An efficient algorithm to determine probabilistic bisimulation

Citation for published version (APA):

DOI:
10.3390/a11090131

Document status and date:
Published: 05/09/2018

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Article

An Efficient Algorithm to Determine Probabilistic Bisimulation

Jan Friso Groote *, Jao Rivera Verduzco and Erik P. de Vink

Department of Mathematics and Computer Science, Eindhoven University of Technology, P.O. Box 512, 5600 MB Eindhoven, The Netherlands; H.J.Rivera.Verduzco@student.tue.nl (J.R.V.); evink@win.tue.nl (E.P.d.V.)

* Correspondence: j.f.groote@tue.nl; Tel.: +31-40-2475003

Received: 10 July 2018; Accepted: 27 August 2018; Published: 3 September 2018

Abstract: We provide an algorithm to efficiently compute bisimulation for probabilistic labeled transition systems, featuring non-deterministic choice as well as discrete probabilistic choice. The algorithm is linear in the number of transitions and logarithmic in the number of states, distinguishing both action states and probabilistic states, and the transitions between them. The algorithm improves upon the proposed complexity bounds of the best algorithm addressing the same purpose so far by Baier, Engelen and Majster-Cederbaum (Journal of Computer and System Sciences 60:187–231, 2000). In addition, experimentally, on various benchmarks, our algorithm performs rather well; even on relatively small transition systems, a performance gain of a factor 10,000 can be achieved.

Keywords: probabilistic system with nondeterminism; probabilistic labeled transition system; probabilistic bisimulation; partition-refinement algorithm

1. Introduction

In [1], Larsen and Skou proposed the notion of probabilistic bisimulation. Although described for deterministic transition systems, the same notion is also very suitable for probabilistic transition systems with nondeterminism [2,3], i.e. so-called PLTSs. It expresses that two states are equivalent exactly when the following condition holds: if one state can perform an action ending up in a set of states, each with a certain probability, and then the other state can do the same step ending up in an equivalent set of states with the same distribution of probabilities. Two characteristic nondeterministic transition systems of which the initial states are probabilistically bisimilar are given in Figure 1.

![Figure 1. Two probabilistically bisimilar nondeterministic transition systems.](image-url)
complexity $O(mn)$, where $n$ is the number of states and $m$ is the number of transitions (from states to distributions over states; there is no separate measure for the size of the distributions). As far as we know, it is the only practical algorithm for bisimulation à la Larsen-Skou for PLTSs. In essence, other algorithms for probabilistic systems typically target Markov chains without nondeterminism. The algorithm in [4] performs an iterative refinement of a partition of states and a partition of transitions per action label. The crucial point is splitting the groups of states based on probabilities. For this, a specific data structure is used, called augmented ordered balanced trees, to support efficient storage, retrieval and ordering of states indexed by probabilities.

In this paper, we provide a new algorithm for probabilistic bisimulation for PLTSs of time complexity $O\left((m_a + m_p) \log n_p + m_p \log n_d\right)$ and space complexity $O(m_a + m_p)$, where $n_d$ is the number of states, $m_a$ the number of transitions labelled with actions, $n_p$ the number of distributions and $m_p$ the cumulative support of the distributions. Our $n_d$ coincides with the $n$ of Baier et al. We prefer to use $m_a$, $n_p$, and $m_p$ over $m$ as the former support a more refined analysis. A detailed comparison between the algorithms reveals that, if the distributions have a positive probability for all states, the complexities of the algorithms are similar. However, when distributions only touch a limited number of states, as is often the common situation, the implementation of our algorithm outperforms our implementation of the algorithm in [4], both in time as well as in space complexity.

