Automatic heuristic-based generation of MTBDD variable orderings for PRISM models

Internship report

Vivien Maisonneuve
École normale supérieure de Cachan

Under supervision of Dr. David Parker
Oxford University Computing Laboratory
Group of Prof. Marta Kwiatkowska

19 February - 18 July 2009

My internship took place in Oxford, United Kingdom, in the Oxford University Computing Laboratory[1]. I have been working under the supervision of David Parker[2], in the group of Marta Kwiatkowska[3]. This team, among others, develops the probabilistic model checker PRISM[4].

This software is a model checker for probabilistic models. It uses MTBDDs, a specific type of decision diagrams, to represent models and perform model checking. I have been working on reducing these diagrams size, by playing on the order in which the model variables are represented into MTBDDs. Indeed, smaller diagrams lead to a faster computation time and give the ability to handle bigger models. This problem being NP-complete, I had to use heuristics.

The atmosphere in the laboratory was very good and the whole team has been very welcoming with me. I want to thank all of them for this, and especially Dave Parker who was very present and helped me in many ways.

Oxford is also a beautiful city with an harmonious building architecture, and of course I have taken time to visit it. I have been particularly impressed by the university, one of the most famous and eldest ones in Europe, composed of 38 colleges present in lots of districts.

The first section in this report is a short introduction to PRISM, in which we describe the different types of probabilistic models PRISM can handle. Then we define the MTBDD structure and explain how PRISM uses MTBDDs to represent probabilistic models. The third section deals with variable ordering to reduce MTBDDs size, this is the core of my work. Finally, there are two appendices: detailed results of my experiments in PRISM with various heuristics, and a short documentation of the new ordering options I have implemented.

[1] `http://web.comlab.ox.ac.uk/`
[2] `http://web.comlab.ox.ac.uk/David.Parker/`
[3] `http://web.comlab.ox.ac.uk/Marta.Kwiatkowska/`
[4] `http://www.prismmodelchecker.org/`

Contents

1 Background Material

1.1 The PRISM Model Checker

PRISM (probabilistic symbolic model checker) is a tool for the modelling and analysis of systems which exhibit probabilistic behavior. Probabilistic model checking is a formal verification technique. It is based on the construction of a precise mathematical model of a system which is to be analyzed. Properties of this system are then expressed formally in temporal logic and automatically analyzed against the constructed model.

PRISM incorporates several well-known probabilistic temporal logics:

- PCTL (probabilistic computation tree logic).
- CSL (continuous stochastic logic).
- LTL (linear time logic).
- PCTL* (which subsumes both PCTL and LTL).

plus support for costs/rewards and several other custom features and extensions.

PRISM performs probabilistic model checking to automatically analyze such properties. It also contains a discrete-event simulation engine for approximate verification.

More information about PRISM can be found on PRISM website[5], or in David Parker's thesis [Par02].

1.2 Probabilistic Models

Traditional model checking involves verifying properties of labelled state transition systems. In the context of probabilistic model checking, however, we use models which also incorporate information about the likelihood of transitions between states occurring. PRISM can handle three different types of probabilistic models: discrete-time Markov chains, Markov decision processes and continuous-time Markov chains.

1.2.1 Discrete-Time Markov Chains

The simplest of the models handled by PRISM are discrete-time Markov chains (DTMCs). They can be used to model either a single probabilistic system or several such systems composed in a synchronous fashion. We define a DTMC as a tuple (S, s_0, P, l) where:

- S is a finite set of states.
- $s_0 \in S$ is the initial state.
- $P : S \times S \rightarrow [0, 1]$ is the transition probability matrix.
- $l : S \rightarrow 2^{AP}$ is the labelling function.

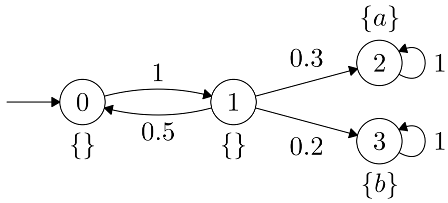
An element $P(s, s')$ of the transition probability matrix gives the probability of making a transition from state s to state s'. We require that $\sum_{s' \in S} P(s, s') = 1$ for all states $s \in S$. Terminating states are modelled by adding a self-loop (a single transition going back to the same state with probability 1). The labelling function l maps states to sets of atomic propositions from a set AP. We use these atomic propositions to label states with properties of interest.

Figure 1 shows a DTMC with four states. In our graphical notation, states are drawn as circles and transitions as arrows, labelled with their associated probabilities. The initial state is indicated by an

[5] `http://www.prismmodelchecker.org/`

additional incoming arrow. The atomic propositions attached to each state, in this case taken from the set $AP = \{a, b\}$, are also shown.

(a)



(b) $P = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0.5 & 0 & 0.3 & 0.2 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Figure 1: A 4 state DTMC and its transition probability matrix

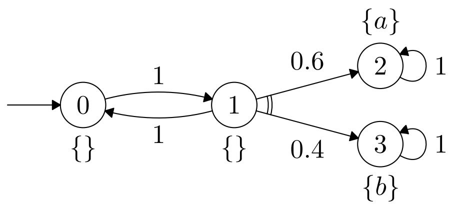
1.2.2 Markov Decision Processes

The second type of model we consider, Markov decision processes (MDPs), can be seen as a generalization of DTMCs. An MDP can describe both nondeterministic and probabilistic behavior. It is well known that nondeterminism is a valuable tool for modeling concurrency: an MDP allows us to describe the behavior of a number of probabilistic systems operating in parallel. Nondeterminism is also useful when the exact probability of a transition is not known, or when it is known but not considered relevant. We define an MDP as a tuple $(S, s_0, Steps, l)$ where:

- S is a finite set of states.
- $s_0 \in S$ is the initial state.
- $Steps : S \to 2^{Dist(S)}$ is the transition function.
- $l : S \to 2^{AP}$ is the labelling function.

The set S, initial state s_0 and labelling function l are as for DTMCs. The transition probability matrix P, however, is replaced by $Steps$, a function mapping each state $s \in S$ to a finite, non-empty subset of $Dist(S)$, the set of all probability distributions over S (i.e. the set of all functions of the form $\mu : S \to [0, 1]$ where $\sum_{s \in S} \mu(s) = 1$). Intuitively, for a given state $s \in S$, the elements of $Steps(s)$ represent nondeterministic choices available in that state. Each nondeterministic choice is a probability distribution, giving the likelihood of making a transition to any other state in S. Figure 2 shows an MDP.

(a)



(b) $Steps =$

	1	2	3	4
1	0	1	0	0
2	1	0	0	0
	0	0	0.6	0.4
3	0	0	1	0
4	0	0	0	1

Figure 2: A 4 state MDP and the matrix representing its transition function

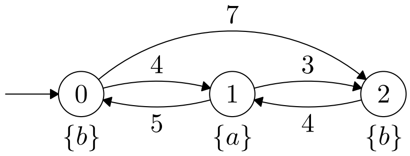
1.2.3 Continuous-Time Markov Chains

The final type of model, continuous-time Markov chains (CTMCs), also extend DTMCs but in a different way. While each transition of a DTMC corresponds to a discrete time-step, in a CTMC transitions can occur in real time. Each transition is labelled with a rate, defining the delay which occurs before it is taken. The delay is sampled from a negative exponential distribution with parameter equal to this rate. We define a CTMC as a tuple (S, s_0, R, l) where:

- S is a finite set of states.
- $s \in S$ is the initial state.
- $R : S \times S \to \mathbb{R}_{\geq 0}$ is the transition rate matrix.
- $l : S \to 2^{AP}$ is the labelling function.

The elements S, s_0 and l are, again, as for DTMCs. The transition rate matrix R, however, gives the rate, as opposed to the probability, of making transitions between states. For states s and s', the probability of a transition from s to s' being enabled within t time units is $1 - e^{-R(s,s')\cdot t}$. Typically, there is more than one state s with $R(s, s') > 0$ (this is known as a race condition). Figure 3 shows a CTMC.

(a)



(b) $R = \begin{pmatrix} 0 & 4 & 7 \\ 5 & 0 & 3 \\ 0 & 4 & 0 \end{pmatrix}$

Figure 3: A 3 state CTMC with its transition rate matrix

1.3 The PRISM Language Fundamentals

The two basic elements of the PRISM language are modules and variables. A model $\mathcal{M}$ is defined as the parallel composition of several interacting modules. Each module has a set of integer-valued, local variables with finite range. We will often refer to these as model variables. The set of model variables in $\mathcal{M}$ is noted $\mathrm{var}(\mathcal{M})$. The local state of a module at a particular time is given by a valuation of its local variables. A global state of the whole model is a valuation of the variables for all modules.

A module makes a transition from one local state to another by changing the value of its local variables. A transition of the whole model from one global state to another comprises transitions for one or more of its component modules. This can either be asynchronous, where a single module makes a transition independently, the others remaining in their current state, or synchronous, where two or more modules make a transition simultaneously.

The behavior of each module, i.e. the transitions it can make in any given state, are defined by a set of commands. Each command consists of a guard, which identifies a subset of the global state space, and one or more updates, each of which corresponds to a possible transition of the module. Intuitively, if the model is in a state satisfying the guard of a command then the module can make the transitions defined by the updates of that command. The probability that each transition will be taken is also specified by the command. The precise nature of this information depends on the type of model being described.

As an example, we consider a description of the 4 state DTMC from figure 1. This is shown in figure 4. The first line identifies the model type, in this case a DTMC. The remaining lines define the modules which make up the model. For this simple example, only a single module m is required.

dtmc
module m
  $v : [0..3]$ **init** 0;
  $[\,]\ (v = 0) \rightarrow 1 : (v' = 1)$;
  $[\,]\ (v = 1) \rightarrow 0.5 : (v' = 0) + 0.3 : (v' = 2) + 0.2 : (v' = 3)$;
  $[\,]\ (v = 2) \rightarrow 1 : (v' = 2)$;
  $[\,]\ (v = 3) \rightarrow 1 : (v' = 3)$;
endmodule

Figure 4: An example of PRISM model file

The first part of a module definition gives its set of local variables, identifying the name, range and initial value of each one. In this case, we have a single variable v with range $[0..3]$ and initial value 0. Hence, the local state space of module m, and indeed the global state space of the whole DTMC, is $[\![0, 3]\!]$.

The second part of a module definition gives its set of commands. Each one takes the form "$[\,]\ g \rightarrow u;$", where g is the guard and u lists one or more updates. A guard is a predicate over all the variables of the model (in this case, just v). Since each state of the model is associated with a valuation of these variables, a guard defines a subset of the model state space. The updates specify transitions that the module can make. These are expressed in terms of how the values of the local variables would change if the transition occurred. In our notation, v' denotes the updated value of v, so "$v' = 1$" implies simply that v's value will change to 1.

Here, since $\mathcal{M}$ is a DTMC model, the likelihood of each possible transition being taken is given by a discrete probability distribution. The second command of m shows an example of this. There is a clear correspondence between this probability distribution and the one in state 1 of the DTMC in figure 1. With an MDP, the syntax would be the same, but several command whose guards are not disjoint would be allowed. In the case of a CTMC model, each command would be assigned rates instead of probabilities.

Commands can be labelled with a name l between square brackets: "$[l]\ g \rightarrow u$". Commands in different modules with the same label are synchronized. In this case, the probability (or the rate) of a synchronous transition is defined to be the product of its component probabilities (or rates). Two synchronized commands

$$[l]\ g \rightarrow \sum_{i=1}^{n} p_i : u_i \quad \text{and} \quad [l]\ g' \rightarrow \sum_{i'=1}^{n'} p'_{i'} : u'_{i'}$$

are equivalent to the command

$$[l]\ g \,\&\, g' \rightarrow \sum_{i=1}^{n} \sum_{i'=1}^{n'} p_i p'_{i'} : u_i \,\&\, u'_{i'}.$$

Thus, a model with synchronized commands can be described by an equivalent model without synchronizations. For this reason, we consider only models with no synchronized commands. In figure 4, every commands are asynchronous.

2 Multi-Terminal Binary Decision Diagrams

Model checking had shown itself to be successful on relatively small examples, but it quickly became apparent that, when applied to real-life examples, explicitly enumerating all the states of the model is impractical. The fundamental difficulty, often referred to as the state space explosion problem, is that the state space of models representing even the most trivial real-life systems can easily become huge.

One of the most well-known approaches for combating this problem is symbolic model checking. This refers to techniques based on a data structure called binary decision diagrams (BDDs). These are directed acyclic graphs which can be used to represent boolean functions $f : \mathbb{B}^n \rightarrow \mathbb{B}$. BDDs were introduced by Lee [Lee59] and Akers [Ake78] but became popular following the work of Bryant [Bry86], who refined the data structure and developed a set of efficient algorithms for their manipulation. In this report, we assume the reader is somewhat familiar with BDDs. If not, an introduction to BDDs can be found in [And97].

In terms of model checking, the fundamental breakthrough was made by McMillan. He observed that transition relations, which were stored explicitly as adjacency lists in existing implementations, could be stored symbolically as BDDs. Because of the reduced storage scheme employed by the data structure, BDDs could be used to exploit high-level structure and regularity in the transition relations.

In PRISM, established symbolic model checking techniques are expanded to the probabilistic case. In addition to algorithms based on graph analysis, probabilistic model checking requires numerical computation to be performed. While graph analysis reduces to operations on a model transition relation and sets of its states, numerical computation requires operations on real-valued matrices and vectors. For this reason, the most natural way to extend BDD-based symbolic model checking is to use multi-terminal binary decision diagrams (MTBDDs).

MTBDDs were first proposed in [CMZ+93] and then developed independently in [CFM+93] and [BFG+93]. MTBDDs extend BDDs by representing functions which can take values from an arbitrary set $\mathcal{D}$, not just $\mathbb{B}$, i.e. functions of the form $f : \mathbb{B}^n \to \mathcal{D}$. In the majority of cases, $\mathcal{D}$ is taken to be $\mathbb{R}$ and this is the policy we adopt here. Note that BDDs are in fact a special case of MTBDDs, in which $\mathcal{D} = \mathbb{B}$.

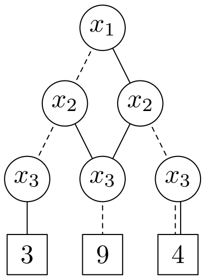
2.1 Definition

Let $\{x_1, \ldots, x_n\}$ be a set of distinct, boolean variables which are totally ordered as follows: $x_1 < \ldots < x_n$. An MTBDD $\mathcal{B}$ over $\bar{x} = (x_1, \ldots, x_n)$ is a rooted, directed acyclic graph. The vertices of the graph are known as nodes. Each node of the MTBDD is classed as either non-terminal or terminal. A non-terminal node b is labelled with a variable $\mathrm{var}(b) \in \bar{x}$ and has exactly two children, denoted $\mathrm{then}(b)$ and $\mathrm{else}(b)$. A terminal node b' is labelled by a real number $\mathrm{val}(b')$ and has no children. We will often refer to terminal and non-terminal nodes simply as terminals and non-terminals, respectively.

