A data structure for the efficient Kronecker solution of GSPNs

Gianfranco Ciardo [*] Andrew S. Miner [†]
Department of Computer Science
College of William and Mary
Williamsburg, VA, 23185
{ciardo,asminer}@cs.wm.edu

Abstract

Kronecker-based approaches have been proposed for the solution of structured GSPNs with extremely large state spaces. Representing the transition rate matrix using Kronecker sums and products of smaller matrices virtually eliminates its storage requirements, but introduces various sources of overhead. We show how, by using a new data structure which we call matrix diagrams, *we are able to greatly reduce or eliminate many of these overheads, resulting in a very efficient overall solution process.*

1. Introduction

Generalized Stochastic Petri Nets (GSPNs) [1] and related models (e.g., stochastic reward nets [15], stochastic activity networks [18]) are widely accepted as one of the best high-level formalisms to define very large and complex continuous-time Markov chains (CTMCs).

Their numerical solution, however, is limited by the well-known state-space explosion problem: while the GSPN modeling a (finite) system might be quite compact, the underlying CTMC can easily have an enormous set of states. Any algorithm for the "exact" solution of a GSPN must then be able to cope with a very large reachability set, or state space, $\mathcal{S}$. This in turn affects the size of the infinitesimal generator $\mathbf{Q}$, a square matrix of dimension $|\mathcal{S}|$ (fortunately very sparse), and of the vectors required by the iterative numerical solution algorithms. We consider the stationary solution of ergodic models, hence, we need to store at the very least one probability vector $\boldsymbol{\pi}$ of dimension $|\mathcal{S}|$, solution of $\boldsymbol{\pi} \cdot \mathbf{Q} = \mathbf{0}$; our work, however, applies just as well to the study of absorbing CTMCs or the transient solution of arbitrary CTMCs.

[*] G. Ciardo's work was supported in part by the National Aeronautics and Space Administration under NASA Grant NAG-1-2168.

[†] A.S. Miner's work was supported by fellowships from the NASA Graduate Student Researchers Program (Grant NGT-1-52195) and the Virginia Space Grant Consortium.

Even more so than the large computational requirements, the storage of $\mathcal{S}$, $\mathbf{Q}$, and $\boldsymbol{\pi}$ is then the main limitation to the applicability of the numerical approach to practical models of interest. Researchers have attacked this problem in various ways. If we restrict our focus on "exact" numerical approaches (i.e., ignoring approximate methods such as truncation and decomposition) for "general" models (i.e., without assuming special properties in the model that would allow us to use ad-hoc solution algorithms), much recent work is geared toward coping with the size of the data structures used to store $\mathcal{S}$ and $\mathbf{Q}$.

For example, Sanders and his group have proposed to store the matrix $\mathbf{Q}$ on secondary memory (a fast, large hard disk) and retrieve it in "chunks" that are operated upon by appropriate block-oriented numerical algorithms [10], or to generate the entries of $\mathbf{Q}$ "on-the-fly" from the high-level model description, as needed [11]. Both approaches have merit, but they also have limitations. The former approach is still limited by the memory available on the hard disk and it forces the use of a numerical algorithm that matches the amount of block computation with the time to retrieve a block from disk; if this delicate balance is perturbed, performance suffers. The latter approach requires that the transition rate from state i to state j be efficiently computable from the high-level model; this is often not the case when the models have many immediate transitions that affect the logical behavior but do not advance the modeled time.

A completely different approach based on the Kronecker description of the matrix $\mathbf{Q}$ has been embraced instead by several researchers, following the publication of Plateau's results on synchronized automata networks [17]. In particular, the approach has been adapted to queuing networks (QNs), GSPNs, and related high-level models by Donatelli [12] and Buchholz and Kemper [4, 6, 13]. Under the structural assumption that the model is composed of K interacting submodels, the matrix $\mathbf{Q}$ can be expressed as the sum of a number of Kronecker products of K small matrices. Then, the memory requirements for the storage of $\mathbf{Q}$ become negligible compared to that of the solution vector $\boldsymbol{\pi}$. However, the computational complexity of the approach can increase in practice by a factor K in the worst case [5]. Furthermore, especially in the initial proposals, this approach was based on using the cross-product $\hat{\mathcal{S}}$ of the state spaces of the individual submodels instead of the state space $\mathcal{S}$ of the overall model. When $\mathcal{S}$ is actually a strict subset of $\hat{\mathcal{S}}$, the approach still works but memory and execution inefficiencies arise. In a badly chosen decomposition, the memory wasted by allocating vectors of size $|\hat{\mathcal{S}}|$ might more than offset the savings obtained by not storing $\mathbf{Q}$ explicitly.

Another data structure requiring in principle much memory is the state space $\mathcal{S}$. While $\mathcal{S}$ is not explicitly used during a standard numerical solution, it is needed before the solution itself, to generate $\mathbf{Q}$, and after the solution, to compute the output measures of interest. $\mathcal{S}$ can be visualized as a matrix with as many columns as the places in the GSPN, $|\mathcal{P}|$, and as many rows as there are reachable markings, $|\mathcal{S}|$. The entries can then be booleans (if the Petri net is safe) or integers, for the general GSPN models we assume. To make things worse, in many GSPNs, most places contain tokens, eliminating the possibility of a simple sparse storage approach. However, it is possible to use much fewer than $|\mathcal{P}| \cdot |\mathcal{S}|$ integers. In [8], we introduced a multi-level technique that uses essentially $|\mathcal{S}|$ integers, an amount of memory certainly no larger than that required for the solution vector. In [14], we further improved on this idea by combining the multilevel approach and its potential for various optimizations with binary decision diagrams [2, 16]. The resulting symbolic generation technique, based on multi-valued decision diagrams [19], can be used to generate transformation state spaces very efficiently in terms of memory and time, indeed so efficiently that, for practical GSPN models, $\mathcal{S}$ becomes a negligible factor in terms of memory usage, just as the Kronecker matrices.

Further progress in this area can then come mostly from two directions. The most important contribution would obviously be to reduce the size of the remaining data structures, π and any other vector required by the numerical solution, which are as of now the only real limitation from a memory standpoint. Unfortunately, this appears to be a very difficult problem, and it might not be at all solvable if we restrict ourselves to exact solutions. If one accepts instead to cope with these large vectors, it is possible to study GSPNs with a few tens of millions of markings, using a modern well-equipped workstation. The second direction for research is then how to speed up the solution process by reducing the overhead inherent in the use of the Kronecker approach. This is the subject of our work.

We investigate the idea of using a new structure, somewhat analogous to the decision diagrams we use for the exploration of the state space [14]. However, instead of $\mathcal{S}$, we use this structure to store the reachability graph and the transition rate between markings, that is, the transition rate matrix $\mathbf{R}$ (which equals $\mathbf{Q}$ except for having zeros on its diagonal). Since the new data structure essentially stores a matrix, we name it "matrix diagram". With it, we can store $\mathcal{S}$ and $\mathbf{R}$ very efficiently. More importantly, this data structure speeds up the Kronecker solution, since it lends itself perfectly well to the type and order of access to the entries of $\mathbf{R}$ that are required by the numerical algorithms based on a Kronecker representation. We stress that, with matrix diagrams, there is no need to store the Kronecker matrices separately: a single data structure encodes both the composition of each marking pair and the rate between them.

The results we report show a substantial improvement over previously known methods, due both to the inherent greater efficiency of the data structure we propose and to the fact that more efficient methods requiring column access to $\mathbf{R}$ (e.g., Gauss-Seidel) can be employed without additional overhead, instead of relying on slower methods that only require row access to $\mathbf{R}$ (e.g., Power or Jacobi).

