Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Information Processing Letters 87 (2003) 309–315

Information Processing Letters

www.elsevier.com/locate/ipl

Optimal state-space lumping in Markov chains ☆

Salem Derisavi [a,*], Holger Hermanns [b,c], William H. Sanders [a]

[a] *University of Illinois at Urbana-Champaign, Coordinated Science Laboratory, Urbana, IL 61801, USA*
[b] *University of Twente, Faculty of Computer Science, 7500 AE Enschede, The Netherlands*
[c] *Universität des Saarlandes, FR 6.2 Informatik, D-66123 Saarbrücken, Germany*

Received 16 August 2002; received in revised form 18 April 2003

Communicated by H. Ganzinger

Abstract

We prove that the optimal lumping quotient of a finite Markov chain can be constructed in $O(m \lg n)$ time, where n is the number of states and m is the number of transitions. Our proof relies on the use of splay trees (designed by Sleator and Tarjan [J. ACM 32 (3) (1985) 652–686]) to sort transition weights.

Keywords: Bisimulation; Computational complexity; Lumpability; Markov chains; Splay trees

1. Introduction

Markov chains are among the most fundamental mathematical structures used for performance and dependability modeling of communication and computer systems. As the size of a Markov chain usually grows exponentially with the size of the corresponding high-level model, one often encounters the infamous state-space explosion problem, which often makes solution of the Markov chain intractable and sometimes makes it impossible. Many approaches to alleviate or circumvent this problem are implicitly or explicitly based on the notion of *lumpability* [13], which allows computation of performance and dependability measures on the quotient of the Markov chains under lumping-equivalence.

In this paper, we study the complexity of constructing the optimal (i.e., coarsest) lumping quotient of a given Markov chain. In [4], Buchholz gives an $O(mn)$ algorithm for this problem, where n is the number of states and m is the number of transitions. We achieve $O(m \lg n)$ time complexity. The algorithm is based on the Paige/Tarjan algorithm for computing bisimilarity on labeled transition systems [16].

The $O(m \lg n)$ result for Markov chains is apparently known, since the result has been claimed by var-

☆ This material is based upon work supported by the National Science Foundation under Grant Nos. 9975019 and 0086096, by the Motorola Center for High-Availability System Validation at the University of Illinois (under the umbrella of the Motorola Communications Center), and by the Netherlands Organization for Scientific Research through a *Vernieuwingsimpuls* award. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

* Corresponding author.
E-mail addresses: derisavi@crhc.uiuc.edu (S. Derisavi), hermanns@cs.utwente.nl (H. Hermanns), whs@crhc.uiuc.edu (W.H. Sanders).

0020-0190/$ – see front matter
doi:10.1016/S0020-0190(03)00343-0

ious authors [2,7,11]. None of their publications, however, contain a proof of the $O(m \lg n)$ time complexity, or a sufficiently detailed description of the algorithm to make the claim obviously true. We show that a naïve algorithm has a time complexity of $O(m \lg^2 n)$, and we do not know how to rigorously prove the claimed better bound for the naïve algorithm. We believe that such a proof would require a clever way for sorting weights that has not yet been devised, to the best of our knowledge. By using statically optimal trees (e.g., splay trees) [18] for the sort operation, we are able to attain the $O(m \lg n)$ time bound. The space complexity is $O(m + n)$.

2. Background

In the remainder of this paper we focus on continuous-time Markov chains (CTMCs), although the given definitions, algorithms, and analyses are easily extended to the case of discrete-time Markov chains.

We consider a finite CTMC $(\mathcal{S}, Q)$ with state space $\mathcal{S} = \{x_1, x_2, \ldots, x_n\}$ by a transition rate matrix $Q : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}^+$, where $\mathbb{R}^+$ is the set of positive reals. We define a relation $\to \subset \mathcal{S} \times \mathcal{S}$ where $x_i \to x_j$ iff $Q(x_i, x_j) > 0$. We use m to denote the cardinality of $\to$, i.e., the number of nonzero transition rates of the CTMC. We further define a cumulative function $q : \mathcal{S} \times 2^{\mathcal{S}} \mapsto \mathbb{R}^+$ by

$$q(x_i, X) = \sum_{x \in X} Q(x_i, x) \quad \text{for } X \subseteq \mathcal{S}, \tag{1}$$

and define the forward and backward images $e, e^{-1} : \mathcal{S} \mapsto 2^{\mathcal{S}}$ by $e(x_i) = \{x_j \mid x_i \to x_j\}$, and $e^{-1}(x_i) = \{x_j \mid x_j \to x_i\}$.

