"ON-THE-FLY" SOLUTION TECHNIQUES FOR STOCHASTIC PETRI NETS AND EXTENSIONS

Daniel D. Deavours and William H. Sanders
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
{deavours,whs}@crhc.uiuc.edu

Abstract

Models of realistic Markov systems using high-level modeling representations, such as stochastic Petri nets, frequently generate very large state spaces and corresponding state-transition-rate matrices. In this paper, we propose a new steady state solution method which avoids explicit storing of the matrix in memory. This method does not impose any structural restrictions on the model, uses Gauss-Seidel and variants as the numerical solver, and uses less memory than current state of the art solvers. An implementation of these ideas show that one can realistically solve very large, general models in relatively little memory.

I. Introduction

Problems of scalability in models and the resulting state-space explosion are daunting. The traditional approach of generating a state-level model from a high-level specification, such as stochastic Petri nets, typically results in very large state spaces for practical models. Such problems are further compounded with even higher-level formalisms, such as stochastic Petri nets with tokens that have attributes. This problem is often called the "largeness problem," and is a major impediment to accurately modeling large and complex systems.

There have been numerous attempts to address the largeness problem, resulting in techniques that produce either exact or approximate results. The exact approaches tend to fall into two complementary categories: those that attempt to reduce the state-space size (e.g., methods based on stochastic well-formed nets [1] or reduced base model construction [2]), and those that attempt to tolerate the large state space. One popular method for tolerating large state spaces avoid the explicit storage of the matrix, and are generally called *matrix-free* methods. Several matrix-free methods take advantage of the fact that some components of a model (called submodels) interact in a limited way with other submodels, so that the state-transition-rate matrix of the model is a function of Kronecker operators on the state-transition-rate matrix of the submodels. Solution methods for stochastic automata networks [3], [4] are an example of this type of method.

More recently, there has been work on superposed generalized stochastic Petri nets (SGSPNs), which are essentially independent submodels that may be joined by synchronizing on a timed transition. This class seems to be more promising as a less restrictive modeling technique. First introduced in [5], solutions for SGSPNs were restricted by the so-called product space (the product of the submodels' state spaces), which could be much larger than the set of tangible reachable

states. Kemper, in [6], [7], devised a method to operate on the tangible space, rather than the product space, by providing a mapping from product space to the tangible reachable space. Ciardo and Tilgner [8] built on Kemper's work by removing some of the imposed restrictions, e.g., by allowing synchronizing transitions to be immediate.

We believe that there are three substantial restrictions with current SGSPN techniques. First, all known methods based on Kronecker operators require models to have a structure such that there are partially independent components with limited interaction between them. While Ciardo and Tilgner relax these requirements significantly, many models still do not exhibit the structure required to use these methods.

Second, the sum of the sizes of the state spaces of the component models must be smaller than the size of state space of the combined model for Kronecker-based methods to be advantageous. This requires the submodels to be approximately the same size.

Third, solution methods of Kronecker-based tools have generally been limited to the power or Jacobi methods, both of which usually exhibit poor convergence behavior. This is particularly undesirable because large systems of equations tend to exhibit worse convergence characteristics than small systems. Also, due to the nature of Kronecker-based methods, solution by the Jacobi method requires three vectors of size equal to the state space. A notable exception is the work of Ciardo [9], who presents algorithms for doing a Gauss-Seidel iteration, although we are unaware of any tool that uses them.

We extend the state of the art in matrix-free methods in several ways. First, we use the set of states and the model to compute a row and column of the matrix. We call this method *on-the-fly*, and it imposes no structural restrictions on the model. Second, we utilize Gauss-Seidel as the core iterative solver, which typically converges more quickly than Jacobi. Third, we acknowledge that matrix-free methods are inherently slow, so we look at two techniques based on Gauss-Seidel to reduce access to the matrix, decreasing solution times. Finally, we derive an implementation of Gauss-Seidel that is better matched to matrix-free methods than Jacobi in that it uses less memory for little additional work.

In order to implement the most memory efficient general methods, we explore a straightforward implementation of Gauss-Seidel in which we develop algorithms that can generate, on-the-fly, the required incoming and outgoing transition rates from a state. We prefer Gauss-Seidel over Jacobi or the power method because it typically converges in fewer iterations, and requires

less memory. For the on-the-fly matrix generation, we give three algorithms: one for standard stochastic Petri nets (SPNs), one for generalized stochastic Petri nets (GSPNs) [10], and one for stochastic activity networks (SANs) [11], [12] and stochastic reward networks (SRNs) [13]. These algorithms are discussed in Section II.

Second, since the generation of the state-transition-rate matrix on-the-fly takes significantly more time than doing an iteration with the matrix in memory, we develop a new iterative solution method in Section III that exhibits locality in its use of data from the state-transition-rate matrix. A new method which we introduce here, called *modified adaptive Gauss-Seidel* (MAGS), reuses generated rows and columns in the state-transition-rate matrix within an iteration in order to reduce the performance penalty incurred by their generation. We also review block Gauss-Seidel (BGS), which also has this property.

Third, solution algorithms based on Gauss-Seidel typically requires access to rows of A in order to solve $Ax = b$, which corresponds to accessing the incoming rates of a state in the corresponding Markov model. Traditional implementations of Gauss-Seidel require only space for the solution vector, but because it requires accessing incoming rates, it can make on-the-fly solutions less efficient. In Section IV, we show a new implementation of Gauss-Seidel, called *column Gauss-Seidel*, that requires access to only outgoing rates, but requires one vector in addition to the solution vector. While this new approach takes more memory than traditional Gauss-Seidel, it takes no more than Jacobi, and for on-the-fly and Kronecker-based methods, it takes less than Jacobi. We show how to extend column Gauss-Seidel to SOR, MAGS, and BGS.

These three contributions, namely on-the-fly rate generation, MAGS or BGS, and column Gauss-Seidel, are somewhat orthogonal in that they are independent contributions that can be applied in other contexts. For example, solutions based on Kronecker operators could benefit from both MAGS and column Gauss-Seidel. However, each contribution enhances the other, and taken as a whole, they present a new solution technique that addresses all restraining aspects of computing a solution to models that are otherwise intractable. To show the usability of these ideas, we implemented them. We built a tool to performn column Gauss-Seidel, column BGS, and column MAGS on a GSPN model, and we present various performance measurements in Section V.

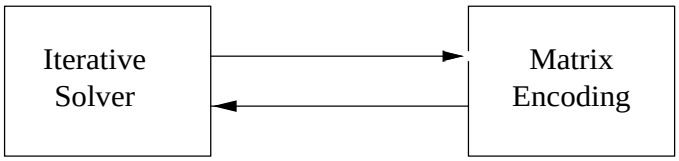

Fig. 1. Solution Paradigm.

II. Forward/Backward Access Algorithms

The first class of algorithms we develop makes use of both incoming and outgoing state transition rates. In this section, we show three algorithms: one for SPN models, one for GSPN models, and one for SAN or SRN models.

Before proceeding, we will introduce some helpful notation. In particular, we will address the solution of a system of simultaneous linear equations written as $\pi Q = 0$, where π is a row vector and Q is the state-transition-rate matrix. Since we focus on numerical solution techniques, we adopt the notation $Ax = b$, or more precisely $Ax = 0$, where $A = Q^T$ and $x = \pi^T$. Here, the off-diagonal i-th row elements of Q represent outgoing rates of state i in the corresponding Markov chain, and the off-diagonal column elements of A represent the same. Similarly, the off-diagonal i-th column elements of Q and the off-diagonal i-th row elements of A represent the incoming rates to state i in the corresponding Markov chain.