Similar to the algorithm of Baier et al., our algorithm keeps track of a partition of states and of distributions (referred to as action states and probabilistic states below) but in line with the classical Paige–Tarjan approach [5] it also maintains a courser partition of so-called constellations. The treatment of distributions in our algorithm is strongly inspired by the work for Markov Chain lumping by Valmari and Franceschinis, but our algorithm applies to the richer setting of non-deterministic labelled probabilistic transition systems. Using a brilliant, yet simple argument, taken from [6], the number of times a probabilistic transition is sorted can be limited by the fan-out of the source state of the transition. This leads to the observation that we can use straightforward sorting without the need of any tailored data structure such as augmented ordered balanced trees or similar as in [4,7]. Actually, our algorithm uses a simplification of the algorithm in [6] since the calculation of so-called majority candidates can be avoided, too.

We implemented both the new algorithm and the algorithm from [4]. We spent quite some effort to establish that both implementations are free from programming flaws. To this end, we ran them side-by-side and compared the outcomes on a vast amount of randomly generated probabilistic transition systems (in the order of millions). Furthermore, we took a number of examples from the field, among others from the PRISM toolset [8], and ran both implementations on the probabilistic transition systems that were obtained in this way. Time-wise, all benchmarks indicated better results for our algorithm compared to the algorithm from [4]. Even for rather small transition systems of about 100,000 states, performance gains of a factor 10,000 can be achieved. Memory-wise the implementation of our algorithm also outperforms the implementation in [4] when the sizes of the probabilistic state space are larger. Both findings are in line with the theoretical complexity analyses of both algorithms. Both implementations have been incorporated in the open source mCRL2 toolset [9,10].

1.1. Related Work

Probabilistic bisimulation preserves logic equivalence for PCTL [11]. In [12], Katoen c.s. reported up to logarithmic state space reduction obtained by probabilistic bisimulation minimisation for DTMCs. Quotienting modulo probabilistic bisimulation is based on the algorithm in [7]. In the same vein, Dehnert et al. proposed symbolic probabilistic bisimulation minimisation to reduce computation time for model checking PCTL in a setting for DTMCs [13], where an SMT solver is exploited to do the splitting of blocks. Partition reduction modulo probabilistic bisimulation is also used as an ingredient in a counter-example guided abstraction refinement approach (CEGAR) for model checking for PCTL by Lei Song et al. in [14].
For CTMCs, Hillston et al. proposed the notion of contextual lumpability based on lumpable bisimulation in [15]. Their reduction technique uses the Valmari–Franceschini algorithm for Markov chain lumping mentioned earlier. Crafa and Renzato [16] characterised probabilistic bisimulation of PLTSs as a partition shell in the setting of abstract interpretation. The algorithm for probabilistic bisimulation that comes with such a characterisation turns out to coincide with that in [4]. A similar result applies to the coalgebraic approach to partition refinement in [17] that yields a general bisimulation decision procedure, which can be instantiated with probabilistic system types.

Probabilistic simulation for PLTSs has been treated in [4], too. In [18], maximum flow techniques are proposed to improve the complexity. Zhang and Jansen [19] presented a space-efficient algorithm based on partition refinement for simulation between probabilistic automata, which improves upon the algorithm for simulation by Crafa and Renzato [16] for concrete experiments taken from the PRISM benchmark suite. A polynomial algorithm, essentially cubic, for deciding weak and branching simulation was provided and it is proven that it correctly calculates probabilistic bisimulation. This section ends with an elaborate example. In Section 4 we provide a detailed version the algorithm with a focus on the implementation details necessary to achieve the complexity. In Section 5, we provide some benchmarking results and a few concluding remarks are made in Section 6.

1.2. Synopsis

The structure of this article is as follows. In Section 2, we provide the notions of a probabilistic transition system as well as that of probabilistic bisimulation. In Section 3, the outline of our algorithm is provided and it is proven that it correctly calculates probabilistic bisimulation. This section ends with an elaborate example. In Section 4 we provide a detailed version the algorithm with a focus on the implementation details necessary to achieve the complexity. In Section 5, we provide some benchmarking results and a few concluding remarks are made in Section 6.