Now consider a partition $\mathcal{Y} = \{Y_1, \ldots, Y_{n'}\}$ of $\mathcal{S}$ (typically $n' \ll n$). $Y_1, \ldots, Y_{n'}$ are called the *blocks* of partition $\mathcal{Y}$. Let $[x]_{\mathcal{Y}}$ denote the (unique) block $Y \in \mathcal{Y}$ containing $x \in \mathcal{S}$. Each partition induces a quotient stochastic process on the state space $\mathcal{Y}$, but not all partitions $\mathcal{Y}$ result in a process with the Markov property. Following [19], we define ordinary lumping as:

Definition 1 [19]. A CTMC $(\mathcal{S}, Q)$ is ordinarily lumpable with respect to a partition $\mathcal{Y}$ of $\mathcal{S}$ iff for any blocks $Y_i, Y_j \in \mathcal{Y}$ and for every $x, x' \in Y_i$ we have $q(x, Y_j) = q(x', Y_j)$. In such a case, the quotient Markov chain $(\mathcal{Y}, Q')$ is determined by $Q'([x]_{\mathcal{Y}}, Y) = q(x, Y)$ for any $Y \in \mathcal{Y}$.

We refer to [3] for theorems on how the state probabilities of the original chain are related to those of the quotient chain. The partitions of $\mathcal{S}$ satisfying the condition above form a complete lattice, in which the trivial finest partition $\{\{x_i\} \mid x_i \in \mathcal{S}\}$ is the minimal element, and the coarsest partition, corresponding to the smallest possible lumped quotient, is the maximal element. This lattice is akin to the lattice induced on a labeled transition system by the notion of bisimulation [8,15], where bisimilarity is the maximal element in the lattice of bisimulations. Like bisimilarity, the coarsest lumping partition can be characterized as a fixpoint of successively finer partitions; this characterization is the basis of the algorithmic procedure we will discuss. For bisimilarity, Kanellakis and Smolka have given such a partition refinement algorithm with time complexity $O(mn)$ [12]. They conjecture that an algorithm exists that reduces the time complexity to $O(m \lg n)$. A few years later Paige and Tarjan designed such an algorithm [16].

3. Algorithm description

In this section, we describe our algorithm for coarsest ordinary lumping. Like other algorithms that claim the $O(m \lg n)$ time complexity, our algorithm is based on Paige and Tarjan's algorithm to compute bisimilarity of labeled transition systems. The pseudocode of the algorithm is given in Algorithm 1. For a given CTMC $(\mathcal{S}, Q)$, LUMPCTMC takes as parameters the initial partition P of the state space $\mathcal{S}$ and the transition rate matrix Q, and returns the transition rate matrix Q' of the smallest ordinarily lumped quotient $(\mathcal{Y}, Q')$ such that $\mathcal{Y}$ is a refinement of P.

The algorithm comprises three phases. Line 1 does the necessary initializations; it assigns to L the blocks of the initial partition P of CTMC states. The role of L will become clear below. Lines 2–4 constitute the main step of the algorithm, in which partition P is refined to the coarsest ordinary lumping partition. Finally, lines 5–12 build the quotient Markov chain with respect to the final partition P; they construct the lumped CTMC with $n' = |P|$ states $\{B_1, \ldots, B_{|P|}\}$. Q' is built according to the equation given in Definition 1.

```
LUMPCTMC(P, Q)
 1  L := blocks of P
 2  while L ≠ ∅
 3     S := POP(L)
 4     SPLIT(S, P, L)
 5  n' := # of blocks in P
 6  allocate n' × n' matrix Q'
 7  initialize Q' to zero
 8  for every block B_k of P
 9     x_i := arbitrary state in B_k
10     for every x_j such that x_i → x_j
11        Let B_l be [x_j]_P
12        Q'(B_k, B_l) := Q'(B_k, B_l) + Q(x_i, x_j)
13  return Q'
```

Algorithm 1. Pseudocode of the lumping algorithm.

