Automated Markov-chain based Analysis for Large State Spaces

Kaitlin N. Smith, Michael A.Taylor, Anna A. Carroll, Theodore W. Manikas, Mitchell A. Thornton
Darwin Deason Institute for Cyber Security
Southern Methodist University
Dallas, Texas, USA
{knsmith, taylorma, aacarroll, manikas, mitch}@smu.edu

***Abstract*—Modeling the dynamic, time-varying behavior of systems and processes is a common design and analysis task in the systems engineering community. A popular method for performing such analysis is the use of Markov chains. Additionally, automated methods may be used to automatically determine new system state values for a system under observation or test. Unfortunately, the state-transition space of a Markov chain grows exponentially in the number of states resulting in limitations in the use of Markov chains for dynamic analysis. We present results in the use of an efficient data structure, the algebraic decision diagram (ADD), for representation of Markov chains and an accompanying prototype analysis tool. Experimental results are provided that indicate the ADD is a viable structure to enable the automated modeling of Markov chains consisting of hundreds of thousands of states due to their ability to provide computation related efficiencies. This result allows automated Markov chain analysis of extremely large state spaces to be a viable technique for system and process modeling and analysis. Experimental results from a prototype implementation of an ADD-based analysis tool are provided to substantiate our conclusions.**

Keywords—Markov chain, Algebraic Decision Diagram, ADD, reliability analysis tool, dynamic system analysis

I. INTRODUCTION

Markov chain models are commonly used to analyze reliability and other system characteristics. Markov chains have also been used to model processes wherein the process is composed of a discrete set of potential steps in some order of application. Recent Markov chain applications include fault analysis of solar array systems [1], communication network analysis [3], energy usage prediction for cellular base stations [4], and prediction of cascading failures in power grids [5]. Markov chains allow prediction of dynamic system performance through various states [2]. Markov analysis examines a sequence of events and determines the tendency of one event to be followed by another event. A Markov chain assumes that the system behavior can be modeled as a set of discrete states and an accompanying discrete time parameter.

A transition matrix represents transitions between the states of a Markov chain model for a system. Various analysis methods are applied to the transition matrix to determine system reliability. However, there is an issue with scalability of this modeling approach. As the number of system states increases in the Markov chain, the size of the matrix will also increase. For N states, the matrix size is N^2. For example, if 10 states are included in a model, the corresponding transition matrix contains 100 elements (*i.e.*, a 10×10 transition matrix). This causes the computational complexity of analyzing the transition matrix to grow rapidly with respect to the number of states in the Markov chain.

While the use of Markov chains in systems engineering is a well-known and commonly used method, a chief limitation on the use of Markov chain analysis is the large computational complexity involved in their representation and manipulation. In particular, Markov chains that are constructed and manipulated with automated methods can rapidly grow in size and require unacceptably large amounts of computer memory storage accompanied with unacceptably large computation times for extracting analysis results from the chain. In the past, this complexity has limited the usage of Markov chains to systems and processes that can be modeled with relatively few states and has prevented their use for applications, particularly automated applications, wherein the Markov chain is constructed through automatic system state discovery and analysis.

A related limitation is the inability to efficiently add newly discovered states to an existing Markov chain. When new states are determined or discovered in an automatic manner, it is desirable to use a data structure that allows for the new state to be added to an existing Markov chain without completely re-constructing the existing chain. Therefore, it is desirable to utilize a data structure that allows for efficient addition of a new state while also having the ability to be easily manipulated such that metrics of interest can be extracted in a timely manner.

In this paper, we present an approach for representing Markov chains that significantly reduces the memory requirements for modeling a system or process and also allows the performance of various analyses methods to improve in terms of computation time. Our experimental results indicate that the approaches described here do provide significant improvements in both computational storage requirements and performance of various algorithms for prediction of system characteristics.

II. MARKOV CHAIN AND ADD BACKGROUND

A. Markov Chains

A Markov chain is a model that can be used to represent a discrete stochastic process. The processes that Markov chains

978-1-5090-4623-2/17/$31.00 ©2017 IEEE

model are memoryless, meaning that the future status of the system is only dependent upon the system's present state and is independent of the history of previous events. Markov chains can theoretically contain an infinite set of states although for practicality, the state set is limited to a finite state space.

A common method for representing a Markov chain is the use of a directed graph, sometimes referred to as a state diagram, where the vertices represent a system state and the directed edges represent transitions among the state set. The directed edges are thus annotated with the transition probabilities from the transition matrix and each graph vertex corresponds to a transition matrix row (or column) index. The states or vertices that comprise the diagram are interconnected by transition probabilities that describe the likelihood that the system will transition from one state to another leading to the notion of a 'current state' and a 'next state,' respectively.