In Sect. 2, we briefly recall the approach based on Kronecker operators for the solution of GSPNs, and discuss its potential pitfalls. In Sect. 3, we describe a new data structure and a set of manipulation routines that can substantially increase the efficiency of a Kronecker implementation. Numerical results for our proposed approach are reported in Sect. 4. Finally, Sect. 5 concludes with a summary of our contribution.

2. The Kronecker approach

In this section, we recall the basic Kronecker operators and their use in the solution of GSPNs, with the help of a running example. Then, we examine several problems that negatively affect the solution complexity in a Kronecker-based solution.

2.1. Definitions

We first recall the definition of the Kronecker product $\mathbf{A} = \bigotimes_{k=1}^{K} \mathbf{A}^k$ of K square matrices $\mathbf{A}^k \in \mathbb{R}^{n_k \times n_k}$. Throughout our presentation, we assume a fixed *mixed-base* sequence $(n_1, \ldots, n_K)$, which, as we will later see, corresponds to the sizes of the "local state spaces" for the subnets in our GSPNs. Given this base, we can then identify a sequence $(l_1, \ldots, l_K)$ with its mixed-base value

$$(\cdots((l_1)n_2 + l_2)n_3 \cdots)n_K + l_K = \sum_{k=1}^{K} \left(l_k \prod_{m=k+1}^{K} n_m \right)$$

and compute the Kronecker product as

$$\mathbf{A}_{(i_1,\ldots,i_K),(j_1,\ldots,j_K)} = \mathbf{A}^1_{i_1,j_1} \cdot \mathbf{A}^2_{i_2,j_2} \cdots \mathbf{A}^K_{i_K,j_K}.$$

The Kronecker sum $\bigoplus_{k=1}^{K} \mathbf{A}^k$ is defined in terms of Kronecker products, as

$$\begin{aligned}\bigoplus_{k=1}^{K} \mathbf{A}^k &= \sum_{k=1}^{K} \mathbf{I}_{n_1} \otimes \cdots \otimes \mathbf{I}_{n_{k-1}} \otimes \mathbf{A}^k \otimes \mathbf{I}_{n_{k+1}} \otimes \cdots \otimes \mathbf{I}_{n_K} \\ &= \sum_{k=1}^{K} \mathbf{I}_{\prod_{m=1}^{k-1} n_m} \otimes \mathbf{A}^k \otimes \mathbf{I}_{\prod_{m=k+1}^{K} n_m},\end{aligned}$$

where $\mathbf{I}_x$ is the identity matrix of size $x \times x$.

We assume that the GSPN model under study is composed of K ergodic GSPNs having disjoint sets of places

$\mathcal{P}^k$, but not necessarily disjoint sets of transitions $\mathcal{T}^k$. The set of transitions of the overall model, $\mathcal{T} = \cup_{k=1}^{K} \mathcal{T}^k$, can be partitioned into a set $\mathcal{T}_S = \{t : \exists k, l, k \neq l, t \in \mathcal{T}^k \cap \mathcal{T}^l\}$ of synchronizing transitions, belonging to one or more submodels, and K sets $\mathcal{T}_L^k = \mathcal{T}^k \setminus \mathcal{T}_S$ of transitions local to each submodel. Synchronizing transitions are usually assumed to be timed, and for simplicity we do so too, although we have shown in [9] how to manage immediate synchronizing transitions. In other words, any immediate transition in the GSPN must be local, and we assume from now on that it has been eliminated, so that all transitions are timed.

Since each submodel is ergodic, we can build the set of (tangible) submarkings of the k^{th} submodel in isolation. If this set contains n_k submarkings, we can define $\hat{\mathcal{S}}^k = \{0, 1, \ldots, n_k - 1\}$, and treat it as if it were a set of reachable submarkings; that is, we identify the submarkings with their *indices*. This is because the actual submarkings, that is the number of tokens in the places of $\mathcal{P}^k$, is not relevant to our data structure, and can then be conceptually stored (once) in a two-dimensional array of size $|\mathcal{P}^k| \times n_k$, since n_k is assumed to be small. To avoid confusion in the remainder of the paper, we use the notation "$[a_1, \ldots, a_x]$" to indicate a *marking* or a submarking, with a_i being the number of tokens in place p_i, while we use the notation "$(a_1, \ldots, a_K)$" to indicate a global *state*, where a_k is the index of the submarking for the k^{th} submodel.

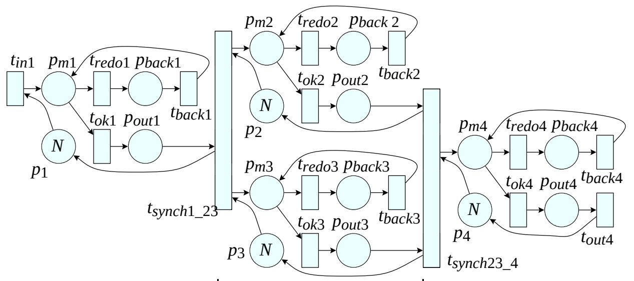
We can define the *potential state space* of the overall model as $\hat{\mathcal{S}} = \hat{\mathcal{S}}^1 \times \cdots \times \hat{\mathcal{S}}^K$; any tangible state $(i_1, \ldots, i_K)$ in the reachability set $\mathcal{S}$ belongs to $\hat{\mathcal{S}}$, but not all elements of $\hat{\mathcal{S}}$ are necessarily reachable.

The key idea in the Kronecker solution of such a GSPN lies in expressing the transition rate matrix $\mathbf{R} \in I\!R^{|\mathcal{S}| \times |\mathcal{S}|}$ as the submatrix, corresponding to the reachable markings only, of the matrix $\hat{\mathbf{R}} \in I\!R^{|\hat{\mathcal{S}}| \times |\hat{\mathcal{S}}|}$ defined as

$$\hat{\mathbf{R}} = \sum_{t \in \mathcal{T}_S} \cdot \bigotimes_{k=1}^{K} \mathbf{W}^k(t) + \bigoplus_{k=1}^{K} \mathbf{R}^k \qquad (1)$$

where $\mathbf{W}^k(t) \in I\!R^{n_k \times n_k}$ describes the effect of synchronizing transition t on the k^{th} submodel, with $\mathbf{W}^k(t) = \mathbf{I}$ for $t \notin \mathcal{T}^k$, and $\mathbf{R}^k \in I\!R^{n_k \times n_k}$ describes the effect of the local transitions $\mathcal{T}_L^k$ on the k^{th} submodel itself.

It should be noted that the diagonal of $\mathbf{Q}$ is guaranteed to contain only nonzero elements, if the CTMC is ergodic. Hence, in practical implementations of a standard numerical solution where $\mathbf{Q}$ is stored explicitly using sparse storage, it is best to store $\mathbf{Q}$'s off-diagonal entries in the matrix $\mathbf{R}$, plus a full vector $\mathbf{h}$ of the expected holding times, where $\mathbf{h}_i = -\mathbf{Q}_{i,i}^{-1}$, instead of storing $\mathbf{Q}$ as a single matrix. In the Kronecker case, the diagonal of $\mathbf{Q}$ can also be encoded as a Kronecker expression, using a second set of local matrices [20]. We follow [20] in the implementation of our tool SMART [7], by storing the row sums of the $\mathbf{W}^k$ and $\mathbf{R}^k$ matrices and using them in a second Kronecker expression to compute the expected holding times. Alternatively, SMART can store $\mathbf{h}$ as a full vector, thereby increasing the



Transition	Rate	Transition	Rate	Transition	Rate
t_{in1}	1.0	t_{redo1}	0.36	t_{ok1}	0.84
t_{out4}	0.9	t_{redo2}	0.42	t_{ok2}	0.98
t_{synch1_23}	0.4	t_{redo3}	0.39	t_{ok3}	0.91
$t_{synch23\_4}$	0.5	t_{redo4}	0.33	t_{ok4}	0.77
$t_{back1,\ldots,4}$	0.3				

Figure 1. Kanban Model

memory requirements ($\mathbf{h}$ occupies as much space as the stationary probability vector $\boldsymbol{\pi}$) while reducing the computation times.

