An Efficient BDD-Based Implementation of Gauss-Seidel for CTMC Analysis

Rashid Mehmood, David Parker and Marta Kwiatkowska

{rxm,dxp,mzk}@cs.bham.ac.uk

School of Computer Science, University of Birmingham, B15 2TT, UK

Abstract

Symbolic approaches to the analysis of Markov models, i.e. those that use BDD-based data structures, have proved to be an effective method of combating the state space explosion problem. One such example is the use of offset-labelled MTBDDs (multi-terminal BDDs). However, a major disadvantage of this data structure is that it cannot be used with efficient iterative methods, in particular, Gauss-Seidel. In this paper, we propose a solution that permits the use of this numerical method by introducing a data structure derived from an offset-labelled MTBDD. This approach provides significant improvements in terms of both time and memory consumption. We present and analyse experimental results for both in-core and out-of-core versions of this implementation on a standard workstation, and successfully perform steady-state probability computation for CTMCs with as many as 600 million states and 7.7 billion transitions.

Keywords: CTMCs, Performance analysis, BDDs, Symbolic model checking

Submission category: Model Checking of Stochastic Systems

Approx. word count: 6,000

Contact author: Rashid Mehmood, Tel: +44 121 4144793, Fax: +44 121 4144281

1 Introduction

Continuous-time Markov chains (CTMCs) are a widely used model for the performance evaluation of communication networks and computer systems. A large variety of useful performance measures can be derived from a CTMC via the computation of its *steady-state probabilities*. This reduces to the more general problem of solving a linear equation system $Ax = b$ of size equal to the number of states in the CTMC. Unfortunately, these models tend to grow extremely large, a phenomenon often known as the *state space explosion problem*.

A great deal of effort has gone into developing efficient implementations of this process, with particular emphasis on the problem of storing large CTMCs. The various approaches can be broadly classified as: (i) *explicit*, where the CTMC is kept in a data structure of size proportional to the number of states and transitions, typically a *sparse matrix*; or (ii) *implicit*, where this explicit storage is avoided. The latter includes *symbolic* (BDD-based) methods [HMKS99, CM99, KNP02b], *on-the-fly* methods [DS98b] and *Kronecker* methods [Pla85]. Implicit techniques offer compact storage by exploiting structure and regularity in the CTMCs, usually derived from their description in some high-level specification formalism, and can hence be applied to larger CTMCs than explicit methods. Another classification is into *in-core* approaches, where data is stored in the main memory of a computer, and *out-of-core* approaches, where it is stored on disk (see e.g. [DS98a]).

Steady-state probability computation for large CTMCs is usually performed using *iterative* numerical solution methods, which generate successive approximations to the solution vector until the desired level of accuracy has been achieved. Two commonly used iterative methods are Jacobi and Gauss-Seidel. The latter is considerably more efficient since it typically converges much faster and requires less storage for the solution vector. In this paper we consider a symbolic implementation of these iterative techniques, specifically using *multi-terminal binary decision diagrams* (MTBDDs). A major weakness of the MTBDD approach, not shared by other implicit methods, is that, although the Jacobi method is supported, Gauss-Seidel is not practical. In this paper, we propose a number of modifications to the MTBDD approach which remedy this situation. We have developed both in-core and out-of-core implementations of our technique and present experimental results to demonstrate that they provide a significant improvement in time and memory efficiency.

This work is part of the ongoing development of the probabilistic model checker PRISM [KNP02a]. PRISM supports *CSL model checking* [BKH99] of CTMCs, a crucial part of which is the computation of steady-state probabilities. MTBDD-based techniques are particularly attractive in a model checking context because they integrate easily with BDDs, which are efficient for computing reachable state spaces, identifying strongly connected components, and other graph-based analyses required for model checking. An efficient symbolic implementation for solving linear equation systems is also desirable for other functions of PRISM, such as model checking of discrete-time Markov chains.

The paper is organised as follows. Section 2 describes the MTBDD data structure and its application to the representation and numerical solution of CTMCs. In Section 3, we explain our modifications to this approach. In Section 4, we present and analyse experimental results from our implementation. Finally, Section 5 concludes the paper.

2 MTBDD-Based Techniques

2.1 Representing CTMCs with MTBDDs

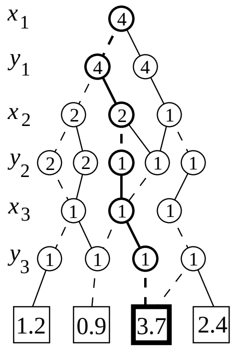
Multi-terminal binary decision diagrams (MTBDDs) [CFM+93, BFG+93] are an extension of binary decision diagrams (BDDs). An MTBDD is a rooted, directed acyclic graph (DAG), which represents a function mapping Boolean variables to real numbers. MTBDDs can be used to encode real-valued vectors and matrices by encoding their indices as Boolean variables. Since a CTMC is described by a real-valued matrix, it can also be represented as an MTBDD.