All of the transition probabilities that correspond to exiting a current state, $Z_0 = i$, and transitioning into one of a set of potential next states, $Z_1 = j$, must sum to unity in a discrete-time Markov stochastic process within the state space, S [6]. This relationship is described by Equation (1) below:

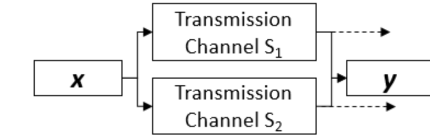
$$\sum_{j \in S} P(Z_0 = i \mid Z_1 = j) = 1, i \in S \qquad (1)$$

In a Markov chain, next-state transitions back to a current state, or self-loops, are possible and often assumed, even if not annotated on the state diagram. States that are only able to self-loop, or that are characterized by a transition probability of $P_{i,i} = 1$, are known as absorbing states. Once an absorbing state is reached, the Markov chain remains in that state indefinitely and can never leave that state. Absorbing states are useful for modeling phenomena such as a non-recoverable failure.

Due to the ability of a Markov chain to capture time-varying behavior, such as system component failure or repair, Markov chains are useful for modeling how dynamic systems evolve over time. Markov chains can be represented in numerous ways with state diagrams and transition matrices being among the most common methods. In order to produce time-sensitive metrics for a system represented with a Markov chain model, the state diagram paths can be traversed in order to determine the probability of the current state evolving through a series of states during a future interval of time. Additionally, linear algebra operations can be performed on the transition matrix in order to derive other system information or metrics of interest.

As an illustrative example, Markov chains are useful in applications involving reliability analysis. Fig. 1 models a portion of a communications network as a simplified block diagram and describes corresponding states of availability at any given instant in time.

The example in Fig. 1 was first presented in [7], and it features states that indicate a fully operational communication system, a partially operational communication system, and a communication system that fails to connect node $\boldsymbol{x}$ with node $\boldsymbol{y}$. Gradual degradation or repair causes the states to move from one extreme to another. The Markov chain model that represents the Fig. 1 communication network is illustrated in Fig. 2 while the transition matrix, **P**, comprised of the transition probability values for the system states is given in Table I.



State	Description
A_1	S_1 and S_2 both available for communication from ***x*** to ***y***
A_2	S_1 fails, but S_2 available for communication from ***x*** to ***y***
A_3	S_1 repairs underway, S_2 available for communication from ***x*** to ***y***
F_1	S_2 fails before S_1 repairs start, no available communication from ***x*** to ***y***
F_2	S_2 fails before S_1 repairs are complete, no available communication from ***x*** to ***y***

Fig. 1 Example communication network and associated states

TABLE I. TRANSITION MATRIX, **P**, OF EXAMPLE COMMUNICATION NETWORK IN FIG. 1

P =

	A1	*A2*	*A3*	*F1*	*F2*
A1	0.998	0.002	0	0	0
A2	0	0.798	0.2	0.002	0
A3	0.004	0	0.994	0	0.002
F1	0	0	0	1	0
F2	0	0	0	0	1

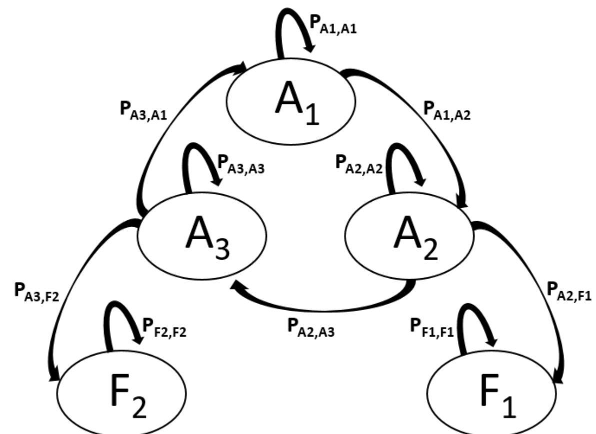


Fig.2. Markov chain state diagram of example communication network in Fig.1

The vertices of the state diagram in Fig. 2 represent the system states in Fig. 1 and the directed edges indicate the available transitions from the current state to the next state.

Within the transition matrix, the rows represent the current state of the system while the columns represent the next state. The probability of transitioning from one state, i, to another, j, can easily be determined by referencing matrix element $P(i,j)$. For example, the probability of transitioning from state A_1 to A_2 would be 0.002. It should be noted that because Markov chains represent stochastic processes, the sum of each row in the transition matrix must equal 1 as in agreement with Equation (1) above.

The transition matrix is a useful tool for predicting behavior and other metrics regarding a system such as that in the Fig. 1 communication system after multiple transitions have occurred. For example, if the system is currently in state A_1 and the probability of ending in state A_2 after 3 transitions is desired, matrix $\mathbf{P}$ must be cubed. After this computation is performed, the solution can be found in matrix element $\mathbf{P}^3(A_1,A_2)$. The point that the system reaches a steady state can also be determined by increasing the power of the transition matrix, $\mathbf{P}$, until probability values stabilize and cease to fluctuate. Examples of different methods used for calculating more detailed information about a Markov chain from the transition matrix can be found in [6].