2. Preliminaries

Let $S$ be a finite set. A distribution $f$ over $S$ is a function $f : S \rightarrow [0, 1]$ such that $\sum_{s \in S} f(s) = 1$. For each distribution $f$, its support is the set $\{ s \in S \mid f(s) > 0 \}$. The size of $f$ is defined as the number of elements in its support, written as $|f|$. The set of all distributions over a set $S$ is denoted by $\mathcal{D}(S)$. Distributions are lifted to act on subsets $T \subseteq S$ by $f[T] = \sum_{s \in T} f(s)$.

For an equivalence relation $R$ on $S$, we use $S/R$ to denote the set of equivalence classes of $R$. We define $s / R = \{ t \in S \mid s R t \}$ and, for a subset $T$ of $S$, we define $T / R = \{ s \in S \mid \exists t \in T: s R t \}$. A partition $\pi = \{ B_i \subseteq S \mid i \in I \}$ is a set of non-empty subsets such that $B_i \cap B_j = \emptyset$ for all $i, j \in I$ and $\bigcup_{i \in I} B_i = S$. Each $B_i$ is called a block of the partition. Slightly ambiguously, we use $S/R$ to denote the set of equivalence classes of $R$ with respect to $S$. Clearly, the set of equivalence classes of $R$ forms a partition of $S$. Reversely, a partition $\pi$ of $S$ induces an equivalence relation $R_\pi$ on $S$, by $sR_\pi t$ iff $s, t \in B$ for some block $B$ of $\pi$. A partition $\pi$ is called a refinement of a partition $\varrho$ iff each block of $\pi$ is a subset of a block of $\varrho$. Hence, each block in $\varrho$ is a disjoint union of blocks from $\pi$.

We use probabilistic labeled transition systems as the canonical way to represent the behaviour of systems.

**Definition 1.** (Probabilistic Labeled Transition System). A probabilistic labeled transition system (PLTS) for a set of actions $\mathcal{A}$ is a pair $A = (S, \rightarrow)$ where

- $S$ is a finite set of states, and
- $\rightarrow \subseteq S \times \mathcal{A} \times \mathcal{D}(S)$ is a finite transition relation relating states and actions to distributions.

It is common to write $s \xrightarrow{a} f$ for $(s, a, f) \in \rightarrow$. For $s \in S$, $a \in \mathcal{A}$, and a set $F \subseteq \mathcal{D}(S)$ of distributions, we write $s \xrightarrow{a} F$ if $s \xrightarrow{a} f$ for some $f \in F$. Similarly, we write $s \xrightarrow{\varrho} F$ if there is no distribution $f \in F$ such that $s \xrightarrow{a} f$. For the presentation below, we associate a so-called probabilistic state $u_f$ with each distribution $f$ provided there is some transition $s \xrightarrow{a} f$ of $\mathcal{A}$. We write $U$ for $\{ u_f \mid \exists s \in S, a \in \mathcal{A}: s \xrightarrow{a} f \}$, with typical element $u$. Note that, since $\rightarrow$ is finite, $U$ is also finite.
We also use the notation \( s \xrightarrow{u} f \) if \( s \xrightarrow{u} f \) for some \( f \in \mathcal{D}(S) \). As a matter of notation, we write \( u_f[T] \) for \( f[T] \) if probabilistic state \( u_f \) corresponds to the distribution \( f \). We sometimes use a so-called probabilistic transition \( u_f \xrightarrow{a} p \) for \( 0 < p \leq 1 \) and \( s \in S \) if \( u_f(s) = p \). To stress \( S \cap U = \emptyset \), we refer to states \( s \in S \) as action states.

Below, in particular in the complexity analysis, we use \( n_a = |S| \) as the number of action states, \( n_p = |U| \) as the number of probabilistic states, \( m_a = |\rightarrow| \) as the number of action transitions and \( m_p = \sum_{u_f \in U} |f| \) as the cumulative size of the support of the distributions corresponding to all probabilistic states. Note that \( m_p \geq n_p \) as every distribution has support of at least size 1.

The following definition for probabilistic bisimulation stems from [1].