The advantage of using MTBDDs (and other symbolic data structures) to store CTMCs is that they can often give extremely compact storage, provided that the CTMCs exhibit a certain degree of structure and regularity. In practice, this is very often the case since they will have been specified in some (inherently structured) high-level description formalism. Intuitively, the reason that MTBDDs can exploit such regularity is that the data structure is stored in a reduced form, with redundant (identical) nodes of the graph being merged. This means that, where possible, identical portions of the matrix are stored only once.

In fact, in this paper we use a variant of the data structure called an *offset-labelled MTBDD*. The principal difference is the addition of *offsets* to each node of the graph, used to allow conversion between the *potential* and *actual* (reachable) state spaces, the former often being significantly larger than the latter and hence inefficient to deal with. For brevity, in this paper we will simply refer to this data structure as an MTBDD.

Figure 1 illustrates the MTBDD representation of an 8×8 matrix, which might occur in the numerical solution of a CTMC. Note that, to preserve structure in the symbolic representation, the diagonal elements are stored separately as an array. Hence, the diagonal entries of the matrix in Figure 1 are all zero. The MTBDD comprises two types of nodes: *non-terminal*

$$\begin{matrix} 0 & 1.2 & 0 & 0 & 0 & 0 & 0.9 & 0 \\ 0.9 & 0 & 0 & 0 & 0 & 0 & \mathbf{3.7} & 0 \\ 0 & 0 & 0 & 1.2 & 0.9 & 0 & 0 & 0 \\ 0 & 0 & 0.9 & 0 & 3.7 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3.7 & 2.4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0.9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 3.7 & 0 & 0 & 0 \end{matrix}$$



Matrix entry	Encoding						Path						Offsets						Reachable entry
	x_1	x_2	x_3	y_1	y_2	y_3	x_1	y_1	x_2	y_2	x_3	y_3	x_1	y_1	x_2	y_2	x_3	y_3	
$(0,1) = 1.2$	0	0	0	0	0	1	0	0	0	0	0	1	-	-	-	-	-	1	$(0,1) = 1.2$
$(0,6) = 0.9$	0	0	0	1	1	0	0	1	0	1	0	0	-	4	-	1	-	-	$(0,5) = 0.9$
$(1,0) = 0.9$	0	0	1	0	0	0	0	0	0	0	1	0	-	-	-	-	1	-	$(1,0) = 0.9$
$\mathbf{(1,6) = 3.7}$	**0**	**0**	**1**	**1**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**-**	**4**	**-**	**1**	**1**	**-**	$\mathbf{(1,5) = 3.7}$
$(2,3) = 1.2$	0	1	0	0	1	1	0	0	1	1	0	1	-	-	2	2	-	1	$(2,3) = 1.2$
...etc...																			

Figure 1: Representing an 8×8 matrix as an (offset-labelled) MTBDD

nodes, drawn as circles, and *terminal* nodes, drawn as squares. Non-terminal nodes are labelled with integer offsets, terminal nodes with real values. Each row of nodes in the data structure is associated with a Boolean variable which is written on the far left of the row. The MTBDD represents a function over these Boolean variables. For the example in Figure 1, the function is over the variables $x_1, y_1, x_2, y_2, x_3, y_3$. For a given valuation of these variables, the value of the function can be computed by tracing a path from the top of the MTBDD to the bottom, at each node taking the dotted edge if the associated Boolean variable is 0 and the solid edge if it is 1. The value can be read from the label of the terminal node reached. If there is no such path, the value is 0. For example, if $(x_1, y_1, x_2, y_2, x_3, y_3) = (0, 1, 0, 1, 1, 0)$, the function returns 3.7 (as highlighted in the figure). If $(x_1, y_1, x_2, y_2, x_3, y_3) = (0, 1, 0, 1, 1, 1)$, however, the function returns zero.

The matrix represented by the MTBDD is encoded by this function as follows. The x_i variables are for row indices, and the y_i variables are for column indices. Since these variables are all Boolean, but the row and column indices are integers in the range $0 \ldots 7$, the information has to be encoded. We assume the standard binary representation of integers for this purpose. Consider the matrix entry $(1, 6) = 3.7$. The row index is 1 so we code this as 001 ($x_1 = 0, x_2 = 0, x_3 = 1$). The column index is 6 so we code this as 110 ($y_1 = 1, y_2 = 1, y_3 = 0$). Note that the x_i and y_i variables are interleaved in the MTBDD (this is a common heuristic in BDD-based representations to reduce their size) so the entry $(1, 6)$ is represented by the path 010110 which, as we have seen above, leads to the value 3.7. In the left portion of the table in Figure 1, we illustrate this encoding for the first five non-zero entries of the matrix.

The *actual* row index (in terms of reachable states only) is determined by summing the offsets on x_i nodes from which the solid edge is taken (i.e. if $x_i = 1$). The actual column index is computed similarly for y_i nodes. In the example in Figure 1, state 5 is not reachable. For the previous example of matrix entry $(1, 6)$, the actual row index is 1 and the actual column index is $4 + 1 = 5$, i.e. $(1, 6)$ becomes $(1, 5)$.

