# Simple $O(m \log n)$ Time Markov Chain Lumping

Antti Valmari[1] and Giuliana Franceschinis[2]

[1] Tampere University of Technology, Department of Software Systems
P.O. Box 553, FI-33101 Tampere, Finland
Antti.Valmari@tut.fi

[2] Dip. di Informatica, Univ. del Piemonte Orientale
viale Teresa Michel 11, 15121 Alessandria, Italy
Giuliana.Franceschinis@mfn.unipmn.it

**Abstract.** In 2003, Derisavi, Hermanns, and Sanders presented a complicated $O(m \log n)$ time algorithm for the Markov chain lumping problem, where $n$ is the number of states and $m$ the number of transitions in the Markov chain. They speculated on the possibility of a simple algorithm and wrote that it would probably need a new way of sorting weights. In this article we present an algorithm of that kind. In it, the weights are sorted with a combination of the so-called possible majority candidate algorithm with any $O(k \log k)$ sorting algorithm. This works because, as we prove in the article, the weights consist of two groups, one of which is sufficiently small and all weights in the other group have the same value. We also point out an essential problem in the description of the earlier algorithm, prove the correctness of our algorithm in detail, and report some running time measurements.

## 1 Introduction

Markov chains are widely used to analyze the behaviour of dynamic systems and to evaluate their performance or dependability indices. One of the problems that limit the applicability of Markov chains to realistic systems is state space explosion. Among the methods that can be used to keep this problem under control, *lumping* consists of aggregating states of the Markov chain into "macrostates", hence obtaining a smaller Markov chain while preserving the ability to check desired properties on it.

We refer to [4,8] for different lumpability concepts and their use in the analysis of systems. For the purpose of this article it suffices that in the heart of their use is the problem of constructing the coarsest lumping quotient of a Markov chain. We define this problem formally in Section 2, and call it "the lumping problem" for brevity.

Let $n$ denote the number of states and $m$ the number of transitions in the Markov chain. An $O(n + m \log n)$ time algorithm for the lumping problem was given in [6,5]. It is (loosely) based on the Paige–Tarjan relational coarsest partition algorithm [10] of similar complexity. Unless the input is pathological with many isolated states, we have $n = O(m)$ implying $O(n + m \log n) = O(m \log n)$. Therefore, it is common practice to call these algorithms $O(m \log n)$.

The Paige–Tarjan algorithm starts with an initial partition of the set of states and refines it until a certain condition is met. Sets of the partition are traditionally called *blocks*. A basic operation in the Paige–Tarjan algorithm is the splitting of a block to at most three subblocks. We call one of the subblocks the *middle group*, and another one the *left block*. Their precise definitions will be presented in Section 6.

When applying the Paige–Tarjan algorithm to the lumping problem, the block splitting operation has to be modified. The middle group may have to be split further to one or more *middle blocks*. On the other hand, the rather complicated mechanism used by the Paige–Tarjan algorithm for separating the left block from the middle group is not needed any more, because the refined splitting operation can do that, too.

The authors of [6] first discussed a general balanced binary tree approach to implementing the refined splitting operation. They proved that it yields $O(m \log^2 n)$ time complexity to the algorithm as a whole. Then they proved $O(m \log n)$ time complexity for the special case where the trees are splay trees.

The authors of [6] speculated whether $O(m \log n)$ time complexity could be obtained with a simpler solution than splay trees. In this article we show that this is the case. Instead of always processing the left block and middle group together with a binary search tree, our algorithm processes them separately when necessary. Separation is obtained with the so-called possible majority candidate algorithm. The left block need not be split further. The splitting of the middle group is based on sorting it with just any $O(k \log k)$ time algorithm, where $k$ is the number of items to be sorted. To show that this yields the desired complexity, we take advantage of a special property of middle blocks that sets an upper bound to the number of times each state can be in a middle block. The left block lacks this property. Our algorithm sometimes separates some middle block instead of the left block, but when this happens, the left block is so small that it does not matter.

The articles [6,5] do not show a correctness proof of their algorithm. Indeed, the description and pseudocode in them ignore an essential issue. This makes direct implementations produce wrong results every now and then, as we show in Section 4 with an example. The splitting operation uses one block, called *splitter*, as input. If block $B$ has been used as a splitter and is then itself split to $B_1$, $B_2$, $\ldots$, $B_k$, then it suffices that all but one of them are used as splitters later on. The good performance arises from not using a biggest one among the $B_i$ in the future. However, if $B$ has not been used as a splitter, then every $B_i$ must be used in the future. The articles [6,5] fail to say that. Because of this, we felt it appropriate to discuss the correctness issue in great detail in this article.

In Section 2 we describe the lumping problem rigorously. Section 3 introduces the less well known old algorithms and data structures that our new algorithm uses. Our new algorithm is presented in Section 4 and proven correct in Section 5. That it runs in $O(n + m \log n)$ (or $O(m \log n)$) time is shown in Section 6. Section 7 presents some measurements made with a test implementation, and Section 8 presents our conclusions.

## 2 The Lumping Problem

The input of the lumping problem consists of a weighted directed graph $(S, \Delta, W)$ together with an initial partition $\mathcal{I}$. In the definition, $\Delta \subseteq S \times S$, and $W$ is a function from $\Delta$ to real numbers. The elements of $S$, $\Delta$, and $W$ are called *states*, *transitions*, and *weights*, respectively. We let $n$ denote the number of states and $m$ the number of transitions. For convenience, we extend $W$ to $S \times S$ by letting $W(s, s') = 0$ whenever $(s, s') \notin \Delta$. We also extend $W$ to the situation where the second argument is a subset of $S$ by $W(s, B) = \sum_{s' \in B} W(s, s')$. By $s \to s'$ we mean that $(s, s') \in \Delta$. If $B \subseteq S$, then $s \to B$ denotes that there is some $s' \in B$ such that $s \to s'$.

In many applications, the values $W(s, s')$ are non-negative. We do not make that assumption, however, because there are also applications where $W(s, s)$ is deliberately chosen as $-W(s, S \setminus \{s\})$, making it usually negative. It is also common that $W(s, S)$ is the same for every $s \in S$, but we do not make that assumption either.

