Compact Representation of Solution Vectors in Kronecker-Based Markovian Analysis

Peter Buchholz[1], Tuğrul Dayar[2](✉), Jan Kriege[1], and M. Can Orhan[2]

[1] Informatik IV, Technical University of Dortmund, 44221 Dortmund, Germany
{peter.buchholz,jan.kriege}@cs.tu-dortmund.de
[2] Department of Computer Engineering, Bilkent University, TR-06800 Bilkent, Ankara, Turkey
{tugrul,morhan}@cs.bilkent.edu.tr

Abstract. It is well known that the infinitesimal generator underlying a multi-dimensional Markov chain with a relatively large reachable state space can be represented compactly on a computer in the form of a block matrix in which each nonzero block is expressed as a sum of Kronecker products of smaller matrices. Nevertheless, solution vectors used in the analysis of such Kronecker-based Markovian representations still require memory proportional to the size of the reachable state space, and this becomes a bigger problem as the number of dimensions increases. The current paper shows that it is possible to use the hierarchical Tucker decomposition (HTD) to store the solution vectors during Kronecker-based Markovian analysis relatively compactly and still carry out the basic operation of vector-matrix multiplication in Kronecker form relatively efficiently. Numerical experiments on two different problems of varying sizes indicate that larger memory savings are obtained with the HTD approach as the number of dimensions increases.

Keywords: Markov chains · Kronecker products · Hierarchical Tucker decomposition · Reachable state space · Compact vectors

1 Introduction

Modelling and analysis of multi-dimensional Markov chains (MC) on high end desk-top computers is an area of research with ongoing interest. When a discrete-event dynamic system is composed of interacting subsystems, it may be possible to provide a state-based mathematical model for its behaviour as a multi-dimensional MC with each dimension of the MC representing a different subsystem and a number of events that trigger state changes at certain transition rates. In this kind of model, subsystems can change state locally by themselves, that is, independently of states the other subsystems are in, or they can change state synchronously with some or all the other subsystems depending on their local states. The state space of such a model is therefore determined by the combination of states the subsystems can be in under the operational semantics of the system. Hence, a subset of the Cartesian product of the subsystem state spaces forms the so called reachable state space. Usually not all states from the Cartesian product are reachable because synchronized transitions prohibit some specific combinations of subsystem states to be reachable [3,6]. It is important to be able to represent this reachable state space and the transitions among its states compactly and then analyse the steady-state or transient behaviour of the underlying system as accurately and as efficiently as possible.

When the reachable state space at hand is relatively large but finite, the infinitesimal generator underlying the MC can be represented as a block matrix in which each nonzero block is expressed as a sum of Kronecker products of smaller rectangular matrices [7]. This is the form of the Kronecker representation in hierarchical Markovian models [3], where rectangularity of the smaller matrices is possible due to the product state space of the modelled system being larger than its reachable state space [9]. When the product state space is equal to the reachable state space, the smaller matrices turn out to be square as in stochastic automata networks [19,20].

For Kronecker-based Markovian representations, analysis methods employ vector-Kronecker product multiplication as the basic operation [21]. Therein, the challenge is to perform this operation in as little of memory and as fast as possible. When the factors in the Kronecker product terms are relatively dense, the operation can be performed efficiently by the shuffle algorithm [10]. When the factors are relatively sparse, it may be more efficient to obtain nonzeros of the generator in Kronecker form on the fly and multiply them with corresponding elements of the vector [6]. Recently, the shuffle algorithm has been modified so that relevant elements of the vector are multiplied with submatrices of factors in which zero rows and columns are omitted [8]. This approach is shown to avoid unnecessary floating-point operations (flops) that evaluate to zero during the course of the multiplication and possibly reduces the amount of memory used. In many cases, a smaller number of flops than the shuffle algorithm and the algorithm that generates nonzeros on the fly is possible. Nevertheless, the memory allocated for the vectors in all mentioned algorithms is still proportional to the size of the reachable state space, and this size increases rapidly with the number of dimensions.

The current paper takes a different approach and attempts to reduce the amount of memory allocated to solution vectors in Kronecker-based Markovian analysis by using the hierarchical Tucker decomposition (HTD) [14,15]. HTD is originally conceived in the context of providing a compact approximate representation for dense multi-dimensional data [12] in a manner similar to the tensor-train decomposition [18], but is somewhat more suitable to our requirements in that the decomposition is available through a tree data structure with logarithmic depth in the number of dimensions. Both decompositions have the special feature of possessing approximation errors that can be user controlled, and hence, approximations accurate to machine precision are computable using them. Clearly, with such decompositions it is always possible to trade quality of approximation for compactness of representation, and how compact the solution vector in HTD format remains throughout the solution process is an interesting question to investigate. The tensor train decomposition has been applied in [13] to approximate the solution vector for models where the product space is reachable using an alternating least squares approach. HTD has, to the best of our knowledge, not been applied to structured Markov chains yet.

Here, we show that a compact solution vector in HTD format can be multiplied with a sum of Kronecker products to yield another compact solution vector in HTD format. In doing this, we note that the multiplication of the compact solution vector in HTD format with a Kronecker product term does not increase the memory requirements of the compact vector, but the addition of two compact vectors does, which necessitates some kind of truncation, hence, approximation, to be introduced to the addition operation only. Then, starting from an initial solution, the compact vector in HTD format is iteratively multiplied with the uniformized generator matrix of a given MC in Kronecker form until a predetermined stopping criterion is met. Indeed, we are interested in observing how the memory requirements of the compact solution vector in HTD format changes over the course of iterations due to the sequence of multiply, add, and truncate operations in each iteration, together with the average time it takes to perform the iteration and the influence of the approximation error on the quality of the solution. The same numerical experiment is performed with a flat solution vector as long as the reachable state space size using the modified shuffle algorithm. The two approaches are compared for their memory and timing requirements, leading us to the conclusion that compact vectors in HTD format become relatively more memory efficient as the number of dimensions increases.

In passing to the organization of the paper, we remark that compact representations for solution vectors in Markovian analysis have also been considered from the perspective of binary decision diagrams [5,16]. The proposed compact structures therein have not been time-wise competitive, whereas the approach investigated in this paper seems to be a step forward. The organization of the paper is as follows. In Sect. 2, we provide background information on HTD and the related algorithms that are be used in our Kronecker setting. In Sect. 3, we discuss implementation issues associated with using HTD within the NSolve package of the Abstract Petri Net Notation (APNN) toolbox [1,2]. In Sect. 4, we present results of numerical experiments on two different problems of varying sizes and having transitions that take place at different time scales. Section 5 concludes the paper.