The ordering $<$ over the boolean variables is imposed upon the nodes of the MTBDD. For two non-terminals, b_1 and b_2, if $\mathrm{var}(b_1) < \mathrm{var}(b_2)$, then $b_1 < b_2$. If b_1 is a non-terminal and b_2 is a terminal, then $b_1 < b_2$. We require that, for every non-terminal b in an MTBDD, $b < \mathrm{else}(b)$ and $b < \mathrm{then}(b)$. The boolean variable ordering for an MTBDD over $\bar{x}$ is noted by a list of variables: $\omega = [x_1, \ldots, x_n]$.

Figure 5(a) shows an example of an MTBDD. The nodes are arranged in horizontal levels, one per boolean variable. The variable $\mathrm{var}(b)$ for a non-terminal b is given by its label. The two children of a node b are connected to it by edges, a solid line for $\mathrm{then}(b)$ and a dashed line for $\mathrm{else}(b)$. Terminals are drawn as squares, instead of circles, and are labelled with their value $\mathrm{val}(b)$. For clarity, we omit the terminal with value 0 and any edges which lead directly to it.

(a)



(b)

x_1	x_2	x_3	$f_{\mathcal{B}}$
0	0	1	3
0	1	0	9
1	1	0	9
1	0	0	4
1	0	1	4
otherwise			0

Figure 5: An MTBDD $\mathcal{B}$ and its function $f_{\mathcal{B}}$.

An MTBDD $\mathcal{B}$ over variables $\bar{x} = (x_1, \ldots, x_n)$ represents a function $f_{\mathcal{B}}(x_1, \ldots, x_n) : \mathbb{B}^n \to \mathbb{R}$. The value of $f_{\mathcal{B}}(x_1, \ldots, x_n)$ is determined by tracing a path in $\mathcal{B}$ from the root node to a terminal, for each non-terminal b, taking the edge to $\mathrm{then}(b)$ if $\mathrm{var}(b)$ is 1 or $\mathrm{else}(b)$ if $\mathrm{var}(b)$ is 0. The function represented by the MTBDD in figure 5(a) is shown in figure 5(b).

The reason that MTBDDs can often provide compact storage is because they are stored in a reduced form: if two nodes b and b' are identical, i.e. if

$$\mathrm{var}(b) = \mathrm{var}(b') \quad \text{and} \quad \mathrm{then}(b) = \mathrm{then}(b') \quad \text{and} \quad \mathrm{else}(b) = \mathrm{else}(b')$$

then only one copy of the node is stored. We refer to this as sharing of nodes.

In this report we assume that all MTBDDs are fully reduced in this way. Under this assumption, and for a fixed ordering of boolean variables, the data structure can be shown to be canonical, meaning that there is a one-to-one correspondence between MTBDDs and the functions they represent:

$$\mathcal{B} = \mathcal{B}' \quad \text{iff} \quad f_{\mathcal{B}} = f'_{\mathcal{B}}.$$

2.2 Variable Ordering

The size of an MTBDD $\mathcal{B}$ is defined as the number n of nodes contained in the data structure. This is particularly important because it affects both

- The total amount of memory required to store $\mathcal{B}$, directly proportional to its number of nodes n.
- The time complexity of operations on $\mathcal{B}$, typically proportional to n.

An important consideration from a practical point of view is variable ordering, the size of an MTBDD representing a given function being extremely sensitive to the ordering of its boolean variables.

Let us consider a boolean function with m variables $f(x_1, \ldots, x_m) : \mathbb{B}^m \to \mathbb{R}$. The size of an MTBDD representing f is in $\mathcal{O}(m)$ at the best and in $\mathcal{O}(2^m)$ in the worst case, depending upon the ordering of boolean variables $x_1, \ldots, x_m$. This can be proved analyzing the boolean function

$$f(x_1, \ldots, x_p, y_1, \ldots, y_p) = \bigwedge_{i=1}^{p} (x_i = y_i).$$

If constructed with the variable ordering (a) $[x_1, \ldots, x_p, y_1, \ldots, y_p]$, the MTBDD representing f contains $3 \cdot 2^p - 2 = \mathcal{O}(2^p)$ nodes. If we choose to use the ordering (b) $[x_1, y_1, \ldots, x_p, y_p]$ instead of (a), the MTBDD will contain only $3p+1 = \mathcal{O}(p)$ nodes. Figure 6 shows the MTBDDs constructed with orderings (a) and (b), and $n = 2$.

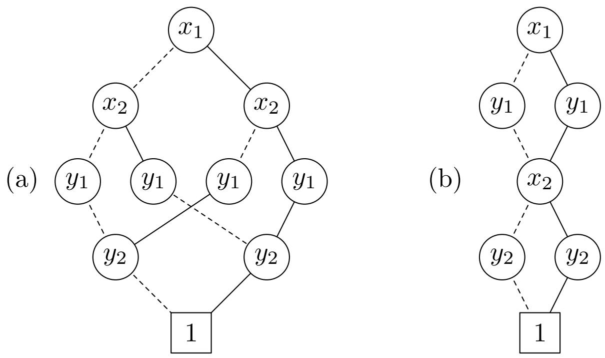


Figure 6: Two MTBDDs representing the same function, with different orderings

It is of crucial importance to care about variable ordering when applying this data structure in practice. The problem of finding the best variable ordering is NP-complete [BW96]. The best known algorithm, relying on a dynamic programming approach, has a time complexity $\mathcal{O}(n^2 3^n)$ where n is the number of boolean variables to order [FS87]. In practice, it cannot be used for problems with more than approximately 16 boolean variables, which is in practice far too limiting. Furthermore, for any constant $c > 1$, it is even NP-hard to compute a variable ordering resulting in a MTBDD with a size that is at most c times larger than the optimal one [Sie02]. The consequence is that we absolutely need to use heuristics to tackle this problem. This discussed in sections 3.

2.3 MTBDD Model Construction

We now present ways in which probabilistic models can be encoded as MTBDDs. This translation proceeds in three phases. The first task is to establish an encoding of the model's state space into MTBDD variables. Secondly, using the correspondence between PRISM and MTBDD variables provided by this encoding, an MTBDD representing the model is constructed from its description. Thirdly, we compute from the constructed model the set of reachable states. All unreachable states, which are of no interest, are then removed.

2.3.1 Encoding Into MTBDD Variables

In our case, a model's state space is defined by a number of integer-valued PRISM variables. To represent it in terms of MTBDD variables, PRISM's technique is to encode each model variable with its own set of MTBDD variables. For the encoding of each one, we use the standard binary representation of integers.

Consider a model with three PRISM variables, v_1 , v_2 and v_3 , each of range $\{0, 1, 2\}$. Our structured encoding would use 6 MTBDD variables, say $x_1, \ldots, x_6$, with two for each PRISM variable, i.e. x_1 , x_2 for v_1, x_3, x_4 for v_2 and x_5, x_6 for v_3. The state $(2, 1, 1)$, for example, would become $(1, 0, 0, 1, 0, 1)$.

An interesting consequence of this encoding is that we effectively introduce a number of extra states into the model. In our example, 6 MTBDD variables encode $2^6 = 64$ states, but the model actually only has $3^3 = 27$ states, leaving 37 unused. We refer to these extra states as dummy states. Happily, dummy states will not increase the MTBDD size, as MTBDDs contain no nodes to encode irrelevant information.

Compared to other encoding techniques, this scheme generally leads to smaller MTBDDs [Par02]. Two other important advantages result from the close correspondence between PRISM variables and MTBDD variables. Firstly, it facilitates the process of constructing an MTBDD, i.e. the conversion of a description in the PRISM language into an MTBDD representing the corresponding model. Since the description is given in terms of PRISM variables, this can be done with an almost direct translation. Secondly, we find that useful information about the model is implicitly encoded in the MTBDD. When using PRISM, the atomic propositions used in PCTL or CSL specifications are predicates over PRISM variables. It is therefore simple, when model checking, to construct an MTBDD which represents the set of states satisfying such a predicate by transforming it into one over MTBDD variables. With most other encodings, it would be necessary to use a separate data structure to keep track of which states satisfy which atomic propositions.

2.3.2 MTBDD Construction

The second step consists in constructing an MTBDD representing the set of possible transitions in the model from its description, using the correspondence between PRISM and MTBDD variables. We begin by considering the problem of representing DTMCs and CTMCs. Such a model is described by a real-valued square matrix P whose indices are global states $s \in S$ (that is to say a valuation of all the model variables), or equivalently by a function

$$f_P : S \times S \to \mathbb{R}.$$

Yet, given our encoding of model variables into boolean variables, described in section 2.3.1, we can represent each variable value in a global state s as a set of boolean variables values. Thus, a global state s can be seen as a tuple of boolean values $\tilde{s} \in \mathbb{B}^n$. P can then be described with a function

$$\tilde{f}_P : \mathbb{B}^n \times \mathbb{B}^n \to \mathbb{R} = \mathbb{B}^{2n} \to \mathbb{R}$$

which can be represented with an MTBDD. (If $\tilde{s}$ or $\tilde{s}'$ is a dummy state, $\tilde{f}_P(\tilde{s}, \tilde{s}')$ simply returns 0, stating the transition is impossible.)

To clarify it, we consider the 4 state DTMC in figures 1 and 4. As described in section 2.3.1, the unique model variable $v \in [\![0, 3]\!]$ is encoded with two boolean variables $(v_1, v_2) \in \mathbb{B}^2$ such as $v = 2v_1 + v_2$. Figure 7(a) represents the model transition matrix indexed by (v_1, v_2), and figure 7(b) the corresponding MTBDD. As previously, notations v',v_1', v_2' denote updated values of v, v_1 and v_2.

We can check that a path in the resulting MTBDD from the root node to a terminal node b corresponds to a couple $(\tilde{s}, \tilde{s}') \in S^2$ of global states, while $\mathrm{val}(b)$ is the probability label of the transition $\tilde{s} \to \tilde{s}'$ in the model. For example, the path $v_1 = v_1' = v_2 = v_2' = \top$, consisting in following only solid lines from the root node, leads to the probability 1. This path corresponds to the command $(v = 3) \to (v' = 3)$ (see figure 4). On the other hand, we can also notice there is no path $v_1 = v_1' = v_2 = v_2' = \bot$, as well as there is no command $(v = 0) \to (v' = 0)$ in the model.

Efficient construction of an MTBDD from a matrix is beyond the scope of this document. This topic is investigated in [CFM+93] and [BFG+93].

Representing MDPs with MTBDDs is more complex than DTMCs or CTMCs since the nondeterminism must also be encoded. An MDP is not described by a transition matrix over states, but by a function

(a) $\tilde{P} = \begin{array}{c|cccc|} & 0,0 & 0,1 & 1,0 & 1,1 \\ \hline 0,0 & 0 & 1 & 0 & 0 \\ 0,1 & 0.5 & 0 & 0.3 & 0.2 \\ 1,0 & 0 & 0 & 1 & 0 \\ 1,1 & 0 & 0 & 0 & 1 \\ \hline \end{array}$

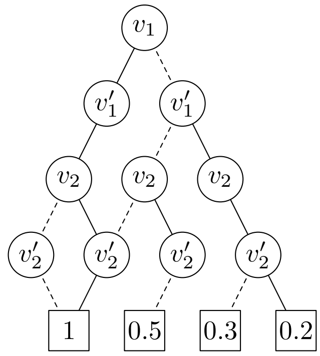
(b) 

Figure 7: A re-indexed transition matrix and the corresponding MTBDD

Steps mapping each state to a set of nondeterministic choices, each of which is a probability distribution over states.

Assuming, however, that the maximum number of nondeterministic choices in any state is m, and letting S denote the set of states of the MDP, we can reinterpret *Steps* as a function of the form $S \times [\![1, m]\!] \times S \rightarrow [0, 1]$. We have already discussed ways of encodings a model's state S into boolean variables. If we encode the set $[\![1, m]\!]$ in a similar fashion, we can consider *Steps* as a function mapping boolean variables to real numbers, and hence represent it as an MTBDD. Thus, we use as usual boolean variables to range over source and destination states, along with extra boolean variables to encode $[\![1, m]\!]$. These new variables are referred as nondeterministic variables, since they represent nondeterministic choices in the model. Their encoding scheme is addressed more in detail in [Par02].

2.3.3 Reachability

MTBDDs constructed in the previous section represent a model's transitions relation, but do not take into account reachability. The last step to convert a model description into an MTBDD representing it is to compute the set of reachable states and encode it into the MTBDD: all unreachable states, which are of no interest, have then to be removed.

Computing the set of reachable states of the model can be done via a breadth-first search of the state space starting with the initial state. First, an MTBDD representing the initial state is computed. Reachability is then performed iteratively using this MTBDD along with the transition MTBDD defined in section 2.3.2.

It should be noted that, in non-probabilistic model checking, determining the reachable states of a model may actually be sufficient for model checking. In our case, though, we usually need to perform probability calculations. Since these must be performed on the entire, reachable model, reachability is part of the construction phase.

Another observation we make here is that the removal of unreachable states from the model often causes an increase in the size of the MTBDD. This looks paradoxical, as both the number of states and the number of transitions in the model decrease. An explanation for this phenomenon is that the regularity of the model is also reduced, which decreases the ratio of shared nodes in the MTBDD. It is, however, impractical to retain the unreachable states since this would result in extra work being performed at the model checking stage.

3 Reducing MTBDDs Size

The size of an MTBDD is defined as the number of nodes contained in the data structure. This notion is particularly important for several reasons. First, it affects the amount of memory required for storage: as each MTBDD node is stored in memory, the memory footprint of an MTBDD is typically proportional to the number of nodes it contains. Second, time complexity of operations on MTBDDs also depends on their size, and so construction and model checking times too.

As a consequence, reducing the average size of MTBDDs in PRISM would have two benefits:

- First, it decrease time required for MTBDD construction and model checking.
- Second, this would allow to handle more complex models with the same computation capacities.

We have seen in section 2.2 that the size of an MTBDD representing a given model is extremely sensitive to the ordering of its boolean variables. Consequently, a way to reduce size of MTBDDs in PRISM is to generate boolean variable orderings leading to small MTBDDs. Unfortunately, the problem of finding the best variable ordering is NP-hard [BW96]. Thus, we will have to use heuristics to find good variable orderings, despite not the best ones, in a reasonable time.

We will focus only on the ordering of boolean variables deriving from model variables. We have briefly seen in section 2.3.2 that MTBDD representations of MDP models also contain extra boolean variables, known as nondeterministic variables, to represent nondeterministic behaviors. The position these variables should be given in orderings will not be addressed in this report. This problem is discussed in [Par02]. For our part, eventual nondeterministic boolean variables will be systematically placed in top of MTBDD orderings, before boolean representations of model variables.

3.1 Static Heuristics vs Dynamic Heuristics

There are several heuristic approaches to find fairly good variable orderings. Variable ordering heuristics can be divided into two groups: static and dynamic heuristics. Static heuristics compute an ordering of variables from the syntactic description of the model to be represented before MTBDD construction. Dynamic heuristics attempt to minimize MTBDD size by improving variable ordering after the MTBDD has been partially or completely constructed.