A *partition* of a set $A$ is a collection $\{A_1, A_2, \ldots, A_k\}$ of pairwise disjoint nonempty sets such that their union is $A$. The initial partition $\mathcal{I}$ is a partition of $S$. The elements of a partition of $S$ are traditionally called *blocks*. A partition $\mathcal{B}'$ is a *refinement* of a partition $\mathcal{B}$ if and only if each element of $\mathcal{B}'$ is a subset of some element in $\mathcal{B}$.

A partition $\mathcal{B}$ of $S$ is *compatible* with $W$ if and only if for every $B \in \mathcal{B}$, $B' \in \mathcal{B}$, $s_1 \in B$, and $s_2 \in B$ we have $W(s_1, B') = W(s_2, B')$. Let *croip* be an abbreviation for "compatible refinement of initial partition", that is, a partition of $S$ that is a refinement of $\mathcal{I}$ and compatible with $W$. The objective of the lumping problem is to find the coarsest possible croip, that is, the croip whose blocks are as big as possible. Our new algorithm solves it.

Sometimes a variant problem is of interest where compatibility is defined in a different way. In it, compatibility holds if and only if for every $B \in \mathcal{B}$, $B' \in \mathcal{B} \setminus \{B\}$, $s_1 \in B$, and $s_2 \in B$ we have $W(s_1, B') = W(s_2, B')$. The variant problem can be solved by, for each state $s$, replacing $W(s, s)$ by $-W(s, S \setminus \{s\})$, and then using the algorithm for the lumping problem [5]. This is an instance of a more general fact, given by the next proposition.

**Proposition 1.** *For every $I \in \mathcal{I}$, let $w_I$ be an arbitrary real number and $U_I = I \cup \bigcup \mathcal{I}_I$, where $\mathcal{I}_I$ is an arbitrary subset of $\mathcal{I}$. Let $W'$ be defined by*

$$\begin{aligned} W'(s, s') &:= W(s, s') \text{ when } s' \neq s, \text{ and} \\ W'(s, s) &:= w_I - W(s, U_I \setminus \{s\}), \text{ where } I \text{ is the } I \in \mathcal{I} \text{ that contains } s. \end{aligned}$$

*Then the coarsest lumping-croip with $W'$ is the coarsest variant-croip with $W$.*

*Proof.* The value of $W(s, B)$ has no role in the definition of variant-compatibility whenever $s \in B$. This implies that the value of $W(s, s)$ has never any role. So $W$ and $W'$ yield the same variant-croips. The claim follows, if we now show that with $W'$, every lumping-croip is a variant-croip and vice versa.

It is immediate from the definitions that every lumping-croip is a variant-croip. To prove the opposite direction with $W'$, let $\mathcal{B}$ be a variant-croip, $B \in \mathcal{B}$,

$s_1 \in B$, and $s_2 \in B$. We have to prove that $W'(s_1, B') = W'(s_2, B')$ for every $B' \in \mathcal{B}$. This is immediate when $B' \neq B$ by the definition of variant-croips. We prove next that $W'(s_1, B) = W'(s_2, B)$, completing the proof.

Let $I$ be the initial block that contains $s_1$, and let $B_1$, $B_2$, …, $B_k$ be the blocks to which the blocks in $\{I\} \cup \mathcal{I}_I$ have been split in $\mathcal{B}$. Clearly $s_2 \in I$, $B$ is one of the $B_i$, $B_1 \cup \cdots \cup B_k = U_I$, and $\sum_{i=1}^{k} W'(s, B_i) = W'(s, U_I) = w_I$ when $s \in I$. Without loss of generality we may index the $B_i$ so that $B = B_1$. Then $W'(s_1, B) = w_I - \sum_{i=2}^{k} W'(s_1, B_i) = w_I - \sum_{i=2}^{k} W'(s_2, B_i) = W'(s_2, B)$, because $W'(s_1, B') = W'(s_2, B')$ when $B' \neq B$. □

## 3 Background Data Structures and Algorithms

In this section we introduce those algorithms and data structures that are needed in the rest of the article, not new, but not presented in typical algorithm text-books either.

**Refinable Partition.** Our lumping algorithm needs a data structure for maintaining the blocks. We present two suitable data structures that provide the following services.

They make it possible in constant time to find the size of a block, find the block that a given state belongs to, mark a state for subsequent splitting of a block, and tell whether a block contains marked states. They also facilitate scanning the states of a block in constant time per scanned element, assuming that states are not marked while scanning. Finally, there is a block splitting operation that runs in time proportional to the number of marked states in the block. It makes one subblock of the marked states and another of the remaining states, provided that both subblocks will be nonempty. If either subblock will be empty, it does not split the block. In both cases, it unmarks the marked states of the block. It is important to the efficiency of the lumping algorithm that the running time of splitting is only proportional to the number of marked, and not all, states in the block.

A traditional refinable partition data structure represents each block with two doubly linked lists: one for the marked states and another for the remaining states [1, Sect. 4.13]. The record for the block contains links to the lists, together with an integer that stores the size of the block. It is needed, because the size must be found fast. The record for a state contains a link to the block that the state belongs to, and forward and backward links.

Marking of an unmarked state consists of unlinking it from its current list and adding it to the list of the marked states of its block. In the splitting, the new block is made of the marked states, and unmarked states stay in the old block. This is because all states of the new block must be scanned, to update the link to the block that the state belongs to. The promised running time does not necessarily suffice for scanning the unmarked states. (For simplicity, we ignore the other alternative where the smaller subblock is made the new block.)

A more recent refinable partition data structure was inspired by [9] and presented in [12]. In it, states and blocks are represented by numbers. All states

```
$count$ := 0
for $i$ := 1 to $k$ do
    if $count = 0$ then
        $pmc := A[i]$ ; $count$ := 1
    else if $A[i] = pmc$ then
        $count := count + 1$
    else
        $count := count - 1$
```

**Fig. 1.** Finding a possible majority candidate

(that is, their numbers) are in an array *elems* so that states that belong to the same block are next to each other. The segment for a block is further divided to a first part that contains the marked states and second part that contains the rest. There is another array that, given the number of a state, returns its location in *elems*. A third array denotes the block that each state belongs to.