B. Algebraic Decision Diagrams

Algebraic Decision Diagrams, or ADDs, are directed acyclic graphs based on the concepts of reduced ordered Binary Decision Diagrams, or BDDs that were introduced in [8].

A BDD represents a discrete-valued function that is dependent upon binary-valued (or switching) variables and consists of a vertex set wherein the vertices are classified as either terminal or non-terminal nodes. A non-terminal node represents a function variable and has two exiting directed edges that are each annotated or labeled with one of the possible valuations of the particular variable that is represented by the non-terminal node. The terminal nodes represent function values and have no exiting directed edges. One of the non-terminal nodes has no edges pointing to it and is designated as the initial or root node. A particular valuation of the represented function is then obtained by following a path from the initial node to a terminal node wherein the path is indicated by a set of variable valuations.

Due to the reduction rules provided in [8], BDDs can represent a switching function in a very compact manner (typically) as compared to a binary tree that contains 2^n vertices for a switching function of n variables. When the reduction rules are applied to a BDD, the total number of required vertices ranges from $O(n)$ in the best case to $O(2^n)$ in the worst case. For many functions of interest, the required number of BDD vertices is not exponential in number, allowing the BDD to be a very compact and convenient means for discrete function representation.

ADDs, like BDDs, are also directed acyclic graphs that are comprised of an initial root vertex (or node) and a set of non-terminal nodes that represent the variables of a discrete function. Also, like the BDD, ADD nodes or vertices are connected by directed edges. For BDDs, the diagrams are structured to represent binary, or two-valued, variables, and each non-terminal vertex has exactly two exiting edges representing a variable valuation of one of two possible values (usually 0 or 1). Reduction rules are likewise applicable to ADDs that result in removing redundant nodes and sharing isomorphic sub-graphs. These reduction rules allow ADDs to be very efficient data structures in terms of the required amount of memory or storage as long as the number of terminal nodes stays finite [9]. The size of the diagram can also be further reduced if a proper variable ordering for the non-terminal nodes is selected.

BDDs are further restricted to binary-valued functions as well as dependent upon binary-valued variables. Thus, BDDs have terminal nodes typically annotated with either 0 or 1. ADDs are a generalization of BDDs in that they allow for the representation of discrete functions with more than two values. Thus, ADDs provide more flexibility as compared to BDDs since they represent functions that are not limited to only two terminal node values. For example, the ADD representation of a switching function with multiple outputs could use integer-valued terminal nodes to indicate the function value in radix-10 form. As an example, a pair of binary-valued functions that depend upon the same set of binary-valued variables could be represented with a single ADD instead of two BDDs and the terminal nodes could be labeled with '0,' '1,' '2,' or '3' corresponding to the ordered pairs of binary function values '00,' '01,' '10,' or '11.' Additionally, the terminal nodes of an ADD can be real values labeled with a floating point representation that directly represent the elements within a transition matrix for a Markov chain. In this case, the non-terminal vertices of the ADD do not represent function values, rather they represent the position of a probability value within a transition matrix. Because the non-terminal vertices of an ADD have only two exiting edges, each row and column index of a transition probability matrix is represented as a binary value and each bit in the row and column index is assigned a non-terminal vertex.

Our motivation for using an ADD to represent a Markov chain transition matrix is based upon the following observations:

1) Many transition matrices of interest contain a significant degree of sparseness thereby causing the reduction rules to have a large degree of freedom resulting in a relatively small data structure.
2) In an automated environment where new system of process states may be iteratively discovered, it is relatively easy to add a new state to an existing ADD representing a transition matrix without reformulating the entire structure.
3) Extracting a particular probability is accomplished through a single traversal of one path within the ADD from the initial to the terminal node.

4) Extracting an *m*-step probability that corresponds to *m* transitions of the Markov chain is accomplished through *m* single path traversals of the ADD representing the transition matrix.
5) Other Markov chain computations of interest are implemented as directed graph algorithms over the ADD wherein the ADD is often a very compact structure.

As an illustrative example, consider the Markov chain represented by the state transition diagram in Fig. 3 and the probability transition matrix in Equation (2).