2.2. Running example

As a running example, we use the GSPN in Fig. 1, taken from [9], modeling a kanban manufacturing system. The input parameter N affects the size of the underlying CTMC. When describing in detail the state space and transition rate matrix underlying the GSPN, we use the value $N = 1$; in our experimental results of Sect. 4, we increase N up to seven. We partition the net into four subnets such that subnet k contains places $\{p_{m_k}, p_{back_k}, p_k, p_{out_k}\}$. This yields two synchronizing transitions, t_{synch1_23} and $t_{synch23\_4}$, while the others are local transitions.

The local state spaces (all possible submarkings) are:

Ψ_k	$(p_{m_k}, p_{back_k}, p_k, p_{out_k})$	Ψ_k	$(p_{m_k}, p_{back_k}, p_k, p_{out_k})$
0	[0, 0, 1, 0]	1	[1, 0, 0, 0]
2	[0, 1, 0, 0]	3	[0, 0, 0, 1]

and $\hat{\mathcal{S}}^1 = \hat{\mathcal{S}}^2 = \hat{\mathcal{S}}^3 = \hat{\mathcal{S}}^4 = \{0, 1, 2, 3\}$. Then, for instance, the global state $(3, 0, 0, 2)$ corresponds to the marking with one token in places p_{out_1}, p_2, p_3 and p_{back_4}, and zero tokens in the remaining places. From this state, we can fire transition t_{synch1_23} to reach state $(0, 1, 1, 2)$ or fire transition t_{back4} to reach state $(3, 0, 0, 1)$.

The resulting Kronecker matrices for the kanban net are shown in Fig. 2. We see that the transition rate from state $(3, 0, 0, 2)$ to state $(0, 1, 1, 2)$ is given by the only nonzero term in row $3 \cdot 4^3 + 0 \cdot 4^2 + 0 \cdot 4^1 + 2 \cdot 4^0 = 194$ and column $0 \cdot 4^3 + 1 \cdot 4^2 + 1 \cdot 4^1 + 2 \cdot 4^0 = 22$ for the sum of the Kronecker products in Eq. (1),

$$\begin{aligned} &\mathbf{W}^1(t_{synch1\_23})[3,0] \cdot \mathbf{W}^2(t_{synch1\_23})[0,1] \cdot \\ &\mathbf{W}^3(t_{synch1\_23})[0,1] \cdot \mathbf{W}^4(t_{synch1\_23})[2,2] \quad = \quad 0.4. \end{aligned}$$

Similarly, the transition rate from state $(3, 0, 0, 2)$ to state $(3, 0, 0, 1)$ is given by the nonzero term in row 194 and column $3 \cdot 4^3 + 0 \cdot 4^2 + 0 \cdot 4^1 + 1 \cdot 4^0 = 193$ for the Kronecker

$$\mathbf{W}^1(t_{synch1\_23}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.4 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{W}^1(t_{synch23\_4}) = \mathbf{I} \qquad \mathbf{R}^1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0.36 & 0.84 \\ 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{W}^2(t_{synch1\_23}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{W}^2(t_{synch23\_4}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{R}^2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0.42 & 0.98 \\ 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{W}^3(t_{synch1\_23}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{W}^3(t_{synch23\_4}) = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{R}^3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0.39 & 0.91 \\ 0 & 0.3 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{W}^4(t_{synch1\_23}) = \mathbf{I} \qquad \mathbf{W}^4(t_{synch23\_4}) = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{R}^4 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0.33 & 0.77 \\ 0 & 0.3 & 0 & 0 \\ 0.9 & 0 & 0 & 0 \end{bmatrix}$$

Figure 2. Kronecker matrices

sum in Eq. (1),

$$\mathbf{I}[3,3] \cdot \mathbf{I}[0,0] \cdot \mathbf{I}[0,0] \cdot \mathbf{R}^4[2,1] = 0.3.$$

These correspond to the rates for transitions t_{synch1_23} and t_{back4}, respectively.

2.3. Problems with the Kronecker approach

As shown in the previous section, the Kronecker approach allows us to represent the matrix $\hat{\mathbf{R}}$ very efficiently in terms of memory. However, this compactness comes at a price. In this section, we examine some of the inefficiencies connected with the Kronecker approach. We have analyzed these problems in depth in [5]; in the next section we show how these can be alleviated or eliminated using our data structures.

Potential vs. actual state space. As stated in the introduction, initial approaches based on Kronecker operators worked on the potential state space $\hat{\mathcal{S}} = \hat{\mathcal{S}}_1 \times \cdots \hat{\mathcal{S}}_K$. This makes the storage of the overall state space unnecessary (only the local state spaces $\hat{\mathcal{S}}_k$ must be stored), but it also forces us to allocate solution vectors of size $|\hat{\mathcal{S}}|$ instead of $|\mathcal{S}|$, resulting in a potentially fatal problem ($\hat{\mathcal{S}}$ can be orders of magnitude larger than $\mathcal{S}$).

Need to skip unreachable states. Using the potential instead of the actual state space can also affect the efficiency of the approach. In earlier implementations, the numerical solution approach used the Power or Jacobi method, since these can be implemented so that they access the matrix $\hat{\mathbf{R}}$ by rows. Since "we cannot reach an unreachable state from a reachable state", if the initial probability vector is initialized with probability mass only on some reachable states (e.g., the initial state with probability 1 and every other state with probability 0) and the process is ergodic, we are guaranteed that exactly and only the reachable states will have nonzero entries in the solution vector. In the Jacobi iterations, we can then avoid ever considering any entry in $\hat{\mathbf{R}}$ that is related to an unreachable state by skipping over any row index whose corresponding entry in the current iteration vector is zero. Alternatively, we either need to keep track of the set of indices for which the iterate has nonzero entries, or, equivalently, we can build the set of reachable states before beginning the numerical computation, and retrieve their indices (their lexicographic position in $\hat{\mathcal{S}}$) at each iteration of the numerical method. This latter method is of course preferred when $\hat{\mathcal{S}}$ is much larger than $\mathcal{S}$, since, in this case, the cost of testing for zero entries in the iteration vector could dominate the computation.

Difficulties with access by columns. Since the Power and Jacobi method have slow convergence, later approaches have shown how to use faster methods such as Gauss-Seidel. Unfortunately, these require sequential access to each column of $\mathbf{R}$, and since "we may reach a reachable state from an unreachable state", these newer approaches must prevent accessing such spurious entries. If the vectors are of size $|\hat{\mathcal{S}}|$, the problem is automatically solved, in a way, as long as the probability of any unreachable state is initially set to zero. However, the complexity is now affected by the number of spurious entries in the columns corresponding to reachable states.

Logarithmic overhead. Another way of dealing with the problem of spurious entries, and at the same time solving the even more important memory limitations inherent in using data structure of size $|\hat{\mathcal{S}}|$, is to use vectors of size $|\mathcal{S}|$ throughout. However, since $\hat{\mathbf{R}}$ and not $\mathbf{R}$ is encoded by the Kronecker expression in Eq. (1), we still need to map "potential indices" (from 0 to $|\hat{\mathcal{S}}| - 1$) to "actual indices" (from 0 to $|\mathcal{S}| - 1$). Every approach published so far [5, 13, 21] maps these indices using a binary search of some type on the data structure used to store $\mathcal{S}$, resulting in a logarithmic overhead.