These two categories of heuristics rely on different principles. In static heuristics, we try to avoid most of computations necessary to an MTBDD construction. So, they involve less calculations and are usually faster to run than dynamic heuristics. But they lead to less good results, since they cannot take into consideration some fine phenomenon occurring in MTBDDs. They are also most of the time less general: a given static heuristic may be suited for a model or a category of models, but will have poor results with some others. On the other hand, dynamic heuristics are very general and do not depend on the category of the model being encoded into MTBDDs. They give usually better results at the expense of a slower computation time.

Both categories of heuristics are interesting, and they can be used in conjunction for a better efficiency: first we get a boolean variable ordering from the model thanks to a static heuristic, we can then start the MTBDD construction, and try to improve it during or after the construction phase with a dynamic heuristic. In PRISM, all the MTBDD stuff rely on the CUDD (Colorado University Decision Diagram) package[6] of F. Somenzi. It provides a rich set of dynamic reordering algorithms, some of them being slight variations of existing techniques and other having been developed specifically for it.

As many dynamic heuristics were provided directly by CUDD, I have been mainly working on static heuristics during my internship. My task was to create new heuristics or adapt existing BDDs heuristic to MTBDD representations of probabilistic models, to implement them into PRISM and to compare results quality and computation time of these heuristics. Consequently, the sequel of this document principally deals with static heuristics, although dynamic heuristics are briefly discussed in section 3.4

[6] http://vlsi.colorado.edu/~fabio/CUDD/

3.2 Static Heuristics

Most of static heuristics are based on the following simple observations. First, we notice that locating strongly dependent variables close to each other typically reduces MTBDDs size. Second, in a group of variables, variables appearing more important should be placed higher in the variable ordering. We will refer to these two key ideas with notations ① and ②.

These two ideas do not correspond to very precise rules. The dependence between two variables v_1, v_2 denotes the influence the value of v_1 may have on the value of v_2, and vice versa. If knowing the value of v_1 massively reduces the number of possible values for v_2, then v_2 is strongly dependent on v_1. This is difficult to evaluate precisely while trying to limit computation time, so it may be evaluated by the number of commands containing both v_1 and v_2, or the distance between these variables in the formulas of a model. The importance of a variable v is an indicator of the number of variables strongly dependent on v, that is to say the way its value influences the value of many other variables. Once again, as we try to minimize computation time, this will be interpreted by the number of commands containing v, or the number of variables related to v. In fact, the interpretation and the consideration given to ① and ② depends on the heuristic.

Figure 6 illustrates the idea ①. Consider what happens when we traverse the MTBDDs from top to bottom, trying to determine the value of the function for some assignment of the variables. Effectively, each node encodes the values of all the variables in levels above it. For example, in the second MTBDD, after two levels, we have established whether or not $x_1 = y_1$. If so, we will be positioned at the single node on the x_2 level. If not, we will have already moved to the zero constant node. In either case, from this point on, the values of x_1 and y_1 are effectively irrelevant, since in the function being represented, x_1 and y_1 relate only to each other. In the second MTBDD, however, there is a gap in the ordering between x_1 and y_1. After the first level, we "know" the value of x_1 but cannot "use" it until the third level. In the meantime, we must consider all possible values of x_2, causing a blow-up in the number of nodes required.

Many heuristics are performed using the model abstract syntax tree (AST), the syntactic structure of a given PRISM model $\mathcal{M}$. For this, we regard the PRISM commands as terms over the signature that contains constant symbols and the primed and unprimed versions of the model variables as atoms, and uses symbols like $+$, $*$, $=$, $<$, $\rightarrow$ as function symbols (the probabilities attached to updates are irrelevant and can simply be ignored). The node set in the AST $\mathcal{A}$ for model $\mathcal{M}$ consists of all commands in $\mathcal{M}$ and their subterms: the primed and unprimed versions of the variables of $\mathcal{M}$ and nodes for all function symbols that appear in the commands of $\mathcal{M}$ (like comparison operators, arithmetic operators, the arrows between guard and sum of updates in commands). Furthermore, $\mathcal{A}$ contains a special root node that serves to link all commands. The edge relation in the AST is given by the "subterm relation". That is, the leaves stand for the primed or unprimed variables (they are merged) or constants. (At the bottom level, leaves representing the same variable or constant are collapsed, so, in fact, the AST is a directed acyclic graph, and possibly not a proper tree.) The children of each inner node a represent the maximal proper subterms of the term represented by node a. The children of the root note are the nodes representing the commands. As an example, figure 8 represents the AST of the second command in PRISM model file in figure 4.

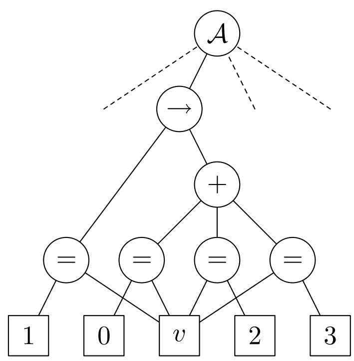


Figure 8: A PRISM model partial AST

Let a be an AST node, $\mathrm{var}(a)$ is defined as the set of $\mathcal{M}$ variables which appear in node a or its children:

$$\mathrm{var}(a) = \begin{cases} \{v\} & \text{if } a \text{ is a variable leave } v \\ \emptyset & \text{if } a \text{ is a constant leave} \\ \displaystyle\bigcup_{a' \text{ child of } a} \mathrm{var}(a') & \text{if } a \text{ is an inner node} \end{cases}$$

In particular, $\mathrm{var}(\mathcal{A}) = \mathrm{var}(\mathcal{M})$.

3.2.1 Boolean Variable Interleaving

A prominent idea is to interleave the boolean variables $v_1, \ldots, v_n, v'_1, \ldots, v'_n$ representing a model variable v values before an update and after an update. In the ordering, this leads to the sequence $[\ldots, v_1, v'_1, \ldots, v_n, v'_n, \ldots]$. This idea was first presented in [EFT91].

This is a consequence of ①. Indeed, each traversal through the MTBDD corresponds to a single transition in the model. In a typical transition, only a few PRISM variables will actually change value, the rest remaining constant. Since there is a direct correspondence between PRISM variables and MTBDD variables, this argument also applies at the MTBDD level: generally, each v'_i variable is most closely related to v_i. For instance, if we consider a model with two variables v, w, the following commands are equivalent:

$$\begin{array}{rlcl} & w = 0 & \rightarrow & w' = 1 \\ \Longleftrightarrow & w = 0 & \rightarrow & w' = 1 \ \&\ v' = v \\ \Longleftrightarrow & w = 0 & \rightarrow & w' = 1 \ \&\ v'_1 = v_1 \ \&\ \ldots\ \&\ v'_n = v_n \end{array}$$

So, boolean variables v_i, v'_1 should be grouped together in the ordering according to ①. Hence, the interleaved variable ordering is beneficial.

Figure 9 demonstrates the effect of this on some typical transition matrices. We use the polling system case study of [IT90], but results are the same for all PRISM examples. We present MTBDD sizes for both the interleaved and non-interleaved orderings. The difference is clear: it is simply not feasible to consider non-interleaved orderings.

N	States	MTBDD size	
		Interleaved	Non-interleaved
5	240	271	1,363
7	1,344	482	6,766
9	6,912	765	39,298
11	33,792	1,096	178,399
13	159,744	1,491	794,185

Figure 9: MTBDD sizes for interleaved and non-interleaved variable orderings

Considering these results, boolean variable interleaving will be systematically used. The rest of this section deals with heuristics providing variable orderings for model variables only. Coupled with this boolean variable interleaving, they can be used to construct complete boolean variable orderings. Considering only model variables also improves running time of most of heuristics, since the amount of variables to handle is smaller and there is no need to convert variables in model descriptions into sets of boolean variables before running heuristics.

We are going to study several general heuristics, which can be used with any category of models. Results are given farther in this section, and detailed results in appendix A. There also exist specialized heuristics, designed to handle models representing a precise structure, for example the very efficient Noack's algorithm for Petri nets [Noa99]. These heuristics give generally better results on the category of models they are conceived for than general heuristics. We have not implemented them in PRISM for two reasons. First, most of PRISM probabilistic models simply does not correspond to an usual structure but are rather a conjunction of probabilistic automatons with no obvious property. Second, it would be slow and difficult to infer which structure a PRISM model file possibly corresponds to, and convert it into this structure to be able to apply a suited heuristic, if such a heuristic exists. For this reason, only general heuristics are discussed next.

3.2.2 Greedy Algorithms

A first category of heuristics rely on greedy algorithms. The basic idea is to build a complete model variable ordering step by step, adding a model variable at every step to a partial ordering ω (initially empty), until ω contains all the model variables. To choose which variable should be added to ω at every step, a weight function $\text{weight}(v, \omega, \mathcal{M})$ is computed for every model variable $v \in \text{var}(\mathcal{M}) \setminus \omega$ not yet in ω. This function depends on the considered variable v, the content of the partial ordering ω when the function is called, and the model $\mathcal{M}$. The variable with the highest weight is then added to the partial ordering ω. The mechanism of a greedy algorithm is represented in figure 10. ($\cdot$ is the list concatenation operator.)

```
var ω = [ ]
while var(M) \ ω ≠ ∅ do
    for every model variable v ∈ var(M) \ ω do
        var w_v = weight(v, ω, M)
    end for
    var v_max = v ∈ var(M) \ ω such as w_v is maximal
    ω := ω · [v_max]
end while
return ω
```

Figure 10: Mechanism of a greedy algorithm

Clearly, the core of a greedy algorithm is its weight function. Its complexity also depends on this function. Let n be the number of model variables in $\mathcal{M}$ and suppose time complexity of $\text{weight}(v, \omega, \mathcal{M})$ is in $\mathcal{O}(f(\mathcal{M}))$, the overall time complexity of a greedy algorithm is

$$T_c = \mathcal{O}\left(\sum_{i=1}^{n} i \cdot f(\mathcal{M})\right)$$

where term "$i \cdot f(\mathcal{M})$" arises from the for loop and the sum from the while loop (time to select $v_{\max}$ is supposed negligible compared to $f(\mathcal{M})$).

If $f(\mathcal{M}) = \mathcal{O}(n)$, then $T_c = \mathcal{O}\left(\sum_{i=1}^{n} i \cdot n\right)$ so the global run time is $\mathcal{O}(n^3)$. If $\text{weight}(v, \omega, \mathcal{M})$ only depends on v, $\mathcal{M}$, and not on ω, the weight w_v of each model variable v has to be computed only once, which makes the for loop useless, so time complexity is only $\mathcal{O}(n \cdot f(\mathcal{M}))$.

Concerning memory footprint, this algorithm just requires to store a partial ordering ω whose length is limited by n, n other bytes to store at most n model variable weights in the *for* loop and eventually some room to run the weight function.

Presence In Commands

Let us present a first greedy heuristic. We consider a model $\mathcal{M}$ and note $\mathcal{C}$ the set of commands in $\mathcal{M}$ (i.e. the set of root node's children in the model AST). Then, the weight of a model variable v in a partial ordering ω is defined as the sum, for every command c in $\mathcal{C}$ containing v, of the number of variables in $\omega \cdot [v]$ which are also present in c. In other words,

$$\text{weight}(v, \omega, \mathcal{M}) = \sum_{\substack{c \in \mathcal{C} \\ x \in \text{var}(c)}} |(\omega \cdot [v]) \cap \text{var}(c)|$$

We can show this algorithm respects ideas ① and ② leading to a good variable ordering. Indeed, considering a model variable v and a partial ordering ω, there are two conditions so that $\text{weight}(v, \omega, \mathcal{M})$

returns a high value. First, v should appear in the same commands as many variables which are yet in the partial ordering, in order to maximize the intersection term. As a consequence, close variables are kept together in the ordering, which corresponds to ①. Second, the weight of a model variable v depends on the number of commands containing it, so important variables, which appear in a lot of commands, have a more important weight and tends to be placed first in the ordering, with regards to ②.

Fan-in Heuristic

A well-known BDD variable ordering algorithm is fan-in heuristic [MWB88], by Malik et al. This heuristic was primarily designed for logic circuits, but can be adapted to MTBDDs and probabilistic models rather easily and still give good results. We study here this later variant.

Fan-in heuristic relies on walks on a model's AST $\mathcal{A}$. It consists in two steps. First, a breadth-first search is performed, starting from the leaves in $\mathcal{A}$ (i.e. variables and constant symbols), which labels all nodes of the tree with the maximum distance to a leave node. The label $l(a)$ of a node a is formally defined by:

$$l(a) = \begin{cases} 0 & \text{if } a \text{ is a leave} \\ 1 + \max\{l(a') \,|\, a' \text{ is a child of } a\} & \text{if } a \text{ is an inner node} \end{cases}$$

The second step of the heuristic is to perform a depth-first search of variable leaves, starting at the root node, with the additional property that the depth-first search order in each node a that is visited is according to a descending ordering of the label values of its children. The visiting order of the variables then yields a promising variable ordering for the model's MTBDD.

Figure 11(a) represents the AST of command $x = y \rightarrow x' = y * z$. Each node a is labelled by $l(a)$. We can check that terminal nodes are labelled by 0, the star node by 1, because its two children are terminal nodes, the rightmost equal node by $2 = 1 + \max\{0, 1\}$, etc. Once the AST is labelled, it can be traversed to get a variable ordering. With respect to fan-in heuristic traversal rule, starting at the arrow node, the rightmost equal node has to be explored first since its label is greater than its brother's, then the star node, which leads to the ordering $[y, z, x]$ or $[z, y, x]$.

Fan-in heuristic is based on the assumption that variables which are accessed via longer paths are more important, and so, to respect the principle ②, have to be ordered first. As ordering construction relies on a depth-first AST traversal, this algorithms also groups close variables together, which corresponds to the first idea ①. It was primarily designed to be efficient for models based on logic circuits, but results are rather good for most of PRISM models.

Another good point, this algorithm just require to traverse twice the model AST and is consequently very fast.

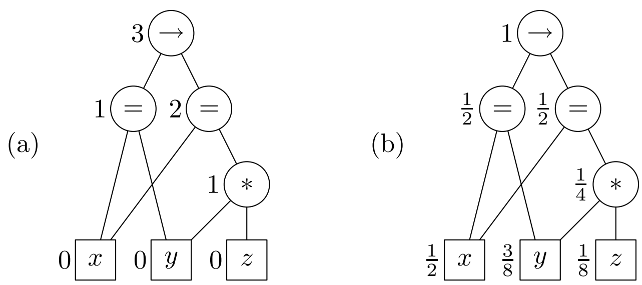


Figure 11: A command AST labelled by fan-in and weight heuristics

Weight Heuristic

Another ordering technique is given by the weight heuristic [MIY90]. It relies on an iterative approach that assigns weights to all nodes in the AST $\mathcal{A}$ and in each iteration the variable with the highest weight is the next in the variable ordering. This variable as well as any node that cannot reach any other variable is then removed from $\mathcal{A}$ and the next iteration yields the next variable in the ordering. (We suppose here that initially the leaves representing constants are removed from the AST.) In each iteration the weights are obtained as follows. We start with the root node and assign weight 1 to it and then propagate the weight to the leaves by means of the formula:

$$l(a) = \begin{cases} 1 & \text{if } a \text{ is root} \\ \displaystyle\sum_{a' \text{ father of } a} \frac{l(a')}{\text{number of children of } a'} & \text{if } a \text{ is not root} \end{cases}$$

(As at the at the bottom level, leaves representing the same variable or constant are collapsed, the AST is a direct acyclic graph and possibly not a proper tree, so an outer node may have several fathers.)