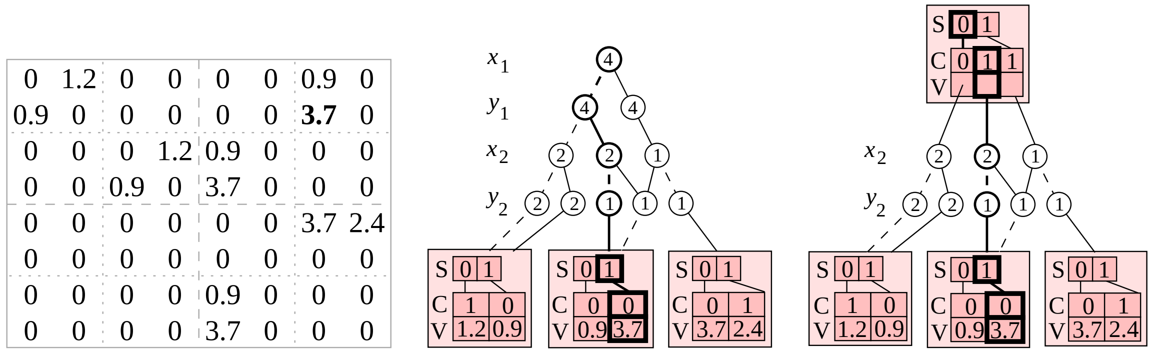
2.2 Numerical Solution with MTBDDs

Numerical solution of CTMCs performed entirely using MTBDDs (i.e. for both matrices and vectors) [HMPS96, HMKS99] has proved to be inefficient due to the lack of structure in the solution vector. The solution usually adopted is to combine symbolic (MTBDD-based) storage of the matrix with explicit (array-based) storage of the solution vector [KNP02b, Par02]. For some iterative methods, notably Jacobi, each iteration is based essentially on a matrix-vector multiplication operation. This requires access to each matrix element exactly once (and in any order). When using MTBDD matrix storage, this can be achieved via a single depth-first traversal of the data structure since each matrix element corresponds to a path through the MTBDD. Unfortunately, the overhead associated with this process makes it slower than the equivalent operation with sparse matrices since it it is significantly easier to read the matrix entries directly from array-based storage.

Fortunately, significant optimisation of the process is possible [KNP02b, Par02]. Note that each non-terminal node of an MTBDD represents a submatrix of the matrix represented by the whole MTBDD. In Figure 1, for example, x_2 nodes and x_3 nodes represent 4×4 and 2×2 submatrices, respectively, of the 8×8 matrix (see Figure 2(a)). The nodes near the bottom of the MTBDD, in particular, are visited many times during its traversal. It is much faster to extract entries of the matrix if we replace some of these nodes with an explicit representation of their corresponding submatrix, eliminating the need to traverse the nodes below this point.

We define a *level* of an MTBDD to be an adjacent pair of rows of nodes. We count levels from the top of the MTBDD, i.e. level i contains all the x_i and y_i nodes. The total number of levels is denoted l_{total}. Our strategy for the scheme outlined above is to select a value $l_{sm} \leq l_{total}$ and replace the x_i nodes in layer $l_{total} - l_{sm} + 1$ with an explicit representation of their corresponding submatrices. This means that nodes in the bottom l_{sm} levels do not need to be traversed and can be removed entirely. For this explicit storage, we employ the *compressed sparse row* (CSR) sparse matrix scheme, which uses three arrays: `Val` and `Col`, which contain the value and column index, respectively, of each non-zero matrix entry, stored row by row, and `Starts`, which contains indices into `Val` and `Col`, indicating where the entries for each row are stored. We illustrate this idea in Figure 2 on the MTBDD from Figure 1 using $l_{sm} = 1$ ($l_{total} = 3$); the three arrays for CSR are denoted `V`, `C` and `S`, respectively. Generally, as l_{sm} is increased, the time for each iteration of numerical solution (i.e. a single traversal) decreases, but the required memory increases. Our strategy is to choose l_{sm} as high as possible without exceeding some predefined memory limit (here, we use 1MB).

A problem with the MTBDD approach described above is that, although the Jacobi



(a) (b) (c)

Figure 2: Optimisations to the MTBDD storage scheme

iterative method can be efficiently implemented, Gauss-Seidel cannot because it requires row-wise access to matrix entries. A depth-first traversal of the MTBDD does not allow matrix entries to be extracted in this order. Of course, it would be possible to access each element of each row individually, going from top to bottom of the MTBDD each time, but this would be very inefficient.