$Q'(B_k, B_l)$, the rate from B_k to B_l in the quotient chain, is $q(x_i, B_l) = \sum_{x_j \in B_l} Q(x_i, x_j)$, where x_i is an *arbitrary* element of B_k.[1]

We assume that the initial partition P is given by the single block $\{\mathcal{S}\}$, but finer initial partitions are also possible. One example of using a non-trivial initial partition is the scenario in which reward structures are defined on states of the Markov chain, in which case the model is said to be a Markov reward model [10]. For such a model, all states with the same reward are put in the same block in the initial partition. Since the elements of the blocks of a partition in this algorithm are actually states of a CTMC, we will use the words *state* and *element* interchangeably. The proof of the correctness of the algorithm is not given in this paper, but can be adapted from [6].

P may be *refined* in each iteration of the while loop in Algorithm 1. Refining is a step in which at least one of the blocks of P is partitioned into at least two (smaller) blocks. L plays the role of a list of "potential" *splitters* of P. A splitter S with respect to a partition P is a set that satisfies the following condition:

$$\exists B \in P,\ x, y \in B \quad \text{s.t.} \quad q(x, S) \neq q(y, S), \tag{2}$$

which means that the condition in Definition 1 is violated by the pair x, y and hence B needs to be split. SPLIT(S, P, L), given in Algorithm 2, performs this

```
SPLIT(S, P, L)
 1  L', L'' := ∅
 2  for every x_j ∈ S
 3     for every x_i → x_j
 4        x_i.sum := 0
 5  for every x_j ∈ S
 6     for every x_i → x_j
 7        x_i.sum := x_i.sum + Q(x_i, x_j)
 8        L' := L' ∪ {x_i}
 9  for each x_i ∈ L'
10     B := block of x_i
11     delete x_i from B
12     INSERT(B_T, x_i)
13     if B ∉ L'' add B to L''
14  for every B ∈ L''
15     B_l := largest block of {B, V_{k_1}, ..., V_{k_|B_T|}}
16     L := L ∪ {B, V_{k_1}, ..., V_{k_|B_T|}} − {B_l}
```

Algorithm 2. Pseudocode of SPLIT procedure.

crucial refinement step. It refines P with respect to splitter S, and may also add a number of new potential splitters to L. In other words, SPLIT finds every block B of P that satisfies condition (2), and splits each such block B into sub-blocks $\{B_1, \ldots, B_k\}$ such that

$$\forall 1 \leqslant i \leqslant k,\ \forall x, y \in B_i, \quad q(x, S) = q(y, S),$$

$$\forall 1 \leqslant i \neq j \leqslant k,\ \forall x \in B_i, y \in B_j, \quad q(x, S) \neq q(y, S),$$

and adds all B_i's except the largest one to L.[2] If L is empty, it means that the partition is refined with respect to all potential splitters, and the while loop in Algorithm 1 finishes.

Algorithm 2 shows the pseudocode for the refinement operation. Line 1 initializes L' and L''. L' stores the set of states that have at least one transition to any of the states in S (i.e., $L' = \bigcup_{x \in S} e^{-1}(x)$). L'' is the list of all blocks of P that are partitioned in a refinement step. Lines 2–4 initialize the cumulative function q to zero. Lines 5–8 compute $q(., S)$ as defined in Eq. (1).

Lines 9–13 split each block B based on the values of the cumulative function of elements in B. Each block B has a corresponding binary search tree (BST) B_T, which we call the *sub-block tree*. Each node of

[1] x_i can be *arbitrary* because the CTMC is ordinarily lumpable with respect to P.

[2] If there is more than one block of maximal size, we add all of them except for one.