**Definition 2. (Probabilistic Bisimulation).**

Consider a PLTS \( \mathcal{A} = (S, \rightarrow) \). An equivalence relation \( R \subseteq S \times S \) is called a probabilistic bisimulation for \( \mathcal{A} \) iff for all states \( s, t \in S \) such that \( s \xrightarrow{a} f \) and \( s \xrightarrow{a} g \), for some action \( a \in \text{Act} \) and distribution \( f \in \mathcal{D}(S) \), it holds that \( t \xrightarrow{a} g \) for some distribution \( g \in \mathcal{D}(S) \), and \( f[B] = g[B] \) for each \( B \in S / R \).

Two states \( s, t \in S \) are probabilistically bisimilar iff a probabilistic bisimulation \( R \) for \( \mathcal{A} \) exists such that \( s \xrightarrow{a} f \) and \( t \xrightarrow{a} g \), which we write as \( s \simeq_p t \). Two distributions \( f, g \in \mathcal{D}(S) \), and similarly two probabilistic states \( u_f, u_g \in U \), are probabilistically bisimilar iff for all \( B \in S / \simeq_p \) it holds that \( f[B] = g[B] \), which we also denote by \( f \simeq_p g \) and \( u_f \simeq_p u_g \), respectively.

By definition, probabilistic bisimilarity is the union of all probabilistic bisimulations. To be able to speak of probabilistically bisimilar distributions (or of probabilistically bisimilar probabilistic states), probabilistic bisimilarity needs to be an equivalence relation. In fact, probabilistic bisimilarity is a probabilistic bisimulation. See [22] for a proof.

3. A Partition Refinement Algorithm for Probabilistic Bisimulation (Outline)

Many efficient algorithms for standard bisimulation calculate partitions of states [5,23,24]. Here, we consider the construction of a partition \( \mathcal{B} \) of the sets of action states \( S \) and of probabilistic states \( U \) for some fixed PLTS \( \mathcal{A} \) over a set of actions \( \text{Act} \). Below blocks of the partition always contain either action states or probabilistic states.

3.1. Stability of Blocks and Partitions

An important notion underlying the algorithm introduced below is that of the stability of a block of a partition. If a block is not stable, it contains states that are not bisimilar. These states either have different transitions or different distributions. We first define the notion of stability more generically on sets instead of on blocks. Then, we lift it to partitions.

**Definition 3. (Stable Sets and Partitions).**

1. A set of action states \( B \subseteq S \) is called stable under a set of probabilistic states \( C \subseteq U \) with respect to an action \( a \in \text{Act} \) iff \( s \xrightarrow{a} C \) whenever \( t \xrightarrow{a} C \) and vice versa for all \( s, t \in B \). The set \( B \) is called stable under \( C \) iff \( B \) is stable under \( C \) with respect to all actions \( a \in \text{Act} \).
2. A set of probabilistic states \( B \subseteq U \) is called stable under a set of action states \( C \subseteq S \) iff \( u[C] = v[C] \) for all \( u, v \in B \).
3. A set of states \( B \) with \( B \subseteq S \), respectively \( B \subseteq U \), is called stable under a partition \( \mathcal{C} \) of \( S \cup U \), with \( C \subseteq S \) or \( C \subseteq U \) for all \( C \in \mathcal{C} \), iff \( B \) is stable under each \( C \in \mathcal{C} \) with \( C \subseteq U \), respectively \( C \subseteq S \).
4. A partition \( \mathcal{B} \) is called stable under a partition \( \mathcal{C} \) iff all blocks \( B \) of \( \mathcal{B} \) are stable under \( \mathcal{C} \).

There are two simple but important properties stating that stability is preserved when splitting sets. The first one says that subsets of stable sets are also stable.
Lemma 1. Let \( B \subseteq S \) be a set of action states and \( C \subseteq U \) a set of probabilistic states. If \( B \) is stable under \( C \), then any \( B' \subseteq B \) is also stable under \( C \). Similarly, if \( C \) is stable under \( B \), then any \( C' \subseteq C \) is also stable under \( B \).