To facilitate understanding our new approach, we present in Figure 1 a simple paradigm for viewing the solution process. Instead of viewing the matrix as data, we view it as a function returning the requested portion of the matrix. Hence, when the matrix is stored explicitly in memory, the function may be quite trivial and efficient in terms of computation, but costly in terms of memory consumption. In this paradigm, the superposed GSPN methods use Kronecker operators on smaller matrices and a mapping function to generate an element of A. Thus, accessing an element of A requires more computation, but (usually) less memory. Kronecker-based methods have the disadvantage of requiring a special structure in the model in order to work efficiently. In contrast, our methods act directly on the net representation to generate a row or column of A. This requires significant computation, but it will work with any model and will always take memory proportional to the size of the model.

More specifically, let s_i represent an encoding of the i-th state of the model. The encoding may be a simple concatenation of the bit encoding of the number of tokens in places, or a more sophisticated encoding suggested by Kemper [6], [7], called a *mix*. The encodings of all the states

in the model form the set $S = \{s_1, s_2, \ldots, s_n\}$ (computed initially by a state space search). To compute the i-th column of A, we take the state encoding s_i with the model and compute the successor states and the rates to those states. The significant computational requirements to compute a column in A are to

1. Decode s_i
2. Determine all enabled timed transitions in s_i
3. Fire all enabled transitions and possibly search a network of immediate transitions to determine the rate to each successor state j
4. For each successor state j:
   (a) Encode s_j
   (b) Search for s_j in S to determine index j

If we must do a binary search to look for an element in S (for the most efficient use of space), the most expensive operation is probably 4 (b), which takes time $O(\log n)$, where n is the number of states in the model. If we are willing to use more memory, we may use a hash table to do the lookup in $O(1)$ time, but this is usually at the expense of additional memory.

Generating a column of A is therefore straightforward, but accessing A only by columns limits our choice of iterative methods to Jacobi or the power method (unless the new approach from Section IV is used). We would prefer to use Gauss-Seidel because it typically converges in fewer iterations and requires memory only for the solution vector. In order to use Gauss-Seidel, or variants of it, we need to have access to rows of A. To illustrate the need for access to rows of A, consider the basic action in the Gauss-Seidel method that we call a Gauss-Seidel *step*:

$$x_i^{(k+1)} = \frac{-1}{a_{ii}} \left(\sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} + \sum_{j=i+1}^{n} a_{ij} x_j^{(k)} - b_i \right) \tag{1}$$

where $x^{(k)}$ is the solution vector after k iterations. Doing a Gauss-Seidel step for i from 1 to n is called an *iteration*. In order to explicitly do the summation as shown above, one must have access to a row of A. Entries in the i-th row correspond to the incoming rates from predecessor states of s_i, so the task is to find the predecessor states and the corresponding incoming rates. The approach we take to finding these predecessor states is basically to execute the model one step "backwards" in time. Algorithms for performing this are given in this section.

Finding the diagonal element, a_{ii} in Equation 1, is also non-trivial, since it is defined as the negative sum of the outgoing rates. In general, we must either store the diagonal or compute

/* Return the vector of off-diagonal row i */

$a = 0$

for each $t \in T_i^{-1}$ do

$s_j \xleftarrow{r(t)} s_i$

if $s_j \in S$

$a_j = a_j + r(t)$

return a

Fig. 2. Algorithm to get i-th column for SPN model.

it each time it is needed. Storing the diagonal requires an additional vector, and although it may save a significant amount of computation, we show in Section IV that if we have space for a second vector, we can perform Gauss-Seidel without executing the model "backwards." Thus, the algorithms in this section are apropriate only for the most memory-limited problems, as one Gauss-Seidel step requires executing the model one step "backwards" and "forwards."

A. SPNs

For SPNs, by which we mean Petri nets (with no inhibitor arcs, although inhibitor arcs can be considered a special case of a marking-dependent rate) and exponentially timed transitions, the algorithm is simple. To understand it, we first introduce the notion of a reverse model. A *reverse model* is the corresponding model where the directions of all the arcs are reversed. The firing rules are the same except that any marking-dependent rates are determined after a transition fires. We let T_i be the set of (timed) transitions enabled in state i of the model, and T_i^{-1} be the set of transitions enabled in the reverse model. The notation $s_i \xrightarrow{r(t)} s_j$ means state i goes to state j with rate $r(t)$ by firing transition t. Similarly, $s_j \xleftarrow{r(t)} s_i$ means state i goes to state j with rate $r(t)$ in the reverse model, or, equivalently, state j goes to state i with rate $r(t)$ in the forward model. The symbol S denotes the set of reachable states. The algorithm for computing the non-diagonal row entries is shown in Figure 2.

The SPN modeling paradigm is simple, but modeling complex systems with simple SPNs is difficult. We present this algorithm because SPNs are simple and fast, and also because it gives us a framework on which we can build more complex algorithms.

B. GSPNs

The procedure for computing the outgoing states and rates for a GSPN model is a straightforward extension of SPNs and is generally well known. However, it is less trivial to compute the incoming states and rates, or correspondingly, the i-th off-diagonal row elements of A. Figure 3 shows the algorithm we propose to do this. This algorithm allows for general marking-dependent rates and weights, so we can replace inhibitor arcs with transitions with marking-dependent rates or weights. The new notation is as follows: T_i is the set of transitions enabled in state s_i, T_i^{-1} is the set of transitions enabled in the reverse model in state s_i, T^{-1} is a set containing transitions enabled in the reverse model that have become enabled exclusively by the firing of some immediate transition, and $s_j \overset{w(m)}{\longleftarrow} s_i$ means s_i goes to s_j by firing a single immediate transition m with weight $w(m)$ in the reverse model. I_i is the set of immediate transitions enabled in state s_i in the forward model, and I_i^{-1} is the set of immediate transitions enabled in state s_i in the reverse model.

The algorithm consists of two basic procedures that correspond to searching timed and immediate transitions. At a high level, we simply reverse the directions of the arcs and search all paths involving the firing of any number of immediate transitions followed by the firing of a timed transition. This algorithm does this in an organized way.

In particular, the algorithm starts by searching predecessor states reached by firing timed transitions in the reverse model. Those are the states that lead to the current state by firing only a single timed transition. After those are searched, set T^{-1} set to $\{\}$. Add transitions to T^{-1} only as they become enabled by firing an immediate transition in the reverse model. An intuitive explanation for this is that in the forward model, a stable marking goes to a stable marking by firing a timed transition followed by a number of immediate transitions. Therefore, if we trace the same path backwards in the reverse model, the path can not end with the firing of a timed transition that does not become enabled by the firing of immediate transitions along the path. We can avoid examining potentially many vanishing states this way and therefore prevent unnecessary computation.

The `search_back_im` procedure recursively searches through the network of immediate transitions. After an immediate transition is fired in the reverse model, we determine the probability $\hat{r}$ and try firing each $t \in T^{-1}$ to see if it results in a stable marking. We believe that computing I_i can be done efficiently and can prevent unnecessary searching in S for a vanishing marking

/* Return the vector of off-diagonal column i */

$a = 0$

for each $t \in T_i^{-1}$ do

$s_j \xleftarrow{r(t)} s_i$

if $s_j \in S$

$a_j = a_j + r(t)$

set $T^{-1} = \{\}$

call `search_back_im(`s_i`, 1)`

procedure `search_back_im(`s_i`, `r`)`

for each $m \in I_i^{-1}$

$s_j \xleftarrow{w(m)} s_i$ (update T^{-1})

$\hat{r} = r \times w(m) / \sum_{\forall k | s_j \xrightarrow{\hat{w}(k)} s_k} \hat{w}(k)$

for each $t \in T^{-1}$ do

$s_k \xleftarrow{\bar{r}(t)} s_j$

if $I_k = \{\}$ and $s_k \in S$

$a_k = a_k + \hat{r}\bar{r}(t)$

call `search_back_im(`s_j`, `$\hat{r}$`)`

Fig. 3. Algorithm to get i-th column for GSPN model.

(which is usually computationally more expensive).