Interleaving reduces overhead, precludes column access. The logarithmic overhead just mentioned is $O(\log |\mathcal{S}|)$ using a straightforward implementation. At best, the logarithmic overhead can be reduced to $O(\log |\hat{\mathcal{S}}_K|)$ by "interleaving" the components of row and column indices in the multiplication algorithms [5]. Also, each entry of $\mathbf{R}$ is conceptually obtained as a product of K real numbers, but many entries share (sub)products of $2, 3, \ldots, K-1$

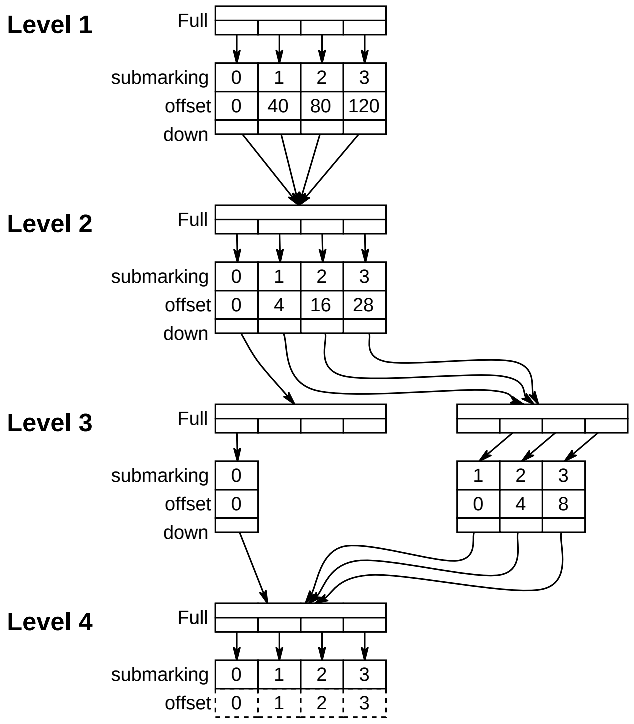


Figure 3. State space data structure

of these numbers; an important advantage of interleaving is that it amortizes the multiplications that must be performed, by better exploiting these common subproducts. However, the interleaving approach presented in [5] precludes access by columns, so we cannot use Gauss-Seidel or any other algorithm with similar access requirements. As pointed out in [5], this forces us to choose between a slower-converging method with smaller per-iteration cost, and a faster-converging method with higher per-iteration cost.

3. Our data structures

3.1. Decision diagrams to store the state space

The structure we use to represent the state space is essentially a variant of a multi-valued decision diagram (MDD) [19]. Our technique for generating and representing the state space using decision diagrams (either binary or multi-valued) is described in detail in [14], so we discuss it only briefly here, for completeness.

State space generation with decision diagrams utilizes two key ideas. The first is to represent a state space in levels, or hierarchically, and was presented in [8]. A set of K-element integer vectors (such as $\mathcal{S}$) is represented by storing a set of integers, where each integer i_1 has associated with it a set of $(K-1)$-element vectors of integers. These $(K-1)$-element vectors are combined with i_1 to form K-element vectors. The second key idea is to generate the state space *symbolically*: each iteration can potentially discover a large set of reachable states. As these iterations only require manipulations of decision diagrams (instead of manipulating individual states), symbolic generation using decision diagrams is typically extremely efficient in terms of both memory and CPU requirements.

Once the state space has been generated using decision diagrams, it is transformed into a less dynamic structure containing some additional information for computing state indices. The final representation of the actual state space of our running example is shown in Fig. 3. For example, following the downward pointer below submarking 3 at level 1, we obtain all the states of the form $(3, \bullet, \bullet, \bullet)$. If we then follow the downward pointer below submarking 0 at level 2, we obtain all the states of the form $(3, 0, \bullet, \bullet)$. Repeating this process, we can determine that state $(3, 0, 0, 2)$ is reachable, while $(3, 0, 2, 2)$ is not. Notice that many of the downward pointers lead to the same structure, indicating equivalent sets. For instance, we can quickly see that for any reachable state of the form $(3, \bullet, \bullet, \bullet)$, state $(2, \bullet, \bullet, \bullet)$ is also reachable, and vice-versa.

An important property of our structure is that each level maintains both full and sparse storage. The *Full* pointers allow us to determine if a given submarking is reachable in one operation. For instance, following the downward pointers from submarking 3 at level 1 and then submarking 0 at level 2, we can quickly tell that no reachable state of the form $(3, 0, 2, \bullet)$ exists, since element 2 of the *Full* array is empty. The rest of the structure uses sparse storage, which not only saves memory, but also allows us to quickly iterate over the reachable states, skipping the unreachable ones. If we did not use array *Full*, a binary search could be used to find a given submarking index at each level, but this would add a logarithmic-time overhead [5].

Finally, the index of a state is determined using the *offset* quantities. The offset specifies the number of accumulated reachable states so far, not including the current submarking. For instance, looking at the level 1 structure, we see that there are 40 states of the form $(0, \bullet, \bullet, \bullet)$ and 120 states of the form $(\{0, 1, \text{or } 2\}, \bullet, \bullet, \bullet)$. Looking at the level 2 structure, we see that there are 4 states of the form $(3, 0, \bullet, \bullet)$ and 16 states of the form $(3, \{0 \text{ or } 1\}, \bullet, \bullet)$. Offsets are not explicitly stored at the last level: they can be determined from the *Full* pointers. They are depicted in Fig. 3 using dotted lines only for clarity. To determine the index of a state, we follow the downward pointers and add the offsets. So, for instance, the index of state $(3, 0, 0, 2)$ is $120+0+0+2$. For the states reachable from $(3, 0, 0, 2)$, we discover that the index of state $(0, 1, 1, 2)$ is $0+4+0+2$, and the index of state $(3, 0, 0, 1)$ is $120+0+0+1$. Note that the index of $(0, 0, 0, 0)$ is 0 and the index of $(3, 3, 3, 3)$ is 159. Thus, the indexing mechanism maps the set of reachable states onto the integers $\{0, \ldots, |\mathcal{S}|-1\}$. Also, note that the indices count the states in lexicographical order: $(0, 0, 0, 1)$ has index 1, $(0, 0, 0, 2)$ has index 2, and so on. Given any reachable state $(s_1, \ldots, s_K)$, we can easily determine the "next" reachable state (meaning the reachable state with index one higher than the current reachable state). Using the sparse structure of the decision diagram, we determine the next reachable substate at level K. If there is none, we find the next reachable substate at level $K-1$. If there are no next reachable substates all the way back to level 1, then the current state is the last reachable state. Otherwise, if we stopped at level k, we leave $s_1, \ldots, s_{k-1}$ unchanged, and move to s'_k, the next reachable substate after s_k for level k. Finally, we find the first reachable substates at levels $k+1$ through K. This operation requires $2K-1$ steps in the worst case, and 1 step in the best (and most common) case.

Decision diagrams are extremely efficient both in terms of memory and CPU: we can build and store enormous reachability sets since the execution time and the memory requirements are related to the size of the decision diagram, not to the set encoded by it. In practical models solved numerically, the memory for the decision diagram is essentially negligible in comparison to the probability vector $\boldsymbol{\pi}$.