2 Compact Vectors in Kronecker Setting

Let us consider a d-dimensional Markovian system, where $\mathcal{S}_h$ denotes the state space of the hth ($h = 1, \ldots, d$) component in the d-dimensional MC, and assume that $\mathcal{S}_h$ are defined on consecutive nonnegative integers starting from 0. We denote the reachable state space of the system by $\mathcal{S} \subseteq \times_{h=1}^{d} \mathcal{S}_h$, where $\times_{h=1}^{d} \mathcal{S}_h$ is the product state space. Now, let $\mathcal{S}^{(i)} = \times_{h=1}^{d} \mathcal{S}_h^{(i)}$, where $\mathcal{S}_h^{(i)}$ is a partition of $\mathcal{S}_h$ in the form of consecutive integers for $i = 1, \ldots, J$. Then $\mathcal{S}^{(1)}, \ldots, \mathcal{S}^{(J)}$ is a Cartesian product partitioning of $\mathcal{S}$ if $\mathcal{S} = \cup_{i=1}^{J} \mathcal{S}^{(i)}$ and $\mathcal{S}^{(i)} \cap \mathcal{S}^{(j)} = \emptyset$ for $i \neq j$ and $i, j = 1, \ldots, J$ [9].

The infinitesimal generator $\mathbf{Q}$ underlying the MC can be viewed as a $(J \times J)$ block matrix induced by the Cartesian product partitioning of $\mathcal{S}$ as in [7,9]

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}^{(1,1)} & \dots & \mathbf{Q}^{(1,J)} \\ \vdots & \ddots & \vdots \\ \mathbf{Q}^{(J,1)} & \dots & \mathbf{Q}^{(J,J)} \end{bmatrix}.$$

Block (i,j) of $\mathbf{Q}$ for $i,j = 1,\dots,J$ is given by

$$\mathbf{Q}^{(i,j)} = \begin{cases} \sum_{k \in \mathcal{K}^{(i,j)}} \mathbf{Q}_k^{(i,j)} + \mathbf{Q}_D^{(i)} & \text{if } i = j, \\ \sum_{k \in \mathcal{K}^{(i,j)}} \mathbf{Q}_k^{(i,j)} & \text{otherwise,} \end{cases}$$

where

$$\mathbf{Q}_k^{(i,j)} = \alpha_k \bigotimes_{h=1}^{d} \mathbf{Q}_{k,h}^{(i,j)}, \quad \mathbf{Q}_D^{(i)} = -\sum_{j=1}^{J} \sum_{k \in \mathcal{K}^{(i,j)}} \alpha_k \bigotimes_{h=1}^{d} \mathrm{diag}(\mathbf{Q}_{k,h}^{(i,j)} \mathbf{e}),$$

$\otimes$ is the Kronecker product operator, α_k is the rate associated with continuous-time transition k, $\mathcal{K}^{(i,j)}$ is the set of transitions in block (i,j), $\mathbf{e}$ represents a column vector of ones, $\mathrm{diag}(\mathbf{a})$ denotes the diagonal matrix with the entries of vector $\mathbf{a}$ along its diagonal, and $\mathbf{Q}_{k,h}^{(i,j)}$ is the submatrix of the transition matrix $\mathbf{Q}_{k,h}$ whose row and column state spaces are $\mathcal{S}_h^{(i)}$ and $\mathcal{S}_h^{(j)}$, respectively [3]. In practice, the matrices $\mathbf{Q}_{k,h}$ are sparse [7] and held in sparse row format since the nonzeros in each of its rows indicate the possible transitions from the state with that row index. The advantage of partitioning the reachable state space is the elimination of unreachable states from the set of rows and columns of the generator to avoid unnecessary flops due to unreachable states. We also remark that the continuous-time transition rate of a Kronecker product term, α_k, can be eliminated by scaling one factor in the term with that rate.

To simplify the discussion and the notation, we consider the multiplication of a single block of $\mathbf{Q}$ from the left with a (sub)vector, and therefore, omit the indices (i,j) and write the index k associated with the transition as a superscript in parentheses above the matrices forming the block. Hence, we concentrate on the operation

$$\mathbf{y}^T := \mathbf{x}^T \sum_{k=1}^{K} \bigotimes_{h=1}^{d} \mathbf{Q}_h^{(k)},$$

where $\mathbf{Q}_h^{(k)}$ is a $(m_h \times n_h)$ matrix, implying $\bigotimes_{h=1}^{d} \mathbf{Q}_h^{(k)}$ is a $(\prod_{h=1}^{d} m_h \times \prod_{h=1}^{d} n_h)$ matrix, and $\mathbf{x}$ is a $(\prod_{h=1}^{d} m_h \times 1)$ vector. K is equal to the number of terms in the sum, i.e., $|\mathcal{K}^{(i,j)}|$ if we consider block (i,j). Observe that this is the operation that takes place when each block of a block matrix in Kronecker form such as $\mathbf{Q}$ gets multiplied on the left by an iteration subvector. In fact, the same subvector multiplies all blocks in a row of the matrix in Kronecker form.

To be consistent with the literature, we consider in the following multiplications of Kronecker products $\otimes_{h=1}^{d} \mathbf{A}_h^{(k)}$ with column vector $\mathbf{x}$ and their summation in the usual matrix-vector form

$$\mathbf{y} := \sum_{k=1}^{K} \left(\bigotimes_{h=1}^{d} \mathbf{A}_h^{(k)} \right) \mathbf{x},$$

where $\mathbf{A}_h^{(k)}$ is the transpose of $\mathbf{Q}_h^{(k)}$ and of size $(n_h \times m_h)$. In particular, we are interested in its implementation as

$$\mathbf{y}^{(1)} := \mathbf{0}, \quad \mathbf{x}^{(k)} := \left(\bigotimes_{h=1}^{d} \mathbf{A}_h^{(k)} \right) \mathbf{x}, \quad \mathbf{y}^{(k+1)} := \mathbf{y}^{(k)} + \mathbf{x}^{(k)} \quad \text{for } k = 1, \ldots, K,$$

and $\mathbf{y} := \mathbf{y}^{(K+1)}$, where $\mathbf{0}$ is a column vector of 0's. Now, we turn to the HTD format.

2.1 HTD Format

Assuming without loss of generality that d is a power of 2, the $(\prod_{h=1}^{d} m_h \times 1)$ vector $\mathbf{x}$ in (orthogonalized) HTD format can be expressed as

$$\mathbf{x} = (\mathbf{U}_1 \otimes \cdots \otimes \mathbf{U}_d)\mathbf{c},$$

where $\mathbf{U}_h$ for $h = 1, \ldots, d$ are $(m_h \times r_h)$ orthogonal basis matrices for the different dimensions in the model and

$$\mathbf{c} = (\mathbf{B}_{1,2} \otimes \cdots \otimes \mathbf{B}_{d-1,d}) \cdots (\mathbf{B}_{1,\ldots,d/2} \otimes \mathbf{B}_{d/2+1,\ldots,d})\mathbf{B}_{1,\ldots,d}$$

is a $(\prod_{h=1}^{d} r_h \times 1)$ vector in the form of a product of $\log_2 d$ matrices each of which except the last is a Kronecker product of a number of transfer matrices $\mathbf{B}_t$ related to each other as in the full binary tree of Fig. 1. The transfer matrix