Figure 3 shows the basic algorithm, but there are some possible improvements. We noted above that inhibitor arcs are a special case of marking-dependent values, which is the simplest way to deal with them. We could also build data structures that could help to tell us if a transition in the reverse model is "inhibited," that is, that there is no need to fire a transition in the reverse model because it would result in a state where that transition is inhibited in the forward model.

C. SANs/SRNs

Stochastic activity networks and stochastic reward nets are the most general modeling class we consider in that they can specify any arbitrary marking change upon completion of an activ-

ity/transition. This makes it virtually impossible to define a reverse model in closed or simple algorithmic form.

There is no single best way to solve SAN and SRN models on-the-fly because they can have wildly different characteristics. We propose a method that will work well for many models that are both sparsely connected and have small bounds on the number of tokens in places. The heuristic is that by observing bounds (during the initial state-space search) on the number of tokens in places, we can do an exhaustive search of all the possible combinations of markings that could have led to state i. To keep the terminology simple, we call a *transition* the mechanism which causes a delay, the enabling function, and all state changing mechanisms associated with it, including any connected immediate transitions. Furthermore, we let t_{out} be the set of places that transition t may modify when it fires, t_{in} be the set of places t uses to determine whether it is enabled, and t_* be $t_{in} \cup t_{out}$.

When performing the state space search, we note the range of the marking of all the places in t_{out} before and in t_* after t fires. The minimum value a place $p \in t_{out}$ takes while t is enabled is denoted $\texttt{minpre}(p,t)$; similarly the maximum is $\texttt{maxpre}(p,t)$. After t fires, the minimum value of the marking of each place $p \in t_*$ is recorded by $\texttt{minpost}(p,t)$, and the maximum by $\texttt{maxpost}(p,t)$. We present other auxiliary functions used in the algorithm in Figure 4, and the explanations are as follows: `enabled` returns true if transiton t is enabled in marking M, `fire` fires transition t in marking M and return the new marking, `mark` returns the number of tokens in place p in marking M, `rate` returns the firing rate of transition t in marking M, `state` returns a unique integer corresponding to marking M, and `setmark` sets the number of tokens in p to *integer* in state M and returns the new marking.

Now when trying to find all predecessor states of a state s_i, we examine each transition and eliminate those we know could not have fired to lead to s_i based on `minpost` and `maxpost`. Those transitions that are not eliminated are added to T^{-1}, where now T^{-1} is the set of transitions that *could* be enabled in the reverse model. Then, for each $t \in T^{-1}$, test exhaustively all possible combinations of each $p \in t_{out}$ based on `minpre` and `maxpre`. If a combination is found, the `state` function is used to find the corresponding state in the state table. Figure 5 shows the full algorithm.

We note that there are many conditions where this algorithm will perform poorly. It works best when the bounds of the markings of places are relatively small and the t_{out} set is small,

/* Auxiliary functions */

/* t: transition */

/* M: marking */

/* p: place */

`enabled`$(t, M) \rightarrow$ *boolean*

`fire`$(t, M) \rightarrow M$

`mark`$(p, M) \rightarrow$*integer*

`min/maxpre`$(p, t) \rightarrow$*integer*

`min/maxpost`$(p, t) \rightarrow$*integer*

`rate`$(t, M) \rightarrow \mathcal{R}$

`state`$(M) \rightarrow$*integer*

`setmark`$(p,$*integer*$, M) \rightarrow M$

Fig. 4. Auxiliary functions for SAN/SRN algorithm.

which we argue is typical of most nets. This method is really only computationally tractable if the number of combinations is small and the transition firing procedures are fast.

Since we don't know which markings are valid, we may place the model in an unreachable state and execute some marking-dependent function. We must impose that the modeler be aware of this possibility and write functions that will always give a reasonable answer (i.e. a function that sets the number of tokens in a place to be negative in any marking is illegal).

Other more effective techniques for eliminating combinations exist. In particular, the algorithm could keep more information than just the upper and lower bound of the number of tokens in places, but keep more detailed information, or information about linear combinations of tokens in places. We propose bounds because the memory requirement is at most $4|T||P|$, where T is the set of transitions, and P is the set of places. This holds the memory requirement to roughly the size of the model.

D. Comments

Searching "backwards" in the reverse model may be simple and efficient, as in SPNs or GSPNs with few immediate activities, or less efficient with complex GSPNs, SANs, or SRNs. The

$a = 0$

$T^{-1} = \emptyset$

$M_0 = M =$ current marking

for each transition t

  for each $p \in t_*$

    if $\texttt{minpost}(p,t) \leq \texttt{mark}(p,M) \leq \texttt{maxpost}(p,t)$

      $T^{-1} = T^{-1} \cup \{t\}$

for each $t \in T^{-1}$

  for each $p \in t_{out}$

    $M = \texttt{setmark}(p, \texttt{minpre}(p,t), M)$

  and $\texttt{mark}(p, \bar{M}) \leq \texttt{maxpre}(p,t), p \in t_{out}\}$

    if $\texttt{enabled}(t,M)$

      $\hat{M} = \texttt{fire}(t,M)$

      if $M_0 = \hat{M}$

        $r = \texttt{rate}(t,M)$

        $j = \texttt{state}(M)$

        $a_j = a_j + r$

Fig. 5. Algorithm to get i-th column for SAN/SRN model.

benefit of using the above algorithms is the lack of any structural requirements on the model, the ability to use Gauss-Seidel, and the minimal memory requirements (S and x). What makes this approach less attractive is that it must search both the reverse and forward model for each Gauss-Seidel step.

In the next section, we show that our extension to adaptive Gauss-Seidel can use the successor states to potentially reduce the number of Gauss-Seidel steps to achieve convergence. Section IV shows that with one exra vector we can efficiently implement Gauss-Seidel and variants without computing predecessor states.

III. Numerical Solution Methods That Exhibit Locality

We now have algorithms to generate a row and column of A on-the-fly, and we can perform Gauss-Seidel to compute a solution. Typically, A is very sparse, and it is obvious that generating

a row and column of A takes much more time than the time to perform one Gauss-Seidel step. (Our implementation in Section V shows a difference of a factor of about 80.) To improve solution times, we would like to find methods which exhibit locality. This allows us to generate a portion of the matrix and store it temporarily in a software cache, then use the portion of the matrix repeatedly in the solution process, then generate another portion of the matrix and continue. We are willing to perform more work in the solution process in order to reduce the amount of matrix generation in order to speed up the overall time to solution.

In this section we will present two methods that exhibit locality. The first, block Gauss-Seidel, is well known, and is reviewed briefly for completeness. The second, modified adaptive Gauss-Seidel, is new and an innovative extension to adaptive Gauss-Seidel [14], [15].

A. Block Gauss-Seidel

Block Gauss-Seidel (BGS) (e.g., [4]) solves the large system $Ax = b$ by solving many smaller systems $A_{ii}\hat{x}_i = r_i$, where

$$r_i = -(\sum_{\substack{j=1 \\ j \neq i}}^{N} A_{ij}\hat{x}_j),$$

where now A_{ij} is a submatrix of A and $\hat{x}_j$ is a subvector of x. Instead of solving a smaller system for $\hat{x}_i$ exactly at each step, we do a small number of Gauss-Seidel iterations, called *inner iterations*, to get $A_{ii}\hat{x}_i$ closer to $\hat{b}_i$ (an approximate solution). In general, the more inner iterations, the fewer outer iterations required to converge, up to a point of diminishing returns.

The strategy is to generate the portions of the matrix necessary to compute A_{ii} and r_i. Once this is computed, A_{ii} and r_i remain in memory while the solver does a number of Gauss-Seidel iterations. Although the iterative solver may do more work to get to the solution, it typically takes fewer iterations and therefore makes fewer accesses to the transition-rate-matrix, which satisfies the desired property. We can tune the solver by varying the number of inner iterations to get an optimum solution time.