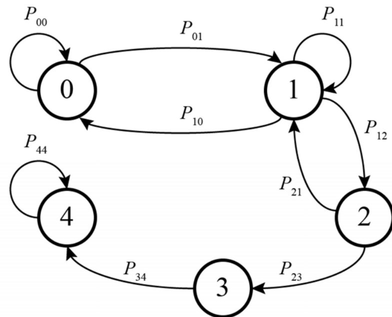


Fig.3. Markov chain state diagram of example

$$\mathbf{P} = \begin{bmatrix} P_{00} & P_{01} & 0 & 0 & 0 \\ P_{10} & P_{11} & P_{12} & 0 & 0 \\ 0 & P_{21} & 0 & P_{23} & 0 \\ 0 & 0 & 0 & 0 & P_{34} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \qquad (2)$$

Representing a matrix with an ADD requires that the row and column indices of the matrix are in the form of binary identifiers that can be represented as variables with ADD non-terminal nodes. Then, graph traversal can be accomplished by traversing a path of the row and column index values in order to retrieve the matrix element represented by a terminal node. When the matrix has some degree of sparseness or repetition in the elements it contains, the reduction rules offered by ADDs allow the matrix to be represented in a compact form requiring a significantly reduced amount of memory as compared to other more common data structures for sparse matrix representations.

To illustrate the compactness, consider the Markov chain represented in Fig. 3 and by Equation (2). The row indices of the matrix have values ranging from zero to four with the topmost row corresponding to zero and the bottommost corresponding to four. Because the ADD utilizes binary-valued non-terminal vertices, the row index variables are expressed in binary as 000_2, 001_2, 010_2, 011_2, and 100_2 from the top to the bottom row. We utilize a subscripted value to indicate that the base or radix of the values is two (binary). The ADD variable representing the row index values are denoted as the triplet (a,b,c) where a is the most significant bit and c is the least significant bit. As an example, the index value for row 3 (*i.e.*, the fourth row from the top of the transition matrix) is $abc=011_2$. Likewise, the column indices increase in value from left to right, range from 000_2 through 100_2, and are represented by the triplet of variables (d,e,f). Fig. 4 contains a graphical illustration of the ADD representing the example Markov chain that corresponds to the state transition diagram in Fig. 3 and the probability transition matrix of Equation (2).

It should be noted that a particular ADD corresponds to a particular variable order. While the particular variable order is irrelevant regarding the Markov chain that is represented, certain variable orders allow for the reduction rules to be more effective than others. In the example ADD shown in Fig. 4, the variable order $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$ is used and may not necessarily result in the absolute minimally-sized ADD. It is also the case that some of the non-zero transition probabilities, represented symbolically as P_{ij} in Fig. 4, may have equivalent numeric values. In the case where multiple P_{ij} have the same

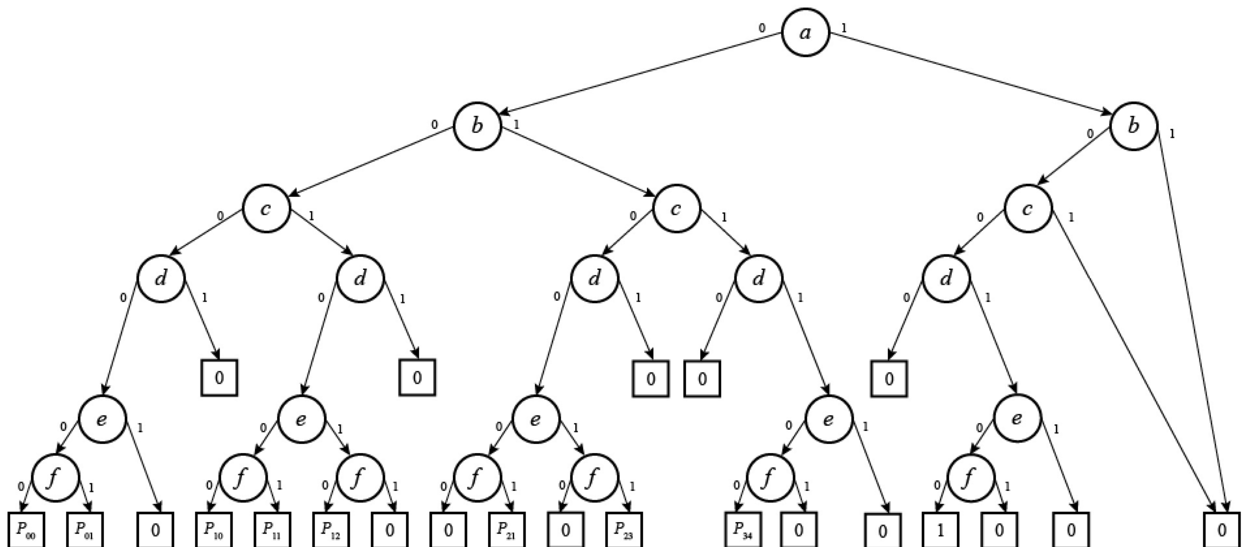


Fig.4. ADD representation of example Markov chain

value, additional reduction in the size of the ADD will result.

An ADD representation is more compact when the Markov chain transition probability matrix it represents is sparse. Many Markov chains do have sparse, or at least banded, transition probability matrices thus enhancing the compactness of the ADD data structure.