A solution can be found by again making use of the fact that MTBDDs allow convenient access to matrix blocks. Descending one level from the top of an MTBDD splits the matrix which it represents into 4 blocks. Descending l_b levels, for some $l_b \leq l_{total}$, gives a decomposition into B^2 blocks, where $B = 2^{l_b}$. If we explicitly store pointers to the nodes which represent these blocks, we can quickly access them without having to traverse the top part of the MTBDD. For large l_b, many of the matrix blocks will actually be empty so we can use a sparse matrix (again, the CSR scheme, but storing node pointers instead of numerical values). Nodes in the top l_b levels of the MTBDD (like the bottom l_{sm} levels) can now be removed entirely. This is illustrated in Figure 2(c) for $l_b = 1$, i.e. $B = 2$. Note that there are only 3 node pointers stored, not $B^2 = 4$, since one block is empty. We also require an additional array `Offsets` which gives us the (global) index of the first row of each block in terms of reachable states. This information replaces the offsets on the top layer of MTBDD nodes which have now become obsolete. Note that, because the rows and columns of the matrix which correspond to reachable states of the CTMC are distributed unevenly, the matrix blocks are of varying (and unpredictable) size.

Figure 2(c) also shows that the divisions of a matrix into *submatrices* (by ascending from the bottom of the MTBDD) and into *blocks*[1] (by descending from the top of the MTBDD) can happily coexist, provided that the top and bottom layers do not overlap, i.e. $l_{sm} + l_b \leq l_{total}$.

We can now efficiently access matrix elements, if not by individual rows, then at least by rows of blocks. This facilitates the implementation of iterative solution methods which access the matrix a block at a time. In [Par02, KMNP02], a compromise between Jacobi and Gauss-Seidel called Pseudo Gauss-Seidel is used (see Section 3.1 for details of this method). Using this method, increasing the value of l_b reduces both the number of iterations required and the amount of memory for vector storage, although not to the same extent as (standard) Gauss-Seidel. Increasing l_b does, however, also lead to exponential growth in the amount of memory required for storage of the top layer of the data structure. Hence, we choose l_b as high as possible such that the memory does not exceed some limit (here, we use 1MB).

3 Two-Layer Matrix Storage

We now propose improvements to the techniques described in the previous section. Recall that the MTBDD storage scheme has three layers, the top and bottom layers of which comprise sparse matrices, and the middle layer of which contains MTBDD nodes. We consider first the top layer. As before, we assume that this layer constitutes l_b levels of the MTBDD, that

[1]Although the two terms are in general interchangeable, for convenience we will consistently distinguish between "submatrices" and "blocks" in this way.

$B = 2^{l_b}$, and that the sparse storage scheme used is CSR (compressed sparse row). We assume also that only $nnzB_{mat}$ of the B^2 matrix blocks are non-empty. Hence, the top layer of the data structure comprises four arrays: `Val` ($nnzB_{mat}$ node pointers), `Col` ($nnzB_{mat}$ integers), `Starts` (B integers), and `Offsets` (B integers).

The bottleneck in the storage of the top layer is the two arrays `Starts` and `Offsets`, the size of which ($4B$ bytes each) is exponential in l_b. Since the matrix is sparse, structured, and contains many unreachable states, it is more efficient to make these two arrays of length $nnzB$, where $nnzB$ is the number of non-zero rows of blocks in the matrix (or, equivalently, blocks in the vector). Furthermore, we use the *compact modified sparse row* (compact MSR) format [KM02], instead of CSR. This results in the following changes to our top-layer storage scheme. Firstly, we store each distinct MTBDD node pointer once only, in a separate array `Dist` of size d. The array `Val` can now store indices into this array, instead of actual pointers. Since d is typically small, each index `Val`$[i]$ into `Dist` can actually be stored in spare bits of `Col`$[i]$, eradicating the need for the array `Val` entirely (see [KM02, Meh03] for details). Secondly, the array `Starts` now stores the number of non-zero blocks in each row of matrix blocks, which is equivalent to the information stored previously, but requires only one byte per entry instead of four bytes. In total, we now have four arrays: `Dist` (d node pointers), `Col` ($nnzB_{mat}$ integers), `Starts` ($nnzB$ bytes), `Offsets` ($nnzB$ integers).

Further reductions in memory can also be made by using compact MSR instead of CSR for the storage of the bottom layer of the data structure. Together, these savings mean that we are now able to select much larger values for l_b and l_{sm}. In fact, we can actually choose values such that $l_b + l_{sm} = l_{total}$, i.e. the *blocks* indexed by the top layer and the *submatrices* described on the bottom layer coincide, the middle layer of the data structure being removed entirely. In Figure 3(a), we show this new *two-layer* data structure for the running example

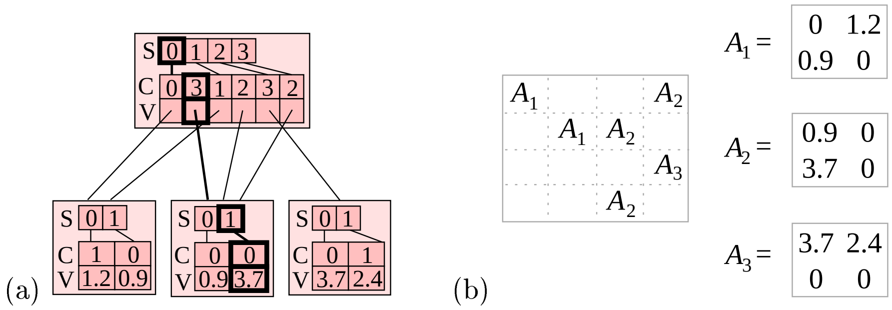


Figure 3: (a) The two-layer data structure (b) Block partitioning of the 8×8 matrix

of Figures 1 and 2. For clarity of presentation, the sparse matrix scheme used for both the top and bottom layers in this figure is actually CSR, rather than compact MSR. Figure 3(b) illustrates the resulting partitioning of the 8×8 example matrix.