Adaptive Gauss-Seidel. Modified adaptive Gauss-Seidel (MAGS) is an extension to adaptive Gauss-Seidel [14], [15] that exhibits locality. To motivate its formulation, we first review adaptive Gauss-Seidel. Adaptive Gauss-Seidel (AGS) is based intuitively on the observation that some elements sometimes converge or change more quickly than others, that is, $|x_i^{(k+1)} - x_i^{(k)}| > |x_j^{(k+1)} - x_j^{(k)}|$. If this is true, then a Gauss-Seidel step on x_i is considered more *effective* than a Gauss-Seidel step on x_j, and therefore more work should be done on x_i. The intuition is that

```
procedure AGS(ε)
M = s_1, ..., s_n
while M ≠ {}
    choose state s_i ∈ M
    M = M \ {s_i}
    t = x_i
    Gauss-Seidel_Step(i)
    if |t − x_i| > ε
        for all j ≠ i, a_ji ≠ 0
            M = M ∪ {s_j}
```

Fig. 6. Adaptive Gauss-Seidel iteration.

x_i is getting to the solution faster, so we should do steps on it more frequently. AGS is thus a variant of Gauss-Seidel where Gauss-Seidel steps are not necessarily performed in sequential order. Adaptive Gauss-Seidel is based on the methods of Rüde [16], for which he shows rigorously the effectiveness of the algorithm for the case where A is symmetric, positive definite. Since A is not symmetric or positive definite for Markov models, we use AGS as a heuristic. Our belief in its effectiveness is based on the fact that Horton [15] shows empirically that AGS needs significantly fewer floating point operations than standard Gauss-Seidel to solve certain Markov models to the same accuracy. Therefore, an implementation of adaptive Gauss-Seidel with on-the-fly matrix generation should compute a solution in less time than Gauss-Seidel.

If a Gauss-Seidel step on element i yields a large difference in $|x_i^{(k+1)} - x_i^{(k)}|$, then that work is considered effective, and large changes in x_i may also yield large changes in the values of the successors of state i. To quantify simply, if $|x_i^{(k+1)} - x_i^{(k)}| > \epsilon$, then we should also schedule the successors of s_i to receive steps. In Section II, we noticed that in order to compute a_{ii}, we need to compute the outgoing rates of state i. Adaptive Gauss-Seidel can take advantage of this by noticing the successor states of s_i (i.e., the non-zero entries of the i-th column).

In particular, let M be the set of states on which we need to perform work, which is initially set to S. Figure 6 shows the algorithm in detail for a given ϵ. The algorithm continues until M is empty. We call this one AGS iteration. The strategy to get a solution efficiently is to pick an initial large ϵ_0, call `AGS`, and then repeat the process with a successively smaller ϵ.

The way to decrease ϵ at each iteration is a difficult problem. Horton [14], [15] proposes decreasing it by a multiplicative constant $\Delta\epsilon$, shown here.

$\epsilon = \epsilon_0$
while not converged
    `AGS(`ϵ`)`
    $\epsilon = \epsilon \times \Delta\epsilon$

Choosing a good $\Delta\epsilon$ is also difficult. If we choose a value near one, it makes MAGS work like normal Gauss-Seidel. If $\Delta\epsilon$ is too small, fast-changing elements may start to converge to the wrong values, resulting in unnecessary work. Horton suggests values around 0.5, and our experimentation shows that this value sometimes works well for AGS and our modification as well. The stopping criteria could be any of the known criteria, or it could be a sufficiently small ϵ. This seems to be at least as good as the commonly used $||x^{(k+1)} - x^{(k)}||$ method.

Modified Adaptive Gauss-Seidel. Although AGS may speed convergence, since it works on states according to an "effectiveness" criterion, it does not ensure any kind of locality for data re-use. In particular, we note that AGS does not specify which state should be removed from M. We have modified the algorithm to narrow the choices in order to create locality. Specifically, we modify AGS by adding another set C, which is used to represent a software cache of M. The set C has two types associated with each element: *activated* and *deactivated*. We modify AGS by first limiting our working set to C, and when AGS would add s_i to M, instead check whether $s_i \in C$, and if it is, activate s_i; otherwise add s_i to M. The algorithm for modified AGS (MAGS) is given in Figure 7.

In practice, the order in which we choose elements from M or C plays a very significant role in the convergence characteristics. Experience has shown that the best convergence occurs when elements are chosen from C or M in a breadth first order. Experience has also shown that MAGS, while it is a valid implementation of AGS, does not usually perform as well as Horton's implementation of AGS. This tells us that the convergence characteristics of AGS are very dependent on the order in which elements are removed from M or C. In the worst performance, MAGS can perform worse that Gauss-Seidel, although with $\Delta\epsilon = 1$, they will always perform the same. We believe that more work needs to be done on choosing a good value for $\Delta\epsilon$ before AGS or MAGS can be an effective general solver.

```
procedure MAGS(ε)
M = s_1, ..., s_n
while M ≠ {}
    C ⊂ M
    M = M \ C
    while there exists an active element in C
        choose an active s_i ∈ C
        deactivate s_i in C
        t = x_i
        call Gauss-Seidel_Step(i)
        if |t − x_i| > ε
            for all j ≠ i, a_ji ≠ 0
                if s_j ∈ C then activate s_j in C
                else M = M ∪ {s_j}
```

Fig. 7. Modified adaptive Gauss-Seidel iteration.

IV. Forward Solution Methods

The complexity in applying the above iterative solution techniques comes because they are based on Gauss-Seidel iteration steps, and hence require row access to A. It may seem that there are two choices: access A inefficiently by rows and columns for solution technique that uses less memory and typically converges faster (Gauss-Seidel and variants), or access A more efficiently by columns and settle for a solution technique that uses more memory and typically converges slower (Jacobi and JOR). What we show in this section is a compromise that in some ways has the best of both.

We show that with little additional work and memory requirements identical to those for the Jacobi method (one additional vector the size of a solution vector), we can also perform Gauss-Seidel-based methods by accessing A only by columns. Note that Kronecker-based methods typically use the Jacobi method and explicitly store the diagonal, requiring a total of three vectors (current estimate, next estimate, and diagonal). Our result is important since it allows use of the more powerful Gauss-Seidel with less memory and approximately the same amount of

work. This generalizes for both on-the-fly and Kronecker techniques, and other iterative solution methods based on Gauss-Seidel (such as SOR, BGS, and MAGS).

With this new method, we can now realistically compute solutions on-the-fly to the more expressive modeling paradigms (SANs and SRNs). Although the method works for all iterative solution techniques based on Gauss-Seidel steps, we develop it in terms of Gauss-Seidel first, and then show how it can be used in more sophisticated variants, such as SOR, MAGS, and BGS.

A. Column Gauss-Seidel

To understand how we can eliminate the need for row access, we recall that the basic operation in many Gauss-Seidel-based iteration schemes is the Gauss-Seidel step, given in (1). By using this step as the basic unit of computation, we can seamlessly replace Gauss-Seidel steps with the new variant, which requires only column (successor) access in other iterative methods.

We introduce our strategy with a vector δ, which we define as

$$\delta_i = x_i^{(k+1)} - x_i^{(k)},$$

so that a Gauss-Seidel step on element i is equivalent to setting $x_i^{(k+1)} = x_k^{(k)} + \delta_i$. We show how to initialize δ, and then given δ, we show how doing a Gauss-Seidel step on element i affects δ_j for all $j \neq i$.