Three arrays are indexed by block numbers. They tell where the segment for the block in *elems* starts and ends, and where is the borderline between the marked and other states. An unmarked state is marked by swapping it with the first unmarked state of the same block, and moving the borderline one step.

**Possible Majority Candidate.** A possible majority candidate *pmc* of an array $A[1 \dots k]$ is any value that has the following properties. If some value occupies more than half of the positions of $A$, then *pmc* is that value. Otherwise *pmc* is just any value that occurs in $A$.

The algorithm in Figure 1 finds a possible majority candidate in linear time [3, Sect. 4.3.3]. To see that it works, let $f(x) = count$ when $pmc = x$ and $f(x) = -count$ when $pmc \neq x$. When $A[i] = x$, then $f(x)$ increases by one independently of the value of *pmc*, and when $A[i] \neq x$, then $f(x)$ increases or decreases by one. If $x$ occurs in more than half of the positions, then $f(x)$ increases more times than decreases, implying that at the end of the algorithm $f(x) > 0$. This guarantees that $pmc = x$, because otherwise *count* would have to be negative, and the tests in the code prevent it from becoming negative.

## 4 The Lumping Algorithm

Our new lumping algorithm is shown in Figure 2. The grey commands on lines 1 and 3 are not part of it. They are added because of the needs of the proofs of the correctness and performance of the algorithm. They will be discussed in Sections 5 and 6. We will prove that their presence or absence does not affect the output of the algorithm.

The input to the algorithm consists of $S$, $\Delta$, $W$, and $\mathcal{I}$. We assume that $\Delta$ is available as the possibility to scan the input transitions of each state in constant time per scanned transition, and define $\bullet s' = \{s \mid s \rightarrow s'\}$.

The algorithm maintains a refinable partition of states. The initial value of the partition is $\mathcal{I}$. Each block has an identity (number or address) with which it

```
 1  $\mathsf{U_B} := \mathcal{I}$ ; $\mathsf{B_T} := \emptyset$ ; $w[s] :=$ unused for every $s \in S$ ; $\mathcal{C} := \{S \cup \{s_\perp\}\}$
 2  **while** $\mathsf{U_B} \neq \emptyset$ **do**
 3      let $B'$ be any block in $\mathsf{U_B}$ ; $\mathsf{U_B} := \mathsf{U_B} \setminus \{B'\}$ ; $\mathcal{C} := \mathcal{C} \setminus \{C_{B'}\} \cup \{B', C_{B'} \setminus B'\}$
 4      $\mathsf{S_T} := \emptyset$
 5      **for** $s' \in B'$ **do for** $s \in \bullet s'$ **do**
 6          **if** $w[s] =$ unused **then** $\mathsf{S_T} := \mathsf{S_T} \cup \{s\}$ ; $w[s] := W(s, s')$
 7          **else** $w[s] := w[s] + W(s, s')$
 8      **for** $s \in \mathsf{S_T}$ **do if** $w[s] \neq 0$ **then**
 9          $B :=$ the block that contains $s$
10          **if** $B$ contains no marked states **then** $\mathsf{B_T} := \mathsf{B_T} \cup \{B\}$
11          mark $s$ in $B$
12      **while** $\mathsf{B_T} \neq \emptyset$ **do**
13          let $B$ be any block in $\mathsf{B_T}$ ; $\mathsf{B_T} := \mathsf{B_T} \setminus \{B\}$
14          $B_1 :=$ marked states in $B$ ; $B :=$ remaining states in $B$
15          **if** $B = \emptyset$ **then** give the identity of $B$ to $B_1$ **else** make $B_1$ a new block
16          $pmc :=$ possible majority candidate of the $w[s]$ for $s \in B_1$
17          $B_2 := \{s \in B_1 \mid w[s] \neq pmc\}$ ; $B_1 := B_1 \setminus B_2$
18          **if** $B_2 = \emptyset$ **then** $\ell := 1$ **else**
19              sort and partition $B_2$ according to $w[s]$, yielding $B_2, \ldots, B_\ell$
20              make each of $B_2, \ldots, B_\ell$ a new block
21          **if** $B \in \mathsf{U_B}$ **then** add $B_1, \ldots, B_\ell$ except $B$ to $\mathsf{U_B}$
22          **else** add $[B,]^?$ $B_1, \ldots, B_\ell$ except a largest to $\mathsf{U_B}$
23      **for** $s \in \mathsf{S_T}$ **do** $w[s] :=$ unused
```

**Fig. 2.** The coarsest lumping algorithm

can be found via an index or pointer. We saw in Section 3 that when a block is split, the splitting operation decides which subblock inherits the identity of the original block.

The array $w$ has one slot for each $s \in S$. It stores numbers. One value that could not otherwise occur is reserved for a special purpose and denoted with "unused" in the pseudocode. In our implementation, unused = `DBL_MAX`, that is, the maximal double precision floating point value of the computer.

The algorithm maintains a set $\mathsf{U_B}$ of "unprocessed" blocks, that is, blocks that have to be used later for splitting. Similarly, $\mathsf{S_T}$ maintains a set of "touched" states and $\mathsf{B_T}$ a set of "touched" blocks that will be processed later. The algorithm has been designed so that only very simple operations are needed on them. In particular, when something is being added, it is certain that it is not already there. It is thus easy to implement these sets efficiently as stacks or other data structures. The sets contain indices of or pointers to blocks and states, not copies of the block and state data structures. Therefore, when a block that is in $\mathsf{U_B}$ is split, the subblock that inherits its identity also inherits the presence in $\mathsf{U_B}$.

Initially $\mathsf{U_B}$ contains all blocks. The body of the main loop of the algorithm (lines 3 to 23) takes and removes an arbitrary block $B'$ from $\mathsf{U_B}$ and splits all blocks using it. The splitting operation may add new blocks to $\mathsf{U_B}$. This is repeated until $\mathsf{U_B}$ becomes empty. A block that is used in the role of $B'$ is called a *splitter*.