Although, the new data structure is made up entirely of sparse matrix storage, it should still be seen as a symbolic approach. The data structure is constructed directly from the MTBDD representation and is completely reliant on the exploitation of regularity that this provides. We also observe here that the resulting storage scheme can actually be thought of as a special case of the *matrix diagram* data structure of [CM99] (where the number of levels is restricted to two and the single top-level matrix contains only ones and zeros). We speculate that the two-layer matrix could also be constructed from a matrix diagram (or some other implicit) representation of a CTMC.

3.1 Implementing Gauss-Seidel

Our focus is on the steady-state solution of CTMCs, which reduces to the problem of solving a linear equation system $Ax = b$ where $b = 0$. We now show how the data structure from the previous section allows us to apply well-known *block iterative* techniques (see e.g. [Ste94]) to do this and how, consequently, this provides us with a way to implement Gauss-Seidel. We assume a block partitioning of A into B^2 blocks, A_{ij}, for $0 \leq i, j < B$. Vectors are split into partitions of matching sizes; e.g. the solution vector at the kth iteration, $x^{(k)}$, is split up into sub-vectors $X_i^{(k)}$ for $0 \leq i < B$. Given this partitioning, an iteration of the *block Gauss-Seidel* method is formulated as follows:

$$X_i^{(k)} = A_{ii}^{-1}\tilde{X}, \quad \text{where} \quad \tilde{X} = -\sum_{j<i} A_{ij}\ X_j^{(k)} - \sum_{j>i} A_{ij}\ X_j^{(k-1)} \tag{1}$$

Each phase of the kth iteration requires us to first compute the vector $\tilde{X}$ and then solve (or partially solve) the linear equation system $X_i^{(k)} = A_{ii}^{-1}\tilde{X}$. In fact, if we apply just a single iteration of Gauss-Seidel to this (inner) linear equation system, then the block Gauss-Seidel method reduces to (standard) Gauss-Seidel. Since the two-level data structure provides efficient access both to each row of matrix blocks, and to each individual row within these blocks, we can now use it to implement Gauss-Seidel in this way. The standard MTBDD approach, on the other hand, provides a division into matrix blocks, but not access to each individual row of these blocks (because we still have to perform traversal of MTBDD nodes in the middle layer). The approach taken in [KNP02b, Par02] is to instead perform a single iteration of Jacobi to the (inner) linear equation system; the resulting numerical method is referred to in [KNP02b, Par02] as Pseudo Gauss-Seidel (PGS).

4 Experimental Results

In this section, we analyse the performance of the two-layer approach described in the previous section. We performed steady-state probability computation on three sets of widely used benchmark CTMCs: a flexible manufacturing system (FMS) [CT93], a Kanban system [CT96] and a cyclic server polling system [IT90]. These models were generated using the probabilistic model checker PRISM [KNP02a]. Experiments were run on a 440MHz 512MB UltraSPARC-II workstation running SunOS 5.8. Iterative methods were terminated when the maximum

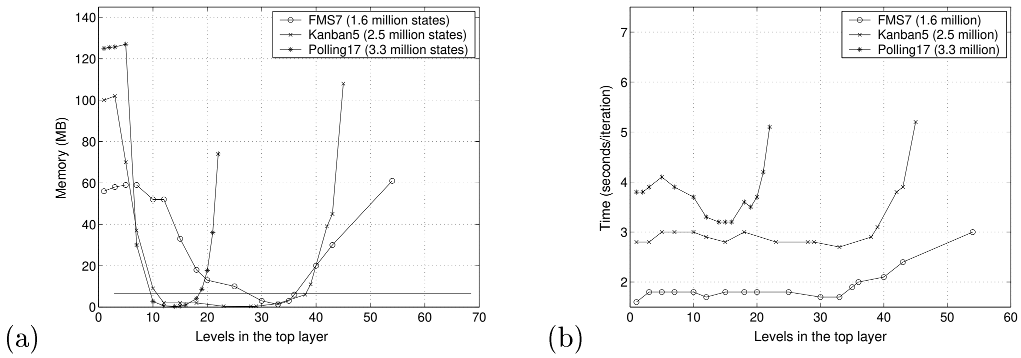


Figure 4: Performance for different values of l_b: (a) Memory (b) Time per iteration

relative difference between vector entries reached 10^{-6}. In all cases, the time taken to generate the symbolic CTMC matrix representation was negligible compared to numerical solution, and hence we present results solely for the latter.