In particular, let $x^{(0)}$ be some initial guess. We initialize δ by the following:

$$\begin{aligned}
&\delta = 0 \\
&\text{for } i = 1 \text{ to } n \\
&\qquad \text{for } j = 1 \text{ to } n | j \neq i \\
&\qquad\qquad \delta_j = \delta_j + a_{ji} x_i^{(0)} \\
&\text{for } i = 1 \text{ to } n \\
&\qquad \delta_i = (b_i - \delta_i)/a_{ii} - x_i^{(0)}
\end{aligned}$$

This essentially does a Jacobi iteration and places $x^{(1)} - x^{(0)}$ in δ. This is what we want because if we choose to start Gauss-Seidel at x_i, then $x_i^{(1)} = x_i^{(0)} - \delta_i$. (The first Gauss-Seidel step is identical to the first Jacobi step.)

Now we may do a Gauss-Seidel step on any element by simply doing the computation $x_i^{(1)} = x_i^{(0)} + \delta_i$. Once we do the computation, however, δ is in general obsolete. We now show how to update δ after each Gauss-Seidel step. Say we do a Gauss-Seidel step on x_i, in the most general

form

$$x_i^{(k)} = \frac{1}{a_{ii}} \left(- \sum_{\substack{j=1 \\ j \neq i}}^{n} a_{ij} x_j^* + b_i \right) \ ,$$

where x_j^* is the most recently computed value of x_j. After this step, $\delta_i = 0$. Now say we do a Gauss-Seidel step on x_c, and then observe the effects of this computation on δ_i.

$$x_c^{(p+1)} = x_c^{(p)} + \delta_c$$

$$x_i^{(k+1)} - x_i^{(k)} = \frac{-1}{a_{ii}} \left(a_{ic} x_c^{(p+1)} - a_{ic} x_c^{(p)} \right)$$

Finally,

$$\delta_i = \frac{-a_{ic} \delta_c}{a_{ii}} \ .$$

Now let us assume $\delta_i \neq 0$. We denote δ_i^0 as the value of δ_i before performing a Gauss-Seidel step on x_k. Inductively, we can show that after we do a Gauss-Seidel step on x_c, we can compute the new δ_i from the value of δ_i^0 and δ_c.

$$x_i^{(k+1)} - x_i^{(k)} = \delta_i^0 + \frac{-1}{a_{ii}} \left(a_{ic} x_c^{(p+1)} - a_{ic} x_c^{(p)} \right)$$

$$\delta_i = \delta_i^0 - \frac{a_{ic} \delta_c}{a_{ii}}$$

Now we can see that updating δ_i after performing a Gauss-Seidel step on x_c requires access to the c-th column of A. In addition, computing δ_i also needs a_{ii}, but this dependency is easy to eliminate. If we let $d_i = a_{ii} \delta_i$, and d_i^0 is the value of d_i before performing a Gauss-Seidel step on x_c, then

$$d_i = d_i^0 - a_{ic} \delta_c \ . \tag{2}$$

Then, when doing the Gauss-Seidel step on x_i, simply divide d_i by a_{ii} to get δ_i, and update all $d_j | \, a_{ji} \neq 0$. Thus, in an implementation, the values for x and d must be kept explicitly. This is sufficient to perform a Gauss-Seidel step on any element by accessing only the i-th column of A.

Successive Over-Relaxation. We now show how to easily extend this method to Successive Over-Relaxation (SOR). Recall the basic step for SOR:

$$x_c^{(k+1)} = \omega \bar{x}_c^{(k+1)} + (1 - \omega) x_c^{(k)} \ ,$$

where $\bar{x}_i$ is the Gauss-Seidel iterate. We computed above

$$\bar{x}_c^{(k+1)} = x_c^{(k)} + \delta_c \ ,$$

and by substitution,

$$x_c^{(k+1)} = x_c^{(k)} + \omega\delta_c \ .$$

Again, the vector d is stored and δ_i is computed at the time the SOR step on x_i is performed. The updating of d_i, $\forall i \neq c$ is done by (2). From this, we can see that the column-only SOR step involves only a minor extension to the column-only Gauss-Seidel.

Algorithm. Figure 8 shows the algorithm for doing a Gauss-Seidel step using only column access. We show the algorithm with the feature of over-relaxation parameter ω, which is set to 1 for standard Gauss-Seidel, but in general can take on values $\omega \in (0, 2)$. The initialization step in the figure is different than that shown above because we are initializing d instead of δ. Note that the initialization procedure has two steps. The first step requires one sweep of the matrix, and the second step requires access to the diagonal of the matrix. If the initial guess $x^{(0)}$ is set to some known value, e.g., $x_i^{(0)} = 1/n$, then the space allocated for x can store the diagonal of the matrix during the first sweep, avoiding a need to access the matrix at all in the second step.

Notice that the column Gauss-Seidel step on x_i accesses only the i-th column of A. Thus, column Gauss-Seidel allows us to perform any Gauss-Seidel step using only successor states and corresponding rates of state s_i. The diagonal element, a_{ii}, is easily computed as the negative sum of the rates to successor states.

As an example of the use of this implementation of a Gauss-Seidel step, we show an implementation of standard Gauss-Seidel.

call `cGauss-Seidel_Step_Init()`
x = initial guess
while x not converged
    for $i = 1$ to n
        call `cGauss-Seidel_Step(`i`)`

We call this algorithm *column Gauss-Seidel*.

To illustrate the memory and computation costs of the algorithm, we compare column Gauss-Seidel with traditional implementations of Gauss-Seidel and Jacobi with respect to operation count and memory requirements in Table I. The column Gauss-Seidel we illustrate in the table is without over-relaxation and assumes a sparse $n \times n$ matrix A with m non-zero entries is stored explicitly in memory. Notice that we can do column Gauss-Seidel with the same memory

/* Matrix $A \in \mathcal{R}^{n \times n}$ */

/* arrays x, b, and $d \in \mathcal{R}^n$ */

/* Solve $Ax = b$ using d. */

procedure `cGauss-Seidel_Step_Init()`

$d = 0$

for $i = 1$ to n

    for $j = 1$ to $n | \, j \neq i$

        $d_j = d_j + a_{ji} x_i$

for $i = 1$ to n

    $d_i = (b_i - d_i / a_{ii} - x_i) a_{ii}$

procedure `cGauss-Seidel_Step(`int i`)`

$\delta = \omega d_i / a_{ii}$

$x_i = x_i + \delta$

for $j = 1$ to $n | \, j \neq i$

    $d_j = d_j - a_{ji} \times \delta$

Fig. 8. Gauss-Seidel step requiring only column access to A.

requirements as Jacobi, and after an initialization cost, the same number of operations per iteration as Jacobi and Gauss-Seidel.

The table may be misleading for matrix-free techniques because the matrix may be accessed effeciently only by columns. Thus, normal Gauss-Seidel is infeasable (unless the algorithms in Section II are used), and the Jacobi method requires an extra vector to hold the diagonal. This is not the case for Column Gauss-Seidel. Thus, on-the-fly and Kronecker methods are better matched with column Gauss-Seidel than Jacobi. The only disadvantage of column Gauss-Seidel is the initialization step, which requires the work of approximately one Jacobi iteration and one sweep of the matrix.

Furthermore, column Gauss-Seidel has some improved numerical properties relative to Gauss-Seidel or Jacobi. As the iteration process approaches the solution, the algorithm keeps d to full precision, even while variations in x are small. Column Gauss-Seidel may thus proceed as if x were kept to greater precision, because all the important information about x (namely $x^{(k+1)} - x^{(k)}$)

TABLE I

Measures for different algorithms.

Operation	Gauss-Seidel	Jacobi	Column Gauss-Seidel
Initialization	n/a	n/a	$m+n$ mult n div $m+2n$ add
Iteration	m - n mult n div m add	m - n mult n div m add	m - n mult n div m add
Memory	m + n	m + 2n	m + 2n

is stored in d; this is useful when elements in x vary in size by many orders of magnitude and the user requires a high degree of accuracy. The algorithm is not self-correcting, however. If somehow (due to rounding errors, for example,) x is perturbed, the algorithm will converge to the wrong answer. An easy solution to this is to reinitialize d when the iteration process is near the solution, or after every several digits of accuracy acquired. This reinitialization, however, will require two sweeps of the matrix, an additional vector, or saving and restoring the solution vector from disk.