Figure 11(b) represents the AST of command $x = y \to x' = y * z$, before any node deletion occurred. Each node a is labelled by $l(a)$ as defined above. We can check the root node is labelled by 1, its two children by $\frac{1}{2}$, the star node by $\frac{1}{4} = \frac{1/2}{2}$ because its father is labelled by $\frac{1}{2}$ and has two children, etc. The variable with the highest weight is x so the resulting ordering starts with variable x. To pursue the ordering computation, x node should be removed as well as both edges leading to it, and labels of remaining nodes should be recomputed. Finally, the resulting ordering with weight heuristic is $[x, y, z]$.

Using this heuristic requires to delete parts of the AST and then to label some remaining nodes again each time a variable is added to the ordering. Weight heuristic is by consequence slower than fan-in heuristic.

3.2.3 Variable graphs

Another approach is to represent some properties of a probabilistic model $\mathcal{M}$ with a graph. A model variable graph is defined as a complete, undirected edge-labeled graph $G = (V, E, l)$ such as:

- Vertices V are model variables: $V = \text{var}(\mathcal{M})$.
- $E = V \times V$ is the set of graph edges. An edge is a couple of model variables.
- $l : E \to \mathbb{R}$ is a label function.

Variable graphs are simplified descriptions of models. The label function l maps an edge, that is to say a pair of model variables to a value, thus representing a relation on variables. Graph algorithms can then be used to find a hamiltonian path in the graph, that is to say a complete model variable ordering.

TSP Heuristics

A first heuristic consists in defining $l(v_1, v_2)$ as the distance between model variables v_1, v_2 in the model's AST $\mathcal{A}$. There are several ways to define a distance between two variables. We can envisage to use the length of the shortest path p in $\mathcal{A}$ linking v_1 and v_2, but results are not satisfying. Indeed, v_1 and v_2 can be very close in a given sub-formula of model description and quite distant anywhere else, in which case v_1 and v_2 should not be considered that close in the graph. It is far better to consider an average distance between v_1 and v_2.

The definition which gave the best results is the following. Considering two model variable v_1 and v_2, let $\mathcal{P}_1$ (respectively $\mathcal{P}_2$) the set of paths $(\mathcal{A} \to \cdots \to v_1)$ (respectively $(\mathcal{A} \to \cdots \to v_2)$) in the AST starting from root node and leading to the leave v_1 (respectively to v_2). Let

$$p_1 = (a_1 = \mathcal{A} \to a_2 \to \cdots \to a_m = v_1) \in \mathcal{P}_1 \quad \text{and} \quad p_2 = (b_1 = \mathcal{A} \to b_2 \to \cdots \to b_n = v_2) \in \mathcal{P}_2$$

be two such paths, the distance d between p_1 and p_2 is given by

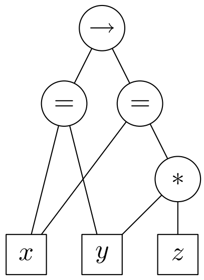
$$d(p_1, p_2) = (m - k) + (n - k) \quad \text{where} \quad k = \max\{i \in [\![1, \min(m, n)]\!] \mid a_i = b_i\}$$

that is to say the length of the shortest AST path joining v_1 and v_2 whose all nodes are in $p_1 \cup p_2$. (As both p_1 and p_2 start with the root node, such a path always exists.) The distance between v_1 and v_2 is then defined by

$$l(v_1, v_2) = \frac{1}{|\mathcal{P}_1| \cdot |\mathcal{P}_2|} \cdot \sum_{\substack{p_1 \in \mathcal{P}_1 \\ p_2 \in \mathcal{P}_2}} d(p_1, p_2).$$

Once the model graph is constructed, we want to find a model variable ordering $\omega = [v_1, \ldots, v_n]$, i.e. a walk on the graph, which minimizes the total variable distance $\sum_{i=1}^{n-1} l(v_i, v_{i+1})$ given by the sum of distances between adjacent variables in ω. This is simply a traveling salesman problem (TSP) instance. As TSP is an NP-complete problem, we do not try to get the best solution using an exact method (this would be slow, destructive in regard to our goals). Instead, we use a TSP heuristic to find a fairly good hamiltonian path considering the overall distance criterion. There are many TSP heuristics, most of them consisting in performing random permutations in a default solution while trying to minimize the overall distance. They may give a different solution each time they are called on a given graph, leading to different orderings and of course different MTBDD sizes. From a certain quality of TSP solutions, there is no clear correlation between the quality of a solution and the size of resulting MTBDD, which legitimates *a posteriori* the use of TSP heuristics.

Figure 12 represents the AST of command $x = y \rightarrow x' = y * z$ and the associated variable graph. After distance computation, we get $d(x, y) = d(y, z) = 3.5$ and $d(x, z) = 4$. TSP solver then returns ordering $[x, y, z]$ or $[z, y, x]$, in which the most distant variables x and z are separated by y.

(a) 

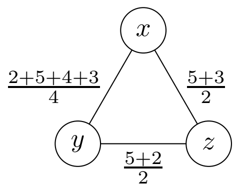
(b) 

Figure 12: A variable graph labelled by distances

The assumption in this heuristic is that related variables are close in the model AST. So, finding a walk which minimizes the global distance helps providing an ordering such as related variables are close together.

Let $\mathcal{M}$ be a model and $\mathcal{C}$ the set of commands in $\mathcal{M}$. An alternative TSP-based heuristic consists in using the label function

$$l(v_1, v_2) = |\{c \in \mathcal{C} \mid v_1 \in \mathrm{var}(c) \wedge v_2 \in \mathrm{var}(c)\}|$$

and then finding a path as long as possible with a TSP solver. An advantage of this technique compared to the previous heuristic is that labelling is far fastest: there is no need to traverse several times the model AST, but just to collect the variable set of each command. Here, the underlying assumption is that variables appearing in the same commands are strongly related and should be grouped together in the ordering.

A default of these TSP-based heuristics is that they cannot take into consideration absolute position of model variables in orderings. For example, two reversed paths $\omega_1 = [v_1, \ldots, v_n]$ and $\omega_2 = [v_n, \ldots, v_1]$ correspond to the same overall distance in an undirected variable graph, and thus have the same quality according to a TSP heuristic criterion. In other words, the second key idea ② is simply ignored. This problem can be tackled with metrics, which also take advantage of these heuristics' nondeterminism (see section 3.3).

Clusters

It may also be interesting, in the case of a large model with lots of variable, to clusterize the variable graph $G = (V, E, l)$, that is to say to partition the node set V into subsets $V_1, \ldots, V_m$ such as

$$\forall i \in [\![1, m]\!], V_i \neq \emptyset \quad \text{and} \quad V = \biguplus_{i=1}^{m} V_i.$$

Typically, each subset V_i contains a set of closely related model variables. Once graph nodes are partitioned, we have to find partial variable orderings ω_i for each cluster V_i, and also an ordering of hypergraph clusters $[V_{i_1}, \ldots, V_{i_m}]$. The complete model variable ordering is then given by $\omega = \omega_{i_1} \cdot \ldots \cdot \omega_{i_m}$. For problems with a huge number of variables, we can envisage to perform several clustering steps: each cluster V_i would be split into sub-clusters $v_{i_1}, \ldots, v_{i_n}$, and this process could possibly be iterated.

Graph clustering requires to make new choices. First, the clusterization algorithm and its parameters (for most of algorithms, we have to set how many clusters should be created). Second, an algorithm to construct an ordering for clusters. And finally, ordering technique(s) to use inside clusters. To address the first issue, there are many clustering algorithms. I had good results with the Markov Cluster Algorithm (MCL) [vD00], a fast and scalable unsupervised cluster algorithm for graphs based on simulation of stochastic flow in graphs. This algorithm does not require to be parametrized with the number of clusters to create. Concerning the second point, we can draw our inspiration from the usual ordering techniques, considering clusters as meta-variables, gathering properties of contained variables. Some cluster-based ordering techniques are detailed in [NW07].

In PRISM however, most of models are composed of a relatively small number of variables, typically between 5 and 100, which are deeply interdependent. Consequently, variable graphs are rather isomorphic, and clustering algorithms have difficulties to find relatively independent, medium-sized substructures in a model variable graph. This approach, despite being potentially fast and efficient for very large, deeply structured models (see [NW07]), is less effective than previous approaches on most of PRISM files.

Hypergraphs

The idea of variable graphs can also be extended with hypergraphs. Hypergraphs are a generalization of graphs, where edges can connect any number of vertices (and not only two as in graphs). Formally, a hypergraph H is a pair $H = (V, E)$ where V is as for graphs a set of nodes, and E a set of non-empty subsets of V called hyperedges or links. Therefore, E is a subset of $\mathcal{P}(V) \setminus \emptyset$, where $\mathcal{P}(V)$ is the power set of V. While graph edges are pairs of nodes, hyperedges are arbitrary sets of nodes, and contain an arbitrary number of nodes.

As for model graphs, an hypergraph used to study a model $\mathcal{M}$ satisfies $V = \text{var}(\mathcal{M})$. Its hyperedges are also labelled with a label function l, which maps hyperedges (i.e. sets of model variables) to values. The advantage of using hypergraphs rather than graphs is the possibility to represent a relation on not only two but any number of variables, and so to construct a more precise representation of a model. Hypergraphs also provide a more general point of view on models, since it is possible to encode information concerning large sets of model variables.

Once a model's hypergraph representation is constructed, specific algorithms on hypergraphs are used to derive a variable ordering from it. A technique using hypergraphs is the MINCE (min-cut, etc.) heuristic [AMS04]. It relies on min-cut algorithm, used to split a hypergraph into two subgraphs while minimizing the sum of hyperedge labels connecting vertices in different partitions. First, for every set of variables (i.e. hyperedge) $e \in E$, its average span (formally defined in section 3.3.1) in the model AST is computed. This value defines $l(e)$. Hypergraph vertex set is then recursively bisected via min-cut linear placement, leading to a variable ordering.

Unfortunately, because of the potentially large number of hyperedges, hypergraphs construction is usually slower and they require more memory for storage. As for clusters, the limited size of PRISM models makes hypergraphs refinements less useful. Furthermore, metrics (see section 3.3) can take into account the same information as hypergraphs.

3.2.4 Results

To compare different heuristics, we have been running PRISM on a set of test files. For each heuristic, we show

- The resulting MTBDD size.
- The heuristic running time: this is time to get a model variable ordering.
- The MTBDD construction time, including reachability computation time.
- The model checking time.

We present here the average results for many models, so possible disparities between models cannot be seen. For more detailed results, see appendix A.

The following ordering techniques are compared:

1. Random model variable ordering (MTBDD variables are interleaved).
2. Default PRISM variable ordering, corresponding to the order in which variable are defined in PRISM files.
3. TSP heuristic considering distances between variables (see section 3.2.3).
4. TSP heuristic considering the number of commands containing two variables (see section 3.2.3).
5. Presence greedy heuristic (see section 3.2.2).
6. Weight greedy heuristic (see section 3.2.2).
7. Fan-in greedy heuristic (see section 3.2.2).

Results are displayed on figure 13. Each line corresponds to an ordering technique.

Technique	MTBDD size	Ordering (ms)	Construction (ms)	Model checking (ms)
1	39,863	1	2,205	36,739
2	12,190	1	699	27,086
3	9,489	1,251	651	29,302
4	8,174	49	910	26,595
5	19,790	58	1,189	23,466
6	18,965	303	1,171	23,296
7	9,002	88	519	17,215

Figure 13: Results for static heuristics

We can see that the default PRISM variable ordering is far better than the random order, since local variables used in a module are defined inside it: in that sense, PRISM default order respects ①. Most of heuristics are better than default. Heuristics 5 and 6 seems to give bad results, but this is an average number: if we look at more detailed results, we see they can construct interesting orderings for CTMCs, but are not suited for MDPs.

3.3 Metric Optimization Heuristics

In this section, we study another category of variable ordering techniques: metric optimization heuristics. A metric μ is a function taking as argument an ordering ω and returning a value $x \in \mathbb{R}$.

$$\mu : \begin{cases} \mathfrak{S}(\mathrm{var}(\mathbb{M})) & \rightarrow & \mathbb{R} \\ \omega & \mapsto & x \end{cases}$$

μ is used to evaluate quality of orderings, i.e. their ability to lead to small MTBDDs model representations. A metric optimization heuristic seeks to minimize (or maximize) its metric function μ. It consists in comparing several variable orderings in a comparison set $\Omega = \{\omega_1, \dots, \omega_n\}$ with μ, and selecting $\omega \in \Omega$ which minimizes (or maximizes) μ, in other words the best ordering in Ω according to μ.

Metrics are not constructive heuristics, in that sense they are not used to construct variable orderings from a model description. This allow more freedom while defining metrics: we just have to set a criterion to evaluate quickly quality of orderings while, unlike constructive heuristics, there is no need to care on how construct orderings well satisfying this criterion.

However, we have to use constructive heuristics to build the comparison set $\Omega = \{\omega_1, \dots, \omega_n\}$. It is possible to use a number of different heuristics to get n variable orderings, but this may be slow since each heuristic require some initial computations to be initialized, e.g. construct a graph or label AST nodes. Instead, we can notice some heuristics are not deterministic: for example, a TSP-based heuristic may give different results for the same model if the underlying TSP solver is randomized. With a nondeterministic heuristic, we can quickly construct several different orderings since there is only one heuristic initialization. Thus, using a nondeterministic heuristic is a good idea to compute Ω rapidly.

Naturally, the metric definition is of crucial importance in a metric optimization heuristic. The chosen metric should be defined so as to capture enough relevant information about the model under consideration to prove applicable in establishing good variable orders.

