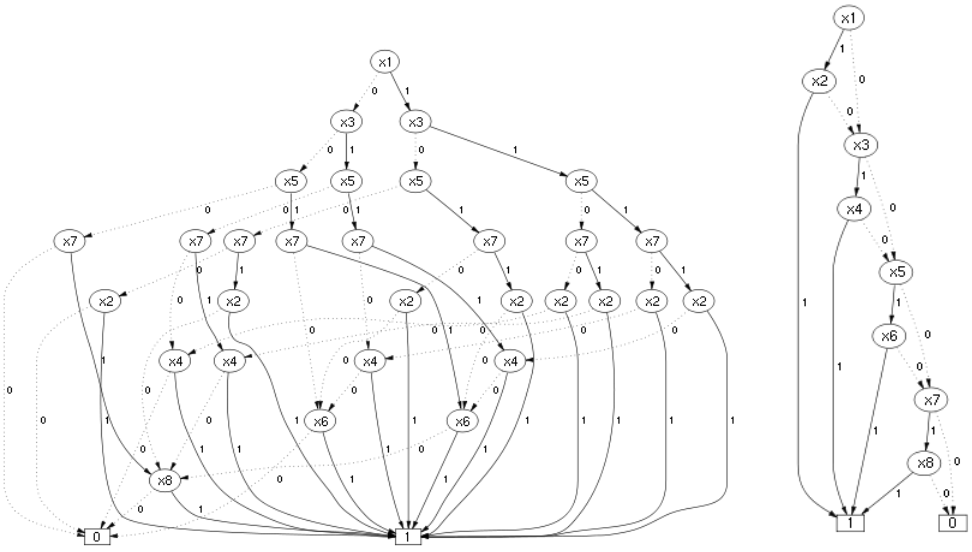


Figure 1.3: The same BDDs with different variable orders

In the case of tiny diagrams, a human can reorder it and let the computer handle the rest of the computation. However, for common systems, the number of variables can easily reach a thousand and more. It is then impossible for a person to even represent the diagram, making it that much more difficult to reduce its size, and considering that the state space of a system has no link whatsoever with the state space of another system, the automatic modification of the variables order seems challenging. One algorithm has nonetheless caught our attention : FORCE.

Chapter 2

FORCE : A powerful algorithm that can easily be enhanced

Taking root in the natural laws of gravity, the FORCE algorithm has been described in *FORCE : A Fast and Easy-To-Implement Variable-Ordering Heuristic* [3]. The algorithm itself is as simple as using forces of attraction between variables. The variables are ordered on a line, given a position, and are attracted to the variables they are linked to. The diagram will thus organize itself in a simpler way, and should be easier to deal with afterwards.

First, the variables are defined, and then the hyperedges which represent the links between variables. The particularity of an hyperedge is that it can bind together more than 2 variables. Then the variables are given an initial position on a line representing the initial order. After that, 3 steps are repeated until a suitable order is reached :

- Compute the center of gravity of each hyperedge (the mean value of the position of the variables it connects.)
- Assign each variable a new position, based on the mean value of the center of gravity of the hyperedges connected to the variable.
- Reorder the variables according to their new position.

To determine the quality of an order and know when to stop, the algorithm computes the span of the order (figure 2.1). It is the sum of the span of every hyperedge, which is the difference between the position of its first and last variables (the ones that are the most apart from each other).

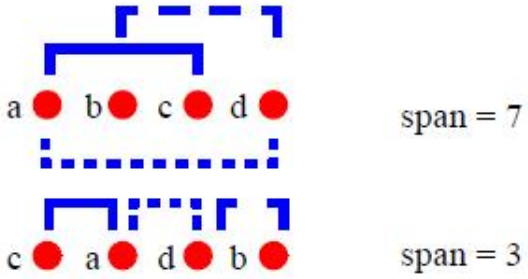


Figure 2.1: A BDD before and after being reordered

Once the 'length' of the order has settled, the algorithm is complete and gives the final order.