B. Column Modified Adaptive Gauss-Seidel and Column Block Gauss-Seidel

As mentioned earlier, the approach used to obtain column-only Gauss-Seidel can be extended to all Gauss-Seidel-based algorithms. In this case of modified adaptive Gauss-Seidel, this is very straightforward; the routine `cGauss-Seidel_Step` is a direct replacement for the routine `Gauss_Seidel_Step`. This substitution results in an algorithm that can solve any model class, especially SAN or SRN models, with much greater speed than the variant that requires row access. The cost of using column Gauss-Seidel is some extra time spent in initialization (negligible) and the extra memory to hold d. If this memory is available, the column-only variant should be used.

In the case of modified block Gauss-Seidel, we can take a similar approach to that done for Gauss-Seidel. The details of how this is done deserve some elaboration, however, since we are dealing with matrices instead of scalars. In particular, let A be divided into $N \times N$ sub-matrices,

labeled A_{11} to A_{NN}. Likewise, x and b are divided into N vectors $\hat{x}_i$ and $\hat{b}_i$. Traditional implementations of block Gauss-Seidel solve explicitly (in the general case)

$$\hat{x}_i^{(k+1)} = -A_{ii}^{-1} \left(\sum_{\substack{j=0 \\ j \neq i}}^{N} A_{ij} \hat{x}_j^* - \hat{b}_i \right). \tag{3}$$

By letting r_i be an $\frac{n}{N}$-sized vector such that $r_i = -\sum_{\substack{j=0 \\ j \neq i}}^{N} A_{ij}\hat{x}_j + \hat{b}_i$, block Gauss-Seidel reduces to repeatedly solving $A_{ii}\hat{x}_i = r_i$. Note that a direct implementation of Equation 3 requires accessing A by block rows. We present a method that only accesses A by block columns.

Let $\hat{x}_i^{(k+1)} - \hat{x}_i^{(k)} = \hat{\delta}_i$. Say $\hat{\delta}_i^0$ is the correct and current value for $\hat{\delta}_i$. Now we observe what happens to $\hat{\delta}_i$ as a result of a change in $\hat{x}_c$ by $\hat{\delta}_c$.

$$\hat{x}_c^{(p+1)} = \hat{x}_c^{(p)} + \hat{\delta}_c$$

$$\hat{\delta}_i = \hat{\delta}_i^0 - A_{ii}^{-1} \left(A_{ic}\hat{x}_c^{(p+1)} - A_{ic}\hat{x}_c^{(p)} \right)$$

$$\hat{\delta}_i = \hat{\delta}_i^0 - A_{ii}^{-1} A_{ic}\hat{\delta}_c$$

If we let $\hat{d}_i = A_{ii}\hat{\delta}_i$, and $\hat{\delta}_i^0$ be $\hat{\delta}_i$ before the BGS step on $\hat{x}_c$,

$$\hat{d}_i = \hat{d}_i^0 - A_{ic}\hat{\delta}_c,$$

and $\hat{d}$ can be updated by accessing only the c-th block column of A.

Here is the algorithm for one step of column Block Gauss-Seidel with over-relaxation parameter ω.

procedure `BGS_Step`(int i)
solve approximately $A_{ii}\hat{\delta} = \hat{d}_i$ for $\hat{\delta}$
$\hat{\delta} = \omega\hat{\delta}$
$\hat{x}_i = \hat{x}_i + \hat{\delta}$
for $j = 1$ to $n | \, j \neq i$
    $\hat{d}_j = \hat{d}_j - A_{ji}\hat{\delta}$

For the initialization procedure, one can use an approach similar to the initialization procedure given in Figure 8. Indeed, if one uses Gauss-Seidel inner iterations, the exact initialization procedure in Figure 8 may be used.

```
/* Queue Q, Tree S */
/* succ(s) returns set of successor states */
Q.enqueue(s_0)
S.insert(s_0)
while Q.notempty()
    s_i = Q.dequene()
    for each s_j ∈ succ(s_i)
        if not S.find(s_j)
            S.insert(s_j)
            Q.enqueue(s_j)
```

Fig. 9. State space search algorithm.

V. Prototype Implementation Performance Comparison

To show the usability of the on-the-fly method, we implemented some of the ideas into a tool and took performance measurements. The tool allows three solution methods: column Gauss-Seidel, column block Gauss-Seidel, and column modified adaptive Gauss-Seidel. The implementation is a prototype, so we don't expect optimal performance, but we do believe the results are roughly indicative.

A. Solution Process

The solver begins by reading in the model specification from a file, and then performs a state space search. A marking is encoded into a state, which is represented by a bit array where each place uses only the necessary number of bits. The states are stored in a tree, where the states are held in an array, and the "left" and "right" child pointers use the same space that is later used for the solution vector. States are explored in a breadth-first ordering, and a queue is also allocated to facilitate this. After the states have been explored as shown in Figure 9, the state array is sorted to make efficient use of space and $O(\log n)$ lookup time. Unfortunately, this has the result of essentially permuting the matrix formed from the natural breadth-first ordering of states.

Next, the column Gauss-Seidel initialization is called to initialize the d array, and then assign a uniform initial guess for x. We note that the order in which a tool performs Gauss-Seidel steps

```
/* State array S, bit array M */
sort S
call Gauss-Seidel_Step_Init()
set x to initial guess
while ||d|| > ε
    i = initial state
    Q.enqueue(i)
    M = 0
    while Q.notempty()
        i = Q.dequene()
        call cGauss-Seidel_Step(i)
        for each s_j ∈ succ(s_i)
            if M_j = 0
                M_j = 1
                Q.enqueue(j)
```

Fig. 10. Column Gauss-Seidel implementation

has a substantial impact on the convergence rate, and it is very desirable to perform Gauss-Seidel steps in a breadth-first order, the same order in which the states are explored. To accomplish this, the tool uses the queue (Q of Figure 9), and starting from the initial state, it performs a breadth-first ordering of the states. A bit vector M is used to keep track of the visited states.

The tool allocates space for the following: S, x, d, M, and Q. Therefore, the memory usage of the tool is $|S| + |x| + |d| + |M| + |Q|$, or on a typical unix workstation, 129 bits plus $|s|$ per state, plus the size of the queue.

For improved performance, we implemented column block Gauss-Seidel. A direct implementation of column BGS (see Section IV-B) is difficult because we would like to perform BGS on the states in a breadth-first order (for fast convergence). Since the tool is essentially solving a permuted form of the matrix, it difficult to determine whether a matrix entry is within the diagonal block. Instead, we choose a simpler implementation which generates a block of rows, stores them in a software cache, then performs column Gauss-Seidel on those rows a fixed number of times. This method is mathematically identical to the method shown in Section IV-B where

the smaller matrix is solved by Gauss-Seidel, but takes more computation to do inner iterations because r is implicitly formed during each inner iteration. We are willing tolerate this increased work because we typically do only a small number of inner iterations and the time to do the inner iterations is small compared to the time spent computing the rows.

The size of the cache is specified by the maximum number of rows and the maximum number of nonzero entries for all the rows. Rows are added to the cache until the cache is full, and then a fixed number of column Gauss-Seidel steps are performed on the states corresponding to the rows in the cache.

The modified adaptive Gauss-Seidel implementation performs the algorithm shown in Figure 7. Elements are chosen from M and placed in C in a breadth-first order. Unlike Gauss-Seidel or BGS where the matrix is accessed in breadth-first order exactly once per iteration, MAGS can access the matrix many times in the breadth-first order, skipping the elements not in M. Thus, we can not (efficiently) use the successor states and Q to find the breadth-first ordering of the states, so we instead use an integer array to order the states. Elements are put in C in a breadth-first order of activated elements, and activated elements are removed from C by round-robin. This assures that each sweep through C performs Gauss-Seidel steps on activated states in a breadth-first order. The cache keeps a count of the number of activated elements, and when it is zero, the cache is filled with new elements. When M is empty, one MAGS iteration is complete.