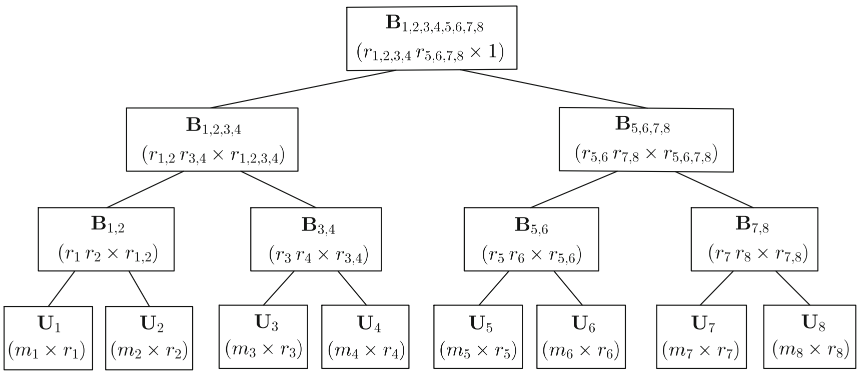

$\mathbf{B}_{1,2,3,4,5,6,7,8}$
$(r_{1,2,3,4}\, r_{5,6,7,8} \times 1)$
$\mathbf{B}_{1,2,3,4}$
$(r_{1,2}\, r_{3,4} \times r_{1,2,3,4})$
$\mathbf{B}_{5,6,7,8}$
$(r_{5,6}\, r_{7,8} \times r_{5,6,7,8})$
$\mathbf{B}_{1,2}$
$(r_1\, r_2 \times r_{1,2})$
$\mathbf{B}_{3,4}$
$(r_3\, r_4 \times r_{3,4})$
$\mathbf{B}_{5,6}$
$(r_5\, r_6 \times r_{5,6})$
$\mathbf{B}_{7,8}$
$(r_7\, r_8 \times r_{7,8})$
$\mathbf{U}_1$ $(m_1 \times r_1)$
$\mathbf{U}_2$ $(m_2 \times r_2)$
$\mathbf{U}_3$ $(m_3 \times r_3)$
$\mathbf{U}_4$ $(m_4 \times r_4)$
$\mathbf{U}_5$ $(m_5 \times r_5)$
$\mathbf{U}_6$ $(m_6 \times r_6)$
$\mathbf{U}_7$ $(m_7 \times r_7)$
$\mathbf{U}_8$ $(m_8 \times r_8)$

Fig. 1. Matrices forming $\mathbf{x}$ in HTD format for $d = 8$.

$\mathbf{B}_t$ is of size $(r_{t_l} r_{t_r} \times r_t)$ with the node index t defined as $t := t_l, t_r$, and $r_{1,\ldots,d} = 1$ since $\mathbf{B}_{1,\ldots,d}$ is at the root of the tree [14, pp. 5–6].

The $(d-1)$ intermediate nodes of the binary tree in Fig. 1 store the transfer matrices $\mathbf{B}_t$ and its leaves store the basis matrices $\mathbf{U}_h$ so that each intermediate node has two children. In orthogonalized HTD format of $\mathbf{x}$, one can also conceive of orthogonal basis matrices $\mathbf{U}_t = (\mathbf{U}_{t_l} \otimes \mathbf{U}_{t_r})\mathbf{B}_t$, at intermediate nodes with r_t columns that relate the orthogonal basis matrices $\mathbf{U}_{t_l}$ and $\mathbf{U}_{t_r}$ for the two children of transfer matrix $\mathbf{B}_t$ with the transfer matrix itself. In fact, the orthogonal matrix $\mathbf{U}_t$ has in its columns the singular vectors associated with the largest r_t singular values [11, pp. 76–79] of the matrix obtained by taking index t as row index, the remaining indices in order as column index of the d-dimensional data at hand (i.e., with a slight abuse of notation, $\mathbf{x}(t, \{1, \ldots, d\} - t)$). Hence, we have the concepts of "hierarchy of matricizations" and "higher-order singular value decomposition (HOSVD)", and r_t is the rank of the truncated HOSVD. More detailed information regarding this can be found in [12,14]. We remark that $\mathbf{B}_t$ may also be viewed as a 3-dimensional array of size $(r_{t_l} \times r_{t_r} \times r_t)$ having as many indices in each of its three dimensions as the number of columns in the matrices in its two children and itself, respectively. The number of transfer matrices in the lth factor forming $\mathbf{c}$ is the Kronecker product of $2^{\log_2 d - l}$ transfer matrices for $l = 1, \ldots, \log_2 d - 1$. In fact, $\mathbf{c}$ is a product of Kronecker products, and so is $\mathbf{x}$, but neither has to be formed explicitly.

When d is not a power of 2, it is still useful to keep the tree in a balanced form, for instance, as in Fig. 2 for which

$$\mathbf{x} = (((\mathbf{U}_1 \otimes \mathbf{U}_2)\mathbf{B}_{1,2}) \otimes \mathbf{U}_3 \otimes \mathbf{U}_4 \otimes \mathbf{U}_5)(\mathbf{B}_{1,2,3} \otimes \mathbf{B}_{4,5})\mathbf{B}_{1,2,3,4,5}.$$

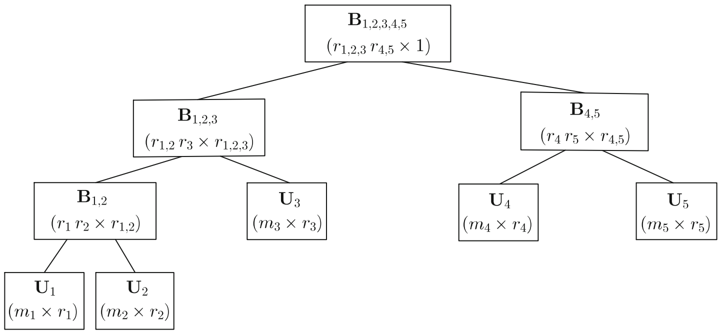


Fig. 2. Matrices forming $\mathbf{x}$ in HTD format for $d = 5$.

Assuming that $r_{\max} = \max_t(r_t)$ and $m_{\max} = \max(m_1, \ldots, m_d)$, memory requirement for matrices in the binary tree associated with HTD format is bounded by $dm_{\max}r_{\max}$ at the leaves, $r_{\max}^2$ at the root, and $(d-2)r_{\max}^3$ at other intermediate nodes, thus, totally $dm_{\max}r_{\max} + (d-2)r_{\max}^3 + r_{\max}^2$. In the next subsection, we show how a particular rank-1 vector can be represented in HTD format.