4.1 An Optimal Choice for the Parameter l_b

The selection of l_b (which also determines l_{sm}) controls the size of the top and bottom parts of the two-layer data structure and is a crucial factor for its performance. In this section, we analyse the issue in more detail. Although determining the optimal value is likely to be expensive or infeasible, we seek a heuristic which provides good performance. In Figure 4, we plot both the memory required to store the matrix using our data structure and average time per iteration, observed when using a range of values of l_b on three representative CTMCs.

Consider first the plot for memory usage (Figure 4(a)). We see that, for all three examples, the minimum and maximum values of l_b result in very high memory usage. This is unsurprising since, in these extreme cases, the sparse matrix storage for either the bottom or top layers, respectively, constitutes almost the whole of the data structure. In these cases, none of the regularity and compactness of the original MTBDD is exploited, and hence the memory usage is high. However, we see that for values of l_b in the middle of this range, we can easily find a compromise between storage on the top and bottom layers which gives a dramatic drop in memory. For times per iteration (Figure 4(b)), we see that there are some fluctuations as l_b is varied and a consistent increase as it reaches its maximum. Overall, though, the variations in time are not nearly as significant as for the memory. Based on these statistics, we adopt the heuristic $l_b = 0.6 \times l_{total}$ for the results presented in the next two sections.

4.2 Speed of Numerical Solution

We now compare our technique against implementations based on two existing data structures: sparse matrices (compact MSR) and MTBDDs (as implemented in PRISM). In each case, we use the most efficient numerical solution method available, i.e. Gauss-Seidel and Pseudo Gauss-Seidel, respectively. Table 1 reports timing results. The first four columns give details of each CTMC used, its size n (reachable states) and average number of non-zeros per row (a/n). For each implementation, we give the average time per iteration, the number of iterations and the total time. To give a better indication of the trends in these statistics, in Figures 5(a) and (b), we plot time per iteration and total time against number of states, respectively.[2]

Our first observation is that the average time per iteration for the two-layer approach represents an improvement over the original MTBDD implementation. The principal reason for this is that, in the former, we avoid the traversal of MTBDD nodes in the middle layer of the data structure. This also means that the two-layer approach provides a much more consistent performance across the three examples (see Figure 5(a)). MTBDDs are more reliant on structure in the CTMCs; the FMS models, for example, exhibit less regularity, and hence

[2]For these plots we have collated results on a machine with equivalent CPU (440MHz) but with more RAM (5GB), in order to illustrate trends over a larger range of state spaces.

Model	k	States (n)	a/n	Time/iter. (secs)			Iterations		Total time (secs)		
				Sparse (GS)	MTBDD (PGS)	Two-layer (GS)	PGS	GS	Sparse (GS)	MTBDD (PGS)	Two-layer (GS)
FMS	6	537,768	7.8	0.3	0.72	0.50	1,027	812	244	739	406
	7	1,639,440	8.3	1.1	3.44	1.61	1,207	966	1063	4,152	1,555
	8	4,459,455	8.6	3.2	11.8	4.68	1,392	1,125	3600	16,426	5,265
	9	11,058,190	8.9	–	37.7	12.3	1,581	1,287	–	59,604	15,830
	10	25,397,658	9.2	–	100	29.2	1,775	1,454	–	177,500	42,457
Kanban	4	454,475	8.8	0.3	0.52	0.44	414	323	96.9	215	142
	5	2,546,432	9.6	1.8	3.13	2.71	590	461	830	1,847	1,249
	6	11,261,376	10.3	–	15.9	12.9	794	622	–	12,625	8,024
Polling system	15	737,280	8.3	0.5	0.69	0.62	179	32	16.0	124	19.8
	16	1,572,864	8.8	1.1	1.57	1.45	196	33	36.3	308	47.9
	17	3,342,336	9.3	2.4	3.78	3.23	213	34	81.6	805	110
	18	7,077,888	9.8	5.5	8.11	7.17	229	34	187	1,857	244
	19	14,942,208	10.3	–	19.3	16.3	255	35	–	4,922	571
	20	31,457,280	10.8	–	–	36.8	–	36	–	–	1,325

Table 1: Timing results: A comparison with existing implementations

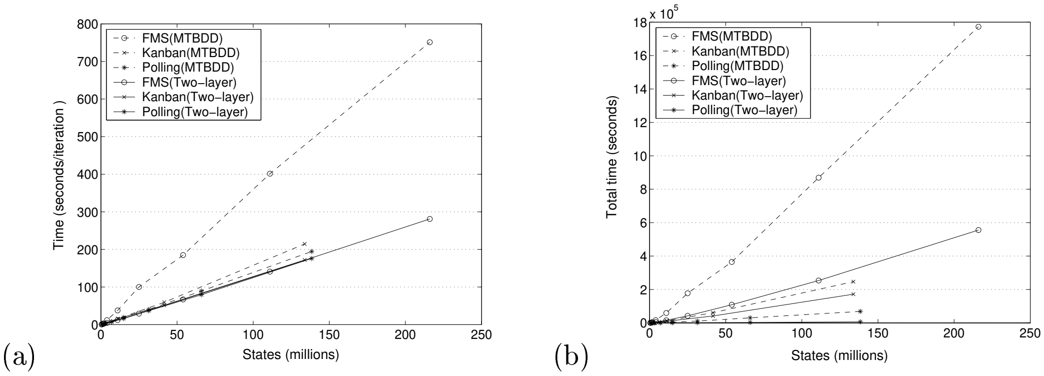


Figure 5: Timing results, plotted against number of states

show a greater improvement between the two approaches. Secondly, since using Gauss-Seidel over Pseudo Gauss-Seidel reduces the number of iterations required, the total solution times show an even more impressive improvement.