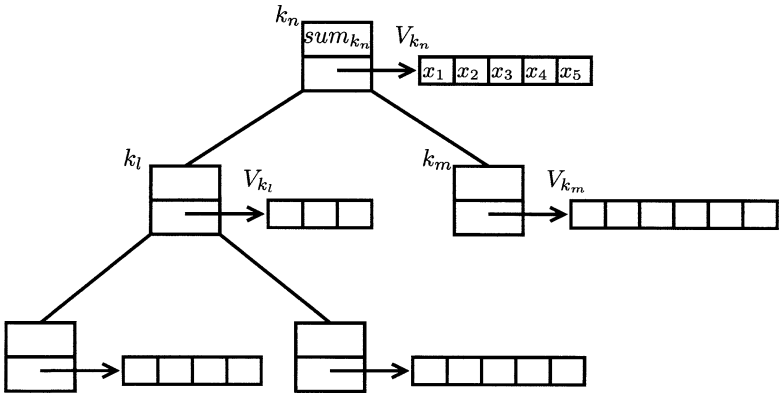

Fig. 1. The sub-block tree B_T associated with each block B.

B_T stores the list of elements of B that have the same value of $q(., S)$. Any implementation of a BST data structure (e.g., the red-black tree or the AVL tree) can be used as the sub-block tree. In Section 4, we will see how exploiting the full potential of an appropriate BST helps us achieve the $O(m \lg n)$ running time for LUMPCTMC.

The set of nodes of B_T is denoted by $\{k_1, \ldots, k_{|B_T|}\}$. Each node k_j of B_T contains a key sum_{k_j} and also a set V_{k_j} (k_j's *V-set*) of states, each of which has the same total outgoing rate to states in S (Fig. 1). Accordingly, the set of V-sets of B_T's nodes is denoted by $\{V_{k_1}, \ldots, V_{k_{|B_T|}}\}$. For any node k_j and any $x_i \in V_{k_j}$, we have $\text{sum}_{k_j} = x_i.\text{sum} = q(x_i, S)$. INSERT$(B_T, x_i)$ in line 12 adds x_i to the V_{k_j} for which $k_j.\text{sum} = x_i.\text{sum}$.

If there is no transition from x_i to any state in S, we will have $x_i.\text{sum} = 0$; x_i will be neither inserted into B_T nor removed from B. All such elements (if any) will remain in B, and, along with the V-sets of B_T, constitute the sub-blocks of the original block B. Lines 14–16 update the list of potential splitters L by adding to L all the sub-blocks of a block except for one of the largest one(s). Excluding the largest sub-block from the set of potential splitters is similar to the "process the smaller half" strategy given by Hopcroft [1,9]. The difference is that in Hopcroft's strategy a set B is partitioned into two subsets, while in our case B can be partitioned into more than two subsets. The similarity is that in both approaches the largest subset is not processed, ensuring that for each subset $A \subset B$ to be processed, $|A| \leqslant |B|/2$. We can neglect the largest block (or any other single block), because its power of splitting other blocks is maintained by the remaining sub-blocks. For an example, see Fig. 1. V_{k_m} is the largest V-set (having 6 states); therefore, whichever is larger, V_{k_m} or the set of remaining elements of B, will be the only sub-block that is excluded from the set of new potential splitters that are added to L in line 16.

4. Time complexity

In this section, we will prove that when we use splay trees [18] as the sub-block trees, the refinement step of LUMPCTMC (lines 1–4 in Algorithm 1) runs in $O(m \lg n)$ time. In order to show how the properties of splay trees (compared to other balanced BSTs) affect the running time of the algorithm, we are going to analyze the running time of LUMPCTMC in two configurations. In the first configuration, we use a general balanced BST for the sub-block trees. We assume that insertion and query of a node in a general balanced BST take $O(\lg k)$ time in the worst case, where k is the number of nodes in the tree. For this configuration, we will show that the worst case running time of the algorithm is $O(m \lg^2 n)$. In the second configuration, we use a splay tree for the sub-block trees. Using the *static optimality* property of splay trees, we show that we can take an $O(\lg n)$ factor out of the time complexity we obtained when we used a general balanced BST; in other words, we achieve the $O(m \lg n)$ running time. As our algorithm is a variant of the one proposed by Paige and Tarjan, both

of our running time proofs are based on the running time proof given in [16].

Except for the tree data structure used for the sub-block tree, all of the data structures are common between the two configurations. Like a graph, the CTMC is stored using an adjacency list data structure. Each vertex (state x) has two associated linked lists: one for successor vertices (given by $e(x)$) and one for predecessor vertices (given by $e^{-1}(x)$).[3] The transition rates in Q are stored as edge weights in the adjacency list data structure. A partition is stored as a doubly-linked list whose elements are the blocks. Each block points to a doubly-linked list whose elements are the elements of the block. Each element (state) has a pointer to its block. Therefore, finding the block of an element, or adding or deleting a block or an element of a block, takes constant time. The sets L, L', and L'' are also stored as doubly-linked lists.