2.2 Uniform Distribution in HTD Format

Let $\mathbf{x} = \mathbf{e}/m$ be the $(m \times 1)$ uniform distribution vector, where $m = \prod_{h=1}^{d} m_h$. Then $\mathbf{x}$ may be represented in HTD format with all matrices having rank-1 for which the basis matrices given by $\mathbf{U}_h = \mathbf{e}/\sqrt{m_h}$ are of size $(m_h \times 1)$ for $h = 1, \ldots, d$ and the transfer matrices given by

$$\mathbf{B}_t = \begin{cases} (\prod_{h=1}^{d} \sqrt{m_h})/m & \text{if } t \text{ corresponds to root} \\ 1 & \text{otherwise} \end{cases}$$

are (1×1). Note that memory taken up by flat representation of $\mathbf{x}$ is m nonzeros, whereas that with HTD format is $d - 1 + \sum_{h=1}^{d} m_h$ nonzeros since the $(d-1)$ transfer matrices are all scalars equal to 1 except the one corresponding to the root. In passing to the multiplication of a compact vector with a Kronecker product, we remark that each basis matrix $\mathbf{U}_h$ for the uniform distribution has only a single column and that column is unit 2-norm, implying all $\mathbf{U}_h$ are orthogonal.

2.3 Multiplication of Vector in HTD Format with Kronecker Product

Assuming that $\mathbf{x}$ is in HTD format with orthogonal basis matrices $\mathbf{U}_h$ and transfer matrices $\mathbf{B}_t$ forming vector $\mathbf{c}$, the operation

$$\mathbf{x}^{(k)} := \left(\bigotimes_{h=1}^{d} \mathbf{A}_h^{(k)} \right) \mathbf{x} \quad \text{is equivalent to performing} \quad \mathbf{x}^{(k)} := \left(\bigotimes_{h=1}^{d} \mathbf{A}_h^{(k)} \mathbf{U}_h \right) \mathbf{c}$$

since $\mathbf{x} = (\otimes_{h=1}^{d} \mathbf{U}_h)\mathbf{c}$. Hence, the only thing that needs to be done to carry out the computation of $\mathbf{x}^{(k)}$ in HTD format is to multiply the $(n_h \times m_h)$ Kronecker factor $\mathbf{A}_h^{(k)}$ with the corresponding $(m_h \times r_h)$ orthogonal basis matrix $\mathbf{U}_h$ for $h = 1, \ldots, d$. Clearly, the $(n_h \times r_h)$ product matrix $\mathbf{A}_h^{(k)}\mathbf{U}_h$ need not be orthogonal. But this does not pose much of a problem, since $\mathbf{x}^{(k)}$ can be transformed into orthogonalized HTD format if the need arises by computing the QR decomposition [11, pp. 246–250] of $\mathbf{A}_h^{(k)}\mathbf{U}_h = \tilde{\mathbf{U}}_h\mathbf{R}_h$ for $h = 1, \ldots, d$, propagating the triangular factors $\mathbf{R}_h$ into the transfer matrices, and orthogonalizing the updated transfer matrices at intermediate nodes in a similar manner up to the root as in Algorithm 1 in [14, p. 12]. However, the situation is not as good for the addition of two compact vectors.

2.4 Addition of Two Vectors in HTD Format and Truncation

Addition of two matrices $\mathbf{Y}$ and $\mathbf{X}$ with given singular value decompositions (SVDs) [11, pp. 76–79]

$$\mathbf{Y} = \mathbf{U_Y}\mathbf{\Sigma_Y}\mathbf{V_Y}^T \quad \text{and} \quad \mathbf{X} = \mathbf{U_X}\mathbf{\Sigma_X}\mathbf{V_X}^T$$

results in

$$\mathbf{Y} + \mathbf{X} := (\mathbf{U_Y}\ \mathbf{U_X}) \begin{pmatrix} \mathbf{\Sigma_Y} & \\ & \mathbf{\Sigma_X} \end{pmatrix} (\mathbf{V_Y}\ \mathbf{V_X})^T .$$

Here, $\mathbf{\Sigma_Y}, \mathbf{\Sigma_X}$ are diagonal matrices of singular values, whereas $\mathbf{U_Y}$, $\mathbf{U_X}$ and $\mathbf{V_Y}$, $\mathbf{V_X}$ are orthogonal matrices of left and right singular (row) vectors associated with matrices $\mathbf{Y}$, $\mathbf{X}$, respectively. SVD is a rank revealing factorization in that the number of nonzero singular values of a matrix corresponds to its column rank. This implies that the sum $(\mathbf{Y} + \mathbf{X})$ has a rank equal to the sum of the ranks of the two matrices that are added.

The situation for the sum $\mathbf{y}^{(k+1)}$ of the two vectors $\mathbf{y}^{(k)}$ and $\mathbf{x}^{(k)}$ in HTD format is no different if one replaces the SVD with HOSVD. This is conveniently illustrated for $d = 4$ by Fig. 5 in [14, p. 11]. For the following steps performing the addition and representing the resulting vector in HTD format, we exploit the algorithms presented in [14]. Among three alternative approaches that have been investigated therein for computing $\mathbf{y}$, the best seems to be to multiply, add and then truncate K times as demonstrated in Fig. 11 of [14]. This approach is coded in Algorithm 7 of [14, p. 23] which works by calling Algorithm 3 that takes care of the reduced Gramians computations of a compact vector in non-orthogonalized HTD format. Recall that the compact vector $\mathbf{x}^{(k)}$ obtained after multiplication does not need to be in orthogonal HTD format even though $\mathbf{x}$ might have been. Once Algorithm 3 is executed, Algorithm 7 takes over and computes the truncated HOSVD for the sum of two vectors $\mathbf{y}^{(k)}$ and $\mathbf{x}^{(k)}$ in HTD format without initial orthogonalization. The output $\mathbf{y}^{(k+1)}$ of Algorithm 7 is a truncated compact vector in orthogonalized HTD format and this operation is repeated K times until $\mathbf{y}$ is obtained. The number of flops executed by Algorithm 7 is $O(dK^2 r_{max}^2 (n_{max} + r_{max}^2 + K r_{max}))$, where $n_{\max} = \max(n_1, \dots, n_d)$. The significance of this algorithm is that one can impose an accuracy of ϵ on the truncated HOSVD by choosing rank r_t in node t based on dropping the smallest singular values whose squared sum is less than or equal to $\epsilon^2/(2d-3)$ [14, pp. 18–19]. This is a very nice result but also implies that the truncation leads to an approximate solution vector.

2.5 Computing the 2-Norm of a Vector in HTD Format

Normally, it is more relevant to compute the maximum (i.e., infinity) norm of a solution vector in probabilistic analysis even though all norms are known to be equivalent [11, pp. 68–70]. However, the computation of the maximum value (in magnitude) of the elements of a compact vector requires being able to know which indexed value is the largest and also its value, which seems to be costly for a compact vector in HTD format. Therefore, we consider the computation of the 2-norm of vector $\mathbf{y}$ given by $||\mathbf{y}||_2 = \sqrt{\mathbf{y}^T\mathbf{y}}$.