3.2. Matrix diagrams for the transition rate matrix

We now introduce a new data structure for the storage of real matrices. Since this structure is somewhat analogous to decision diagrams, we call it a *matrix diagram*. A matrix diagram imposes a K-level hierarchical structure on each element of the matrix it represents. The element corresponding to row $(r_1, \ldots, r_K)$ and column $(c_1, \ldots, c_K)$ is computed as the product of element $[r_1, c_1]$ of a level 1 matrix with element $[r_2, c_2]$ of a level 2 matrix, and so on. Associated with each element (except for level K) is a pointer indicating which matrix must be used at the next level. For instance, if $\mathbf{B}[1,2] = 1.5$ and $\mathbf{A}[0,0] = (2.0, \mathbf{B})$ then the matrix represented by $\mathbf{A}$ has the value $2.0 \cdot 1.5$ at row $(0,1)$ and column $(0,2)$. A matrix diagram element is actually a set; this allows for additions. Thus, if $\mathbf{C}[1,2] = 1.7$ and $\mathbf{A}[1,0] = \{(2.0, \mathbf{B}), (4.0, \mathbf{C})\}$, then the matrix represented by $\mathbf{A}$ has the value $2.0 \cdot 1.5 + 4.0 \cdot 1.7$ at row $(1,1)$ and column $(0,2)$. Only single-element sets are needed at level K, since we can replace multiple elements with their sum without changing the value of the matrix represented.

A matrix diagram representing the transition rate matrix $\mathbf{R}$ for our running example is shown in Fig. 4. Each element is either the empty set (indicated by blank space) or, in our case, a set of cardinality one (indicated by a box). Notice that the level 3 matrices have different sizes. This is because rows and columns corresponding to unreachable states have been removed. To determine the transition rate from state $(3,0,0,2)$ to state $(0,1,1,2)$, we follow the appropriate path and compute the product, in this case $0.4 \cdot 1.0 \cdot 1.0 \cdot 1.0$. Similarly, the transition rate from state $(3,0,0,2)$ to state $(3,0,0,1)$ is $1.0 \cdot 1.0 \cdot 1.0 \cdot 0.3$. The numbers to the left of the matrices are the *row offsets*. These are analogous to the offsets used in decision diagrams: the index of a row is computed by summing the row offsets. As with decision diagram offsets, the row offsets do not need to be explicitly stored at the last level.

Notice that this representation corresponds exactly to the transition rate matrix $\mathbf{R}$: we do not have extra rows or columns. For instance, if we try to find the transition

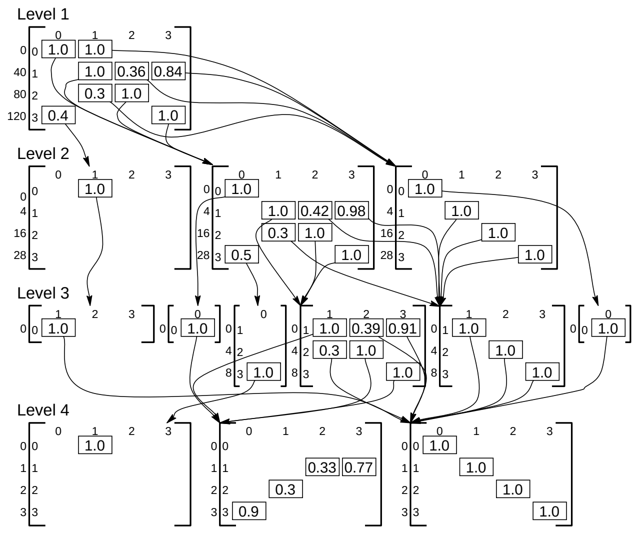


Figure 4. Matrix data structure

rate between unreachable states, such as from $(3,0,1,2)$ to $(3,0,2,2)$, we find that our level 3 matrix does not have a row 1 or column 2. With the standard Kronecker approach, instead, the transition rate is given by the nonzero term of the Kronecker sum in Eq. (1)

$$\mathbf{I}[3,3] \cdot \mathbf{I}[0,0] \cdot \mathbf{R}^3[1,2] \cdot \mathbf{I}[2,2] = 0.39$$

which corresponds to the rate for transition t_{redo3}. Matrix diagrams are able to eliminate these rows and columns because, in this case, the level 3 matrices can depend upon the rows and columns selected at levels 1 and 2. The Kronecker representation is unable to exploit dependencies of this type (generalized Kronecker products allow entries to depend on "global" indices, but this requires to store matrix entries as functions not just real numbers, and their computational complexity still suffers from the problem of unreachable states: evaluating a function only to find out that it is zero has a cost).

3.3. Kronecker implementation

Building a matrix diagram representation of $\hat{\mathbf{R}}$ using Eq. (1) is straightforward. To construct a matrix diagram for the Kronecker product $\mathbf{A} = \bigotimes_{k=1}^{K} \mathbf{A}^k$ we simply copy each matrix $\mathbf{A}^k$ as a level k matrix diagram. Except at the last level, each nonzero element of $\mathbf{A}^k$ is associated with a downward pointer to the level $k+1$ matrix diagram representing matrix $\mathbf{A}^{k+1}$. The level K matrix does not have any downward pointers; it is merely a copy of $\mathbf{A}^K$.

Addition of two K-level matrix diagrams is done by taking the union of the elements of the level 1 matrices. The result is a legal matrix diagram; however, in some cases, we can reduce the size of the resulting set. For instance, if two elements in a set have the same downward pointer, they can be merged by replacing the rate with the sum of the two rates. That is, if a set contains elements $(r_1, \mathbf{A})$ and

```
$GetColumn_k$(in: $\mathbf{M}^k$, $(c_k, \ldots, c_K)$; out: $\mathbf{column}$);
[$\mathbf{M}^k$: a level $k$ matrix of the $K$-level matrix diagram;
$c_k, \ldots, c_K$: column indices to select at this level and below;
$\mathbf{column}$: selected column stored as a sparse vector]
 1. if $k = K$ then                          • The bottom level is trivial
 2.     $\mathbf{column} \leftarrow \mathbf{M}^k_{\bullet,c_k}$;
 3. else
 4.     $\mathbf{column} \leftarrow \mathbf{0}$;
 5.     for each non-empty row $r_k$ in column $c_k$ of $\mathbf{M}^k$
 6.         for each element $(value, \mathbf{M}^{\text{down}})$ at $\mathbf{M}^k_{r_k,c_k}$
 7.             $GetColumn_{k+1}(\mathbf{M}^{\text{down}}, (c_{k+1}, \ldots, c_K), \mathbf{co})$;
 8.             $\mathbf{co} \leftarrow value \cdot \mathbf{co}$;
 9.             shift indices of $\mathbf{co}$ by the offset of row $r_k$;
10.             $\mathbf{column} \leftarrow \mathbf{column} + \mathbf{co}$;
```

Figure 5. Obtaining a matrix diagram column

$(r_2, \mathbf{A})$, we can replace those elements with $(r_1 + r_2, \mathbf{A})$. This is equivalent to the transformation

$$r_1 \cdot \mathbf{A} + r_2 \cdot \mathbf{A} \Rightarrow (r_1 + r_2) \cdot \mathbf{A}$$

Similarly, the transformation

$$r \cdot \mathbf{A} + r \cdot \mathbf{B} \Rightarrow r \cdot (\mathbf{A} + \mathbf{B})$$

can be performed by replacing elements $(r, \mathbf{A})$ and $(r, \mathbf{B})$ with $(r, \mathbf{A} + \mathbf{B})$, where $\mathbf{A} + \mathbf{B}$ refers to the addition operator for two $(K-1)$-level matrix diagrams. While the first transformation always reduces both CPU and memory requirements, the second one decreases CPU requirements but may actually use more memory: if $\mathbf{A}$ and $\mathbf{B}$ are used elsewhere, we effectively end up storing $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{A} + \mathbf{B}$.

Once we have a matrix diagram representation $\hat{\mathbf{R}}$, we can obtain one for $\mathbf{R}$ by removing the rows and columns corresponding to unreachable states (the decision diagram representation of $\mathcal{S}$ is used to detect these states). For our example, this transforms the 4×4 matrices at level 3 into the matrices of various sizes shown in Fig. 4.