Let $\bullet B'$ denote the set of states which have transitions to $B'$, that is, $\bullet B' = \{s \mid \exists s' \in B' : s \rightarrow s'\}$. Lines 4 to 7 find those states, collect them into $\mathsf{S_T}$, and compute $W(s, B')$ for them. Each $W(s, B')$ is stored in $w[s]$. The **if** test ensures that each state is added to $\mathsf{S_T}$ only once. The used $w[s]$ are reset back to "unused" on line 23. This is a tiny bit more efficient than resetting the $w[s]$ before use via $\bullet s'$, as was done in [6].

Lines 8 to 11 mark those states in $\bullet B'$ that have $W(s, B') \neq 0$, and collect into $\mathsf{B_T}$ the blocks that contain such states. We saw in Section 3 that the marking operation moves the state to a new place in the refinable partition data structure (to another linked list or to another part of an array). As a consequence, the marking operation interferes with the scanning of states. It would confuse the scanning of $B'$ on line 5, if it were done in that loop. This is the main reason for the seemingly clumsy operation of collecting $\bullet B'$ into $\mathsf{S_T}$ and scanning it anew from there. Another reason is that it makes it easy to get rid of states that have $W(s, B') = 0$.

Lines 12 to 22 scan each block that has at least one $s$ such that $W(s, B') \neq 0$, and split it so that the resulting subblocks are compatible with $B'$. Lines 14 and 15 are the same as the splitting operation in Section 3. They split $B$ to those states that have and those that do not have $W(s, B') \neq 0$. The former are stored in $B_1$ and the latter remain in $B$. The latter include those that do not have transitions to $B'$. If $B$ would become empty, then $B_1$ will not be a new block but inherits the identity (number or address) of $B$. It must be kept in mind in the sequel that $B_1$ may be different from $B$ or the same block as $B$.

Line 16 finds a possible majority candidate among the $w[s]$ of the states in $B_1$. Lines 17 to 20 split $B_1$ to $B_1$, $B_2$, …, $B_\ell$ so that $s_1$ and $s_2$ are in the same $B_i$ if and only if $W(s_1, B') = W(s_2, B')$. After the sorting on line 19, the $s$ with the same $W(s, B')$ are next to each other and can easily be converted to a new block. The sorting operation is new compared to Section 3. However, it is well known how a doubly linked list or an array segment can be sorted in $O(k \log k)$ time, where $k$ is the number of elements to be sorted.

The subblock whose $W(s, B')$ is the possible majority candidate is processed separately because of efficiency reasons. As was mentioned above, the sorting operation costs $O(k \log k)$. As was pointed out in [6], paying $O(k \log k)$ where $k = |B_1 \cup \cdots \cup B_\ell|$ would invalidate the proof of the $O(n + m \log n)$ performance of the algorithm as a whole. The solution of [6] to this problem was to split $B_1$ to $B_1$, …, $B_\ell$ with the aid of splay trees. However, we will prove in Section 6 that $O(k \log k)$ is not too costly, if those states whose $W(s, B')$ is the possible majority candidate are not present in the sorting.

The set of blocks that will have to be used as splitters in the future is updated on lines 21 and 22. There are two cases. If $B$ is in $\mathsf{U_B}$, then all subblocks of the original $B$ must be in $\mathsf{U_B}$ after the operation. Because $B$ is already there, it suffices to put the $B_i$ into $\mathsf{U_B}$. However, $B_1$ may have inherited the identity of $B$ on line 15 and must not be put into $\mathsf{U_B}$ for a second time.

If $B \notin \mathsf{U_B}$, then it suffices that all but one of the subblocks is put into $\mathsf{U_B}$. The good performance of the algorithm relies on putting only such subblocks into $\mathsf{U_B}$

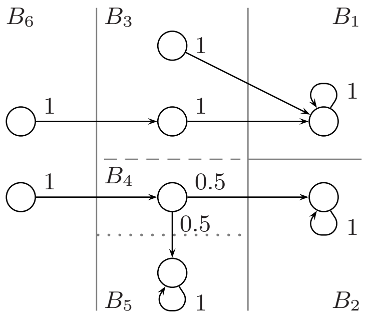

**Fig. 3.** A counter-example to never putting all subblocks into $\mathsf{U_B}$

whose sizes are at most half of the size of the original $B$. This is implemented by finding the largest, or one of the largest if there are many of maximal size, and not putting that subblock into $\mathsf{U_B}$. This works, because there can be at most one subblock whose size is more than half of the original size. The notation $[B,]^?$ reminds that if $B$ and $B_1$ refer to the same block, then only one of them should be considered.

Testing whether $B \in \mathsf{U_B}$ can be made fast, if each block has a bit that is set when the block is put into $\mathsf{U_B}$ and reset when the block is removed from $\mathsf{U_B}$.

The articles [6,5] do not discuss the distinction represented by lines 21 and 22. They seem to always work according to line 22, even if $B \in \mathsf{U_B}$. The example in Figure 3 demonstrates that this is incorrect. The initial partition is $\{B_1, B_2, B_3 \cup B_4 \cup B_5, B_6\}$. If $B_1$ is used as the first splitter, it splits $B_3 \cup B_4 \cup B_5$ to $B_3$ and $B_4 \cup B_5$. Assume that $B_3$ is not and $B_4 \cup B_5$ is put into $\mathsf{U_B}$. If $B_2$ is used as the next splitter, it splits $B_4 \cup B_5$ to $B_4$ and $B_5$. It may be that then $B_5$ is put into $\mathsf{U_B}$ and $B_4$ is not. At this stage, $B_3$ and $B_4$ are not in $\mathsf{U_B}$, and none of the other blocks induces any splitting. Thus $B_6$ is never split, although it should be. This problem makes implementations based directly on [6,5] yield wrong results.

## 5 Correctness

In this section we prove the correctness of the algorithm presented in the previous section. In the proof, we will keep track of some information on blocks that have been used as splitters and then have been split themselves. For this purpose we introduce *compound blocks*. A compound block is always a union of ordinary blocks. The idea is that always on line 2, the splitting that any compound block $C_1$ would cause has already been done, either by having used $C_1$ as a splitter, or by having used $C$, $C_2$, …, $C_k$ as splitters, where $C = C_1 \cup \cdots \cup C_k$ and the $C_i$ are pairwise disjoint. This will be made precise later.