The stopping criteria used for the tool is $||x^{(k+1)} - x^{(k)}||_\infty < \epsilon$, where typically $\epsilon \in \{10^{-6}, 10^{-9}, 10^{-12}\}$. MAGS has a slightly different notion of what an iteration is; a MAGS iteration is complete when M is empty. The stopping criteria used by MAGS is the largest difference between successive estimates of x_i within an interation. Admittedly, these stopping criteria are not particularly good, but they are relatively easy to compute. Stopping criteria based on the residual norm are difficult to compute for matrix-free methods because the residual is never explicitly computed (and explicit computation would take more space), and because within an iteration where A is generated, the solution vector changes.

B. Kanban Manufacturing System Model

To show the performance of the tool, we choose a model of a Kanban manufacturing system. This is a simple model which has appeared in [8], where it is described in detail, and we omit repeating the information here for the sake of space. We chose this model for two reasons. First, we can comparison of measurements to those obtained in [8]. We believe this to be the most

TABLE II

CHARACTERISTICS OF THE KANBAN MODEL.

N	States	NZ Entries	Size (MB)
1	160	616	0.008
2	4,600	28,128	0.34
3	58,400	446,400	5.3
4	454,475	3,979,850	47
5	2,546,432	24,460,416	290

complete implementation of a method based on Kronecker operators, which uses Jacobi as the solution method. Second, the model can generate a range of state space sizes, depending on a parameter value N. Information about the size of the model as a function of N is shown in Table II. The size is the amount of memory it would take to store a matrix in a sparse representation.

We begin the comparison by showing the execution time of the tool in [8], which is repeated in Table III under "Case 1" and "Case 2." In [8], the stopping criteria is $||x^{(k+1)} - x^{(k)}||_\infty < 10^{-6}$, which we note is not a good stopping criteria for models with approximately 10^6 states. Also, it uses a relaxation parameter $\omega = 0.9$, which may increase the convergence rate. It is unfortunate that we do not know how many iterations it took to converge, because this would give us a much better basis of comparison. Case 1 is solved for the general case, where the so-called product space can be larger than the tangible reachable space, and Case 2 is an idealized case, where the product space is the tangible reachable space. The tool was ran on a Sony NWS-5000.

Results from our implementation using column Gauss-Seidel is shown on the bottom section of Table III, where "cGS Init" is the time to perform the state space search and sort the states, "cGS Solve" is the time to solve to the same stopping criteria, "cGS Iter" is the number of Gauss-Seidel iterations required to reach convergence, and "cGS Memory" is the memory allocated by our tool in megabytes. We used $\omega = 1.0$ (no SOR), on an HP C-160 workstation (160 MHz PA-8000), which we estimate is approximately three times faster than the machine used in [8].

One can observe that our tool took approximately the same time as Case 1 of the Kronecker-based tool. Three factors make this difficult to make a quantitative comparison. First, we use different solution methods (Jacobi versus Gauss-Seidel), so the same stopping criteria may mean

TABLE III

COMPARISON OF PERFORMANCE.

N	1	2	3	4	5
Case 1	1	13	310	4721	22,215
Case 2	1	2	2	856	6,055
cGS Init	0	1	13	167	1231
cGS Solve	1	12	345	4415	39,762
cGS Iter	17	30	43	52	69
cGS Memory	.3	.75	1.8	10.9	57

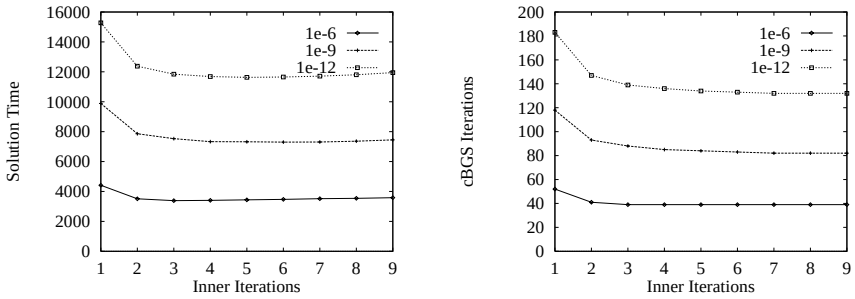

Fig. 11. Vary number of inner iterations, 16,000 cached rows.

different levels of accuracy. Second, we use different overrelaxation constants. We suspect that for Jacobi, using $\omega = 0.9$ may substantially decreases the number of iterations necessary, and we don't know if the Kronecker-based tool took more or fewer iterations. Finally, we estimate that our computer is approximately three times faster. We can speculate, however, that it is likely that the two approaches differ in performance only by a small factor. We believe this a very encouraging result given the greater generality of our approach.

We took several other performance measures on this model for which we set $N = 4$. We were interested in seing how using block Gauss-Seidel would affect performance. We have basically two experiments. First, we choose the number of rows to cache to be 16,000 and vary the number of inner iterations and observe how it affects solution time and number of iterations to three levels of accuracy, corresponding to the stopping criteria of $||x^{(k+1)} - x^{(k)}||_\infty < \{10^{-6}, 10^{-9}, 10^{-12}\}$. These are shown in Figure 11. Again, we performed the same experiment except that we set the number of rows to cache to be 64,000. Figure 12 displays the results from this experiment.

It is interesting to observe that choosing the optimal number of inner iterations may be difficult.

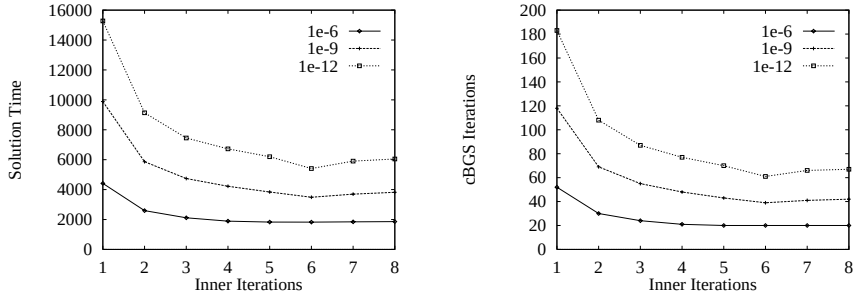

Fig. 12. Vary number of inner iterations, 64,000 cached rows.

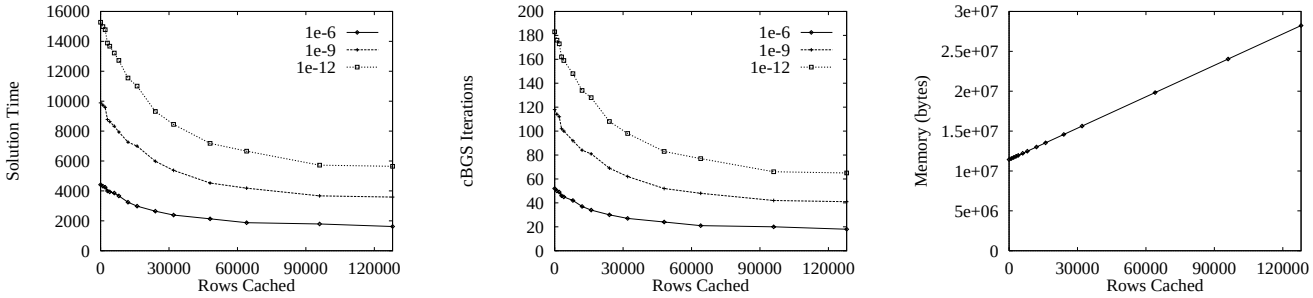

Fig. 13. Vary cache size

With 16,000 rows cached, having more than four inner iterations yielded no decrase in solution time, while with 64,000 rows cached, more than six inner iterations will not decrease solution time. Also, doing more inner iterations than necessary does not substantially increase execution time. In fact, with two inner iterations, we calculat that the average BGS iteration time is 83.88 seconds, while with 9 inner iterations, the average BGS iteration time is 89.95 seconds. As suggested earlier, the time to do inner iterations is small compared to the time to generate a portion of the matrix on-the-fly. One can then safely set the number of inner iterations to be higher than necessary and not worry about any substantial negative affects on performance.