3.3.1 Variable Span

Most of metrics rely on model variable span. To define it, we consider a probabilistic model $\mathcal{M}$ with $n \geq 1$ variables $v_1, \dots, v_n$. Let a be an AST node of $\mathcal{M}$ such as $\mathrm{var}(a) \neq \emptyset$ and $\omega = [v_1, \dots, v_n]$ a variable ordering for $\mathcal{M}$. We define $\mathrm{top}_\omega(a)$ as the index of the topmost occurrence of a variable from $\mathrm{var}(a)$ in the ordering ω. Similarly, $\mathrm{bot}_\omega(a)$ is the index of the bottommost occurrence of an a variable in ω. Formally,

$$\mathrm{top}_\omega(a) = \min\{i \in [\![1, n]\!] \mid v_i \in \mathrm{var}(a)\}$$
$$\mathrm{bot}_\omega(a) = \max\{i \in [\![1, n]\!] \mid v_i \in \mathrm{var}(a)\}$$

The span of a variable set $V = \{v_{i_1}, \dots, v_{i_m}\} \subseteq \mathrm{var}(\mathcal{M})$ in an ordering ω is the length of the shortest sublist $\tilde{\omega}$ in ω such as $V \subseteq \tilde{\omega}$ (if $V = \emptyset$, this is zero). The span of an AST node a is given by the span of its variables $\mathrm{var}(a)$. An equivalent definition is

$$\mathrm{span}_\omega(a) = \begin{cases} \mathrm{bot}_\omega(a) - \mathrm{top}_\omega(a) + 1 & \text{if } \mathrm{var}(a) \neq \emptyset \\ 0 & \text{if } \mathrm{var}(a) = \emptyset \end{cases}$$

For example, we consider a model $\mathcal{M}$ whose variables are $\{v_1, \dots, v_6\}$. Let $a = (v_2 = v_3 + v_5)$ be an AST node of $\mathcal{M}$, the variable set of a is $\mathrm{var}(a) = \{v_2, v_3, v_5\}$. In a variable ordering $\omega = [v_1, \dots, v_6]$, the smallest sublist containing $\mathrm{var}(a)$ is $\tilde{\omega} = [v_2, v_3, v_4, v_5]$, whose length is 4. Thus, $\mathrm{span}_\omega(a) = 4$. This is illustrated in figure 14. We can also remark that $\mathrm{bot}_\omega(a) - \mathrm{top}_\omega(a) + 1 = 5 - 2 + 1 = 4$.

$$\omega = [v_1, \underbrace{v_2, v_3, v_4, v_5}_{\tilde{\omega}}, v_6]$$

Figure 14: Span of a formula in an ordering

Span metric attempts to minimize the sum of spans of every AST node. The underlying idea is that close variables in the AST $\mathcal{A}$ are strongly related, and so should be grouped together in the ordering (principle ①). The value of an ordering ω as returned by the span metric is

$$\mathrm{SPAN}(\omega) = \sum_{a \in \mathcal{A}} \mathrm{span}_\omega(a)$$

3.3.2 Command Span

A variant is Normalized Event Span (NES) metric [CS06]. Let $\mathcal{C}$ the set of commands in $\mathcal{M}$, NES metric is defined as

$$\mathrm{NES}(\omega) = \sum_{c \in \mathcal{C}} \frac{\mathrm{span}_\omega(c)}{|\operatorname{var}(\mathcal{M})| \cdot |\mathcal{C}|}$$

The NES metric computes the average span of all commands (the span is then normalized by $|\operatorname{var}(M)|$) and its value is always between 0 and 1. A low NES indicates that the command spans are small, i.e., that most commands affect only model variables close to each other in the ordering ω.

This metric is faster to compute than span metric, since it does not require to take into account variable spanning of every AST node, but just of command nodes. In return, it captures less accurately information concerning variable proximity.

We generalize this concept by introducing the Weighted Event Span (WES) metric of moment i, WES_i for variable ordering ω as:

$$\mathrm{WES}_i(\omega) = \sum_{c \in \mathcal{C}} \left(\frac{\mathrm{top}_\omega(c)}{|\operatorname{var}(\mathcal{M})|/2} \right)^i \cdot \frac{\mathrm{span}_\omega(c)}{|\operatorname{var}(\mathcal{M})| \cdot |\mathcal{C}|}$$

We observe that WES_0 is exactly equivalent to NES. The WES_1 metric, instead, adds to it a component that reflects the location of the affected region, by assigning higher weights to locations closer to the top. This takes into account that operations applied to nodes in the lower portion of the MTBDD tend to have lower cost than those applied to higher nodes (principle ②). Therefore the span of an event is scaled by $\frac{\mathrm{top}_\omega(c)}{|\operatorname{var}(M)|/2}$, the relative position of the topmost level compared to the average level $|\operatorname{var}(M)|/2$. The weight of an event is thus between $(2/|\operatorname{var}(M)|)^i$ and 2^i , but the average over all events, if their tops were uniformly distributed over the MTBDD, should have an expected value of 1 for WES_1 , like for NES . For larger moments i, the emphasis on the location grows, as the weight multiplies in powers of 2, while strong clustering is relatively less important.

3.3.3 Using MTBDDs

Considering a model $\mathcal{M}$, it is possible to construct MTBDDs representing $\mathcal{M}$ with various variable orderings, and compare directly their size. We name it the MTBDD metric. Without surprise, this metric gives the best results but is very slow compared to other metrics, and is mostly used for comparison purpose or to find a very good variable ordering for a model, if we need to perform intensive model checking.

We have seen previously that the translation of a probabilistic model into an MTBDD proceeds in three phases. The first task is to establish an encoding of the model's state space into MTBDD variables. Secondly, using the correspondence between PRISM and MTBDD variables provided by this encoding, an MTBDD representing the model transitions is constructed from its description. Thirdly, we compute from the constructed model the set of reachable states.

An interesting observation we made is that the removal of unreachable states from the model often causes an increase in the size of its MTBDD. Intensity of this increase depends on the model and is highly variable, it may be negligible or represent an MTBDD growth up to ten times. This is despite the fact that both the number of states and the number of transitions in the model decrease. The explanation for this phenomenon is that regularity of the model is also reduced.

Thus, we can define two metrics relying on MTBDDs:

$$\begin{aligned} \mathrm{MTBDD_{nr}}(\omega) &= \text{size of MTBDD with ordering } \omega \text{ without reachability computation} \\ \mathrm{MTBDD_{r}}(\omega) &= \text{size of MTBDD with ordering } \omega \text{ with reachability computation} \end{aligned}$$

There is no reachability computation in $\mathrm{MTBDD_{nr}}$, so constructed MTBDDs are smaller. Consequently, this heuristic tends to be faster and to require less memory than $\mathrm{MTBDD_{r}}$ while still being very precise.

3.3.4 Results

To compare different metrics, we have been PRISM on the same set of files than in section 3.2.4. As above, for each heuristic we show the MTBDD size, the heuristic running time, the MTBDD construction time and the model checking time. We present here the average results obtained with the following ordering techniques:

1. Random model variable ordering (MTBDD variables are interleaved).
2. Default PRISM variable ordering.
3. SPAN metric on 50 orderings (see section 3.3.1).
4. $\text{WES}_0 = \text{NES}$ metric on 50 orderings (see section 3.3.2).
5. WES_1 metric on 50 orderings (see section 3.3.2).
6. WES_2 metric on 50 orderings (see section 3.3.2).
7. MTBDD_{nr} metric on 10 orderings (see section 3.3.3).
8. MTBDD_{r} metric on 10 orderings (see section 3.3.3).

Results are displayed on figure 15. The comparison set used by these metrics was generated with a TSP heuristic. As heuristics 7 and 8 are slower, they have been run on a smaller set of orderings.

Technique	MTBDD size	Ordering (ms)	Construction (ms)	Model checking (ms)
1	39,863	1	2,205	36,739
2	12,190	1	699	27,086
3	6,226	2,071	645	24,688
4	6,354	288	615	25,178
5	6,649	290	711	30,470
6	6,668	287	681	29,735
7	6,193	1,143	394	24,834
8	5,696	7,232	490	24,887

Figure 15: Results for metrics

These results confirm what was anticipated: heuristic 4 is faster than 3 but less precise, heuristic 7 is better than 6 but considerably slower to run, heuristics 6 and 7 are better than the others. Detailed results are exposed in appendix A.

3.4 Dynamic Heuristics

As we have seen in section 3.1, variable ordering heuristics can be divided into two groups: static and dynamic heuristics. Static heuristics compute variable orderings from the description of the model to be represented before MTBDD construction, while dynamic heuristics attempt to minimize MTBDD size by improving variable ordering after the MTBDD has been partially or completely constructed. Unlike static heuristics, dynamic heuristics are very general and do not depend on the category of models being encoded into an MTBDD. They give usually better results at the expense of a slower computation time.

In PRISM, all the MTBDD operations are based on the CUDD package, which provides a rich set of dynamic reordering algorithms. For this reason, I have mainly focused on studying and implementing static heuristics in PRISM. We are now going to mention very quickly some existing dynamic heuristics.

The key operation of dynamic heuristics is swapping two consecutive MTBDD variables, in attempt to decrease the MTBDD size. In efficient MTBDD implementations, each variable swap has time complexity proportional to the width of the MTBDD.

Sifting algorithm

A well-known dynamic reordering technique is Rudell's sifting algorithm [Rud93]. Sifting algorithm is based on finding the optimal position for a variable, assuming all other variables remain fixed. If there are n variables in the MTBDD (excluding the constant level which is always at the bottom), then there are n potential position for a variable, including its current position. Each variable is considered in turn and is moved up and down in the ordering so that it takes all possible positions. The best position is identified and the variable is returned to that position. Globally, the sifting algorithm requires $\mathcal{O}(n^2)$ swaps of adjacent levels in the MTBDD.

In CUDD, things are a bit more complicated. First, there is a limit on the number of variables that will be sifted. In addition, if the decision diagram grows too much while moving a variable up or down, the current movement is terminated before the variable has reached one end of the order.

There are also two important variants of this technique. First, the algorithm can be iterated to convergence. Second, group of variables can be aggregated according to some criterion (see [PSP94], [PS95] for example). Variables that become adjacent during sifting are tested for aggregation. If test result is positive, they are linked in a group. Sifting then continues with a group being moved, instead of a single variable.

Window permutation algorithm

Window permutation algorithm was presented by Fujita et al. [FMK91] and Ishiura et al. [ISY91]. It proceeds by choosing a level i in the MTBDD and exhaustively searching all $k!$ permutations of the k adjacent boolean variables starting at level i. This is done using $k! - 1$ pairwise exchanges followed by up up to $\frac{k(k-1)}{2}$ pairwise exchanges to restore the best permutation seen. This is then repeated starting from each level until no improvement in the MTBDD is seen.

Because the swap of two adjacent variables is efficient, the window permutation algorithm remains efficient for values of k as large as 4 or 5.

Other approaches

There exist other dynamic variable reordering heuristics, including an approach based on simulated annealing [BLW95], and a genetic algorithm [DBG95]. These methods are potentially very slow and are not widely used.

Conclusions

We have seen in the beginning of this report how PRISM represents different kinds of probabilistic models (DTMC, CTMCs, MDPs) with special data structures: multi-terminal binary decision diagrams, a variant of BDDs which allows to have any number of terminal nodes. This structure is interesting in terms of model checking, since it provides a compact storage of model descriptions and is suited to perform computations efficiently.

Then, we have studied different categories of static heuristics, adapted to probabilistic models, to try to reduce size of MTBDDs before they are constructed: greedy algorithms, which construct orderings step by step by adding a variable at every step, heuristics using a graph representation of models, and metrics to select in an ordering set the most promising one. Most of these heuristics where implemented in PRISM and achieve to improve the default PRISM variable ordering (see appendices). For example, with this new feature, we have been able to run PRISM on a large file which was previously causing a crash because of memory lack. In terms of implementation, I have added around 3000 lines in PRISM's Java source code to add the ordering feature.

It may be interesting to continue in this direction by using CUDD's dynamic heuristics in PRISM, which is not case yet. Unfortunately, this would be difficult to implement in PRISM current source code. As many PRISM models contains several instances of the same structure (for example a server and many identical clients), another idea would be to find an ordering on a similar, but smaller model, and to extend it to the full-sized model.

Appendix

A Detailed Results

Here are detailed results for many heuristics, tested on a large set of PRISM files. Files whose extension is `.pm` are DTMCs models, `.sm` are CTMCs and `.nm` are MDPs. A long dash ("–" symbol) indicates a heuristic always give the same result.

A.1 MTBDD size

Model `brp.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	3,446	571	2,191	4,699
default	3,018	468	1,995	4,030
distance graph	3,423	435	2,538	4,400
command graph	3,075	411	2,401	3,597
presence	3,134	–	–	–
weight	3,825	–	–	–
fan-in	2,633	–	–	–
SPAN	3,219	114	2,523	3,421
NES = WES_0	3,248	27	3,206	3,343
WES_1	3,196	180	2,592	3,506
WES_2	3,177	204	2,592	3,509
$MTBDD_{nr}$	2,613	226	2,353	3,418
$MTBDD_r$	2,475	83	2,248	2,635

Model `cluster.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	3,050	1,107	1,475	8,420
default	1,901	179	1,453	2,119
distance graph	1,639	10	1,619	1,658
command graph	1,840	230	1,229	2,154
presence	2,044	–	–	–
weight	2,111	–	–	–
fan-in	2,355	–	–	–
SPAN	1,793	273	1,200	2,107
NES = WES_0	1,883	195	1,394	2,143
WES_1	1,799	242	1,215	2,123
WES_2	1,846	267	1,200	2,158
$MTBDD_{nr}$	1,467	183	1,215	1,935
$MTBDD_r$	1,451	163	1,203	1,768

Model `coin4.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	5,397	2,673	1,901	14,626
default	1,798	308	1,402	2,432
distance graph	1,428	–	–	–
command graph	1,694	81	1,609	1,772
presence	2,454	–	–	–
weight	2,454	–	–	–
fan-in	2,336	–	–	–
SPAN	1,762	39	1,609	1,772
NES = WES_0	1,717	77	1,609	1,772
WES_1	1,609	–	–	–
WES_2	1,609	–	–	–
$MTBDD_{nr}$	1,609	–	–	–
$MTBDD_r$	1,609	–	–	–

Model `csma3_2.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	21,823	3,350	15,245	28,877
default	12,927	738	11,747	14,542
distance graph	22,984	2,573	18,661	26,316
command graph	17,836	2,115	16,000	23,292
presence	19,854	–	–	–
weight	20,445	–	–	–
fan-in	14,306	–	–	–
SPAN	16,766	886	15,400	18,535
NES = WES_0	16,261	706	15,458	17,473
WES_1	16,637	46	16,322	16,697
WES_2	16,615	95	16,267	16,697
$MTBDD_{nr}$	16,239	393	16,000	17,473
$MTBDD_r$	16,153	298	15,783	17,431

Model `dice.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	66	4	62	71
default	67	4	62	71
distance graph	62	–	–	–
command graph	62	–	–	–
presence	71	–	–	–
weight	71	–	–	–
fan-in	71	–	–	–
SPAN	62	–	–	–
NES = WES_0	62	–	–	–
WES_1	62	–	–	–
WES_2	62	–	–	–
$MTBDD_{nr}$	62	–	–	–
$MTBDD_r$	62	–	–	–

Model `two_dice.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	311	75	188	367
default	196	8	188	203
distance graph	194	–	–	–
command graph	194	–	–	–
presence	365	–	–	–
weight	365	–	–	–
fan-in	203	–	–	–
SPAN	194	–	–	–
NES = WES_0	194	–	–	–
WES_1	194	–	–	–
WES_2	194	–	–	–
$MTBDD_{nr}$	194	–	–	–
$MTBDD_r$	194	–	–	–