The grey statements in Figure 2 maintain the compound blocks. The compound blocks constitute a partition $\mathcal{C}$ of $S \cup \{s_\perp\}$, where $s_\perp$ will be explained soon. Initially $\mathcal{C}$ consists of one compound block that contains all states, including $s_\perp$. On line 3, the compound block $C_{B'}$ that covers the ordinary block $B'$ is split to two compound blocks $B'$ and $C_{B'} \setminus B'$. (The invariant after the proof of Lemma 2 will imply that $C_{B'} \setminus B' \neq \emptyset$.)

The purpose of $s_\perp$ is to make it easier to formulate two invariants that will be used in the last part of the correctness proof. Without $s_\perp$, the last part would be very difficult to follow. The easy formulation needs initially such a compound block $C_\mathsf{i}$ that $S \subseteq C_\mathsf{i}$ and $W(s, C_\mathsf{i})$ is the same for every $s \in C_\mathsf{i}$. Unfortunately, $W(s, S)$ is not necessarily the same for every $s \in S$. Fortunately, we can fix this without affecting the operation of the algorithm by adding a new imaginary state $s_\perp$. Its adjacent transitions are chosen such that $W(s, s_\perp) = -W(s, S)$ when $s \in S$, and $s_\perp$ has no output transitions. Thus $W(s, S \cup \{s_\perp\}) = 0$ for every $s \in S \cup \{s_\perp\}$, and we can let $C_\mathsf{i} = S \cup \{s_\perp\}$. The grey statement on line 1 makes $C_\mathsf{i}$ the only compound block.

We now show that the addition of $s_\perp$ changes the correct answer only by adding $\{s_\perp\}$ as an extra block to it. Clearly $\mathcal{B}$ is a refinement of $\mathcal{I}$ if and only if $\mathcal{B} \cup \{\{s_\perp\}\}$ is a refinement of $\mathcal{I} \cup \{\{s_\perp\}\}$. Furthermore, $W(s_1, B) = W(s_2, B)$ holds trivially when $\{s_1, s_2\} \subseteq \{s_\perp\}$. If $W(s_1, B) = W(s_2, B)$ for every $B \in \mathcal{B}$, then $W(s_1, \{s_\perp\}) = -\sum_{B \in \mathcal{B}} W(s_1, B) = -\sum_{B \in \mathcal{B}} W(s_2, B) = W(s_2, \{s_\perp\})$. From these it can be seen that $\mathcal{B}$ is compatible with the original $W$ if and only if $\mathcal{B} \cup \{\{s_\perp\}\}$ is compatible with $W$ extended with the transitions adjacent to $s_\perp$. So the two systems have the same croips, except for the addition of $\{s_\perp\}$.

The next important fact is that not implementing $s_\perp$ and the grey statements changes the output of the algorithm only by removing $\{s_\perp\}$ from it. The statement $\mathsf{U_B} := \mathcal{I}$ does not put $\{s_\perp\}$ into $\mathsf{U_B}$. (This is similar to line 22, where all except one subblocks of $B$ are put into $\mathsf{U_B}$.) Therefore, $s_\perp$ never occurs as the $s'$ on line 5. Because $s_\perp$ has no output transitions, it cannot occur as the $s$ on line 5 either. Its only effect on the execution of the algorithm is thus that $\{s_\perp\}$ is an extra block that is never accessed. The set $\mathcal{C}$ of compound blocks has no effect on the output, because its content is not used for anything except for the computation of new values of $\mathcal{C}$ on line 3.

We have shown the following.

**Lemma 1.** *Without the grey statements the algorithm in Figure 2 computes the correct result for $S$, $\Delta$, $W$, and $\mathcal{I}$ if and only if with the grey statements it computes the correct result when $s_\perp$ and its adjacent transitions have been added.*

We now prove that the algorithm computes the correct result in the presence of $s_\perp$ and the grey statements. The next lemma states that it does not split blocks unnecessarily.

**Lemma 2.** *Let $s_1 \in S \cup \{s_\perp\}$ and $s_2 \in S \cup \{s_\perp\}$. If the algorithm ever puts $s_1$ and $s_2$ into different blocks, then there is no croip where $s_1$ and $s_2$ are in the same block.*

*Proof.* We show that it is an invariant property of the main loop of the algorithm (that is, always valid on line 2) that if two states are in different blocks of the algorithm, then they are in different blocks in every croip.

If $s_1$ and $s_2$ are in different blocks initially, then they are in different blocks in $\mathcal{I} \cup \{\{s_\perp\}\}$ and thus in every croip.

The case remains where lines 14 to 20 separate $s_1$ and $s_2$ to different blocks. This happens only if $W(s_1, B') \neq W(s_2, B')$. Let $\mathcal{B} \cup \{\{s_\perp\}\}$ be an arbitrary croip. It follows from the invariant that each block of $\mathcal{B} \cup \{\{s_\perp\}\}$ is either disjoint with $B'$ or a subset of $B'$, because otherwise the algorithm would have separated two states that belong to the same block of a croip. Therefore, there are blocks $B'_1, \ldots, B'_k$ in $\mathcal{B} \cup \{\{s_\perp\}\}$ such that $B'_1 \cup \cdots \cup B'_k = B'$. The fact $W(s_1, B') \neq W(s_2, B')$ implies that there is $1 \leq i \leq k$ such that $W(s_1, B'_i) \neq W(s_2, B'_i)$. So $s_1$ and $s_2$ belong to different blocks in $\mathcal{B} \cup \{\{s_\perp\}\}$. □

Proving that the algorithm does all the splittings that it should is more difficult. We first show that the following is an invariant of the main loop.

For each $C$ in $\mathcal{C}$, $\mathsf{U_B}$ contains all but one blocks $B$ that are subsets of $C$.

This is initially true because $\mathsf{U_B}$ contains all blocks except $\{s_\perp\}$, and $\mathcal{C} = \{C_\mathsf{i}\}$ where $C_\mathsf{i} = S \cup \{s_\perp\}$. On line 3, $B'$ is removed from $\mathsf{U_B}$ but also subtracted from $C_{B'}$, so the invariant becomes valid for $C_{B'} \setminus B'$. It becomes valid for the new compound block $B'$, because it consists of one block that is not any more in $\mathsf{U_B}$. Lines 21 and 22 update $\mathsf{U_B}$ so that either $B$ was in $\mathsf{U_B}$ before the splitting operation and all of its subblocks are in $\mathsf{U_B}$ after the operation, or $B$ was not in $\mathsf{U_B}$ beforehand and precisely one of its subblocks is not in $\mathsf{U_B}$ afterwards. Thus they do not change the number of blocks that are subsets of $C$ and not in $\mathsf{U_B}$.