Fortunately, $||\mathbf{y}||_2$ can be obtained using Algorithm 2 in [14, p. 14], which computes inner products of two compact vectors in HTD format. Here, the only difference is that the two vectors are the same vector $\mathbf{y}$. The algorithm starts from the leaves of the binary tree and moves towards the root, requiring the same sequence of operations in the first part of the computation of reduced Gramians in Algorithm 3 in [14, p. 17]. But, this has already been discussed in the previous subsection.

Now, we can move to implementation issues regarding compact solution vectors in HTD format for Kronecker-based Markovian representations.

3 Implementation Issues

The implementation is done within the NSolve package of the APNN Toolbox [1,2]. The binary tree data structure accompanying the HTD format is allocated at the outset depending on the value of d. It is stored in the form of an array of tree nodes from root to leaves level by level so that accessing the children of a parent node or the parent of a child node becomes relatively easy. In a tree node t, there are pointers to matrices $\mathbf{U}_t$ for leaves and $\mathbf{B}_t$ for intermediate nodes which we have seen and accounted for before, but also pointers to matrices $\mathbf{R}_t$ and, as we explain shortly, (2×2) block matrices $\mathbf{M}_t$ and $\mathbf{G}_t$ for each node. Since we expect solution vectors to be dense, the matrices in the compact representation are stored as full matrices including those corresponding to the blocks of $\mathbf{M}_t$ and $\mathbf{G}_t$. The nonzero elements of the full matrices are kept in a one-dimensional real array so that relevant LAPACK methods available at [17] can be called without having to copy vectors. We choose to store transposes of the matrices representing the compact solution vector in row sparse format (meaning they are stored by columns) so that relevant LAPACK methods can be called without having to transpose the input matrices.

The multiplication of the sparse Kronecker factors $\mathbf{A}_h^{(k)}$ with the orthogonal basis matrices $\mathbf{U}_h$ in $\mathbf{x}^{(k)} := \left(\bigotimes_{h=1}^{d} \mathbf{A}_h^{(k)} \mathbf{U}_h\right) \mathbf{c}$ is implemented using straightforward sparse matrix-vector multiplication. After the compact vector $\mathbf{x}^{(k)}$ is computed, the tree nodes of $\mathbf{y}^{(k)}$ are visited and its respective fields are updated so that we have $\mathbf{y}^{(k+1)}$ at hand. Efficient computation of the reduced Gramian matrices $\mathbf{G}_t$ as in Algorithm 3 of [14, p. 17] for $\mathbf{y}^{(k+1)}$ requires exploiting the block structure of the new transfer matrices $\mathbf{B}_t$ whose blocks are already available in the corresponding tree nodes of $\mathbf{y}^{(k+1)}$ after the addition operation. Clearly, there is no need to generate block matrices (or a cubic blocks as in Fig. 5 of [14, p. 11]) with these blocks explicitly. We prefer to store $\mathbf{M}_t$ and $\mathbf{G}_t$ as (2×2) block matrices because of the add a term and then truncate approach followed. Let us next elaborate on this.

Assuming that $r_t(\mathbf{y}^{(k)})$ and $r_t(\mathbf{x}^{(k)})$ denote the ranks of matrices in compact representations of the two vectors that are summed up in node t, $\mathbf{M}_t$ and $\mathbf{G}_t$ become $(r_t(\mathbf{y}^{(k)})r_t(\mathbf{x}^{(k)}) \times r_t(\mathbf{y}^{(k)})r_t(\mathbf{x}^{(k)}))$ matrices, where the first diagonal block is $(r_t(\mathbf{y}^{(k)}) \times r_t(\mathbf{y}^{(k)}))$ and the second diagonal block is $(r_t(\mathbf{x}^{(k)}) \times r_t(\mathbf{x}^{(k)}))$. Then the computation $\mathbf{M}_t := \mathbf{U}_t^T \mathbf{U}_t$ for leaf nodes can be formulated in (2×2) block manner as

$$\mathbf{M}_t^{(i,j)} := (\mathbf{U}_t^{(i)})^T (\mathbf{U}_t^{(j)}) \quad \text{for } i, j = 1, 2,$$

where $\mathbf{U}_t^{(1)}$ and $\mathbf{U}_t^{(2)}$ denote basis matrices of $\mathbf{y}^{(k)}$ and $\mathbf{x}^{(k)}$ at leaf node t, respectively. This computation requires multiplying two full matrices for which the `DGEMM` routine of LAPACK may be used. On the other hand, the computation

$\mathbf{M}_t := \mathbf{B}_t^T(\mathbf{M}_{t_l} \otimes \mathbf{M}_{t_r})\mathbf{B}_t$ for intermediate nodes can be formulated from the bottom of the tree to the root in (2×2) block manner as

$$\mathbf{M}_t^{(i,j)} := (\mathbf{B}_t^{(i)})^T(\mathbf{M}_{t_l}^{(i,j)} \otimes \mathbf{M}_{t_r}^{(i,j)})(\mathbf{B}_t^{(j)}) \quad \text{for } i, j = 1, 2,$$

where $\mathbf{B}_t^{(1)}$ and $\mathbf{B}_t^{(2)}$ denote transfer matrices of $\mathbf{y}^{(k)}$ and $\mathbf{x}^{(k)}$ at node t, respectively.

Similarly, we have reduced Gramian computations, but in opposite direction from root to leaves, that can be formulated in (2×2) block manner for $i, j = 1, 2$ as $\mathbf{G}_t^{(i,j)} := 1$ when t corresponds to root; otherwise,

$$\mathbf{G}_{t_l}^{(i,j)} := (\mathbf{B}_{t:2,3}^{(i)})^T(\mathbf{M}_{t_r}^{(i,j)} \otimes \mathbf{G}_t^{(i,j)})\mathbf{B}_{t:2,3}^{(j)}$$

and

$$\mathbf{G}_{t_r}^{(i,j)} := (\mathbf{B}_{t:1,3}^{(i)})^T(\mathbf{M}_{t_l}^{(i,j)} \otimes \mathbf{G}_t^{(i,j)})\mathbf{B}_{t:1,3}^{(j)},$$