As with decision diagrams, matrix diagrams must be kept *reduced* to alleviate both memory and CPU requirements. A reduced matrix diagram contains no duplicate matrices; instead, a single matrix is stored with multiple incoming pointers. The matrix diagram in Fig. 4 is reduced: no two matrices have the same elements (two elements are equal if their real values and downward pointers are equal). Matrix diagrams are kept reduced during their updates using a uniqueness table. When a matrix is created or modified, the uniqueness table is searched for a duplicate. This technique is analogous to the well-known approach used to maintain reduced binary decision diagrams [3].

3.4. Access by columns

A critical requirement for a Gauss-Seidel numerical solution is efficient access to a given column of the transition rate matrix. A matrix diagram allows for efficient access of a column if the matrices at each level of the matrix diagram have efficient access to the nonzero elements of a given column. An algorithm for obtaining a specific column of the matrix represented by a matrix diagram is depicted in Fig. 5. If the specified matrix is at the bottom level of the

$$\begin{array}{ll}
\text{Level 1} & GetColumn_1(\#1,(3,0,0,1)) \\
 & = GetColumn_2(\#3,(0,0,1)) \left\langle \begin{array}{l}\text{scale } 0.84\\ \text{shift } 40\end{array}\right\rangle \\
 & + GetColumn_2(\#2,(0,0,1)) \left\langle \begin{array}{l}\text{scale } 1.0\\ \text{shift } 120\end{array}\right\rangle = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 0.84 & 0.3 & 0.5 \\ 41 & 122 & 156 \end{bmatrix} \\
\text{Level 2} & GetColumn_2(\#2,(0,0,1)) \\
 & = GetColumn_3(\#2,(0,1)) \left\langle \begin{array}{l}\text{scale } 1.0\\ \text{shift } 0\end{array}\right\rangle \\
 & + GetColumn_3(\#3,(0,1)) \left\langle \begin{array}{l}\text{scale } 0.5\\ \text{shift } 28\end{array}\right\rangle = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 0.3 & 0.5 \\ 2 & 36 \end{bmatrix} \\
 & GetColumn_2(\#3,(0,0,1)) \\
 & = GetColumn_3(\#6,(0,1)) \left\langle \begin{array}{l}\text{scale } 1.0\\ \text{shift } 0\end{array}\right\rangle = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 1.0 \\ 1 \end{bmatrix} \\
\text{Level 3} & GetColumn_3(\#2,(0,1)) \\
 & = GetColumn_4(\#2,(1)) \left\langle \begin{array}{l}\text{scale } 1.0\\ \text{shift } 0\end{array}\right\rangle = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 0.3 \\ 2 \end{bmatrix} \\
 & GetColumn_3(\#3,(0,1)) \\
 & = GetColumn_4(\#1,(1)) \left\langle \begin{array}{l}\text{scale } 1.0\\ \text{shift } 8\end{array}\right\rangle = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 1.0 \\ 8 \end{bmatrix} \\
 & GetColumn_3(\#6,(0,1)) \\
 & = GetColumn_4(\#3,(1)) \left\langle \begin{array}{l}\text{scale } 1.0\\ \text{shift } 0\end{array}\right\rangle = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 1.0 \\ 1 \end{bmatrix} \\
\text{Level 4} & GetColumn_4(\#1,(1)) = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 1.0 \\ 0 \end{bmatrix} \\
 & GetColumn_4(\#2,(1)) = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 0.3 \\ 2 \end{bmatrix} \\
 & GetColumn_4(\#3,(1)) = \begin{array}{r}\text{val}\\ \text{row}\end{array} \begin{bmatrix} 1.0 \\ 0 \end{bmatrix}
\end{array}$$

Figure 6. Computing a column of $\mathbf{R}$

matrix diagram, we simply return the selected column (line 2). Otherwise, for each element of the level k matrix, we follow the downward pointer and recursively compute the appropriate column of this level $k+1$ matrix (line 7). This result is scaled by the value of the current element (line 8) and shifted by the offset of the current row (line 9). Scaling at each level performs the product, while the shifting determines the correct row index. Summing these shifted and scaled results (line 10) gives the selected column.

Computation of a column of $\mathbf{R}$ is shown for our running example in Fig. 6. In the example, we use the matrix diagram represented in Fig. 4 and the decision diagram represented in Fig. 3 as our data structures. A matrix is indicated by its ordinal number; for instance, $GetColumn_3(\#5, \ldots)$ refers to the fifth matrix (counting from the left) in level 3. In the example, we compute column $(3,0,0,1)$ of the matrix diagram, which is column 121 of $\mathbf{R}$. This is done by calling $GetColumn_1(\#1,(3,0,0,1),\mathbf{column})$. We see that $\mathbf{column}$ contains three entries: 0.84 at row 41, 0.3 at row 122, and 0.5 at row 156. These correspond to incoming arcs from states $(1,0,0,1), (3,0,0,2)$ and $(3,3,3,0)$ via transitions t_{ok1}, t_{back4} and $t_{synch23\_4}$, respectively.

To multiply a row vector by a specific matrix column, we can use $GetColumn$ to obtain the column and then perform the inner product of the vector by the column. This is preferred over modifying $GetColumn$ to perform the multiplication directly, as we will discuss in the next section. In particular, one (forward) Gauss-Seidel iteration for the computation of the stationary probability vector $\boldsymbol{\pi}$ is

for $j = 0$ to $|\mathcal{S}| - 1$ do $\boldsymbol{\pi}_j \leftarrow \mathbf{h}_j \cdot \sum_{i:\mathbf{R}_{i,j}>0} \boldsymbol{\pi}_i \cdot \mathbf{R}_{i,j}$,

where the nonzero entries in the column j of $\mathbf{R}$ are extracted from the matrix diagram using $GetColumn$.

3.5. Using a cache to speed up access by columns

Continuing our running example, suppose after accessing column $(3, 0, 0, 1)$ of the matrix diagram we wish to access column $(2, 0, 0, 1)$. We find that our call to $GetColumn_1(\#1, (2, 0, 0, 1))$ again generates calls to $GetColumn_2(\#2, (0, 0, 1))$ and $GetColumn_2(\#3, (0, 0, 1))$. Since these calls are identical to our previous column access, we can avoid duplicate computation, provided we saved those results in a cache of recent operations. Of course, we can also reuse them in the event that duplicate calls to $GetColumn$ occur in a single column access; for instance, $GetColumn_1(\#1, (1, 0, 0, 1))$ generates two calls to $GetColumn_2(\#3, (0, 0, 1))$. Clearly, this has enormous potential for reducing CPU requirements.

Our cache can be implemented by saving only the most recently generated column for each matrix. Thus, for each matrix in the matrix diagram, we require a single vector. Fortunately, these vectors are extremely sparse in practice, and we only need to store their nonzero elements. We can compute the maximum column size (the maximum number of nonzero elements in any column) for each matrix in a bottom-up fashion. Then, we allocate a **column** vector of maximum column size for each matrix before starting the numerical iterations, and use this vector whenever we call $GetColumn$ for that matrix.

We must *clear* the cache at level k (set all the **column** vectors at level k to a null value) whenever the specified column at level k or any level below k changes. For instance, if our last column access was $(3, 0, 0, 1)$, and now we are accessing column $(3, 2, 0, 1)$, we must clear the cache at levels 1 and 2. Thus, to maximize our cache hits, after visiting the reachable column for state $(c_1, \ldots, c_K)$, we should visit next the reachable column for the state that follows $(c_1, \ldots, c_K)$ in lexicographic order *when the string is read in reverse*, i.e., when K is the most significant (slowest changing) index and 1 is the least significant (fastest changing) index (this is not reverse lexicographic order!).