Model `two_dice_knuth.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	237	4	231	240
default	235	4	231	240
distance graph	231	–	–	–
command graph	231	–	–	–
presence	240	–	–	–
weight	240	–	–	–
fan-in	240	–	–	–
SPAN	231	–	–	–
NES = WES_0	231	–	–	–
WES_1	231	–	–	–
WES_2	231	–	–	–
$MTBDD_{nr}$	231	–	–	–
$MTBDD_r$	231	–	–	–

Model `dining_crypt5.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	15,578	4,545	6,659	26,824
default	7,854	2,384	4,169	11,703
distance graph	12,693	2,652	5,743	16,498
command graph	6,007	2,261	3,258	14,349
presence	10,778	–	–	–
weight	9,178	–	–	–
fan-in	4,077	–	–	–
SPAN	6,173	1,745	3,250	11,402
NES = WES_0	3,846	555	3,167	5,089
WES_1	3,880	511	3,258	5,088
WES_2	3,899	672	3,225	6,151
$MTBDD_{nr}$	4,109	908	3,132	7,175
$MTBDD_r$	3,492	161	3,203	3,797

Model `embedded.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	1,182	213	699	1,650
default	1,073	245	646	1,898
distance graph	695	27	668	739
command graph	907	217	612	1,503
presence	740	–	–	–
weight	879	–	–	–
fan-in	875	–	–	–
SPAN	743	132	571	1,117
NES = WES_0	1,001	242	636	1,470
WES_1	747	72	643	1,014
WES_2	779	117	633	1,301
$MTBDD_{nr}$	731	90	578	917
$MTBDD_r$	701	65	586	833

Model `fms.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	286,507	137,886	67,183	585,875
default	28,484	5,043	16,463	37,987
distance graph	32,198	8,287	14,859	54,009
command graph	26,559	8,306	14,594	57,223
presence	23,044	–	–	–
weight	27,286	–	–	–
fan-in	30,953	–	–	–
SPAN	19,598	4,529	14,334	27,014
NES = WES_0	19,304	5,240	13,833	27,847
WES_1	20,998	1,199	19,227	26,237
WES_2	21,428	2,026	19,353	33,221
$MTBDD_{nr}$	20,536	4,921	14,392	33,286
$MTBDD_r$	16,564	3,087	13,850	23,792

Model `leader4.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	26,402	4,482	17,492	35,703
default	14,767	3,529	10,813	20,176
distance graph	14,000	2,542	10,507	20,546
command graph	12,184	2,586	9,983	21,659
presence	26,379	–	–	–
weight	24,758	–	–	–
fan-in	14,100	–	–	–
SPAN	10,243	323	9,739	11,088
NES = WES_0	11,802	1,787	9,770	16,317
WES_1	11,833	1,458	9,770	14,903
WES_2	11,909	1,256	9,643	15,008
$MTBDD_{nr}$	10,892	892	9,861	14,169
$MTBDD_r$	10,175	390	9,564	11,381

Model `leader4_3.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	6,660	989	4,492	9,071
default	5,133	410	4,320	6,047
distance graph	6,844	883	5,355	8,771
command graph	6,960	1,787	3,762	10,650
presence	6,007	–	–	–
weight	4,929	–	–	–
fan-in	6,257	–	–	–
SPAN	6,870	1,389	4,233	8,959
NES = WES_0	6,642	1,661	3,994	10,428
WES_1	5,204	622	3,833	7,250
WES_2	5,029	512	3,852	6,039
$MTBDD_{nr}$	6,022	1,446	3,909	8,987
$MTBDD_r$	4,662	450	3,848	5,825

Model `kncal.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	2,554	184	2,308	2,986
default	2,563	203	2,308	2,995
distance graph	2,740	3	2,736	2,743
command graph	2,739	3	2,736	2,743
presence	2,542	–	–	–
weight	2,542	–	–	–
fan-in	2,736	–	–	–
SPAN	2,736	–	–	–
NES = WES_0	2,736	–	–	–
WES_1	2,736	–	–	–
WES_2	2,736	–	–	–
$MTBDD_{nr}$	2,743	–	–	–
$MTBDD_r$	2,736	–	–	–

Model `mc.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	1,435	121	1,214	1,640
default	1,397	125	1,214	1,592
distance graph	1,525	24	1,498	1,546
command graph	1,503	41	1,443	1,546
presence	1,546	–	–	–
weight	1,546	–	–	–
fan-in	1,386	–	–	–
SPAN	1,546	–	–	–
NES = WES_0	1,546	–	–	–
WES_1	1,443	–	–	–
WES_2	1,443	–	–	–
$MTBDD_{nr}$	1,494	15	1,443	1,498
$MTBDD_r$	1,443	–	–	–

Model `peer2peer5_4.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	3,590	1,261	1,170	5,903
default	6,065	–	–	–
distance graph	3,614	1,206	1,345	5,786
command graph	3,999	1,094	1,591	5,744
presence	4,740	–	–	–
weight	4,740	–	–	–
fan-in	6,065	–	–	–
SPAN	3,908	1,134	1,361	5,918
NES = WES_0	4,020	1,389	1,249	6,008
WES_1	3,478	1,126	1,120	5,537
WES_2	3,819	1,075	1,236	6,020
$MTBDD_{nr}$	2,025	632	826	3,527
$MTBDD_r$	1,927	511	1,165	2,983

Model `phil_lss3.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	10,113	1,722	7,037	13,277
default	7,940	568	7,037	9,166
distance graph	8,192	358	7,887	8,612
command graph	9,671	1,777	7,037	13,277
presence	12,710	–	–	–
weight	7,788	–	–	–
fan-in	9,550	–	–	–
SPAN	7,386	360	7,037	7,801
NES = WES_0	9,889	1,746	7,091	13,277
WES_1	9,979	1,635	7,037	13,277
WES_2	10,032	1,789	7,037	13,277
$MTBDD_{nr}$	8,637	735	7,664	10,049
$MTBDD_r$	7,507	500	7,037	9,166

Model `poll6.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	427	232	240	1,120
default	496	234	265	1,011
distance graph	370	113	265	680
command graph	406	127	277	725
presence	367	–	–	–
weight	390	–	–	–
fan-in	367	–	–	–
SPAN	319	42	265	371
NES = WES_0	319	46	265	384
WES_1	335	43	269	385
WES_2	318	45	265	385
$MTBDD_{nr}$	287	17	265	343
$MTBDD_r$	285	17	265	344

Model `beauquier3.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	127	18	77	154
default	142	9	129	153
distance graph	142	5	135	149
command graph	139	7	129	153
presence	141	–	–	–
weight	77	–	–	–
fan-in	129	–	–	–
SPAN	142	3	131	143
NES = WES_0	143	6	129	153
WES_1	143	6	129	153
WES_2	142	6	129	153
$MTBDD_{nr}$	131	2	129	135
$MTBDD_r$	132	3	129	135

Model `tandem.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	177	31	126	220
default	164	30	126	201
distance graph	161	37	126	201
command graph	161	23	126	201
presence	153	–	–	–
weight	126	–	–	–
fan-in	153	–	–	–
SPAN	201	–	–	–
NES = WES_0	166	–	–	–
WES_1	166	–	–	–
WES_2	166	–	–	–
$MTBDD_{nr}$	128	8	126	166
$MTBDD_r$	128	7	126	153

Model `wlan1.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	8,436	1,355	5,979	11,386
default	7,303	556	5,760	8,287
distance graph	9,295	860	7,571	12,631
command graph	7,376	64	7,283	7,477
presence	6,838	–	–	–
weight	5,730	–	–	–
fan-in	4,484	–	–	–
SPAN	7,477	–	–	–
NES = WES_0	7,477	–	–	–
WES_1	7,436	249	6,927	8,743
WES_2	7,530	386	6,927	8,794
$MTBDD_{nr}$	7,290	25	7,283	7,375
$MTBDD_r$	7,269	70	6,927	7,283

Model `wlan1_collide.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	11,242	1,816	6,922	15,262
default	9,309	758	7,635	11,218
distance graph	10,138	785	9,206	12,643
command graph	8,828	166	8,626	9,871
presence	7,276	–	–	–
weight	6,436	–	–	–
fan-in	6,880	–	–	–
SPAN	8,947	–	–	–
NES = WES_0	8,921	79	8,612	8,947
WES_1	9,018	680	8,552	12,138
WES_2	9,051	831	8,612	12,263
$MTBDD_{nr}$	8,739	52	8,721	8,947
$MTBDD_r$	8,722	24	8,612	8,743

Model `wlan1_time_bounded.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	34,305	5,826	23,068	46,988
default	28,972	4,171	22,619	38,696
distance graph	26,859	2,722	23,366	34,721
command graph	24,886	1,113	23,903	26,310
presence	36,126	–	–	–
weight	31,318	–	–	–
fan-in	17,778	–	–	–
SPAN	24,062	–	–	–
NES = WES_0	24,062	–	–	–
WES_1	27,129	2,301	25,103	33,293
WES_2	26,783	1,936	24,997	33,293
$MTBDD_{nr}$	23,936	40	23,903	23,988
$MTBDD_r$	23,922	36	23,881	23,988

Model `zeroconf.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	9,646	1,322	6,053	12,958
default	7,387	784	5,490	8,858
distance graph	9,772	1,371	5,661	12,741
command graph	7,202	1,160	5,357	10,350
presence	7,529	–	–	–
weight	7,132	–	–	–
fan-in	9,286	–	–	–
SPAN	7,010	670	6,092	8,359
NES = WES_0	7,230	800	5,447	8,846
WES_1	6,987	642	5,845	8,560
WES_2	7,154	587	5,629	8,249
$MTBDD_{nr}$	6,708	1,018	5,555	8,708
$MTBDD_r$	5,793	373	5,402	7,111

Model `mapk_cascade.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	426,313	210,919	115,569	1,181,070
default	71,297	54,569	20,274	258,048
distance graph	43,838	42,721	15,392	225,075
command graph	39,497	23,214	11,331	105,128
presence	297,053	–	–	–
weight	300,005	–	–	–
fan-in	34,064	–	–	–
SPAN	14,469	1,968	10,352	18,082
NES = WES_0	15,074	1,820	11,331	21,288
WES_1	16,628	3,231	11,648	25,640
WES_2	17,245	3,228	11,651	27,035
$MTBDD_{nr}$	19,170	6,046	11,916	39,493
$MTBDD_r$	16,401	3,013	11,916	23,844

Model `sprouty.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	117,550	58,245	43,393	336,180
default	84,271	44,355	18,068	246,159
distance graph	24,178	10,217	8,691	48,825
command graph	20,402	10,922	5,597	52,959
presence	22,615	–	–	–
weight	9,762	–	–	–
fan-in	53,773	–	–	–
SPAN	9,780	4,082	5,374	20,646
NES = WES_0	11,089	4,774	5,995	24,196
WES_1	14,349	4,096	8,356	33,422
WES_2	13,510	3,989	8,455	25,890
$MTBDD_{nr}$	8,839	2,656	5,681	16,662
$MTBDD_r$	8,375	1,851	5,517	13,099

Average result

Heuristic	Avg	Std. dev.	Min	Max
random	39,863	17,558	13,239	93,823
default	12,190	4,787	6,019	27,756
distance graph	9,489	3,113	5,830	20,048
command graph	8,174	2,308	5,175	14,927
presence	19,790	–	–	–
weight	18,965	–	–	–
fan-in	9,002	–	–	–
SPAN	6,226	708	5,157	7,688
NES = WES_0	6,354	846	5,186	8,218
WES_1	6,649	734	5,531	8,976
WES_2	6,668	761	5,526	9,035
$MTBDD_{nr}$	6,193	812	5,192	8,524
$MTBDD_r$	5,696	444	5,101	6,823

A.2 Heuristic Running Time

Time is displayed in milliseconds (ms).

Model `brp.pm`

Heuristic	Avg
random	1
default	1
distance graph	78
command graph	31
presence	42
weight	69
fan-in	17
SPAN	318
NES = WES_0	231
WES_1	231
WES_2	232
$MTBDD_{nr}$	399
$MTBDD_r$	2,686

Model `cluster.sm`

Heuristic	Avg
random	1
default	1
distance graph	32
command graph	27
presence	11
weight	48
fan-in	11
SPAN	334
NES = WES_0	255
WES_1	261
WES_2	258
$MTBDD_{nr}$	286
$MTBDD_r$	525

Model `coin4.nm`

Heuristic	Avg
random	0
default	0
distance graph	55
command graph	25
presence	15
weight	41
fan-in	26
SPAN	317
NES = WES_0	229
WES_1	225
WES_2	227
$MTBDD_{nr}$	241
$MTBDD_r$	2,416

Model `csma3_2.nm`

Heuristic	Avg
random	1
default	1
distance graph	9,999
command graph	92
presence	143
weight	439
fan-in	230
SPAN	1,968
NES = WES_0	390
WES_1	392
WES_2	372
$MTBDD_{nr}$	597
$MTBDD_r$	12,486

Model `dice.pm`

Heuristic	Avg
random	1
default	1
distance graph	30
command graph	28
presence	6
weight	8
fan-in	8
SPAN	203
NES = WES_0	203
WES_1	204
WES_2	202
$MTBDD_{nr}$	138
$MTBDD_r$	138

Model `two_dice.nm`

Heuristic	Avg
random	1
default	1
distance graph	28
command graph	16
presence	8
weight	13
fan-in	10
SPAN	189
NES = WES_0	188
WES_1	187
WES_2	186
$MTBDD_{nr}$	105
$MTBDD_r$	106

Model `two_dice_knuth.pm`

Heuristic	Avg
random	1
default	1
distance graph	56
command graph	32
presence	10
weight	24
fan-in	15
SPAN	213
NES = WES_0	211
WES_1	213
WES_2	212
$MTBDD_{nr}$	144
$MTBDD_r$	142

Model `dining_crypt5.nm`

Heuristic	Avg
random	0
default	0
distance graph	77
command graph	26
presence	24
weight	84
fan-in	17
SPAN	327
NES = WES_0	230
WES_1	233
WES_2	231
$MTBDD_{nr}$	224
$MTBDD_r$	700

Model `embedded.sm`

Heuristic	Avg
random	1
default	1
distance graph	34
command graph	30
presence	11
weight	51
fan-in	11
SPAN	335
NES = WES_0	259
WES_1	255
WES_2	257
$MTBDD_{nr}$	222
$MTBDD_r$	327

Model `fms.sm`

Heuristic	Avg
random	1
default	1
distance graph	137
command graph	37
presence	48
weight	123
fan-in	42
SPAN	378
NES = WES_0	239
WES_1	247
WES_2	251
$MTBDD_{nr}$	935
$MTBDD_r$	4,095

Model `leader4.nm`

Heuristic	Avg
random	1
default	1
distance graph	1,324
command graph	61
presence	121
weight	247
fan-in	84
SPAN	646
NES = WES_0	290
WES_1	287
WES_2	288
$MTBDD_{nr}$	505
$MTBDD_r$	2,298