In figure 2.2, 4 variables linked by 3 hyperedges can be seen. In the first order, the hyperedges are long, and once reordered, they are shorter, meaning they have a lower span. The algorithm will not be able to find a better order, it will stop and give the optimal order.

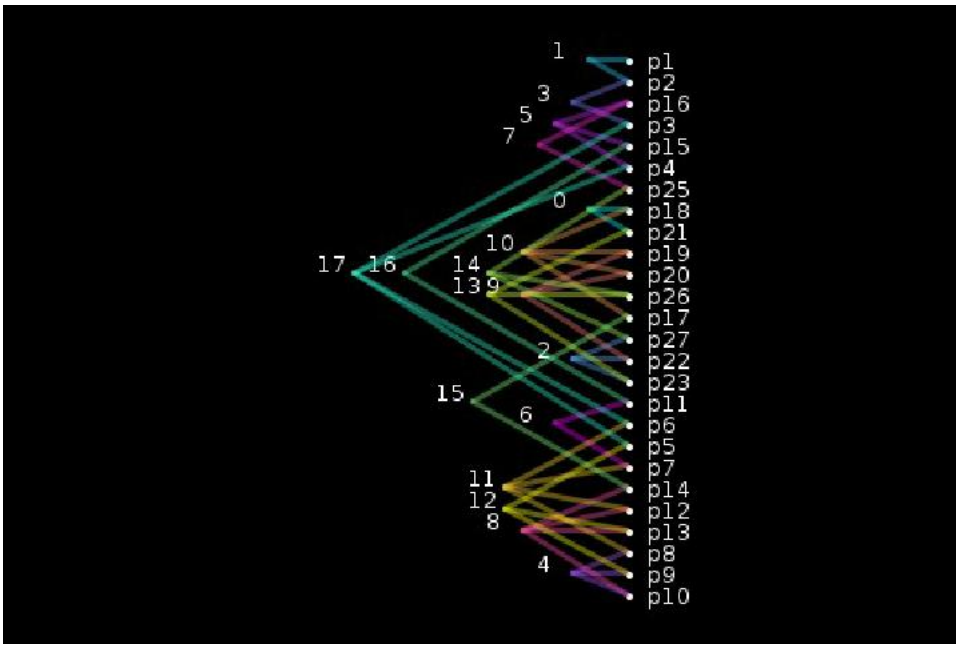


Figure 2.2: A graphic representation of the SDD

To help us in our work, we created a program to display the variable

order. It particularly helps with finding symetries in a graph, or patterns that should be studied further.

On the left are the hyperedges that connect the different variables. The further they are on the left, the higher their span is. On the right are the variables (known as places in this case, hence p1, p2, ...)

FORCE is mainly used to deal with BDDs, but in our case, the SDDs provide another information. As a matter of fact, the links represent operations used on the variables to build the graph, and therefore have a direction. It means that a variable depends on another, and the link between variables can be unidirectional or bidirectional. It seems intuitive to put a variable after the ones it depends on. A variable others depend on is called PRE, a variable depending on a another is a POST, and a bidirectional link involves PRE-POST variables. Using that information before each iteration of the algorithm proved to help decrease the span further in most cases, and can thus be considered a satisfying improvement.

We have adapted that algorithm to SDDs, and despite its simplicity, the results shown in the next part appear to be satisfying in many cases.

Chapter 3

Experimental results

The FORCE algorithm and the enhanced version of FORCE have been tested using some of the most common computer science problems. In each case, we prompted the time needed to apply the model checker and get a solution.