Proof. We only prove the first part as the argument for the second part is essentially the same. If \( s, t \in B' \), then also \( s, t \in B \). As \( B \) is stable under \( C \), it holds that for every action \( a \in \text{Act} \) either both satisfy \( s \xrightarrow{a} C \) and \( t \xrightarrow{a} C \), or neither does. Thus, \( B' \) is stable under \( C \). \( \square \)

The second property says that splitting a set in two parts can only influence the stability of an other set if there is a transition or a positive probability from this other set to one of the parts of the split set.

Lemma 2. Let \( B \subseteq S \) be a set of action states and \( C \subseteq U \) a set of probabilistic states.

1. Suppose \( B \) is stable under \( C \) with respect to an action \( a, C' \subseteq C \), and there is no \( s \in B \) such that \( s \xrightarrow{a} C' \). Then, \( B \) is stable under \( C' \) and \( C \setminus C' \) with respect to \( a \).
2. Suppose \( C \) is stable under \( B, B' \subseteq B \), and \( u[B'] = 0 \) for all \( u \in C \). Then, \( C \) is stable under \( B' \) and \( B \setminus B' \).

Proof. We only provide the proof for the first part of this lemma. If \( s, t \in B \), then both \( s \xrightarrow{a} C' \) and \( t \xrightarrow{a} C' \) by assumption. Thus, \( B \) is stable under \( C' \) with respect to \( a \). Furthermore, \( B \) is stable under \( C \setminus C' \): Suppose \( s, t \in B \) and \( s \xrightarrow{a} C \setminus C' \). Thus, \( t \xrightarrow{a} C \) as \( B \) is stable under \( C \), \( t \xrightarrow{a} C \), and by assumption \( t \xrightarrow{a} C' \). Therefore, \( t \xrightarrow{a} C \setminus C' \). Suppose \( s \xrightarrow{a} C \setminus C' \). Then, also \( s \xrightarrow{a} C \). As \( B \) is stable under \( C \), \( t \xrightarrow{a} C \) and hence, \( t \xrightarrow{a} C \setminus C' \). \( \square \)

The following property, called the stability property, says that a partition stable under itself induces a probabilistic bisimulation. In general, partition based algorithms for bisimulation search for such a stable partition.

Lemma 3. Stability Property. Let \( A = (S, \rightarrow) \) be a PLTS. If a partition \( B \) for \( A \) is stable under itself, then the corresponding equivalence relation \( B \) on \( S \) is a probabilistic bisimulation.

Proof. By the first condition of Definition 3 and stability of all blocks in \( B \) we have that either \( B \subseteq S \) or \( B \subseteq U \), for each block \( B \in B \). We write \( sBt \) iff \( s, t \in B \) for some \( B \in B \). Note that used in this way \( B \) is an equivalence relation on \( S \).

Suppose \( sBf \) for some \( s, t \in S \) and \( s \xrightarrow{f} f \). Let \( u \in U \) correspond to \( f \). Say \( s, t \in B \) and \( u \in B' \) for some blocks \( B, B' \in B \). Then, \( s \xrightarrow{u} B' \). By stability of \( B \) for \( B' \), it follows that \( t \xrightarrow{u} B' \). Hence, \( \nu \in B' \) and \( g \in D(S) \) exist such that \( \nu \) corresponds to \( g \) and \( s \xrightarrow{g} g \). Therefore, for any block \( B'' \in B \) we have \( f[B''] = u[B''] = \nu[B''] = g[B''] \) since the block \( B' \) of \( u \) and \( \nu \) is stable under each block \( B'' \) of \( B \).

Thus, the stable partition \( B \) induces an equivalence relation that satisfies the conditions for a probabilistic bisimulation of Definition 2, as was to be shown. \( \square \)

3.2. Outline of the Algorithm

We present our algorithm in two stages. An abstract description of the algorithm is presented as Algorithm 1; the detailed algorithm is provided as Algorithm 2. The set-up of Algorithm 1 is a fairly standard, iterative refinement of a partition \( B \), in this particular case containing both action states and probabilistic states, which are treated differently. In addition, following the approach of Paige and Tarjan [5], we maintain a coarser partition \( C \), which we call the set of constellations. Each constellation in partition \( C \) is a union of one or more blocks of \( B \), thus \( B \) is a refinement of \( C \). A constellation \( C \subseteq C \) that consists of exactly one block in \( B \) is called trivial. We refine partitions \( B \) and \( C \) until \( C \) only contains trivial constellations (see Line 5 of Algorithm 1).
Algorithm 1 Abstract Partition Refinement Algorithm for Probabilistic Bisimulation.