To implement this second “upside-down” order, we generate a second, upside-down, copy of the state space, and store it in a decision diagram $\mathcal{U}$. For any state $(s_1, \ldots, s_K)$ belonging to $\mathcal{S}$, there exists a state $(s_K, \ldots, s_1)$ belonging to $\mathcal{U}$ and vice-versa. $\mathcal{U}$ is used to determine the order in which to access columns: if we just accessed column $(c_1, \ldots, c_K)$, we reverse the order to obtain $(c_K, \ldots, c_1)$, find the next reachable state $(c'_K, \ldots, c'_1)$ in $\mathcal{U}$, reverse back the order of its components to obtain $(c'_1, \ldots, c'_K)$, find its index in $\mathcal{S}$, which is then next column to access.

These actions can be performed quite efficiently and, thanks to the extreme compactness of decision diagrams, storing the state space twice (i.e., as $\mathcal{S}$ and $\mathcal{U}$) still requires negligible memory. The computation of the index for the next column requires to search twice for a state, once in $\mathcal{U}$ (for a state “close” to the last one searched), and once in $\mathcal{S}$ (for an arbitrary state). The second search is the most expensive, but it still requires only $O(K)$ operations, and it must be performed only once per column (i.e., $|\mathcal{S}|$ times per

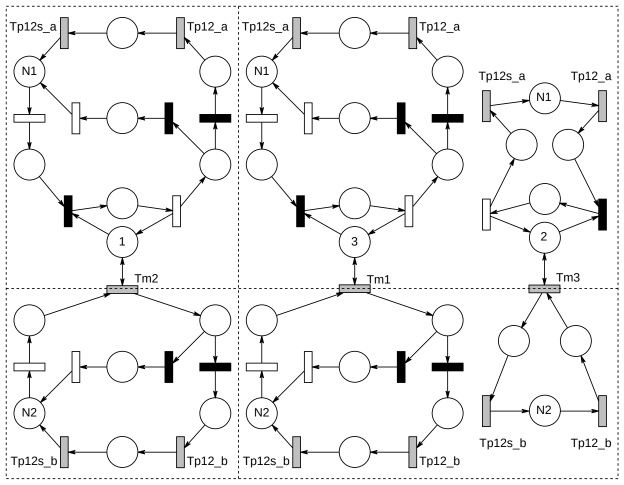


Figure 7. GSPN for the Multiclass QN

iteration); this is much less expensive than the searches required by the algorithms discussed in [5], $O(\log |\mathcal{S}|)$ or at best $O(\log |\mathcal{S}_K|)$, performed once per nonzero entry.

3.6. Benefits of our approach

In this section, we revisit the problems with the Kronecker approach (discussed in Sect. 2.3) and show how our approach addresses these problems.

Actual state space. Like advanced Kronecker approaches, our approach operates on solution vectors of size $|\mathcal{S}|$.

Skip unreachable states. We solve this by storing the state space $\mathcal{S}$, as do advanced Kronecker approaches. For efficient storage of $\mathcal{S}$, we use decision diagrams, which can also be used for Kronecker approaches. Decision diagrams allow us to find the next reachable state quickly, in $O(1)$ operations at best (and on average), and $O(K)$ at worst.

Efficient access by columns. Matrix diagrams allow us to eliminate the rows and columns corresponding to unreachable states; this solves the problem of spurious entries.

No logarithmic overhead. Using the full pointers in our decision diagrams, we can determine if a given state is reachable in $O(K)$ steps, and if a given substate is reachable in $O(1)$ steps. Thus, using “interleaving” Kronecker techniques, the overhead for decision diagrams is only $O(1)$.

Cache to avoid duplication of work. A significant source of overhead in column access using the Kronecker approach is performing the same floating point multiplications several times. By using a cache, we can reduce this wasted effort. With our approach, we visit the columns in an order to maximize our potential cache hits. This allows us to reuse floating point multiplication results as much as possible.

4. Experimental results

A prototype of our approach is implemented in the tool SMART [7]. We test our approach on two models from the literature. The first is our kanban system, our running example. The second is a multiclass QN model from [5], its GSPN shown in Fig. 7 (white transitions are timed and local, black transitions are immediate, and grey transitions are timed and synchronizing; all rates are equal one and all conflicts among immediate transitions are solved using a uniform distribution).

Both models have as input parameters the initial number of tokens in certain places (N for the kanban, N_1 and N_2 for the QN); these affect the size of the underlying CTMC.

The models were run on a 450Mhz Pentium-II workstation with 384Mb of main memory, under the Linux operating system. We compute the stationary probability vector using iterative methods (i.e., Gauss Seidel or Jacobi), stopping when the relative error between subsequent solution vectors is less than 10^{-5}:

$$\max_{i \in \mathcal{S}} \left\{ \left| \frac{\boldsymbol{\pi}_i^{(\mathrm{old})} - \boldsymbol{\pi}_i^{(\mathrm{new})}}{\boldsymbol{\pi}_i^{(\mathrm{new})}} \right| \right\} < 10^{-5}.$$

We use a uniform probability for the initial solution vector. All our timing results refer to runs that did not make use of virtual memory. Data is missing from the table in cases where the solution could not be run due to excessive memory requirements.

The tables compare our new approach, using decision diagrams and matrix diagrams, to the current state of the art Kronecker approaches described in [5]. For comparison, we also show the time requirements for explicit sparse matrix storage, for those cases where the matrix can be stored in memory. In all cases we use a full single-precision vector for the probability vector ($\boldsymbol{\pi}$). The Jacobi method requires an additional accumulator, a full double-precision vector. Holding times are computed as needed, except for the explicit storage case. In the Kronecker case, we use a Kronecker expression for the row sums. For the matrix diagram approach, we use a second matrix diagram to represent the row sums. Using a full vector to store holding times reduces the CPU time by 10% to 20%, at the cost of an extra single-precision vector. For the Kronecker case, we use a multilevel data structure for $\mathcal{S}$ as described in [5] and [8], requiring about one byte per state. If we used decision diagrams in conjunction with the standard Kronecker approach to eliminate the logarithmic overhead present in column accesses, performance would improve by a factor of 15% to 25%.

Results for the kanban model are shown in Table 1. Looking at the state space sizes and the number of nonzero entries in $\mathbf{R}$, we see that matrix $\mathbf{R}$ is extremely sparse: about 10 to 11 nonzeroes per column, on average. The memory required for our matrix diagram structures (including cache space) is about twice as much as for the Kronecker approach, but still an insignificant fraction of the memory required for the probability vector. Looking at the columns reporting the CPU time (in seconds per iteration), we find that our matrix diagram approach requires about as much time per iteration as for the Kronecker approach using Jacobi and much less CPU time (about half) as for the Kronecker approach using Gauss-Seidel. As we expect, Gauss-Seidel requires much fewer iterations than does Jacobi. We see that matrix diagram Gauss-Seidel and Kronecker Gauss-Seidel differ in the number of iterations required; this is because columns are visited in different orders. The fact that in this particular case, matrix diagrams require fewer iterations can only be attributed to good luck. The total CPU time required is found by multiplying the number of iterations by the CPU time per iteration. We see that, using decision diagrams and matrix diagrams, we can solve extremely large systems in a reasonable amount of time: 40 million states in 3 days. For the case $N = 7$, we require 0.22 seconds of CPU time for state space generation and 3,381 bytes for state space storage using decision diagrams. This is clearly a small fraction of the time and memory required to compute $\boldsymbol{\pi}$.

The results for the multiclass QN model are also shown in Table 1. The state space size is determined by parameters N_1 and N_2, corresponding to the number of high and low priority parts in the system, respectively (the rows are listed according to state space sizes). In this case, matrix diagrams require about three times as much memory as Kronecker, and require about a third as much CPU time per iteration as Kronecker for Gauss-Seidel. Again, matrix diagram Gauss-Seidel requires roughly the same time per iteration as does Kronecker Jacobi. In this case, Kronecker and matrix diagram Gauss-Seidel require the same number of iterations.