For the second experiment, we fix the number of inner iterations to be 4, and vary the number of cached rows. These results are reported in Figure 13. As the number of cached rows increases, the memory usage increases as well, which is plotted on the right of Figure 13 as well. Note that there is a clear space/memory tradeoff. One can approximately double performance while doubling the amount of memory used.

Finally, we performed various experiments witht he MAGS implementation. The performance of MAGS for small models was encouraging, but disappointing with larger models. Figure 14 shows a MAGS solution on the Kanban model where $N = 2$ (4,600 states). The cache size is varried from 1 (adaptive Gauss-Seidel) to 400. Not plotted are the Gauss-Seidel solution times,

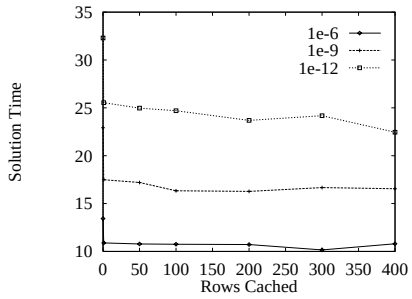

Fig. 14. Vary cache size

which are 13, 23, and 32 seconds for 10^{-6}, 10^{-9}, and 10^{-12} thresholds respectively. With equal size caches, MAGS performs better than BGS when the cache is small, but worse when the cache is larger.

This proved to be somewhat encouraging results. However, for the larger models where $N = 4$ (454,475 states), performance for MAGS with $\Delta\epsilon = 0.5$ got significantly worse. For the three stopping criteria, the AGS (MAGS with one column cached) solution time was 13303, 49237, 73537 seconds, compared to 4415, 9887, 15276 seconds for Gauss-Seidel. The discrepency for $N = 3$ (58,400 states) was not as much, but still significant. Certainly, for a $\Delta\epsilon$ near one, AGS or MAGS will perform like Gauss-Seidel, and so it will do no worse than Gauss-Seidel. Choosing a good value for $\Delta\epsilon$ may decrase the solution time for AGS and MAGS, but ranging $\Delta\epsilon$ over many values to find a good one is both very time consuming and not a practical way to find $\Delta\epsilon$ in general.

We conclude that adaptive Gauss-Seidel is not a good, general purpose solver due to the complexity of finding a good $\Delta\epsilon$, which is probably closely related to the problem of finding a good value for ω for SOR. If some mechanism for automatically generating a good $\Delta\epsilon$ is found in the future, MAGS may be useful as a general solver, because in the worst case $\Delta\epsilon$ can be set to one and MAGS would perform identical to Gauss-Seidel. For a general solution approach, we suggest using BGS with as large of a cache as possible.

VI. Conclusion

Our goal was to extend the types of models that can be solved by memory-efficient matrix-free methods to include a much greater class of models and formalisms. We also wanted to use iterative solvers that converge more quickly than Jacobi or the power method, and we wanted to find solvers that take less memory.

We achieved these goals by three important contributions. First, we introduced the idea of on-the-fly matrix generation, and gave the most memory-efficient algorithms that would work with traditional implementations of Gauss-Seidel. These methods work without regard to any special structure of the model. In particular, we developed algorithms for SPN, GSPN, and SAN and SRN models. These algorithms only require memory to hold the set of reachable states and the solution vector.

Second, we acknowledged that generating a row or column of the matrix is much more time consuming than doing a Gauss-Seidel step, so we looked for ways to store and re-use portions of the matrix. This includes the well-known block Gauss-Seidel and the new modified adaptive Gauss-Seidel methods.

Finally, we introduced column Gauss-Seidel and variants, which is particularly well suited to matrix-free methods. This allows to efficiently perform Gauss-Seidel (or variants) with two vectors, in contrast to the previous requirement of using Jacobi with three vectors. Column Gauss-Seidel, along with row caching, may be applied to other matrix-free methods to increase performance of those particular methods. However, in conjunction with on-the-fly methods, we can now efficiently solve more general models with more powerful iterative solution methods using less memory than previously possible.

References

[1] G. Chiola and G. Franceschinis, "Colored GSPN models and automatic symmetry detection," in *Proceedings of the International Workshop on Petri Nets and Performance Models (PNPM'89)*, Kyoto, Japan, December 1989, pp. 50–60.

[2] W. H. Sanders and J. F. Meyer, "Reduced base model construction methods for stochastic activity networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 1, pp. 25–36, January 1991.

[3] B. Plateau and K. Atif, "A methodology for solving markov models of parallel systems," *IEEE Journal on Software Engineering*, vol. 17, no. 10, pp. 1093–1108, 1991.

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

[5] S. Donatelli, "Superposed generalized stochastic Petri nets: Definition and efficient solution," in *Application and Theory of Petri Nets 1994, Lecture Notes in Computer Science 815 (Proc. 15th Int. Conf. on Application and Theory of Petri Nets, Zaragoza, Spain)*, R. Valette, Ed., pp. 258–277. Springer-Verlag, June 1994.

[6] P. Kemper, "Numerical analysis of superposed GSPNs," in *Sixth International Workshop on Petri Nets and Performance Models*, Durnham, NC, October 1995, pp. 52–61.

[7] P. Kemper, "Numerical analysis of superposed GSPNs," *IEEE Transactions on Software Engineering*, vol. 22, no. 9, September 1996.

[8] G. Ciardo and M. Tilgner, "On the use of Kronecker operators for the solution of generalized stochastic petri nets," ICASE Report #96-35 CR-198336, NASA Langley Research Center, May 1996.

[9] G. Ciardo, "Advances in compositional approaches based on Kronecker algebra: Application to the study of manufacturing systems," in *Third International Workshop on Performability Modeling of Computer and Communication Systems*, Bloomingdale, IL, September 1996, pp. 61–65.

[10] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli, and G. Franseschinis, *Modeling with generalized stochastic Petri nets*, John Wiley & Sons, 1995.

[11] J. F. Meyer, A. Movaghar, and W. H. Sanders, "Stochastic activity networks: Structure, behavior, and application," in *Proc. International Workshop on Timed Petri Nets*, Torino, Italy, July 1985, pp. 106–115.

[12] A. Movaghar and J. F. Meyer, "Performability modeling with stochastic activity networks," in *Proc. 1984 Real-Time Systems Symp.*, Austin, TX, December 1984.

[13] G. Ciardo, A. Blakenmore, P. F. J. Chimento, J. K. Muppala, and K. S. Trivedi, "Automatic generation and analysis of Markov reward models using stochastic reward nets," in *Linear Algebra, Markov Chains, and Queueing Models*, C. Meyer and R. J. Plemmons, Eds., pp. 141–191. Springer-Verlag, 1993.

[14] G. Horton, "Adaptive relaxation for the steady-state analysis of Markov chains," in *Computations with Markov Chains*, pp. 585–586. Kluwer Academic Publishers, Boston, 1995.

[15] G. Horton, "Adaptive relaxation for the steady-state analysis of Markov chains," ICASE Report #94-55 NASA CR-194944, NASA Langley Research Center, June 1994.

[16] U. Rude, "On the multilevel adaptive iterative method," in *Preliminary Proceedings of the Second Copper Mountain Conference on Iterative Methods*, T. Manteuffel, Ed. April 1992, SIAM.