1: function PARTITION-REFINEMENT
2:  \( C := \{ S, U \} \)
3:  \( B := \{ U \} \cup \{ S_A \mid A \subseteq \text{Act} \} \)
4:  where \( S_A = \{ s \in S \mid \forall a \in A \exists u \in U : s \overset{a}{\rightarrow} u \} \)
5: while \( C \) contains a non-trivial constellation \( C \) do
6:  choose block \( B_C \) from \( B \) in \( C \)
7:  replace in \( C \) constellation \( C \) by \( B_C \) and \( C \backslash B_C \)
8: if \( C \) contains probabilistic states then
9:  for all blocks \( B \) of action states in \( B \) unstable under \( B_C \) or \( C \backslash B_C \) do
10:  refine \( B \) by splitting \( B \) into blocks of states with the same actions into \( B_C \) and \( C \backslash B_C \)
11: end for
12: else
13:  for all blocks \( B \) of probabilistic states in \( B \) unstable under \( B_C \) do
14:  refine \( B \) by splitting \( B \) into blocks of states with equal probabilities into \( B_C \)
15: end for
16: end if
17: end while
18: return \( B \)

Algorithm 2 Partition Refinement Algorithm for Probabilistic Bisimulation

1: function PARTITION-REFINEMENT \(( S, U, \rightarrow) \)
2:  \( C := \{ S, U \} \)
3:  \( B := \{ U \} \cup \{ S_A \mid A \subseteq \text{Act} \} \)
4:  where \( S_A = \{ s \in S \mid \forall a \in A \exists u \in U : s \overset{a}{\rightarrow} u \} \)
5: group the incoming action transitions in each block per label \( \rightarrow \) \( O(m_a) \)
6: initialise state_to_constellation_cnt for each transition \( \rightarrow \) \( O(m_a) \)
7: while \( C \) contains a non-trivial constellation \( C \) do
8:  \( \leq n \) iterations
9: choose a block \( B \) from \( B \) in \( C \) such that \( |B_C| \leq \frac{1}{2} |C| \)
10: split constellation \( C \) into \( B_C \) and \( C \backslash B_C \) in \( C \)
11: if \( C \) contains probabilistic states then
12:  for all incoming actions \( a \) of states in \( B_C \) do
13:  \( \leq |\text{Act}| \) iterations
14:  \((B_a, \text{left}_a, \text{mid}_a, \text{right}_a) := \text{aMark}(B, C, B_C, a) \) \( O(nr \text{ of incoming } a \text{ transitions in } B_C) \)
15: for all blocks \( B \in B_a \) do
16:  for all non-empty subsets \( B' \subseteq B \), different from \( \text{left}_a(B), \text{mid}_a(B), \text{right}_a(B) \) do
17:  move \( B' \) out of \( B \) and add \( B' \) as new block to \( B \)
18: else
19:  \((B_p, \text{left}_p, \text{mid}_p, \text{right}_p) := \text{pMark}(B, C, B_C) \) \( O(nr \text{ of incoming prob. transitions in } B_C) \) plus a sorting penalty
20: for all blocks \( B \in B_p \) do
21:  for all non-empty sets of states \( B' \subseteq B \) not equal to \( \text{left}_p(B), \text{mid}_p(B), \text{right}_p(B) \) do
22:  move \( B' \) out of \( B \) and add \( B' \) as a new block to \( B \)
23: end for
24: end if
25: end while
26: return \( B \)

Among others, we preserve the invariant that the blocks in partition \( B \) are always stable under partition \( C \). If all constellations in \( C \) are trivial, then the partitions \( B \) and \( C \) coincide. Hence, the blocks in \( B \) are stable under itself, and according to Lemma 3 we have found a probabilistic bisimulation. Our algorithm works by iteratively refining the set of constellations \( C \). When refining \( C \), we must also refine \( B \) to preserve the above mentioned invariant.