4.1. Using general balanced BSTs

In the balanced BST configuration, we choose an arbitrary balanced BST as the sub-block tree. We assume that the selected data structure provides insertion and query in $O(\lg k)$ time in the worst case.

Lemma 1. *If general balanced BSTs are used,* SPLIT(S, P, L) *takes*

$$O\left(|S| + \sum_{x\in S} \left|e^{-1}(x)\right|(1+\lg n)\right) = \sum_{x\in S} O\left(1 + \left|e^{-1}(x)\right|(1+\lg n)\right)$$

time.

Proof. Line 4 in Algorithm 2 is executed $|e^{-1}(x_j)|$ times, and line 3 is executed $|S|$ times; therefore, lines 2–4 take $O(|S| + \sum_{x\in S} |e^{-1}(x)|)$ time. In the implementation, each element x_i has a flag that shows its membership in L', so line 8 takes constant time. Thus, lines 5–8 also take $O(|S| + \sum_{x\in S} |e^{-1}(x)|)$. Observe that $|L'| \leqslant \sum_{x\in S} |e^{-1}(x)|$. Each of the lines 10–13 runs $|L'|$ times. Lines 10 and 11 take constant time. By our assumption regarding general balanced BSTs, it takes $O(\lg n)$ to execute line 12. By using the same technique used for L', we can execute line 13 in $O(1)$ time. Therefore, lines 9–13 take $O((1 + \lg n)\sum_{x\in S} |e^{-1}(x)|)$. Lines 14 and 15 add all the new potential splitters to L. The number of new potential splitters is bounded by $|L'|$. The worst case happens when each element in L' constitutes its own V-set. Therefore, lines 14 and 15 take $O(|S| + \sum_{x\in S} |e^{-1}(x)|)$ time. □

Lemma 2. *Regardless of the sub-block tree data structure, any element x can appear at most $\lceil \lg(n+1) \rceil$ times in any potential splitter S in all executions of* SPLIT.

Proof. Let B be the block to which x belongs at some point in the algorithm, and assume B is partitioned. Then, x belongs either to the largest sub-block (which is no longer considered a splitter), or to any other sub-block, in which case the sub-block is considered a potential splitter. In the latter case, the size of the sub-block is at most $|B|/2$. In the worst case, each time its block is partitioned, x ends up in a sub-block that is not discarded as the largest, and hence has at most half the size of the original block. Moreover, in the initial partition, none of the blocks, including x's block, will be larger than n. Therefore, for any n and $k \in \mathbb{N}$ such that $2^{k-1} \leqslant n < 2^k$, x belongs to at most k potential splitters. We will then have,

$$2^{k-1} \leqslant n < 2^k \Rightarrow$$
$$k - 1 < \lg(n+1) \leqslant k \Rightarrow$$
$$k = \lceil \lg(n+1) \rceil. \qquad \square$$

Therefore, the running time of the algorithm is

$$\begin{aligned}
&\sum_{\substack{\text{all splitters}\\ S}} \sum_{x\in S} O\left(1 + \left|e^{-1}(x)\right|(1+\lg n)\right) \\
&= O\left(\lg(n+1) \sum_{x\in \mathcal{S}} \left(1 + \left|e^{-1}(x)\right|(1+\lg n)\right)\right) \\
&= O\left(\lg n \times \left(n + m(1+\lg n)\right)\right) \\
&= O(m \lg^2 n).
\end{aligned}$$

Notice that the range of the sum operator is $x \in S$ in the first line and $x \in \mathcal{S}$ in the second line. That leads to the following theorem:

[3] In the actual implementation, only one of the lists is actually stored at any given time.

Theorem 1. *If general balanced BSTs are used as the sub-block trees, the refinement step of* LUMPCTMC *takes* $O(m \lg^2 n)$ *time.*

The part of the algorithm that increased the complexity from $O(m \lg n)$ to $O(m \lg^2 n)$ is line 12. In the next section, we will show how a statically optimal tree (e.g., a splay tree) will allow us to perform a tighter analysis.