Model `leader4_3.pm`

Heuristic	Avg
random	1
default	1
distance graph	259,063
command graph	137
presence	263
weight	4,432
fan-in	652
SPAN	39,534
NES = WES_0	701
WES_1	705
WES_2	697
$MTBDD_{nr}$	1,829
$MTBDD_r$	2,554

Model `knacl.sm`

Heuristic	Avg
random	1
default	1
distance graph	19
command graph	17
presence	6
weight	8
fan-in	6
SPAN	201
NES = WES_0	203
WES_1	202
WES_2	204
$MTBDD_{nr}$	128
$MTBDD_r$	142

Model `mc.sm`

Heuristic	Avg
random	1
default	1
distance graph	20
command graph	25
presence	6
weight	9
fan-in	11
SPAN	231
NES = WES_0	230
WES_1	228
WES_2	228
$MTBDD_{nr}$	201
$MTBDD_r$	223

Model `peer2peer5_4.sm`

Heuristic	Avg
random	1
default	1
distance graph	66
command graph	60
presence	32
weight	62
fan-in	9
SPAN	417
NES = WES_0	366
WES_1	371
WES_2	365
$MTBDD_{nr}$	359
$MTBDD_r$	391

Model `phil_lss3.nm`

Heuristic	Avg
random	1
default	1
distance graph	26,390
command graph	83
presence	122
weight	403
fan-in	365
SPAN	2,175
NES = WES_0	445
WES_1	448
WES_2	448
$MTBDD_{nr}$	484
$MTBDD_r$	2,729

Model `poll6.sm`

Heuristic	Avg
random	1
default	1
distance graph	58
command graph	43
presence	13
weight	46
fan-in	19
SPAN	415
NES = WES_0	304
WES_1	307
WES_2	285
$MTBDD_{nr}$	270
$MTBDD_r$	310

Model `beauquier3.nm`

Heuristic	Avg
random	1
default	1
distance graph	28
command graph	22
presence	7
weight	13
fan-in	7
SPAN	269
NES = WES_0	235
WES_1	234
WES_2	235
$MTBDD_{nr}$	158
$MTBDD_r$	155

Model `tandem.sm`

Heuristic	Avg
random	1
default	1
distance graph	19
command graph	24
presence	5
weight	8
fan-in	6
SPAN	243
NES = WES_0	246
WES_1	228
WES_2	229
$MTBDD_{nr}$	179
$MTBDD_r$	246

Model `wlan1.nm`

Heuristic	Avg
random	1
default	1
distance graph	4,172
command graph	63
presence	72
weight	331
fan-in	151
SPAN	448
NES = WES_0	264
WES_1	263
WES_2	262
$MTBDD_{nr}$	316
$MTBDD_r$	1,334

Model `wlan1_collide.nm`

Heuristic	Avg
random	1
default	1
distance graph	4,092
command graph	71
presence	121
weight	292
fan-in	212
SPAN	595
NES = WES_0	290
WES_1	300
WES_2	297
$MTBDD_{nr}$	393
$MTBDD_r$	2,209

Model `wlan1_time_bounded.nm`

Heuristic	Avg
random	1
default	1
distance graph	6,145
command graph	73
presence	96
weight	305
fan-in	140
SPAN	657
NES = WES_0	282
WES_1	289
WES_2	290
$MTBDD_{nr}$	474
$MTBDD_r$	21,634

Model `zeroconf.nm`

Heuristic	Avg
random	1
default	1
distance graph	492
command graph	57
presence	86
weight	195
fan-in	41
SPAN	508
NES = WES_0	309
WES_1	314
WES_2	315
$MTBDD_{nr}$	18,255
$MTBDD_r$	100,004

Model `mapk_cascade.sm`

Heuristic	Avg
random	2
default	2
distance graph	126
command graph	58
presence	81
weight	126
fan-in	42
SPAN	391
NES = WES_0	260
WES_1	265
WES_2	256
$MTBDD_{nr}$	828
$MTBDD_r$	16,512

Model `sprouty.sm`

Heuristic	Avg
random	2
default	2
distance graph	181
command graph	78
presence	113
weight	193
fan-in	63
SPAN	470
NES = WES_0	342
WES_1	350
WES_2	340
$MTBDD_{nr}$	913
$MTBDD_r$	6,459

Average result

Heuristic	Avg
random	1
default	1
distance graph	12,509
command graph	49
presence	58
weight	303
fan-in	88
SPAN	2,071
NES = WES_0	288
WES_1	290
WES_2	287
$MTBDD_{nr}$	1,143
$MTBDD_r$	7,233

A.3 Construction Time

Time is displayed in milliseconds (ms).

Model `brp.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	466	137	228	906
default	391	88	225	536
distance graph	406	91	217	533
command graph	353	102	207	517
presence	492	–	–	–
weight	480	–	–	–
fan-in	279	–	–	–
SPAN	432	82	261	519
NES = WES_0	422	79	336	515
WES_1	343	33	241	423
WES_2	343	42	240	473
$MTBDD_{nr}$	266	52	233	449
$MTBDD_r$	250	16	215	294

Model `cluster.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	87	23	50	185
default	55	3	49	68
distance graph	50	1	48	54
command graph	54	3	47	63
presence	55	–	–	–
weight	59	–	–	–
fan-in	51	–	–	–
SPAN	54	3	47	62
NES = WES_0	54	3	47	59
WES_1	54	2	51	61
WES_2	54	2	50	59
$MTBDD_{nr}$	55	7	50	87
$MTBDD_r$	54	2	49	60

Model `coin4.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	1,081	515	227	2,600
default	378	103	160	499
distance graph	339	5	333	349
command graph	351	122	229	492
presence	173	–	–	–
weight	173	–	–	–
fan-in	162	–	–	–
SPAN	246	60	230	488
NES = WES_0	312	113	230	470
WES_1	473	7	470	490
WES_2	488	6	478	492
$MTBDD_{nr}$	481	7	469	488
$MTBDD_r$	478	8	469	490

Model `csma3_2.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	2,220	514	1,379	3,858
default	1,176	109	988	1,397
distance graph	2,861	889	1,622	3,953
command graph	1,706	373	1,278	2,486
presence	1,599	–	–	–
weight	1,671	–	–	–
fan-in	1,339	–	–	–
SPAN	1,566	267	1,244	2,263
NES = WES_0	1,467	201	1,233	1,817
WES_1	1,757	25	1,693	1,796
WES_2	1,761	27	1,693	1,847
$MTBDD_{nr}$	1,398	150	1,287	1,954
$MTBDD_r$	1,383	148	1,270	1,909

Model `dice.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	14	–	–	–
default	14	–	–	–
distance graph	14	–	–	–
command graph	14	–	–	–
presence	14	–	–	–
weight	14	–	–	–
fan-in	14	–	–	–
SPAN	14	–	–	–
NES = WES_0	14	–	–	–
WES_1	14	–	–	–
WES_2	14	–	–	–
$MTBDD_{nr}$	14	–	–	–
$MTBDD_r$	14	–	–	–

Model `two_dice.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	23	3	20	39
default	21	1	20	22
distance graph	21	–	–	–
command graph	21	–	–	–
presence	39	–	–	–
weight	39	–	–	–
fan-in	21	–	–	–
SPAN	21	–	–	–
NES = WES_0	21	–	–	–
WES_1	21	–	–	–
WES_2	21	–	–	–
$MTBDD_{nr}$	21	–	–	–
$MTBDD_r$	21	–	–	–

Model `two_dice_knuth.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	38	5	35	45
default	40	5	35	45
distance graph	45	–	–	–
command graph	45	–	–	–
presence	35	–	–	–
weight	35	–	–	–
fan-in	35	–	–	–
SPAN	45	–	–	–
NES = WES_0	45	–	–	–
WES_1	45	–	–	–
WES_2	45	–	–	–
$MTBDD_{nr}$	45	–	–	–
$MTBDD_r$	45	–	–	–

Model `dining_crypt5.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	184	55	88	338
default	101	22	64	139
distance graph	151	36	77	208
command graph	85	24	57	180
presence	128	–	–	–
weight	114	–	–	–
fan-in	65	–	–	–
SPAN	85	17	56	129
NES = WES_0	65	7	57	86
WES_1	65	6	58	81
WES_2	64	7	56	84
$MTBDD_{nr}$	67	11	55	109
$MTBDD_r$	60	2	57	63

Model `embedded.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	35	4	27	44
default	33	4	27	45
distance graph	27	1	26	28
command graph	31	3	25	41
presence	26	–	–	–
weight	32	–	–	–
fan-in	31	–	–	–
SPAN	27	2	24	33
NES = WES_0	31	3	26	39
WES_1	29	1	26	31
WES_2	29	2	26	38
$\mathrm{MTBDD}_{\mathrm{nr}}$	27	1	25	31
$\mathrm{MTBDD}_{\mathrm{r}}$	28	2	25	31

Model `fms.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	9,107	4,585	1,597	17,674
default	574	86	425	837
distance graph	713	168	424	1,155
command graph	589	180	385	1,243
presence	914	–	–	–
weight	1,083	–	–	–
fan-in	1,238	–	–	–
SPAN	425	39	356	515
NES = WES_0	406	24	350	463
WES_1	378	22	345	457
WES_2	390	32	347	514
$\mathrm{MTBDD}_{\mathrm{nr}}$	519	139	389	886
$\mathrm{MTBDD}_{\mathrm{r}}$	459	46	381	572

Model `leader4.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	606	120	378	889
default	349	65	266	456
distance graph	345	55	272	485
command graph	309	53	252	503
presence	485	–	–	–
weight	446	–	–	–
fan-in	352	–	–	–
SPAN	271	12	252	302
NES = WES_0	295	34	250	371
WES_1	300	29	257	356
WES_2	299	26	255	345
$\mathrm{MTBDD}_{\mathrm{nr}}$	283	15	264	339
$\mathrm{MTBDD}_{\mathrm{r}}$	270	12	252	292

Model `leader4_3.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	286	200	93	1,177
default	87	22	64	148
distance graph	133	47	74	276
command graph	321	260	83	1,479
presence	467	–	–	–
weight	120	–	–	–
fan-in	153	–	–	–
SPAN	210	101	103	536
NES = WES_0	265	164	80	868
WES_1	246	100	110	488
WES_2	216	85	101	487
$\mathrm{MTBDD}_{\mathrm{nr}}$	123	28	75	213
$\mathrm{MTBDD}_{\mathrm{r}}$	228	100	97	448

Model `knacl.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	28	1	26	29
default	28	1	26	29
distance graph	28	0	28	29
command graph	29	0	28	29
presence	28	–	–	–
weight	28	–	–	–
fan-in	29	–	–	–
SPAN	29	–	–	–
NES = WES_0	29	–	–	–
WES_1	29	–	–	–
WES_2	29	–	–	–
$\mathrm{MTBDD}_{\mathrm{nr}}$	28	–	–	–
$\mathrm{MTBDD}_{\mathrm{r}}$	29	–	–	–

Model `mc.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	30	3	25	35
default	28	2	25	31
distance graph	29	2	26	31
command graph	29	3	26	33
presence	31	–	–	–
weight	31	–	–	–
fan-in	28	–	–	–
SPAN	31	–	–	–
NES = WES_0	31	–	–	–
WES_1	29	–	–	–
WES_2	29	–	–	–
$\mathrm{MTBDD}_{\mathrm{nr}}$	26	1	26	29
$\mathrm{MTBDD}_{\mathrm{r}}$	29	–	–	–

Model `peer2peer5_4.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	35	3	30	44
default	41	13	36	128
distance graph	36	4	30	52
command graph	36	3	29	44
presence	35	–	–	–
weight	35	–	–	–
fan-in	41	–	–	–
SPAN	38	11	30	100
NES = WES_0	38	12	30	117
WES_1	35	4	29	48
WES_2	36	3	29	45
$\mathrm{MTBDD}_{\mathrm{nr}}$	36	16	28	112
$\mathrm{MTBDD}_{\mathrm{r}}$	32	2	28	35

Model `phil_lss3.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	315	45	191	410
default	255	33	192	305
distance graph	267	15	249	280
command graph	294	53	191	410
presence	388	–	–	–
weight	285	–	–	–
fan-in	307	–	–	–
SPAN	233	45	190	293
NES = WES_0	298	48	192	395
WES_1	299	51	192	395
WES_2	299	53	194	395
$\mathrm{MTBDD}_{\mathrm{nr}}$	270	15	241	304
$\mathrm{MTBDD}_{\mathrm{r}}$	239	42	190	301

Model `poll6.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	27	3	25	43
default	28	3	25	36
distance graph	27	2	25	32
command graph	27	2	24	32
presence	31	–	–	–
weight	27	–	–	–
fan-in	26	–	–	–
SPAN	26	3	25	43
NES = WES_0	26	1	25	30
WES_1	26	1	25	29
WES_2	26	2	25	38
$\mathrm{MTBDD}_{\mathrm{nr}}$	26	1	25	27
$\mathrm{MTBDD}_{\mathrm{r}}$	26	1	25	27

Model `beauquier3.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	16	1	15	18
default	16	0	16	17
distance graph	17	2	15	21
command graph	16	1	15	21
presence	16	–	–	–
weight	15	–	–	–
fan-in	17	–	–	–
SPAN	17	0	16	17
NES = WES_0	17	0	16	17
WES_1	17	0	16	17
WES_2	17	0	16	17
$\mathrm{MTBDD}_{\mathrm{nr}}$	16	0	16	17
$\mathrm{MTBDD}_{\mathrm{r}}$	16	0	16	17

Model `tandem.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	40	6	32	51
default	40	8	32	51
distance graph	41	9	32	51
command graph	38	6	32	51
presence	37	–	–	–
weight	32	–	–	–
fan-in	37	–	–	–
SPAN	51	–	–	–
NES = WES_0	35	–	–	–
WES_1	35	–	–	–
WES_2	35	–	–	–
$MTBDD_{nr}$	32	1	32	35
$MTBDD_r$	32	1	32	37

Model `wlan1.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	419	56	299	516
default	344	28	291	405
distance graph	389	19	368	473
command graph	329	6	321	339
presence	308	–	–	–
weight	276	–	–	–
fan-in	240	–	–	–
SPAN	328	–	–	–
NES = WES_0	328	–	–	–
WES_1	327	16	301	425
WES_2	335	28	303	427
$MTBDD_{nr}$	329	5	326	339
$MTBDD_r$	327	7	303	339

Model `wlan1_collide.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	510	74	366	638
default	413	32	356	486
distance graph	444	40	389	529
command graph	383	14	334	407
presence	356	–	–	–
weight	318	–	–	–
fan-in	326	–	–	–
SPAN	394	1	394	400
NES = WES_0	395	5	381	427
WES_1	385	29	357	522
WES_2	388	40	357	547
$MTBDD_{nr}$	388	6	382	405
$MTBDD_r$	384	8	370	391

Model `wlan1_time_bounded.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	5,482	1,884	2,232	9,896
default	4,485	1,416	2,780	6,856
distance graph	4,208	1,468	2,665	6,206
command graph	3,413	114	3,242	3,564
presence	3,657	–	–	–
weight	2,495	–	–	–
fan-in	4,120	–	–	–
SPAN	3,450	19	3,444	3,514
NES = WES_0	3,447	14	3,444	3,514
WES_1	3,733	858	3,249	6,262
WES_2	3,664	804	3,249	6,273
$MTBDD_{nr}$	3,479	21	3,449	3,542
$MTBDD_r$	3,487	17	3,445	3,542