The invariant implies that each compound block contains at least one ordinary block, namely the one that is not in $\mathsf{U_B}$.

At this point it is easy to prove that the algorithm terminates. Termination of all loops other than the main loop is obvious. Each iteration of the main loop splits one compound block to two non-empty parts. There can be at most $|S|$ splittings, because after them each compound block would consist of a single state, and thus of precisely one block. By the previous invariant, that block is not in $\mathsf{U_B}$, and hence $\mathsf{U_B}$ is empty.

Another important invariant property of the main loop is

For every block $B$, $s_1 \in B$, $s_2 \in B$, and $C \in \mathcal{C}$ we have $W(s_1, C) = W(s_2, C)$.

This is initially true because initially $\mathcal{C} = \{S \cup \{s_\perp\}\}$, and $W(s, S \cup \{s_\perp\}) = 0$ for every $s \in S \cup \{s_\perp\}$. Assume that the invariant holds for $C = C_{B'}$. The splitting of $C_{B'}$ to $B'$ and $C_{B'} \setminus B'$ on line 3 violates the invariant, but the rest of the main loop re-establishes it for $C = B'$. Regarding $C = C_{B'} \setminus B'$, if $s_1$ and $s_2$ are in the same block, then $W(s_1, C_{B'} \setminus B') = W(s_1, C_{B'}) - W(s_1, B') = W(s_2, C_{B'}) - W(s_2, B') = W(s_2, C_{B'} \setminus B')$. So the invariant remains valid.

Lines 1 and 3 imply that each ordinary block is a subset of a compound block. When the algorithm terminates, $\mathsf{U_B} = \emptyset$. Then, by the first invariant, each compound block consists of a single ordinary block. Therefore, ordinary and compound blocks are then the same thing. In this situation, the second invariant reduces to the claim that the partition is compatible. We have proven the following lemma.

**Lemma 3.** *The algorithm terminates, and when it does that, the partition is compatible.*

So the algorithm terminates with a croip. By Lemma 2, all other croips are refinements of the one produced by the algorithm. This means that the output is the coarsest croip. Now Lemma 1 yields Theorem 1.

**Theorem 1.** *The algorithm in Figure 2 (without $s_\perp$ and the grey statements) finds the coarsest refinement of $\mathcal{I}$ that is compatible with $W$.*

We did not assume in the correctness proof of the algorithm that the coarsest croip exists. Therefore, our proof also proves that it exists.

It can be reasoned from the proof that if for every initial block $B$ and every $s_1 \in B$ and $s_2 \in B$ we have $W(s_1, S) = W(s_2, S)$, then it is correct to put initially all but one of the initial blocks into $\mathsf{U_B}$.

## 6 Performance

In this section we show that the algorithm in Figure 2 runs in $O(n + m \log n)$ time, where $n$ is the number of states and $m$ is the number of transitions.

Line 1 runs clearly in $O(n)$ time.

Let us now consider one iteration of the main loop. Lines 3 to 7 run in $O(|B'|) + O(\sum_{s' \in B'} |\bullet s'|)$ time. They find $|\bullet B'| \leq \sum_{s' \in B'} |\bullet s'|$ states and store them into $\mathsf{S_T}$. Lines 8 to 11 and 23 scan the same states and thus run in $O(|\bullet B'|)$ time. Lines 12 to 22 scan a subset of the blocks that contain these states. By Section 3, the running time of lines 14 and 15 is only proportional to the number of these states. Therefore, excluding the sorting operation on line 19, lines 12 to 22 run in $O(|\bullet B'|)$ time. To summarize, excluding the sorting operation, lines 3 to 23 run in $O(|B'|) + O(\sum_{s' \in B'} |\bullet s'|)$ time. The $O(|B'|)$ term can be charged in advance, when $B'$ is put into $\mathsf{U_B}$. This leaves $O(|\bullet s'|)$ time for each $s' \in B'$.

Assume that $B'$ is used as a splitter and later on some $B'' \subseteq B'$ is used as a splitter. There has been a sequence $B'_0, \ldots, B'_k$ of blocks such that $k \geq 1$, $B' = B'_0$, $B'_k = B''$, and $B'_i$ has been created by splitting $B'_{i-1}$ when $1 \leq i \leq k$. When $B'_1$ was created, $B'_0$ was not in $\mathsf{U_B}$ because it had been used as a splitter. When $B'_k$ was created, it was put into $\mathsf{U_B}$ or inherited a position in $\mathsf{U_B}$, because it was later used as a splitter. There is thus at least one $i$ between 1 and $k$ such that $B'_{i-1}$ was not in $\mathsf{U_B}$ and $B'_i$ was put into $\mathsf{U_B}$ when $B'_i$ was created. We see that $B'_i$ was put into $\mathsf{U_B}$ by line 22. As a consequence, $|B'_i| \leq \frac{1}{2}|B'_{i-1}|$. Clearly $|B'_0| \geq |B'_1| \geq \ldots \geq |B'_k|$. So $|B''| \leq \frac{1}{2}|B'|$.

This implies that each time when a state $s'$ is used for splitting, it belongs to a splitter whose size is at most half of the size in the previous time. Therefore, the state can occur in a splitter at most $\log_2 n + 1$ times. The contribution of $s'$ to the execution time of the algorithm as a whole is thus $O((\log n)|\bullet s'|)$ plus the share of $s'$ of the time needed for sorting. When this is summed over every $s' \in S$ and added to the $O(n)$ from line 1, it yields $O(n + m \log n)$, because then $\bullet s'$ goes through all transitions.

We have proven the following lemma.