4.2. Using splay trees

In the second configuration, we use splay trees to represent sub-block trees. Based on the "splaying" heuristic, a splay tree is a BST for which only one restructuring operation, the splaying step, is defined. Each time a node is inserted, deleted, or queried, a number of splaying steps are performed to restructure the tree. For more details see [18].

The key property of splay trees that lets us do a tighter analysis of the running time of the algorithm is given by the following theorem:

Theorem 2 (Static optimality theorem [18]). *Suppose that T is an initially empty splay tree and that a sequence of q accesses* (*insertion, deletion, or query*) *to n different elements is to be done on T. Also, suppose the number of accesses to element i is $q_i > 0$ ($q = \sum_{i=1}^{n} q_i$). Then, the total time to perform the sequence of accesses is*

$$O\left(q + \sum_{i=1}^{n} q_i \lg\left(\frac{q}{q_i}\right)\right).$$

If we amortize the total time given above over all the accesses we will have:

Corollary 1. *Given the assumptions of Theorem* 2, *an access to element i takes* $O(1 + \lg(q/q_i)) = O(1 + \lg q - \lg q_i)$ *amortized time.*

According to the analysis given in Lemma 1, in an execution of SPLIT, lines 1–15 minus line 12 take $O(|S| + \sum_{x \in S} |e^{-1}(x)|)$ time. Using Lemma 2 we observe that lines 1–11 and lines 13–15 take a total time of $O(m \lg n)$ during the execution of LUMPCTMC. Therefore, in order to prove that the refinement step of LUMPCTMC runs in $O(m \lg n)$, it is enough to prove that line 12 of SPLIT takes a total time of $O(m \lg n)$ over all executions of SPLIT.

Lemma 3. *When splay trees are used as the sub-block trees, all executions of* SPLIT *in* LUMPCTMC *take* $O(m \lg n)$ *time.*

Proof. Consider an element $x \in \mathcal{S}$. If x is appended to L' in SPLIT in line 8, it will be inserted into a sub-block tree in line 12. It will then be put into a (possibly smaller) block. In fact, lines 9–13 partition some of the blocks into smaller blocks (the smaller blocks replace the previous ones). Suppose x's block is partitioned j times during the execution of the algorithm. Let B^i be the ith block to which x belongs ($B^0 \supseteq B^1 \supseteq \cdots \supseteq B^j$). In order to partition B^i ($0 \leqslant i < j$) into its sub-blocks, we create the tree B_T^i, into which at most $|B^i|$ elements will be inserted. The number of accesses to a node k_l of B_T^i will be $|V_{k_l}|$; that is the size of the V-set of k_l, and also equals the size of the sub-block associated with k_l. The sub-block of x is B^{i+1}; therefore, by applying Corollary 1 to accesses to B_T^i, it takes $O(1 + \lg|B^i| - \lg|B^{i+1}|)$ time to insert x into B_T^i, i.e., to perform line 12 in SPLIT.

We break this running time into $O(1)$ and $O(\lg|B^i| - \lg|B^{i+1}|)$. As we showed earlier, line 12 is executed $O(m \lg n)$ times during the runtime of LUMPCTMC, i.e., for all elements of $\mathcal{S}$ and for all possible values of i. Therefore, the $O(1)$ part takes $O(m \lg n)$. The $O(\lg|B^i| - \lg|B^{i+1}|)$ part of all executions (i.e., for all i, $0 \leqslant i < j$) of INSERT(B_T^i, x) takes

$$\sum_{i=0}^{j-1} O\big(\lg|B^i| - \lg|B^{i+1}|\big) = O\big(\lg|B^0| - \lg|B^j|\big).$$

Since $1 \leqslant |B^i| \leqslant n$, we have $\lg|B^0| - \lg|B^j| < \lg n$. Therefore, the $O(\lg|B^i| - \lg|B^{i+1}|)$ part of all executions of INSERT(B_T^i, x) takes $O(\lg n)$ time and consequently $O(n \lg n)$ time for all elements of $\mathcal{S}$. Hence line 12, and therefore SPLIT, takes a total of $O(m \lg n)$ time. □

Theorem 3. *Under the assumptions of the second configuration, the refinement step of* LUMPCTMC *takes* $O(m \lg n)$ *time.*

It is worth noting that with an $O(m)$ preprocessing on Q (i.e., reversing all transitions) and no ad-

ditional space requirement, we can also compute the coarsest *exactly* lumpable [17] partition of a CTMC in $O(m \lg n)$ time.