Model `zeroconf.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	3,688	4,081	958	22,015
default	1,620	1,081	609	5,496
distance graph	2,950	3,643	590	22,863
command graph	11,671	17,649	501	53,324
presence	770	–	–	–
weight	674	–	–	–
fan-in	984	–	–	–
SPAN	7,118	7,355	581	40,722
NES = WES_0	6,255	5,961	563	14,722
WES_1	8,116	4,907	589	13,895
WES_2	7,425	5,762	561	14,575
$MTBDD_{nr}$	588	96	466	768
$MTBDD_r$	3,316	10,313	511	44,551

Model `mapk_cascade.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	25,955	12,134	8,259	68,947
default	3,972	3,836	978	17,102
distance graph	1,823	1,861	539	10,784
command graph	1,777	1,352	518	7,996
presence	18,747	–	–	–
weight	20,311	–	–	–
fan-in	1,168	–	–	–
SPAN	604	52	487	714
NES = WES_0	607	52	494	737
WES_1	641	70	461	790
WES_2	648	82	500	960
$MTBDD_{nr}$	787	272	529	1,740
$MTBDD_r$	723	210	500	1,296

Model `sprouty.sm`

Heuristic	Avg	Std. dev.	Min	Max
random	4,436	2,285	1,788	11,661
default	2,984	1,617	720	7,703
distance graph	905	330	413	1,816
command graph	822	449	285	2,590
presence	905	–	–	–
weight	486	–	–	–
fan-in	1,917	–	–	–
SPAN	404	100	266	913
NES = WES_0	478	141	279	1,067
WES_1	368	71	265	640
WES_2	364	83	256	672
$MTBDD_{nr}$	531	139	288	931
$MTBDD_r$	523	113	363	845

Average result

Heuristic	Avg	Std. dev.	Min	Max
random	2,205	1,069	735	5,683
default	699	343	337	1,714
distance graph	651	348	342	2,012
command graph	910	831	328	3,037
presence	1,189	–	–	–
weight	1,171	–	–	–
fan-in	519	–	–	–
SPAN	645	327	341	2,083
NES = WES_0	615	274	341	1,049
WES_1	711	249	356	1,095
WES_2	681	283	356	1,138
$MTBDD_{nr}$	394	39	351	517
$MTBDD_r$	498	442	349	2,227

A.4 Model Checking Time

As model checking is slow, this experiment has been run on a smaller set of files. Time is displayed in milliseconds (ms).

Model `leader4.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	48,521	9,756	32,338	76,392
default	34,894	4,751	27,893	45,427
distance graph	35,067	4,378	26,924	46,168
command graph	30,747	3,697	25,079	40,298
presence	38,223	–	–	–
weight	34,982	–	–	–
fan-in	37,550	–	–	–
SPAN	28,065	1,733	24,495	32,230
NES = WES_0	30,029	3,108	25,436	36,508
WES_1	29,924	2,739	25,470	36,247
WES_2	30,053	2,326	25,164	35,328
$MTBDD_{nr}$	30,220	1,902	25,940	33,967
$MTBDD_r$	28,315	1,741	25,914	33,155

Model `leader4_3.pm`

Heuristic	Avg	Std. dev.	Min	Max
random	114	28	69	195
default	81	7	70	108
distance graph	110	17	84	149
command graph	131	47	62	295
presence	104	–	–	–
weight	75	–	–	–
fan-in	100	–	–	–
SPAN	126	28	69	189
NES = WES_0	121	42	69	261
WES_1	92	16	62	135
WES_2	88	13	69	120
$MTBDD_{nr}$	105	28	64	157
$MTBDD_r$	89	18	66	131

Model `wlan1_time_bounded.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	97,150	42,746	34,024	220,059
default	72,396	28,545	33,450	137,056
distance graph	80,849	18,677	54,523	139,833
command graph	74,418	8,374	61,926	89,399
presence	54,749	–	–	–
weight	57,260	–	–	–
fan-in	29,965	–	–	–
SPAN	69,673	97	69,644	70,001
NES = WES_0	69,658	70	69,644	70,001
WES_1	90,949	13,789	69,515	132,765
WES_2	87,881	11,088	63,128	129,755
$MTBDD_{nr}$	68,165	3,873	64,022	73,708
$MTBDD_r$	70,355	3,621	64,022	73,708

Model `zeroconf.nm`

Heuristic	Avg	Std. dev.	Min	Max
random	1,171	193	792	1,850
default	972	138	632	1,256
distance graph	1,182	160	779	1,476
command graph	1,085	390	698	1,895
presence	787	–	–	–
weight	868	–	–	–
fan-in	1,245	–	–	–
SPAN	887	58	775	1,030
NES = WES_0	903	76	789	1,085
WES_1	914	70	808	1,089
WES_2	918	84	786	1,140
$MTBDD_{nr}$	849	117	725	1,130
$MTBDD_r$	789	261	675	1,820

Average result

Heuristic	Avg	Std. dev.	Min	Max
random	36,739	13,181	16,806	74,624
default	27,086	8,360	15,511	45,962
distance graph	29,302	5,808	20,578	46,907
command graph	26,595	3,127	21,941	32,972
presence	23,466	–	–	–
weight	23,296	–	–	–
fan-in	17,215	–	–	–
SPAN	24,688	479	23,746	25,863
NES = WES_0	25,178	824	23,985	26,964
WES_1	30,470	4,154	23,964	42,559
WES_2	29,735	3,378	22,287	41,586
$MTBDD_{nr}$	24,834	1,480	22,688	27,241
$MTBDD_r$	24,887	1,410	22,669	27,204

B New Ordering Options in PRISM

I have implemented many of the ordering techniques described above in a branch of PRISM. New command line options `-ordering` and `-metric` were added to let an user specify how PRISM should construct a variable ordering from a model description. This appendix is a documentation of this new feature.

B.1 Using a Heuristic: the `-ordering` Switch

The `-ordering` switch can be used to specify an ordering technique. Most of available techniques are heuristics. Syntax of this option is:

`-ordering` *`<heuristic name>`*

Some techniques also accept or require arguments. In this case, syntax is

`-ordering` *`<heuristic name>`*`([`*`<option name>`*` = `*`<value>`*`[, ...]])`

Possible data types for argument values are:

- integer: an expression whose value is a positive integer number.
- boolean: an expression whose value is a boolean or an integer number. In the latter case, 0 corresponds to false, another value to true.
- seed: a seed value to initialize a random number generator, that is to say `selfinit` (self-initialization with the computer clock) or an expression whose value is an integer number.
- variable list: a comma-separated list of variable names, inside brackets or a `list` function. For example, `[a, b, c]` or equivalently `list(a, b, c)`.

Here is a description of the available ordering techniques.

- `default`: in this ordering, nondeterministic variables are put in top of the ordering, and then model variables in the order they are defined in the model description.
  This is the default variable ordering in PRISM.
  Options:
  - `ddgap` (optional, integer, default value: `20`): create a gap in the MTBDD variables, between nondeterministic and model variables. This allows to prepend additional variables, e.g. for constructing a product model when doing LTL model checking.
- `schedinmod`: variant of the default ordering, in which scheduling nondeterministic variables are placed just before the variables of the module they correspond to.
  This variable ordering is used by default with `-mtbdd` and `-o2` switches.
  Options: no option.

From now on, nondeterministic variables are always put in top of orderings.

- `random`: model variables are put in a random order.
  This ordering is mainly used for comparison purpose.
  Options:
  - `kpmodstr` (optional, boolean, default value: `false`): if true, variables of each module are kept grouped together in the ordering.
  - `sameinren` (optional, boolean, default value: `false`): if true and `kpmodstr` is set to true, use the same random permutation in renamed modules.
  - `seed` (optional, seed, default value: `selfinit`): seed value to use in the random number generator.
- `manual`: let the user specify the model variable ordering. This can also be used by an external ordering program to run PRISM with an ordering it generated.
  Options:

- `vars` (mandatory, variable list): the model variable ordering to use.

- `astdstgrph`: use a TSP heuristic in which variable graph edges are labelled by distance in AST. This heuristic is described in section 3.2.3.
  Options:
  - `seed` (optional, seed, default value: `selfinit`): seed value to use in the random number generator.
  - `iters` (optional, integer, default value: `1000`): number of iterations to perform in the TSP solver. A bigger value leads to better TSP solutions at the expense of a slower computation time.
- `cmdgrph`: use a TSP heuristic in which variable graph edges are labelled by the number of commands containing both corresponding variables. This heuristic is described in section 3.2.3.
  Options:
  - `seed` (optional, seed, default value: `selfinit`): seed value to use in the random number generator.
  - `iters` (optional, integer, default value: `1000`): number of iterations to perform in the TSP solver. A bigger value leads to better TSP solutions at the expense of a slower computation time.
- `sumcmd`: use the presence greedy heuristic, described in section 3.2.2.
  Options: no option
- `weight`: use the weight heuristic, described in section 3.2.2.
  Options: no option
- `fanin`: use the fan-in heuristic, described in section 3.2.2.
  Options: no option

B.2 Using a Metric: the `-metric` Switch

We may prefer construct several variable orderings with techniques described above, and compare them with a metric to select the best one. To construct a set of orderings, it is possible to use several times the same nondeterministic heuristic:

```
-ordering <n> * <heuristic>
```

where n is the number of orderings to generate with this heuristic. Nondeterministic ordering techniques are `random`, `astdstgrph` and `cmdgrph`.

We can also use several techniques to produce orderings:

```
-ordering [<n1> * ]<heuristic1> + [<n2> * ]<heuristic2> [+ ...]
```

It is generally faster to produce n orderings with a few nondeterministic heuristics, each one providing many orderings, than to use lots of different techniques. Indeed, most of ordering techniques require some computations to be initialized, thus it is beneficial to minimize the number of initialized techniques.

A metric must also be specified to compare orderings:

```
-metric <metric name>
```

Here is a description of available metrics.

- `span`: use the SPAN metric, described in section 3.3.1.
- `nes`: use the NES metric, described in section 3.3.2.

- `wes0`, `wes1`, `wes2`: use the WES metric, described in section 3.3.2.
  `wes0` corresponds to WES_0 – this is an alias of `nes`.
  `wes1` corresponds to WES_1.
  `wes2` corresponds to WES_2.
- `noreachdd`: use the $\mathrm{MTBDD}_{\mathrm{nr}}$ metric, described in section 3.3.3.
- `reachdd`: use the $\mathrm{MTBDD}_{\mathrm{r}}$ metric, described in section 3.3.3.

List of figures

References

[Ake78] S. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6):509–516, 1978.

[AMS04] F.A. Aloul, I.L. Markov, and K.A. Sakallah. MINCE: a static global variable ordering heuristic for SAT Search and BDD Manipulation. *Journal of universal computer science*, pages 1562–1596, 2004.

[And97] H. R. Andersen. An introduction to binary decision diagrams, 1997.

[BFG+93] I. Bahar, E. Frohm, C. Gaona, G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *Proc. International Conference on Computer-Aided Design (ICCAD'93)*, pages 188–191, 1993.

[BLW95] B. Bollig, M. Löbbing, and I. Wegener. Simulated annealing to improve variable orderings for OBDDs. In *International Workshop on Logic Synthesis*, 1995.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[BW96] B. Bollig and I. Wegner. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 49(9):993–1006, 1996.

[CBGP08] F. Ciesinski, C. Baier, M. Größer, and D. Parker. Generating compatc MTBDD representations from Probmela specifications. In *SPIN '08: Proceedings of the 15th international workshop on Model Checking Software*, pages 60–76, 2008.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, K. McMillan, J. Yang, and X. Zhao. Multi-terminal binary decision diagrams: An efficient data structure for matrix representation. In *Proc. International Workshop on Logic Synthesis (IWLS'93)*, pages 1–15, 1993.

[CMZ+93] E. Clarke, K. McMillan, X. Zhao, M. Fujita, and J. Yang. Spectral transforms for large boolean functions with applications to technology mapping. In ACM Press, editor, *Proc. 30th Design Automation Conference (DAC'93)*, pages 54–60, 1993.

[CS06] G. Ciardo and R. Siminiceanu. New metrics for static variable ordering in decision diagrams. In *TACAS 2006*, pages 90–104, 2006.

[DBG95] R. Drechsler, B. Becker, , and N. Göckel. A genetic algorithm for variable ordering of OBDDs. In *International Workshop on Logic Synthesis*, 1995.

[EFT91] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In *Proc. 3rd International Workshop on Computer Aided Verification (CAV'91)*, pages 203–213, 1991.

[FMK91] M. Fujita, Y. Matsunaga, and T. Kakuda. On variable ordering of binary decision diagrams for the application of multi-level logic synthesis. In *Proceedings of the European Conference on Design Automation*, pages 50–54, 1991.

[FS87] S.J. Friedman and K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. In *Annual ACM IEEE Design Automation Conference*, pages 348–356, 1987.

[ISY91] N. Ishiura, H. Sawada, and S. Yajima. Minimization of binary decision diagrams based on exchanges of variables. In *Proceedings of the International Conference on Computer-Aided Design*, pages 472–475, 1991.

[IT90] O. Ibe and K. Trivedi. Stochastic Petri net models of polling systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1957, 1990.

[Lee59] C. Lee. Representation of switching circuits by binary-decision programs. *Bell System Technical Journal*, 38:985–999, 1959.

[MIY90] S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In *DAC '90: Proceedings of the 27th ACM/IEEE conference on Design automation*, pages 52–57, 1990.

[MWB88] S. Malik, A.R. Wang, and R.K. Brayton. Logic verification using binary decision diagrams in a logic synthesis environment. *ICCAD-88: Digest of technical papers*, pages 6–9, 1988.

[Noa99] A. Noack. A ZBDD package for efficient model checking of Petri nets, 1999.

[NW07] N. Narodytska and T. Walsh. Constraint and variable ordering heuristics for compiling configuration problems. In *IJCAI-07: International Joint Conferences on Artificial Intelligence*, pages 149–154, 2007.

[Par02] D. Parker. *Implementation of Symbolic Model Checking for Probabilistic Systems*. PhD thesis, University of Birmingham, 2002.

[PS95] S. Pando and F. Somenzi. Who are the variables in your neighborhood. In *Proceedings of the International Conference on Computer-Aided Design*, pages 74–77, 1995.

[PSP94] S. Panda, F. Somenzi, and B. F. Plessier. Symmetry detection and dynamic variable ordering of decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 628–631, 1994.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proceedings of the International Conference on Computer-Aided Design*, pages 42–47, 1993.

[Sie02] D. Sieling. The nonapproximability of OBDD minimization. *Information and Computation*, 172:103–138, 2002.

[vD00] S. van Dongen. *Graph Clustering by Flow Simulation*. PhD thesis, University of Utrecht, 2000.