Making a comparison with sparse matrices, we see that the total times for the two-level approach are now much closer than MTBDDs were previously. We also note that the two-layer approach can handle CTMCs approximately an order of magnitude larger than sparse matrices, due to the relatively compact memory requirements. Note also that it can also be applied to slightly larger CTMCs than MTBDDs. This is because its implementation uses

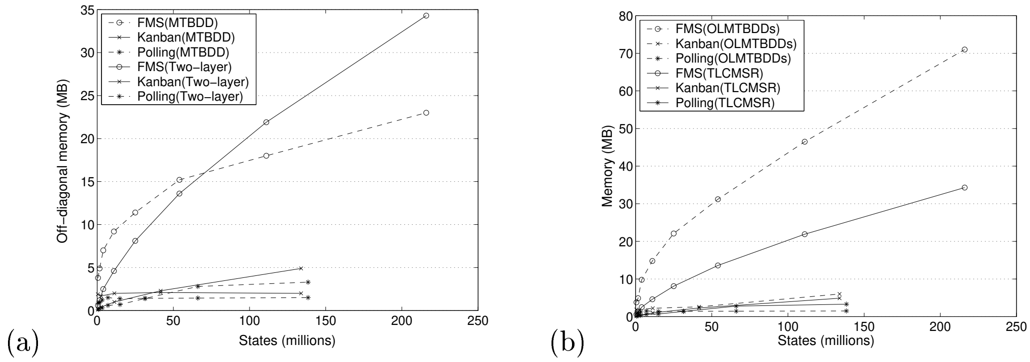


Figure 6: Memory usage, plotted against number of states: (a) Matrix (b) Matrix and vector

two bytes for each element of the diagonal vector, instead of eight bytes, exploiting the small number of distinct values (see [KM02, Meh03] for details).

4.3 Memory Consumption

Now, we consider memory requirements in more detail. For both the MTBDD and two-layer approaches, the memory requirements comprise the matrix, the iteration vector (n doubles), the diagonals vector, and the vector $\tilde{X}$ (see Section 3.1). The storage for diagonal and iteration vectors is, in fact, independent of the approach used, and is thus ignored in our analysis. Clearly, the memory for the matrix will depend heavily on the data structure used. The vector $\tilde{X}$ will also be affected since its size is equal to the maximum dimension of matrix block used, which is governed by the choice of l_b. In Figure 6 we plot memory usage against number of states: (a) shows memory for the matrix alone; (b) shows memory for the matrix and $\tilde{X}$ vector combined.

First, we note that the increase in memory for the two-layer data structure over the MTBDD is reasonably small: the increase in explicit storage is offset by the use of the more efficient compact MSR format. Furthermore, when we consider the memory for matrix and vector combined, we see that the two-layer approach actually requires less memory for large

CTMCs. This is because we are able to select a larger value of l_b, making the maximum block size smaller and reducing the size of $\tilde{X}$. For FMS ($k = 13$), for example, the total memory requirements (matrix and vector) for the two-layer and MTBDD approaches are 35MB and 71MB, respectively. For comparison, we mention here that an implementation of Gauss-Seidel using standard sparse matrix storage, although not needing the $\tilde{X}$ vector, requires 30GB to store the off-diagonal matrix alone.

4.4 Further Results

We conclude by briefly describing a number of additional experimental results. We have developed an *out-of-core* implementation of our technique, where the two-layer storage of the matrix is kept in RAM, as before, but (iteration and diagonal) vectors are stored on disk and retrieved as required. This idea has been used previously in [KMNP02], where an out-of-core version of MTBDD-based numerical solution was implemented in similar fashion. The approach is particularly promising in the context of these symbolic implementations, where the limiting factor is usually the storage required for vectors.

In Table 2, we present average times per iteration and numbers of iterations for numerical solution using the out-of-core implementation of the two-layer approach. We use the same case studies as in the previous section, for brevity only showing the largest few CTMCs for each one. Times are presented for two machines: (1) the workstation used in the previous section (2) a more powerful workstation (2.80GHz, 1GB RAM, 60GB SCSI disk). The results from the first machine show that using out-of-core instead of in-core storage for the vectors allows significantly larger (by approximately an order of magnitude) models to be solved on a standard workstation. The results for the second machine demonstrate the scalability of the out-of-core implementation: we successfully solved a CTMC with over 600 million states.