III. MC Prototype Analysis Tool

A. Data Structure Considerations

Markov chain reliability analysis, when performed using an explicitly represented transition matrix, becomes costly both in terms of required space and computation time. This is especially true whenever the system state space becomes extremely large in size. As the number of states in the Markov chain increases, the corresponding transition matrix grows exponentially in size. While using the transition matrix to evaluate a system may be feasible for a state-space less than 1000 states, an efficient method of storing more massive Markov chains for analysis is desired. Our results indicate that storage of the transition matrix information within an ADD data structure is advantageous. Furthermore, in a system where new states are found or discovered in an iterative fashion, the ADD is likewise advantageous due to the existence of specialized ADD algorithms for vertex insertion, deletion, and translation.

The Markov chain representation of the example communication network, as depicted in Fig. 2, can easily be transformed into an ADD. First, each of the states must be assigned a binary identifier. Many efficient algorithms exist for this process in the form of state encoding techniques that were initially developed in the field of digital circuit design automation algorithms. Since five total states exist, each binary identifier must be at least three bits long. The number of bits needed for each state's binary identifier for a total amount of states, S, in a state-space can be found with Equation (3):

$$ID_{length} = \lceil \log_2(S) \rceil \quad (3)$$

The IDs for the states in this example are the following:

- A_1 => 000_2
- A_2 => 001_2
- A_3 => 010_2
- F_1 => 011_2
- F_2 => 100_2

The length of a string of bits representing a transition from present state to next state is equal to twice the ID_{length}. To create a bit stream that represents a transition, the present state ID and the next state ID should be concatenated. For example, the bit stream indicating a transition from state A_1 to A_2 would be 000001_2. Once binary identifiers have been assigned, variables for the ADD nodes must be determined. In this example, X1, X2, X3 will hold the present state bits while X4, X5, X6 will hold the next state bits. The resulting ADD for the communication system from Fig. 1 is pictured in Fig. 5.

In Fig. 5, the zero terminal node and paths leading to it have been omitted. Transition probabilities are obtained by traversing the paths in the diagram using the binary values of the row and column indices. Using the ADD, only seven floating point terminal nodes must be saved in memory for the Markov chain rather than the 25 elements that would be required if the entire transition matrix was explicitly stored.

It is common to occasionally optimize the ADD through use of various procedures that permute the non-terminal ADD vertex orders in an attempt to minimize the ADD. Certain vertex or ADD variable orders cause the reduction rules to be applied more effectively which, in turn, allow the ADD to be represented with fewer non-terminal vertices. This is incorporated into the Markov chain analysis algorithms by permuting the bitstrings that represent the state transitions in accordance with the current vertex permutations before a path traversal is executed.

As the number of distinct states in a Markov chain increases, the sparsity of the corresponding transition matrix also usually tends to increase due to more zero-valued transition probabilities being present in the represented transition matrix. Additionally, the transition matrices tend to be banded, as can be seen with the example matrix in Table I. These two characteristics help to further reduce the size of the ADD representation of the transition matrix. Additional reduction methods can also be put into place such as limiting the number of terminal nodes. For example, if a reliability analysis only requires a resolution of 0.005 for the probability values, a maximum of only 200 terminal nodes is required for the Markov chain ADD. Transition probabilities from the original transition matrix could be rounded to the nearest 0.005, and in a 1,000+ state system, the resulting directed graph would only require a fraction of the memory of the original full matrix when the probability resolution is arbitrary.

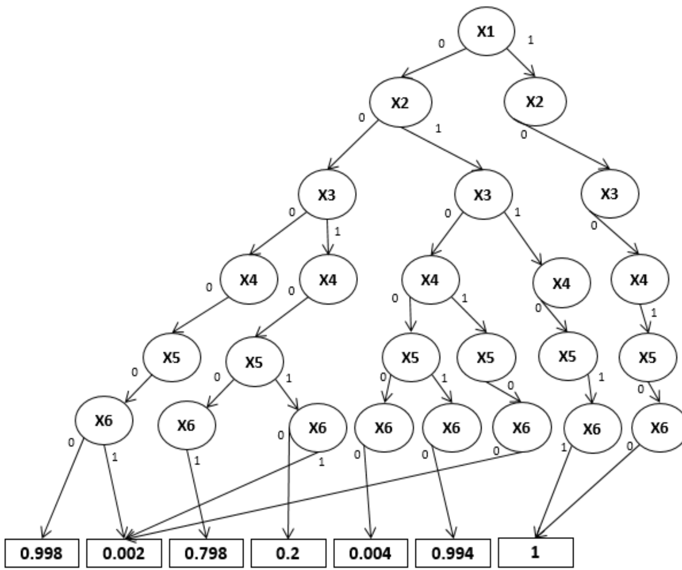


Fig.5. ADD representation of Fig. 1 communication network

B. Markov Chain Analysis Methods in the Prototype Tool

After determining that the ADD is a desirable structure for storing and representing large Markov chains, an analysis library was developed. We identified and implemented 13 different metrics that are computed by our prototype system. These 13 metrics were deemed critical for ascertaining important characteristics of a Markov chain as well as evaluating predictions based upon the Markov chain model. High-performance algorithms for the following calculations were implemented in Python:

1) Probability distribution at convergence

2) Probability distribution after a set number of transitions

3) Probability of a state after a set number of transitions from any starting state

4) Probability of a state after a set number of transitions from a specific starting state

5) Transitions to convergence

6) Transitions required for state probability to reach a threshold percentage

7) Transitions until state probability changes by given percentage

8) Reachability between two states

9) Percentage of reachable states from specific starting state

10) Percentage of reachability between all states

11) Expected number of transitions until each absorbing state is reached

12) Probability of a transient start state being absorbed after a set number of transitions

13) Probability of an absorption state being reached after a set number of transitions from any transient state

These 13 analysis options, coupled with an efficient storage and submatrix extraction system resulted in a high-performance dynamic system analysis tool that performs rapid calculations while also minimizing the amount of storage required for very large Markov chains. This system is amenable to both a human-centered interface where a graphical user interface could be implemented as a front-end data entry mechanism, or it could be used in an automated setting where Markov chain states are iteratively discovered and provided to the ADD engine for the purpose of updating the internal ADD data structure.

IV. EXPERIMENTAL RESULTS

Our implementation was evaluated by using randomly generated right stochastic square matrices ranging in dimension from 5×5 to 10000×10000 in size. Each matrix size consisted of one sparse, one banded, and one dense matrix. Creation of the matrices was performed utilizing a pseudo-random number generator for determining position, count, and value of each of non-zero element. For each matrix, the sum of each row was determined and all elements in the corresponding row were divided by the sum to ensure the matrix was right stochastic. Rows in sparse and banded matrices were generated with a random number of non-zero elements at or below the designated level of sparseness. Non-zero elements in the banded matrices radiate out from the corresponding diagonal value while sparse matrices randomly assign each non-zero value a column position.

Markov chain models created in practice are generally sparse in that each state only has the possibility of transitioning to a small number of other states. Traditionally, matrices that are expected to exhibit sparse connectivity are represented in ways that take advantage of the implicit zero values found within them. One of the most common methods of sparse matrix representation is the row-compressed sparse matrix format. Each row is iterated through in order from top to bottom where every column is then checked in order from left to right. Each time a non-zero value is encountered the current column position is added to an indexing vector and the corresponding value is added to the same position in a separate vector of values. Additionally, each time the first non-zero value is found in a row the vector index position of the column and value is added to a row-indexing vector. The result is that three vectors that can accurately reproduce a sparse matrix without the need to explicitly store zero values or redundant row index values. Given that such data structures are the standard method of storage for sparse matrices, testing was conducted with ADDs and row-compressed sparse matrices.

Row-compressed sparse matrices were tested with the **`NumPy`** and **`SciPy`** Python scientific computation packages. The two packages work together to perform linear algebra computations that are capable of directly handling sparse-matrix formatted data. Basic Linear Algebra Subprograms

(BLAS) and Linear Algebra Package (LAPACK) provided the low level *C* and Fortran functions that **`NumPy`** and **`SciPy`** utilize for quick and efficient computation. The popularity of the Python packages along with their highly reputable underlying *C* and Fortran packages were chosen in an attempt to ensure that the sparse matrix formatted data was tested with current industry standards in terms of linear algebra algorithms.

The algebraic decision diagrams used in our prototype tool were implemented with the University of Colorado Decision Diagram Package (CUDD) [10]. CUDD is a high-performance decision diagram package written in *C* that is capable of building, manipulating, and performing computations with various decision diagrams, including ADDs. The package contains the necessary basic functions to read in a sparse matrix in the form of row and column coordinates and the corresponding element value. The sparse matrix data is used to iteratively construct an algebraic decision diagram. The provided function was designed to process any general form of matrix and as such was altered to more efficiently build ADDs that are restricted to the right stochastic and square transition matrices. Additionally, using the row-compressed sparse matrix format allowed for additional increased efficiency since accounting for coordinate pair input in any possible order was no longer necessary.

Initial testing indicated that the reduction benefits of using ADDs were diminished for the case where the transition matrix elements collectively consisted of many different and unique values and hence caused a large number of ADD terminal vertices to be present. In constructing the ADD, we restricted the terminal nodes to represent a finite number of integers that represent equally sized intervals over the range [0, 1]. The intervals were found to contain comparable numbers of values both above and below the median value of the interval. This relationship allowed the median of each interval to be used for computation with minimal sacrifice in the resulting output resolution of our computational results. The utilization of interval terminals rather than unique floating-point terminals was found to result in a much more efficient data structure without sacrificing accuracy, as shown in Table II. Table II contains the amount of storage required to represent a 1000×1000 (*i.e.*, 1000 Markov chain states) Markov chain transition probability matrix for sparse, banded, and dense transition matrices.