	Computation time (ms)		
Model	Natural	Force	Force Enhanced
prod/kanban10nm.net	1.43573	0.958315	4.46478
tina/eratosthenes-010.pnml.net	0.0022953	0.00206962	0.00186
tina/eratosthenes-100.pnml.net	0.6437	0.1031	0.10439
tina/HouseConstruction-010.pnml.net	36.8247	32.6843	7.2
tina/HouseConstruction-005.pnml.net	2.0089	2.07842	0.64489
pnml/HouseConstruction-005.pnml	0.7947	1.0135	0.7144
tina/l120.net	0.2649	0.2348	0.292
tina/l600.net	10.9645	8.4796	10.3256
prod/ring10nm.net	1.14801	0.88918	0.958829
prod/ring16.net	4.0047	2.9284	3.56533
prod/kanban5.net	0.2273	0.174822	0.46719
prod/kanban20.net	11.8607	6.15946	57.0437
prod/robin10.net	1.0486	0.30424	0.289255
prod/robin20.net	7.2371	1.86381	1.80587
tina/cs_repetitions-2-unfolded.pnml.n	0.7789	0.20387	0.43648
tina/FMS-10.pnml.net	4.28304	0.661379	1.16325
tina/galloc_res-3.pnml.net	2.10228	2.21671	2.48258
pnml/Kanban-5.pnml	0.523066	0.366321	0.54759
tina/l240.net	1.19268	1.02659	1.24548
pnml/MAPK-8.pnml	33.3624	0.1319	0.3239
tina/MAPK-20.pnml.net	25.6881	7.74545	3.85512
tina/millnet100.net	0.2878	0.28959	0.283128
tina/neoelection-2.unf.pnml.net	0.91742	0.2319	0.3062
pnml/Peterson-2.pnml	36.2299	4.1835	3.36189
pnml/dekker-10.pnml	16.44	0.687	0.393

Figure 3.1: Experimental results using the FORCE algorithm and its enhanced version

The tests have shown that although FORCE does give better result in many cases, the most significant results are achieved with the enhanced ver-

sion of FORCE. For the House Construction 010 problem, computation has been improved fourfold, and it gives similar results in many cases. However, the computation time has not been improved in every case. It appears that if the order is already considered 'good', the algorithm tends to worsen it. Further work will determine the reasons behind such behavior.

Chapter 4

A simple matrix representation

Seeing what could be achieved with simple methods, we designed another algorithm based on matrices, the most widely used representation for problems in mathematics. The principle is to put the variables on the lines, and the operations (the hyperedges) on the columns. If a variable is used in an operation, a 1 is put inside the corresponding line and column intersection. Else, a 0 is placed.

To compute the optimal variable order, a specific value is monitored : for each column, the number of 0 between the first 1 and the last 1 is computed, and then we sum those results. It can be seen as the 'area' of the matrix, or its 'void rate'. The more 0 there are inside the columns, the more separated the variables are and the longer the model checking should be.

In order to reduce this 'norm', we can only switch two lines together, which amounts to changing the variable order. Several versions of the algorithm were tested, trying various techniques to reduce the 'norm', unfortunately none has proven significantly useful so far. The progress made thanks to these algorithms were far too slow and small to consider using them. That is why a different technique has been chosen, using a genetic algorithm.

An initial matrix is created, based on the model. Then 100 lines are switched at random, two by two, thus creating 100 'mutated' matrices. Only the 5 matrices with the best 'norm' are kept, and the algorithm starts again by creating 100 new matrices out of these 5, and selecting 5 once more. It is slower than a real norm reduction strategy, but it is far more powerful. This algorithm should be used if time is not of the essence in the model checker that is being run.

Conclusion

The Hierarchical Set Decision Diagrams have been designed to be a fast and efficient representation of state spaces for model checking. We showed that applying a pre-processing algorithm to determine an optimal variable order can sometimes greatly reduce the time needed to run the model checker. The enhanced version of FORCE seems to be efficient in most cases. FORCE takes very little time to be applied and seldom deteriorates the variable order. However, some cases remain where nor the initial order, nor the FORCE order lead to a reasonable computing time.

Future studies should aim at determining why certain models reject the use of FORCE, and what could be done to improve it. A solution might also be to try and imagine new algorithms, but since SDDs are so difficult to grasp, it will be some time before we achieve better results than FORCE. The matrix algorithm is working very well in most cases, but takes a lot more time. This is why it is crucial to have various algorithms available, in order to apply the one suited for the situation.