Model	k	States (n)	a/n	Time per iter. (sec.)		Iterations
				Machine 1	Machine 2	
FMS	12	111,414,940	9.7	2,319	170	1,798
	13	216 427 680	9.9	6,086	327	1,977
Kanban	8	133,865,325	11.3	558	139	999
	9	384,392,800	11.6	2,049	407	1,211
	10	1,005,927,208	12.0	–	1,416	> 50 *
Polling	23	289,406,976	12.3	1,518	264	38
	24	603,979,776	12.8	–	460	38

* Was not run to completion due to excessive time requirements.

Table 2: Timing results for the out-of-core implementation

Computation for the largest model considered (1 billion states) was shown to be feasible but we were unable to wait for the process to complete.

5 Conclusions and Future Work

In this paper, we have presented improvements to MTBDD-based steady-state solution of CTMCs which remove one of its main deficiencies. We showed that, by reducing the storage costs of an MTBDD and then converting it to a two-layer data structure, it is feasible to implement the Gauss-Seidel iterative method. We have given experimental results from our implementation, compared them to existing MTBDD-based implementations and shown that the new approach compares favourably in terms of both time and memory usage. We also developed an out-of-core version of two-layer compact MSR and demonstrated that it allows solution of CTMCs with as many as 600 million states on a standard workstation.

We believe that the technique should be equally applicable to other symbolic approaches, such as those based on the Kronecker representation. The two-level data structure also provides an ideal basis to implement block iterative methods, a direction we wish to pursue in the near future. Furthermore, initial investigations suggest that it is well suited to a parallel implementation. In addition, we are working both to improve the efficiency of the implementation and to integrate the techniques into the probabilistic model checker PRISM.

References

[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. ICCAD'93*, pages 188–191, 1993.

[BKH99] C. Baier, J.-P. Katoen, and H. Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In *Proc. CONCUR'99*, volume 1664 of *LNCS*, pages 146–161. Springer, 1999.

[CFM+93] E. Clarke, M. Fujita, P. McGeer, 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)*, 1993.

[CM99] G. Ciardo and A. Miner. A Data Structure for the Efficient Kronecker Solution of GSPNs. In *Proc. PNPM'99*, Zaragoza, 1999.

[CT93] G. Ciardo and K. S. Trivedi. A Decomposition Approach for Stochastic Reward Net Models. *Performance Evaluation*, 18(1):37–59, 1993.

[CT96] G. Ciardo and M. Tilgner. On the use of Kronecker Operators for the Solution of Generalized Stochastic Petri Nets. ICASE Report 96-35, 1996.

[DS98a] D. D. Deavours and W. H. Sanders. An Efficient Disk-based Tool for Solving Large Markov Models. *Performance Evaluation*, 33(1):67–84, 1998.

[DS98b] D. D. Deavours and W. H. Sanders. "On-the-fly" Solution Techniques for Stochastic Petri Nets and Extensions. *IEEE Transactions on Software Engineering*, 24(10):889–902, 1998.

[HMKS99] H. Hermanns, J. Meyer-Kayser, and M. Siegle. Multi Terminal Binary Decision Diagrams to Represent and Analyse Continuous Time Markov Chains. In *Proc. NSMC'99*, 1999.

[HMPS96] G. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Markovian Analysis of Large Finite State Machines. *IEEE Transactions on CAD*, 15(12), 1996.

[IT90] O. Ibe and K. Trivedi. Stochastic Petri Net Models of Polling Systems. *IEEE Journal on Selected Areas in Communications*, 8(9):1649–1657, 1990.

[KM02] M. Kwiatkowska and R. Mehmood. Out-of-core solution of large linear systems of equations arising from stochastic modelling. In *Proc. PAPM/PROBMIV'02*, volume 2399 of *LNCS*, pages 135–151. Springer, 2002.

[KMNP02] M. Kwiatkowska, R. Mehmood, G. Norman, and D. Parker. A Symbolic Out-of-Core Solution Method for Markov Models. In *Proc. Parallel and Distributed Model Checking (PDMC'02)*, volume 68 issue 4 of *ENTCS*, 2002.

[KNP02a] M. Kwiatkowska, G. Norman, and D. Parker. PRISM: Probabilistic Symbolic Model Checker. In *Proc. TOOLS'02*, volume 2324 of *LNCS*, 2002. www.cs.bham.ac.uk/~dxp/prism.

[KNP02b] M. Kwiatkowska, G. Norman, and D. Parker. Probabilistic Symbolic Model Checking with PRISM: A Hybrid Approach. In *Proc. TACAS 2002*, volume 2280 of *LNCS*, April 2002.

[Meh03] R. Mehmood. Serial Disk-based Analysis of Large Stochatic Models. In *Proc. Dagstuhl Research Seminar*, 2003. To appear in an LNCS Tutorial Volume.

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

[Pla85] B. Plateau. On the Stochastic Structure of Parallelism and Synchronisation Models for Distributed Algorithms. In *Proc. 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, 1985.

[Ste94] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.