TABLE II. FLOATING POINT TERMINALS VS. INTERVAL TERMINALS COMPARISON (1000×1000)

	Floating Point Terminals (Bytes)	10 Interval Terminals (Bytes)	Size Reduction	Similarity After Squaring
Sparse	1099064	280456	74.48%	99.55%
Banded	2095032	109160	94.79%	99.49%
Dense	27785976	400968	98.56%	99.98%

Table III shows the comparison between the memory required (bytes) to build each test transition matrix using the **`NumPy`** and **`SciPy`** structures versus the CUDD ADDs. Table IV provides details about the matrix build times for the traditional matrix representations versus ADD data structures.

TABLE III. BUILD SIZE COMPARISON (BYTES) BETWEEN NUMPY/SCIPY MATRICIES AND CUDD ADDS

	Sparse Matrices	
	NumPy and SciPy	CUDD (10 Interval Terminals)
5x5	204	2376
50x50	2592	15208
100x100	5168	31336
1000x1000	51524	280456
10000x10000	634064	3423048
	Banded Matrices	
	NumPy and SciPy	CUDD (10 Interval Terminals)
5x5	180	2312
50x50	2580	13768
100x100	6332	26920
1000x1000	596804	109160
10000x10000	59293004	1294632
	Dense Matrices	
	NumPy and SciPy	CUDD (10 Interval Terminals)
5x5	200	2376
50x50	20000	4936
100x100	80000	12136
1000x1000	8000000	400968
10000x10000	800000000	24526408

TABLE IV. BUILD TIME COMPARISON BETWEEN NUMPY/SCIPY MATRICIES AND CUDD ADDS

	Sparse Matrices	
	NumPy and SciPy	CUDD (10 Interval Terminals)
5x5	< 1	< 1
50x50	< 1	< 1
100x100	< 1	< 1
1000x1000	6	30
10000x10000	70	1080
	Banded Matrices	
	NumPy and SciPy	CUDD (10 Interval Terminals)
5x5	< 1	< 1
50x50	< 1	< 1
100x100	< 1	< 1
1000x1000	67	460
10000x10000	6974	91270
	Dense Matrices	
	NumPy and SciPy	CUDD (10 Interval Terminals)
5x5	< 1	< 1
50x50	3	< 1
100x100	13	30
1000x1000	1175	8510
10000x10000	222829	2330620

Experimentation shows the savings in memory that occurs when large Markov chains are stored as interval-terminal ADDs. If fewer significant digits are needed to represent the transition probability values, less ADD terminal nodes are necessary and a more compact ADD structure results. Table V provides details about the memory requirements of Markov chains and their corresponding ADD representations with varying levels of precision. The Markov chains in Table V are all banded transition matrices ranging from 100 to 10,000 states. The ADD precision levels in the table include full floating point precision (with no limit on the number of terminal nodes), one significant digit of precision, two significant digits of precision, and three significant digits of precision. Although CUDD functions exist for performing matrix computations on ADDs, such as multiplication and squaring operations, these functions were not found to result in the highest performance in terms of required computation time.

TABLE V. ADD MEMORY REQUIREMENTS VS PRECISION FOR 100, 1000, AND 10,000 STATE BANDED TRANSITION MATRIX

		100 States Banded	1000 States Banded	10000 States Banded
Full Precision	Floating Point	39864, 1223, 1266	2095032, 65447, 65508	107146008, 3348290, 3348375
One Sig. Fig.	10 Term. Inter.	26920, 816, 858	1738848, 3386, 3445	8038432, 40432, 40523
Two Sig. Fig.	100 Term. Inter.	39576, 1189, 1232	3627680, 20600, 20600	8038432, 40432, 40523
Three Sig. Fig.	1000 Term. Inter.	55736, 1469, 1512	5674304, 35542, 35602	62972288, 837580, 837664

DATA = (bytes, nodes, peak nodes)

Our experimentation indicated that timing performance during the linear algebra computations was best when submatrix blocks are extracted from the ADD as needed during the computations and **`NumPy`** and **`SciPy`** are used for the actual computation. This process shortens computation time while maintaining the memory benefits of the transition matrix information being saved as an ADD. Based on this experimentation, we chose to use the ADD to represent the transition matrices, but to use **`NumPy`** and **`SciPy`** with submatrix extraction for the computations rather than implementing the computations as graph algorithms that directly process the ADDs. However, we did also implement the computations as ADD-based graph algorithms as well in our testing and evaluation process. Timing information for the 13 Markov analysis algorithms as described in Section III B is provided in Table VI.