where $\mathbf{B}_{t:2,3}^{(1)}$ and $\mathbf{B}_{t:1,3}^{(1)}$ are transfer matrices $\mathbf{B}_t^{(1)}$ of $\mathbf{y}^{(k)}$ organized respectively as $(r_{t_r}(\mathbf{y}^{(k)})r_t(\mathbf{y}^{(k)}) \times r_{t_l}(\mathbf{y}^{(k)}))$ and $(r_{t_l}(\mathbf{y}^{(k)})r_t(\mathbf{y}^{(k)}) \times r_{t_r}(\mathbf{y}^{(k)}))$ matrices and $\mathbf{B}_{t:2,3}^{(2)}$ and $\mathbf{B}_{t:1,3}^{(2)}$ are transfer matrices $\mathbf{B}_t^{(2)}$ of $\mathbf{x}^{(k)}$ organized respectively as $(r_{t_r}(\mathbf{x}^{(k)})r_t(\mathbf{x}^{(k)}) \times r_{t_l}(\mathbf{x}^{(k)}))$ and $(r_{t_l}(\mathbf{x}^{(k)})r_t(\mathbf{x}^{(k)}) \times r_{t_r}(\mathbf{x}^{(k)}))$ matrices. Such matrices are called matricizations of the given matrix (in this case, the transfer matrix $\mathbf{B}_t^{(1)}$ or $\mathbf{B}_t^{(2)}$ along specific dimensions), and therefore, represent different organizations of the same data. We remark that the off-diagonal blocks of $\mathbf{M}_t$ and $\mathbf{G}_t$ respectively satisfy the relationships $\mathbf{M}_t^{(i,j)} = (\mathbf{M}_t^{(j,i)})^T$ and $\mathbf{G}_t^{(i,j)} = (\mathbf{G}_t^{(j,i)})^T$. Therefore, only one off-diagonal block for these two matrices in each node needs to be computed. The computation of the three blocks of $\mathbf{M}_t$ and $\mathbf{G}_t$ requires multiplications using DGEMM with matricizations and contraction of multi-dimensional data involving $\mathbf{B}_t^{(i)}$ matrices for $i = 1, 2$ as discussed in [14, pp. 9–10, 12–13]. We use two auxiliary vectors of length $\max_{t_l,t_r,t}(r_{t_l}r_{t_r}r_t)$ to implement these operations. The disadvantage of not storing $\mathbf{M}_t$ and $\mathbf{G}_t$ as (2×2) block matrices is that longer auxiliary vectors would need to be allocated.

Truncation of a compact vector requires QR and singular value decompositions [11, pp. 76–79, 246–250] as in Algorithm 7 of [14, p. 23] to be performed. In order to compute these decompositions, DGEQRF and DGESDD routines of LAPACK are used. Since we expect input matrices to be dense, we do not call routines expecting sparse matrices. For a leaf node t, the $(m_t \times (r_t(\mathbf{y}^{(k)}) + r_t(\mathbf{x}^{(k)})))$ input matrix $\mathbf{U}_t$ maybe obtained by concatenating the matrices $\mathbf{U}_t^{(1)}$ and $\mathbf{U}_t^{(2)}$ corresponding to $\mathbf{y}^{(k)}$ and $\mathbf{x}^{(k)}$, respectively. Since the input matrix is also an output matrix, the upper-triangular factor $\mathbf{R}_t$ of the QR decomposition is returned from DGEQRF in the upper-triangular part of the input matrix in which the lower-triangular part has the Householder reflections amounting to the orthogonal factor $\mathbf{Q}_t$. After $\mathbf{R}_t$ is obtained, $\mathbf{R}_t\mathbf{G}_t\mathbf{R}_t^T$ needs to be formed. To this end, we first transform the block matrix $\mathbf{G}_t$ to a dense matrix (with a single block) and multiply this new matrix held as a one-dimensional array from left and right using the DTRMM routine of LAPACK. Note that DTRMM does not accept a trapezoid

$\mathbf{R}_t$; however, this case can be handled by multiplying triangular and rectangular parts of $\mathbf{R}_t$ separately using DTRMM and DGEMM. Hence, there is no need to copy the output of DGEQRF to another matrix including $\mathbf{R}_t$. Once $\mathbf{R}_t\mathbf{G}_t\mathbf{R}_t^T$ is formed, it needs to be decomposed for its singular values and vectors. To this end, we prefer to use the DGESDD routine over the DGESVD routine since it is said to be faster [17]. We remark that this routine computes singular values through the symmetric eigenvalue decomposition, and the singular vectors are truncated at a certain number or possibly by omitting some corresponding to the smaller singular values based on an error tolerance. $\mathbf{S}_t$ ends up being the matrix holding the r_t singular vectors. Then the orthogonal basis matrix $\mathbf{U}_t = \mathbf{Q}_t\mathbf{S}_t$ is computed using the DORMQR routine. In order to avoid storing $\mathbf{S}_t$, we prefer to update $\mathbf{R}_t$ with $\mathbf{S}_t^T$ as in the `htucker` package [15].

The same sequence of operations are carried out level by level from the parents of the leaves to the top of the tree excluding the root. The product $\mathbf{S}_t^T\mathbf{R}_t$ is computed using DTRMM (also possibly with an additional call to DGEMM when $\mathbf{R}_t$ is trapezoid) and stored in the matrix that was allocated for $\mathbf{R}_t$. Note that $\mathbf{S}_t^T\mathbf{R}_t = (\mathbf{F}_t^{(1)}\ \mathbf{F}_t^{(2)})$ is an $(r_t \times (r_t(\mathbf{y}^{(k)}) + r_t(\mathbf{x}^{(k)})))$ matrix with the two blocks $\mathbf{F}_t^{(l)}$ for $l = 1, 2$, where r_t is the rank of node t after truncation. Then for a non-leaf node t, the QR factorization of $\sum_{i=1}^{2}(\mathbf{F}_{t_l}^{(i)} \otimes \mathbf{F}_{t_l}^{(i)})\mathbf{B}_t^{(i)}$ needs to be computed. This computation requires multiplications using DGEMM with matricizations of multi-dimensional data involving $\mathbf{B}_t^{(i)}$ matrices for $i = 1, 2$ as discussed in [14, pp. 9–10]. Finally, the transfer matrix $\mathbf{B}_t = \mathbf{Q}_t\mathbf{S}_t$ is computed using DORMQR.

4 Results of Numerical Experiments

In this section, we consider two example models that have been used as benchmarks in [4]. The first is an availability model with d subsystems in which different time scales occur. Each subsystem models a processing node with 2 processors, one acting as a cold spare, a bus and two memory modules. Time to failure is exponentially distributed with rate 5×10^{-4} for processors, 4×10^{-4} for buses and 10^{-4} for memory modules. Components are repaired by a global repair facility with preemptive priority such that components from subsystem 1 have the highest priority and components from subsystem d have the least priority. Furthermore, the repair of the bus has priority over the repair of the processor which has priority over the repair of the memory module. The repair times of components are exponentially distributed. The repair rates of a processor, a bus, and a memory from subsystem 1 are given respectively as 1, 2, and 4. The same rates for other subsystems are given respectively as 0.1, 0.2, and 0.4. For this model, the reachable state space is equal to the product state space and contains 12^d states. We consider availability models with $d = 3, 4, 5, 6, 7, 8$.

The second example is a model of a polling system of two servers serving customers from d finite capacity queues, which are cyclically visited by the servers. Customers arrive to the system according to a Poisson process with rate 1.5 and are distributed with queue specific probabilities among the queues each of which is assumed to have a capacity of 10. If a server visits a nonempty queue, it serves one customer and then travels to the next queue. On the other hand, a server arriving at an empty queue, skips the queue and travels to the next queue. Service and travelling times of servers are exponentially distributed respectively with rates 1 and 10. Each subsystem in the model describes one queue, and the J partitions of the reachable state space for this model are defined according to the number of servers serving customers at a queue or travelling to the next queue. For each subsystem we obtain 62 states partitioned into 3 subsets. The reachable state space of the complete model has $J = \binom{d+1}{2}$ partitions, and we consider polling system models with $d = 3, 4, 5, 6, 7$.