**Lemma 4.** *Excluding the sorting operations on line 19, the algorithm in Figure 2 runs in $O(n + m \log n)$ time.*

We still have to analyse the time consumption of the sorting operations. For that purpose, consider the $B'$ and $C_{B'} \setminus B'$ of line 3. We say that a subblock $B_i$ of block $B$ on lines 13 to 22 is

– the *left block*, if $W(s, B') \neq 0$ and $W(s, C_{B'} \setminus B') = 0$ for every $s \in B_i$,
– a *middle block*, if $W(s, B') \neq 0$ and $W(s, C_{B'} \setminus B') \neq 0$ for every $s \in B_i$, and
– the *right block*, if $W(s, B') = 0$ for every $s \in B_i$.

This definition covers all subblocks of $B$, because $W(s, C_{B'})$ is the same for every $s \in B$ by the second invariant of Section 5. In particular, every state in the left block has the same $W(s, B')$, because it is $W(s, C_{B'})$. The union of the middle blocks of $B$ is called the *middle group*. The following lemma says an important fact about the middle groups.

**Lemma 5.** *If the middle groups are sorted with an $O(k \log k)$ sorting algorithm (such as heapsort or mergesort), then the total amount of time spent in sorting is $O(m \log n)$. This remains true even if each sorting operation processes also at most as many additional states as is the size of the middle group.*

*Proof.* Let $\#_\mathsf{c}(s)$ denote the number of compound blocks $C$ such that $s \rightarrow C$. Let $s\bullet = \{s' \mid s \rightarrow s'\}$. Clearly $\#_\mathsf{c}(s) \leq |s\bullet|$ and $\sum_{s \in S} \#_\mathsf{c}(s) \leq \sum_{s \in S} |s\bullet| = m$. Each time when $s$ is in a middle block, we have both $s \rightarrow B'$ and $s \rightarrow C_{B'} \setminus B'$, so $\#_\mathsf{c}(s)$ increases by one. As a consequence, if $\#_\mathsf{m}(s)$ denotes the number of times that $s$ has been in a middle block, then $\#_\mathsf{m}(s) \leq \#_\mathsf{c}(s)$. Therefore, $\sum_{s \in S} \#_\mathsf{m}(s) \leq \sum_{s \in S} \#_\mathsf{c}(s) \leq m$.

Let $K$ denote the total number of middle groups processed during the execution of the algorithm, and let $k_i$ be the size of the $i$th middle group. Thus $k_i \leq n$ and $\sum_{i=1}^{K} k_i = \sum_{s \in S} \#_\mathsf{m}(s) \leq m$. We have $\sum_{i=1}^{K} 2k_i \log(2k_i) \leq 2 \sum_{i=1}^{K} k_i \log(2n)$ $= 2\big(\sum_{i=1}^{K} k_i\big) \log(2n) \leq 2m \log(2n) = 2m \log n + 2m \log 2$. Therefore, the total amount of time spent in sorting the middle groups and at most an equal number of additional states with any $O(k \log k)$ sorting algorithm is $\sum_{i=1}^{K} O(2k_i \log(2k_i))$ $= O(m \log n)$. □

The $B_1$ on line 16 is the union of the left block and the middle group. Every state in the left block has the same $W(s, B')$. If the left block contains more states than the middle group, then line 16 assigns its $W(s, B')$ to *pmc*, line 17 separates it from the middle group, and line 19 only sorts the middle group. In the opposite case, $B_1$, and thus its subset $B_2$, contains at most twice as many states as the middle group. Both cases satisfy the assumptions of Lemma 5. This implies that the sorting operations take altogether $O(m \log n)$ time.

The memory consumption of every data structure is clearly $O(m)$ or $O(n)$, and the data structures for the blocks and $\bullet s'$ are $\Omega(n)$ and $\Omega(m)$. Heapsort and mergesort use $O(n)$ additional memory. We have proven the following theorem.

**Theorem 2.** *If the details are implemented as described above, then the algorithm in Figure 2 runs in $O(n + m \log n)$ time and $\Theta(n + m)$ memory.*

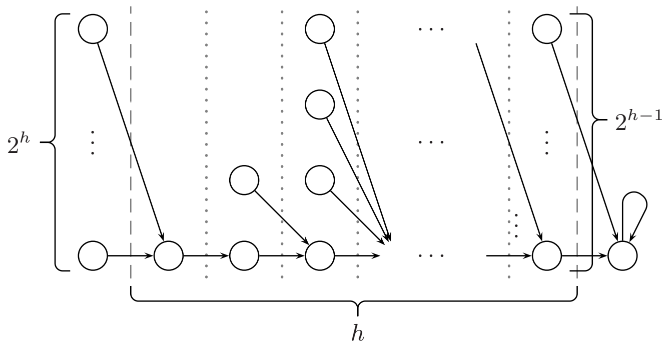

**Fig. 4.** An example where sorting the union of the middle and left blocks with a $\Theta(k \log k)$ algorithm costs too much. Each transition has weight 1.

Processing the possible majority candidate's block separately from $B_2$ is not necessary for correctness. We show now that it is necessary for guaranteeing the performance. Assume that a $\Theta(k \log k)$ sorting algorithm is applied to the union of the middle and left blocks. Consider the family of systems in Figure 4. In the figure, the initial partition is shown by dashed lines.

Assume that the initial block in the center is used as the first splitter. It splits itself into two halves along the rightmost dotted line. The leftmost half is used for further splitting, because it has $2^{h-1} - 1$ states, while the other half has $2^{h-1}$ states. When it is used as a splitter, it splits itself to two halves of sizes $2^{h-2}$ and $2^{h-2} - 1$ states. Again, the leftmost half is smaller. This repeats $h - 1$ times, plus one time which does not cause any splitting. Each time the leftmost initial block is processed as a left block. We have $h$ sorting operations on at least $2^h$ elements each, taking altogether $\Omega(h(2^h \log 2^h)) = \Omega(n \log^2 n) = \Omega(m \log^2 n)$ time, because $m = n = 2^{h+1}$. This is not $O(m \log n)$.

## 7 Testing and Measurements

Our lumping algorithm was implemented in C++ and tested in two different ways.