5. Conclusion

We have shown that with the use of splay trees, the time complexity of constructing the optimal lumping quotient of a finite Markov chain is $O(m \lg n)$. From a more abstract perspective, our algorithm decides Larsen/Skou-style bisimilarity [14] on general weighted automata in $O(m \lg n)$ time.

Currently, we are working on the implementation of the algorithm and its integration into the Möbius framework [5] in order to experimentally observe how the selection of the sub-block tree affects the actual running time of the implementation.

Another area of further work is the extension to reward models. In addition to supporting rewards on states in the implementation, we are going to support rewards on transitions by preprocessing the underlying Markov chain. In the preprocessing, the states of the Markov chain will be augmented with some information regarding the transitions such that transition rewards are integrated into the states of the Markov chain. While that can lead to an a priori blow-up of the state space, the problem of rewards on transitions is then reduced to that of rewards on states, and the model can be optimally lumped with the algorithm presented here.

Acknowledgements

We would like to thank Professor Jeff Erickson of the Computer Science department at the University of Illinois for his helpful discussions and Ms. Jenny Applequist for her editorial assistance.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.

[2] M. Bernardo, R. Gorrieri, A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time, Theoret. Comput. Sci. 202 (1998) 1–54.

[3] P. Buchholz, Exact and ordinary lumpability in finite Markov chains, J. Appl. Probab. 31 (1994) 59–75.

[4] P. Buchholz, Efficient computation of equivalent and reduced representations for stochastic automata, Internat. J. Comput. Systems Sci. Engrg. 15 (2) (2000) 93–103.

[5] D.D. Deavours, G. Clark, T. Courtney, D. Daly, S. Derisavi, J.M. Doyle, W.H. Sanders, P. Webster, The Möbius framework and its implementation, IEEE Trans. Software Engrg. 28 (10) (2002) 956–969.

[6] H. Hermanns, Interactive Markov chains, PhD thesis, Friedrich-Alexander-Universität, Erlangen-Nürnberg, 1998.

[7] H. Hermanns, M. Siegle, Bisimulation algorithms for stochastic process algebras and their BDD-based implementation, in: J.-P. Katoen (Ed.), ARTS'99, 5th Internat. AMAST Workshop on Real-Time and Probabilistic Systems, Vol. 1601, Springer, Berlin, 1999, pp. 144–264.

[8] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, Cambridge, 1996.

[9] J.E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Theory of Machines and Computations, Academic Press, New York, 1971, pp. 189–196.

[10] R.A. Howard, Dynamic Probabilistic Systems, Vol. II: Semi-Markov and Decision Processes, Wiley, New York, 1971.

[11] D.T. Huynh, L. Tian, On some equivalence relations for probabilistic processes, Fund. Inform. 17 (1992) 211–234.

[12] P.C. Kanellakis, S.A. Smolka, CCS expressions, finite state processes, and three problems of equivalence, in: Proc. ACM Symposium on Principles of Distributed Computing, 1983, pp. 228–240.

[13] J.G. Kemeney, J.L. Snell, Finite Markov Chains, D. Van Nostrand, New York, 1960.

[14] K. Larsen, A. Skou, Bisimulation through probabilistic testing, Inform. and Comput. 94 (1) (1991) 1–28.

[15] R. Milner, Communication and Concurrency, Prentice-Hall, London, 1989.

[16] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (6) (1987) 973–989.

[17] P.J. Schweitzer, Aggregation methods for large Markov chains, in: G. Iazeolla, P.J. Courtois, A. Hordijk (Eds.), Mathematical Computer Performance and Reliability, Elsevier, Amsterdam, 1984, pp. 275–302.

[18] D.D. Sleator, R.E. Tarjan, Self-adjusting binary search trees, J. ACM 32 (3) (1985) 652–686.

[19] U. Sumita, M. Rieders, Lumpability and time reversibility in the aggregation–disaggregation method for large Markov chains, Comm. Statist. Stochastic Models 5 (1989) 63–81.