Since the set of states of a PLTS is finite (cf. Definition 1) refinement of the partitions \( B \) and \( C \) cannot be repeated indefinitely. Thus, termination of the algorithm is guaranteed. The partition consisting of singletons of action states and of probabilistic states is the finest that can be obtained, but
Theorem 1. Consider the partition \( B \) resulting from Algorithm 1. We find that (i) two action states are in the same block of \( B \) iff they are probabilistically bisimilar, and (ii) two probabilistic states are in the same block of \( B \) iff they are probabilistically bisimilar.

Proof. Upon termination, because of the while loop of Algorithm 1, all constellations of \( C \) are trivial, i.e., each constellation in \( C \) consists of exactly one block of \( B \). Hence, by Invariant 2, the partitions \( B \) and \( C \) coincide. Thus, by Invariant 3, each block of \( B \) is stable under each block in \( B \). In other words, partition \( B \) is stable under itself.

By the Stability Property of Lemma 3, we have that \( B \) is a probabilistic bisimulation on \( S \). It follows that two action states in the same block of \( B \) are probabilistically bisimilar. Reversely, by Invariant 1, probabilistically bisimilar action states are in the same block of \( B \). Thus, \( \simeq_p \) and \( B \) coincide on \( S \). In other words, two action states are in the same block of \( B \) iff they are probabilistically bisimilar.

To compare \( \simeq_p \) and the relation \( B \) on \( U \), choose probabilistic states \( u, v \in U \) such that \( u \not\simeq_p v \). Thus, \( u \) and \( v \) are in the same block of \( B \). By stability of block \( B \) for \( B \) it follows that \( u[B'] = v[B'] \), for each block \( B' \subseteq S \). Since \( \simeq_p \) and \( B \) coincide on \( S \) this implies \( u[B'] = v[B'] \) for all \( B' \in S/\simeq_p \). Thus, we have \( u \not\simeq_p v \). Reversely, if \( u \simeq_p v \), we have \( u, v \in B \) for some block \( B \) of \( B \) by Invariant 1. Thus, two probabilistic states are in the same block of \( B \) iff they are probabilistically bisimilar.

It is worth noting that in Line 5 of Algorithm 1 an arbitrary non-trivial constellation is chosen and in Line 6 an arbitrary block \( B_C \) is selected from \( C \) (we later put a constraint on the choice of \( B_C \)). In general, there are many possible choices and this influences the way the final partition is calculated. The previous theorem indicates that the final partition is not affected by this choice, neither is the complexity upper-bound, see Section 4.6. However, it is conceivable that practical runtimes can be improved by choosing the non-trivial constellation \( C \) and the block \( B_C \) optimally.
3.3. Refining the Set of Constellations and Restoring the Invariants

As we see from the high-level description of the partition refinement Algorithm 1, a non-trivial constellation \( C \) and a constituent block \( B_C \) are chosen (Lines 5 and 6) and \( C \) is replaced in \( C \) by the smaller constellations \( B_C \) and \( C \setminus B_C \) (Line 7). This preserves Invariants 1 and 2, but Invariant 3 may be violated as stability under \( B_C \) or \( C \setminus B_C \) (or both) may be lost: On the one hand, it may be the case that two actions states \( s \) and \( t \) both have an \( a \)-transition into \( C \), but \( s \) may have one to \( B_C \) but \( t \) to \( C \setminus B_C \) only or vice versa. On the other hand, it may be the case that two probabilistic states \( u \) and \( v \) yield the same value for \( C \) as a whole, i.e. \( u[C] = v[C] \), but by no means this needs to hold for \( B_C \) or \( C \setminus B_C \), i.e. \( u[B_C] \neq v[B_C] \) and \( u[C \setminus B_C] \neq v[C \setminus B_C] \). Therefore, in the remainder of the body of Algorithm 1, the blocks that are unstable under \( B_C \) and \( C \setminus B_C \) are split such that Invariant 3 is restored, both for blocks of actions states (Lines 9 and 10) and for blocks of probabilistic states (Lines 13 and 14). In the next section, the detailed Algorithm 2 describes how this is done precisely.