The first series of tests used as inputs more than 250 randomly generated graphs of various sizes, densities, numbers of initial blocks, and numbers of different transition weights. Unfortunately, there is no straightforward way of fully checking the correctness of the output. Therefore, each graph was given to the program in four different versions, and it was checked that the four outputs had the same number of states and the same number of transitions. Two of the versions were obtained by randomly permuting the numbering of states in the original version, and the first output was used as the fourth input. This is similar to the testing described in [11]. Indeed, the programs written for [11] were used as a starting point when implementing both the lumping program and the testing environment.

**Table 1.** Some timing measurements. The times are in seconds.

| | input | | output | | reading | lumping |
|---|---|---|---|---|---|---|
| source | states | transitions | states | transitions | input | algorithm |
| random | 30 000 | 1 000 000 | 29 982 | 995 044 | 7.3 | 0.7 |
| random | 30 000 | 1 000 000 | 29 973 | 952 395 | 6.9 | 1.0 |
| random | 30 000 | 1 000 000 | 1 | 0 | 6.3 | 0.3 |
| random | 30 000 | 10 000 000 | 29 974 | 9 950 439 | 71.4 | 7.5 |
| random | 30 000 | 10 000 000 | 29 931 | 9 522 725 | 68.9 | 7.6 |
| random | 30 000 | 10 000 000 | 1 | 0 | 63.9 | 3.6 |
| GreatSPN | 184 756 | 2 032 316 | 139 | 707 | 5.2 | 2.0 |
| GreatSPN | 646 646 | 7 700 966 | 139 | 707 | 21.2 | 32.8 |
| GreatSPN | 1 352 078 | 16 871 582 | 195 | 1 041 | 49.5 | 126.6 |
| GreatSPN | 2 704 156 | 35 154 028 | 272 | 1 508 | 111.3 | 825.4 |

The ability of the testing environment to reveal errors was tested by modifying the lumping program so that it initially puts one too few blocks into $\mathsf{U}_\mathsf{B}$. The testing environment detected the error quickly.

The upper part of Table 1 shows some running times on a laptop with 2 GiB of RAM and 1.6 GHz clock rate. The bottleneck in the tests was the capacity of the testing environment and the time spent in input and output, not the time spent by the lumping algorithm.

The Markov chains used in the second set of experiments were made with the GreatSPN tool [2,7] from a family of stochastic Petri net models. The nets exhibit symmetries, making it possible for GreatSPN to also compute the lumped Markov chains directly. The sizes of the results obtained by running our program on unlumped Markov chains produced by GreatSPN were compared to the sizes of lumped Markov chains produced directly by GreatSPN, and found identical. Correctness was also checked by computing some performance indices.

These experiments were made on a laptop with 2 GiB of RAM and 2.2 GHz clock rate. Their results are reported in the lower part of Table 1. They suggest that our program has good performance even with more than $10^6$ states and $10^7$ transitions.

## 8 Conclusions

We presented an $O(m \log n)$ time algorithm for the lumping problem, where $n$ is the number of states and $m$ is the number of transitions. It is not the first algorithm for this problem with this complexity. However, it is much simpler than its predecessor [6], because the use of splay trees was replaced by an application of just any $O(k \log k)$ sorting algorithm together with a simple possible majority algorithm. We also believe that our presentation is the first that is sufficiently detailed and non-misleading from the point of view of programmers. Thus we hope that this article is of value to solving the lumping problem in practice.

Our simplification is based on the observation that the sum of the sizes of the so-called middle blocks during the execution of the Paige–Tarjan algorithm is at most $m$. Therefore, the extra time taken by sorting them is so small that it does not add to the overall time complexity of $O(m \log n)$ of the Paige–Tarjan algorithm. We demonstrated with an example that this does not extend to so-called left blocks. As a consequence, the left blocks must often be processed separately. Fortunately, this was easy to do with the possible majority candidate algorithm.

Our algorithm does not implement the compound blocks of [10]. However, we used compound blocks extensively in the proofs. They are a handy way of keeping track of splitting that has already been done. Without referring to them it would be impossible to define the middle blocks and justify the correctness of the technique that underlies the good performance, that is, sometimes not putting some block into $\mathsf{U}_\mathsf{B}$. Compound blocks are thus essential for understanding the algorithm, although they are not explicitly present in it.

**Acknowledgments.** We thank the reviewers of this article for exceptionally many good comments.

## References

1. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading (1974)
2. Baarir, S., Beccuti, M., Cerotti, D., De Pierro, M., Donatelli, S., Franceschinis, G.: The GreatSPN tool: recent enhancements. SIGMETRICS Performance Evaluation Review, Special Issue on Tools for Performance Evaluation 36(4), 4–9 (2009)
3. Backhouse, R.C.: Program Construction and Verification. Prentice-Hall International Series in Computer Science, UK (1986)
4. Buchholz, P.: Exact and ordinary lumpability in finite Markov chains. Journal of Appl. Prob. 31, 309–315 (1994)
5. Derisavi, S.: Solution of Large Markov Models Using Lumping Techniques and Symbolic Data Structures. Dissertation, University of Illinois at Urbana-Champaign (2005)
6. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in Markov chains. Information Processing Letters 87(6), 309–315 (2003)
7. GRaphical Editor and Analyzer for Timed and Stochastic Petri Nets, `http://www.di.unito.it/%7egreatspn/` (last update September 25, 2008)
8. Kemeny, J.G., Snell, J.L.: Finite Markov Chains. Springer, Heidelberg (1960)
9. Knuutila, T.: Re-describing an algorithm by Hopcroft. Theoret. Comput. Sci. 250, 333–363 (2001)
10. Paige, R., Tarjan, R.: Three partition refinement algorithms. SIAM J. Comput. 16(6), 973–989 (1987)
11. Valmari, A.: Bisimilarity minimization in $O(m \log n)$ time. In: Franceschinis, G., Wolf, K. (eds.) PETRI NETS 2009. LNCS, vol. 5606, pp. 123–142. Springer, Heidelberg (2009)
12. Valmari, A., Lehtinen, P.: Efficient minimization of DFAs with partial transition functions. In: Albers, S., Weil, P. (eds.) STACS 2008, Symposium on Theoretical Aspects of Computer Science, Bordeaux, France, pp. 645–656 (2008), `http://drops.dagstuhl.de/volltexte/2008/1328/`