Table 1. Properties of availability and polling models

	Availability		Polling	
d	J	$\lvert\mathcal{S}\rvert$	J	$\lvert\mathcal{S}\rvert$
3	1	1,728	6	25,443
4	1	20,736	10	479,886
5	1	248,832	15	8,065,860
6	1	2,985,984	21	125,839,395
7	1	35,831,808	28	1,863,521,121
8	1	429,981,696		

The goal of this paper is to compare the memory and timing requirements for a vector-matrix product computation using the full vector and the HTD format approaches. Furthermore, we have to evaluate the accuracy of the computation if truncation is performed in the HTD format. Therefore, we consider in the following iteration steps of the Power method. This is not the most efficient solution method for steady-state analysis, but similar iteration steps can be applied in more advanced iterative techniques and they can be directly used in uniformization for transient analysis. For each model, the solution vector $\boldsymbol{\pi}^{(it)}$ at iteration it is multiplied with

$$\mathbf{P} := \mathbf{I} + \Delta\mathbf{Q}, \quad \text{where } \Delta := 0.999/\max_{s\in\mathcal{S}} |q_{s,s}|,$$

starting with the uniform distribution in $\boldsymbol{\pi}^{(0)}$, so that we have

$$\boldsymbol{\pi}^{(it)} := \boldsymbol{\pi}^{(it-1)}\mathbf{P} \quad \text{for } it = 1, 2, \ldots, \texttt{maxit}$$

with the associated error vector $\mathbf{e}^{(it)} := \boldsymbol{\pi}^{(it)} - \boldsymbol{\pi}^{(it-1)}$. Note that $\mathbf{e}^{(it)} = \Delta\boldsymbol{\pi}^{(it-1)}\mathbf{Q}$, the scaled residual vector corresponding to the previous iteration vector. Here, `maxit` is the maximum number of iterations and we set `maxit` $:= 1,000$. The numerical experiments are performed on an an Intel Core i7 Quad-Core 3.6 GHz processor with 32 GB of main memory.

Table 2 contains the results for the availability model. Time is in seconds and Memory indicates the number of allocated real array elements. For the chosen truncation accuracy of $\epsilon \in [10^{-9}, 10^{-7}]$, the norm of the final error vector is the same for the full and HTD representations. Due to the reduced memory requirements of the vector, the compact representation results even in smaller iteration times when d increases. It should be mentioned that the model is not symmetric due to the priority repair strategy but, as it is common in availability models, the probability distribution becomes unbalanced because repair rates are higher than failure rates.

Table 2. Numerical results for availability models

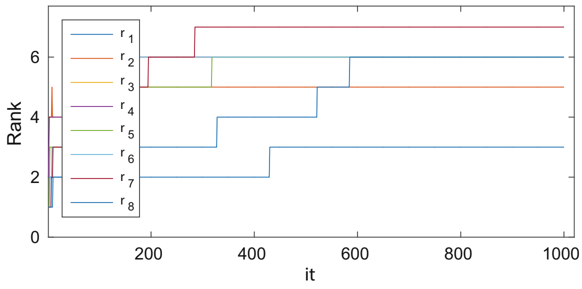
	Full			Compact			
d	Time	Memory	$\|\|\mathbf{e}^{(\mathtt{maxit})}\|\|_2$	ϵ	Time	Memory	$\|\|\mathbf{e}^{(\mathtt{maxit})}\|\|_2$
3	0	5,391	5×10^{-7}	10^{-7}	1	2,102	7×10^{-7}
				10^{-8}	1	2,298	5×10^{-7}
				10^{-9}	2	3,400	5×10^{-7}
4	0	62,598	3×10^{-6}	10^{-7}	3	2,809	2×10^{-6}
				10^{-8}	4	4,107	3×10^{-6}
				10^{-9}	6	6,938	3×10^{-6}
5	2	747,132	1×10^{-5}	10^{-7}	6	3,719	9×10^{-6}
				10^{-8}	9	5,327	1×10^{-5}
				10^{-9}	13	9,278	1×10^{-5}
6	38	8,958,897	3×10^{-5}	10^{-7}	13	6,756	3×10^{-5}
				10^{-8}	18	9,398	3×10^{-5}
				10^{-9}	28	14,120	3×10^{-5}
7	513	107,496,741	7×10^{-5}	10^{-7}	15	6,726	7×10^{-5}
				10^{-8}	28	10,381	7×10^{-5}
				10^{-9}	43	16,150	7×10^{-5}
8	6,329	1,289,946,786	2×10^{-6}	10^{-7}	22	9,078	9×10^{-5}
				10^{-8}	37	12,340	9×10^{-6}
				10^{-9}	66	26,041	3×10^{-6}

The situation is more ambiguous for the polling example whose results are given in Table 3. For the larger configurations, we obtain savings in memory by several orders of magnitude even with the smallest truncation accuracy of ϵ. Time-wise the conventional approach is faster for small configurations, but it is outperformed by the compact representation for larger state spaces (i.e. $d = 6$), if ϵ is not too small. The largest configuration with $d = 7$ can only be analysed with the compact vector representation.

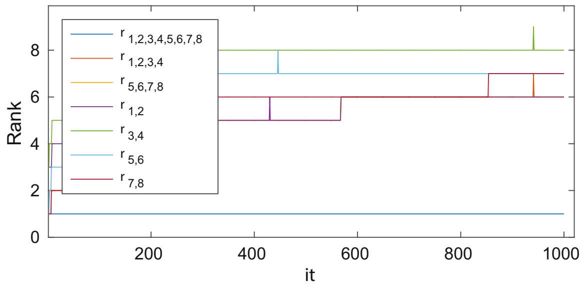
In Fig. 3 the ranks of the different matrices forming the HTD are shown for a truncation accuracy of $\epsilon = 10^{-7}$. It can be seen that the ranks remain moderate. The matrices are fairly dense such that a sparse storage of the matrices for the vector representation is not necessary which can be seen in Fig. 4.