TABLE VI. TIMING DATA FOR MARKOV CALCULATIONS USING 10, 100, 1000, AND 10,000 STATE TRANSITION MATRICIES (ms)

	10 States		100 States		1000 States		10000 States	
	5 Iters	10 Iters	5 Iters	10 Iters	5 Iters	10 Iters	5 Iters	10 Iters
Test 1[a]	6.14		7.96		277.76		85058.56	
Test 2	5.11	5.15	5.97	6.25	49.81	58.13	28692.09	43639.91
Test 3	5.11	5.36	6.05	6.24	46.32	56.29	29577.83	40276.04
Test 4	5.12	5.21	5.91	5.95	47.28	60.14	27160.34	37067.94
Test 5[a]	9.11		44.65		4631.79		1266425.46	
Test 6[b]	7.89		30.97		1600.16		129846.39	
Test 7[c]	8.07		32.21		1559.09		131800.06	
Test 8[d]	0.04		5.97		149.06		23294.28	
Test 9	5.54		7.35		274.18		86666.18	
Test 10	0.12		1.12		573.87		86414.47	
Test 11[e]	5.58		13.86		69.19		16080.84	
Test 12	5.05	5.14	6.15	6.29	51.89	64.75	28641.25	28641.25
Test 13	5.12	5.17	6.42	6.53	48.87	60.75	27510.52	37607.48

Notes:

a *convergence, no set number of iterations*

b *longer of the two times between rise above 50% and fall below 50%*

c *using 10%*

d *2 random states chosen, stops at states being reachable or convergence of system*

e *1 random state is converted to absorbing state*

In Table VI, measurement readings for calculation duration are provided in units of milliseconds. Transition matrices representing 10, 100, 1000, and 10,000 state Markov chains are used with the 13 analysis options to generate the reported timing data.

V. Conclusion

Representing a Markov chain as an ADD proves to be an efficient alternative as opposed to storing information in either a traditional transition matrix that is large and dense or other more efficient and common data structures for storing matrices. This method has been shown to outperform more common data structures for representing matrices, including both dense and banded matrices. Due to the space-saving characteristic of ADDs, it is possible to store extremely large Markov chains in a much more compact manner. Also, very importantly for our application, use of the ADD allows states to be added to an existing Markov chain in a very efficient manner that avoids the reconstruction of the entire structure.

In conclusion, we have developed a high-performance analysis tool that is implemented by extracting submatrices from an ADD structure and using the basic linear algebraic operators from the **`NumPy`** and **`SciPy`** libraries to implement 13 different analysis options over a Markov chain model. Coupling the spatial savings provided by ADDs with the efficient linear algebraic operations of **`NumPy`** and **`SciPy`** resulted in an automated analysis prototype tool that can efficiently store and process extremely large Markov chains while also providing the capability to efficiently add newly discovered states to an existing chain.

References

[1] M. Ammar, K. A. Hoque and O. A. Mohamed, "Formal analysis of fault tree using probabilistic model checking: A solar array case study," 2016 Annual IEEE Systems Conference (SysCon), Orlando, FL, USA, 2016, pp. 1-6.

[2] Graham, C. Markov Chains: Analytic and Monte Carlo Computations. Somerset: Wiley, 2014.

[3] M. K. Ishak, G. Herrmann and M. Pearson, "Performance evaluation using Markov model for a novel approach in Ethernet based embedded networked control communication," 2016 Annual IEEE Systems Conference (SysCon), Orlando, FL, 2016, pp. 1-7.

[4] J. Leithon, T. J. Lim and S. Sun, "Renewable energy management in cellular networks: An online strategy based on ARIMA forecasting and a Markov chain model," 2016 IEEE Wireless Communications and Networking Conference, Doha, 2016, pp. 1-6.

[5] M. Rahnamay-Naeini and M. M. Hayat, "Cascading Failures in Interdependent Infrastructures: An Interdependent Markov-Chain Approach," in IEEE Transactions on Smart Grid, vol. 7, no. 4, pp. 1997-2006, July 2016.

[6] N. Privault, *Understanding Markov Chains: Examples and Applications*, Singapore: Springer Singapore, 2013

[7] J. B. DeMercado, "Reliability Prediction Studies of Complex Systems Having Many Failed States," in *IEEE Trans. on Reliability*, vol. R-20, no. 4 pp. 223-230, Nov. 1971

[8] R. E. Bryant, "Graph-Based Algorithms for Boolean Function Manipulation," *IEEE Trans. on Computers*, vol. C-35, no. 8, pp. 677-691, Aug. 1986

[9] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, F. Somenzi, "Algebraic Decision Diagrams and Their Applications," in *IEEE/ACM Int. Conf. on CAD*, 1993, pp. 188-191.

[10] F. Somenzi, "CUDD: CU Decision Diagram Package Release 3.0.0," Department of Electrical, Computer, and Energy Engineering, University of Colorado at Boulder, Dec. 2015.