5. Conclusion

We have introduced a new efficient data structure, *matrix diagrams*, which can be used to store the very large transition rate matrices $\mathbf{R}$ that arise in the study of structured GSPNs. For practical problems, the memory requirements for the storage of the state space $\mathcal{S}$ and of $\mathbf{R}$ are quite small, usually negligible in comparison to the memory needed to store the numerical solution, a vector of size $|\mathcal{S}|$.

Such enormous memory savings for the storage of $\mathbf{R}$ have already been demonstrated by the various Kronecker implementations recently proposed. However, matrix diagrams also decrease the execution overhead inherent in the Kronecker representation when accessing the entries of $\mathbf{R}$, while, at the same time, allowing efficient access "by-column" to the entries of $\mathbf{R}$. Such a type of access is essential when using faster-converging numerical algorithms such as Gauss-Seidel.

Our results show a speedup factor of two or greater in the solution times with respect to the fastest algorithms previously proposed. Since the difference is partially due to having eliminated a logarithmic overhead factor, these differences are going to become even more relevant as improvements in hardware allow us to tackle larger models.

References

[1] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franceschinis. *Modelling with generalized stochastic Petri nets*. John Wiley & Sons, 1995.

	$\vert\mathcal{S}\vert$	$\eta(\mathbf{R})$	Matrix diagram			Kronecker					Explicit	
N	**Kanban**		Memory	Gauss-Seidel		Memory	Gauss-Seidel		JOR ($\omega = 0.9$)		Gauss-Seidel	
			(bytes)	Iters	sec/iter	(bytes)	Iters	sec/iter	Iters	sec/iter	Iters	sec/iter
3	58,400	446,400	7,599	67	1.46	3,746	97	2.56	240	1.34	97	0.34
4	454,475	3,979,850	13,518	99	12.33	6,096	149	23.69	370	11.99	149	3.04
5	2,546,432	24,460,016	21,667	139	73.09	9,486	214	147.70	527	74.09	214	18.51
6	11,261,376	115,708,992	32,702	185	336.21	14,106	289	723.30	713	359.15	-	-
7	41,644,800	450,455,040	46,678	238	1,289.91	20,388	374	2,922.80	-	-	-	-
N_1 N_2	**Multiclass QN**		Memory	Gauss-Seidel		Memory	Gauss-Seidel		Jacobi		Gauss-Seidel	
3 3	425,104	3,389,626	34,126	116	11.51	12,341	116	32.07	406	13.00	116	2.73
5 2	981,720	8,130,330	52,519	121	27.25	28,812	121	80.28	394	30.90	121	6.36
4 3	1,560,888	13,439,073	45,654	131	44.31	19,378	131	127.77	463	49.57	131	10.55
5 3	4,741,344	43,178,076	72,124	142	138.50	31,932	142	418.13	510	155.24	-	-
4 4	5,731,236	53,120,700	78,570	173	169.16	25,613	173	492.63	617	190.33	-	-
5 4	17,409,168	169,728,572	113,060	183	528.80	38,167	183	1,670.61	662	596.09	-	-

Table 1. Results for the Kanban and multiclass QN models

[2] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comp.*, 35(8):677–691, Aug. 1986.

[3] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comp. Surv.*, 24(3):393–318, 1992.

[4] P. Buchholz. Numerical solution methods based on structured descriptions of Markovian models. In G. Balbo and G. Serazzi, editors, *Computer performance evaluation*, pages 251–267. Elsevier Science Publishers B.V. (North-Holland), 1991.

[5] P. Buchholz, G. Ciardo, S. Donatelli, and P. Kemper. Complexity of Kronecker operations on sparse matrices with applications to the solution of Markov models. ICASE Report 97-66 (NASA/CR-97-206274), Institute for Computer Applications in Science and Engineering, Hampton, VA, 1997. Submitted for publication.

[6] P. Buchholz and P. Kemper. Numerical analysis of stochastic marked graphs. In *Proc. 6th Int. Workshop on Petri Nets and Performance Models (PNPM'95)*, pages 32–41, Durham, NC, Oct. 1995. IEEE Comp. Soc. Press.

[7] G. Ciardo and A. S. Miner. SMART: Simulation and Markovian Analyzer for Reliability and Timing. In *Proc. IEEE International Computer Performance and Dependability Symposium (IPDS'96)*, page 60, Urbana-Champaign, IL, USA, Sept. 1996. IEEE Comp. Soc. Press.

[8] G. Ciardo and A. S. Miner. Storage alternatives for large structured state spaces. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 1245, pages 44–57, St. Malo, France, June 1997. Springer-Verlag.

[9] G. Ciardo and M. Tilgner. On the use of Kronecker operators for the solution of generalized stochastic Petri nets. ICASE Report 96-35, Institute for Computer Applications in Science and Engineering, Hampton, VA, May 1996.

[10] D. D. Deavours and W. H. Sanders. An efficient disk-based tool for solving very large Markov models. In R. Marie, B. Plateau, M. Calzarossa, and G. Rubino, editors, *Proc. 9th Int. Conf. on Modelling Techniques and Tools for Computer Performance Evaluation*, LNCS 1245, pages 58–71, St. Malo, France, June 1997. Springer-Verlag.

[11] D. D. Deavours and W. H. Sanders. "On-the-fly" solution techniques for stochastic Petri nets and extensions. In *Proc. 7th Int. Workshop on Petri Nets and Performance Models (PNPM'97)*, pages 132–141, St. Malo, France, June 1997. IEEE Comp. Soc. Press.

[12] S. Donatelli. Superposed Stochastic Automata: a class of stochastic Petri nets with parallel solution and distributed state space. *Perf. Eval.*, 18:21–26, 1993.

[13] P. Kemper. Numerical analysis of superposed GSPNs. *IEEE Trans. Softw. Eng.*, 22(4):615–628, Sept. 1996.

[14] A. S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In H. Kleijn and S. Donatelli, editors, *Application and Theory of Petri Nets 1999, Lecture Notes in Computer Science 1639 Proc. 20th Int. Conf. on Applications and Theory of Petri Nets, Williamsburg, VA, USA*. Springer-Verlag, June 1999. To appear.

[15] J. K. Muppala, G. Ciardo, and K. S. Trivedi. Modeling using Stochastic Reward Nets. In *Proc. 1st Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS'93)*, pages 367–372, San Diego, CA, USA, Jan. 1993. IEEE Comp. Soc. Press.

[16] E. Pastor, O. Roig, J. Cortadella, and R. Badia. Petri net analysis using boolean manipulation. In R. Valette, editor, *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th Int. Conf. on Applications and Theory of Petri Nets, Zaragoza, Spain)*, pages 416–435. Springer-Verlag, June 1994.

[17] B. Plateau. On the stochastic structure of parallelism and synchronisation models for distributed algorithms. In *Proc. 1985 ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems*, pages 147–153, Austin, TX, USA, May 1985.

[18] W. H. Sanders and J. F. Meyer. Reduced base model construction methods for stochastic activity networks. *IEEE J. Sel. Areas in Comm.*, 9(1):25–36, 1991.

[19] A. Srinivasan, T. Kam, S. Malik, and R. K. Brayton. Algorithms for discrete function manipulation. In *International Conference on CAD*, pages 92–95. IEEE Computer Society, 1990.

[20] W. J. Stewart, K. Atif, and B. Plateau. The numerical solution of stochastic automata networks. *Europ. J. of Oper. Res.*, 86:503–525, 1995.

[21] M. Tilgner, Y. Takahashi, and G. Ciardo. SNS 1.0: Synchronized Network Solver. In *1st International Workshop on Manufacturing and Petri Nets*, pages 215–234, Osaka, Japan, June 1996.