Table 3. Numerical results for polling models

	Full			Compact			
d	Time	Memory	$\|\|\mathbf{e}^{(\mathtt{maxit})}\|\|_2$	ϵ	Time	Memory	$\|\|\mathbf{e}^{(\mathtt{maxit})}\|\|_2$
3	0	82,599	4×10^{-6}	10^{-7}	50	49,297	4×10^{-6}
				10^{-8}	83	72,281	4×10^{-6}
				10^{-9}	108	89,257	4×10^{-6}
4	5	1,496,563	5×10^{-6}	10^{-7}	285	143,436	5×10^{-6}
				10^{-8}	1,175	397,349	5×10^{-6}
				10^{-9}	3,272	774,834	5×10^{-6}
5	103	24,791,966	5×10^{-6}	10^{-7}	409	221,850	5×10^{-6}
				10^{-8}	2,522	800,742	5×10^{-6}
				10^{-9}	10,951	2,136,401	5×10^{-6}
6	1,896	383,988,648	3×10^{-6}	10^{-7}	254	177,534	3×10^{-6}
				10^{-8}	2,448	883,360	3×10^{-6}
				10^{-9}	19,423	3,564,320	3×10^{-6}
7	n/a	5,661,610,381	n/a	10^{-7}	196	217,254	3×10^{-6}
				10^{-8}	1,831	900,220	2×10^{-6}
				10^{-9}	21,668	5,037,050	2×10^{-6}

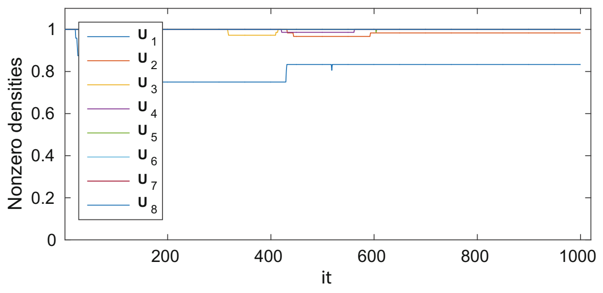


(a) Basis matrices

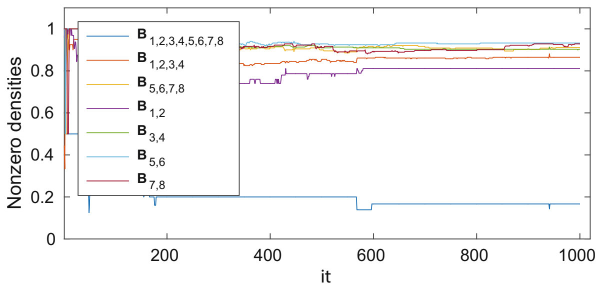


(b) Transfer matrices

Fig. 3. Ranks of basis and transfer matrices forming $\boldsymbol{\pi}^{(it)}$, availability $d = 8$ (Color figure online)



(a) Basis matrices



(b) Transfer matrices

Fig. 4. Densities of basis and transfer matrices forming $\boldsymbol{\pi}^{(it)}$, availability $d = 8$ (Color figure online)

5 Conclusion

We present in this paper a compact representation for the iteration vector of large structured Markov models which has been adopted from numerical analysis where the techniques have been developed in the recent years. It is shown that this vector representation can be combined naturally with a hierarchical Kronecker representation of generator matrices of structured Markov models. The basic step of iterative numerical algorithms to compute transient or steady-state solutions can be conveniently combined with the compact vector representation. Our first examples indicate that in contrast to previously tried compact representations for the vector (e.g., [5,16]), the new approach is memory and also relatively time efficient such that it bears the potential to increase the size of solvable models on a given computer significantly.

There are several things to be done. In particular, more experiments are necessary to confirm our results. The vector-matrix multiplications have to be embedded in more advanced solution techniques like projection or multi-level solution techniques. However, this can be easily done with the available software environment.

Acknowledgement. This work is supported by the Alexander von Humboldt Foundation through the Research Group Linkage Programme. The research of the last author is supported by The Scientific and Technological Research Council of Turkey.

References

1. APNN-Toolbox. Abstract Petri Net Notation Toolbox. http://www4.cs.uni-dortmund.de/APNN-TOOLBOX
2. Bause, F., Buchholz, P., Kemper, P.: A toolbox for functional and quantitative analysis of DEDS. In: Puigjaner, R., Savino, N.N., Serra, B. (eds.) TOOLS 1998. LNCS, vol. 1469, pp. 356–359. Springer, Heidelberg (1998)
3. Buchholz, P.: Hierarchical structuring of superposed GSPNs. IEEE Trans. Softw. Eng. **25**(2), 166–181 (1999)
4. Buchholz, P., Dayar, T.: On the convergence of a class of multilevel methods for large, sparse Markov chains. SIAM J. Matrix Anal. Appl. **29**(3), 1025–1049 (2007)
5. Buchholz, P., Kemper, P.: Compact representations of probability distributions in the analysis of superposed GSPNs. In: Proceedings of the 9th International Workshop on Petri Nets and Performance Models, Aachen, Germany, pp. 81–90. IEEE Press, New York, September 2001
6. Buchholz, P., Ciardo, G., Donatelli, S., Kemper, P.: Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. INFORMS J. Comput. **12**(3), 203–222 (2000)
7. Dayar, T.: Analyzing Markov Chains using Kronecker Products: Theory and Applications. Springer, New York (2012)
8. Dayar, T., Orhan, M.C.: On vector-Kronecker product multiplication with rectangular factors. SIAM J. Sci. Comput. **37**(5), S526–S543 (2015)
9. Dayar, T., Orhan, M.C.: Cartesian product partitioning of multi-dimensional reachable state spaces. Probab. Eng. Inf. Sci. **30**(3), 413–430 (2016)
10. Fernandes, P., Plateau, B., Stewart, W.J.: Efficient descriptor-vector multiplications in stochastic automata networks. J. ACM **45**(3), 381–414 (1998)
11. Golub, G.H., Van Loan, C.F.: Matrix Computations, 4th edn. Johns Hopkins University Press, Baltimore (2012)
12. Hackbusch, W.: Tensor Spaces and Numerical Tensor Calculus. Springer, Heidelberg (2012)
13. Kressner, D., Macedo, F.: Low-rank tensor methods for communicating Markov processes. In: Norman, G., Sanders, W. (eds.) QEST 2014. LNCS, vol. 8657, pp. 25–40. Springer, Heidelberg (2014)
14. Kressner, D., Tobler, C.: `htucker` — A Matlab toolbox for tensors in hierarchical Tucker format. Technical report 2012-02, Mathematics Institute of Computational Science and Engineering, Lausanne, Switzerland, August 2012. http://anchp.epfl.ch/htucker
15. Kressner, D., Tobler, C.: Algorithm 941: `htucker` — a matlab toolbox for tensors in hierarchical Tucker format. ACM Trans. Math. Softw. **40**(3), 22 (2014)
16. Kwiatkowska, M., Mehmood, R., Norman, G., Parker, D.: A symbolic out-of-core solution method for Markov models. Electron. Notes Theor. Comput. Sci. **68**(4), 589–604 (2002)
17. Netlib, A.: Collection of Mathematical Software, Papers, and Databases. http://www.netlib.org
18. Oseledets, I.V.: Tensor-train decomposition. SIAM J. Sci. Comput. **33**(5), 2295–2317 (2011)

19. Plateau, B.: On the stochastic structure of parallelism and synchronization models for distributed algorithms. Perform. Eval. Rev. **13**(2), 147–154 (1985)
20. Plateau, B., Fourneau, J.-M.: A methodology for solving Markov models of parallel systems. J. Parallel Distrib. Comput. **12**(4), 370–837 (1991)
21. Stewart, W.J.: Introduction to the Numerical Solution of Markov Chains. Princeton University Press, Princeton (1994)