The general situation when splitting a block \( B \) for a constellation \( C \) containing a block \( B_C \) is depicted in Figure 2, at the left where \( B \) contains action states and at the right where \( B \) consists of probabilistic states. We first consider the case at the left.

![Figure 2. Splitting a non-stable block \( B \) into left, middle and right.](image)

In this case, block \( B \subseteq S \) is stable under constellation \( C \subseteq U \) and \( C \) is non-trivial. Thus, \( C \) properly contains a block \( B_C \) of \( B \), and we distinguish two non-empty subsets of \( C \), the block \( B_C \) on its own and the remaining blocks together in \( C \setminus B_C \). As \( B \) is stable under \( C \), the block \( B \) can only be unstable under \( B_C \) or \( C \setminus B_C \) if there is an action \( a \in \text{Act} \) and a state \( s \in B \) such that \( s \xrightarrow{a} B_C \) (Lemma 2.1). Thus, we only investigate and split blocks, for which such a transition \( s \xrightarrow{a} B_C \) exists.

We can restore stability by splitting \( B \) into the following three subsets:

\[
\begin{align*}
\text{left}_a(B) &= \{ s \in B \mid s \xrightarrow{a} B_C \wedge s \xrightarrow{a} C \setminus B_C \}, \\
\text{mid}_a(B) &= \{ s \in B \mid s \xrightarrow{a} B_C \wedge s \xrightarrow{a} C \setminus B_C \}, \text{ and} \\
\text{right}_a(B) &= \{ s \in B \mid s \xrightarrow{a} B_C \wedge s \xrightarrow{a} C \setminus B_C \}.
\end{align*}
\]

Note that the remaining set \( \{ s \in B \mid s \xrightarrow{a} B_C \wedge s \xrightarrow{a} C \setminus B_C \} \) must be empty; if not, this would imply that there is some action state \( t \) such that \( t \xrightarrow{a} C \). However, due to the existence of state \( s \) such that \( s \xrightarrow{a} B_C \), this would mean that block \( B \) is unstable under \( C \), contradicting Invariant 3.
Checking that the sets left$_a(B)$, mid$_a(B)$, right$_a(B)$ are stable under $C$ is immediate. As subsets of stable sets are also stable (Lemma 1) and $B$ is stable all other configurations of $C$, the sets left$_a(B)$, mid$_a(B)$, and right$_a(B)$ are stable under all other configurations of $C$ too.

Note that, due to the existence of state $s$ with $s \xrightarrow{a} B_C$, it is not possible that both left$_a(B)$ and mid$_a(B)$ are equal to the empty set. It is however possible that left$_a(B) = B$ or mid$_a(B) = B$, leaving the other two sets empty.

Lines 9 and 10 can now be read as follows. For all $a \in Act$, investigate all blocks $B$ such that there is an action state $s \in B$ with $s \xrightarrow{a} B_C$ as these blocks are the only candidates to be unstable. Replace each such block $B$ in $B$ by $\left\{ \text{left}_a(B), \text{mid}_a(B), \text{right}_a(B) \right\} \setminus \emptyset$ to restore stability under $B_C$ and $C \setminus B_C$.

Invariants 1 and 2 are preserved as these blocks are the only candidates to be unstable. Replace $C$ by splitting $B$. For Invariant 2, this is trivial by construction. For Invariant 1, note that the states in different blocks among left$_a(B)$, mid$_a(B)$, right$_a(B)$ cannot be probabilistically bisimilar as they have unique transitions to states $B_C$ and $C \setminus B_C$ and these target states cannot be bisimilar by Invariant 1. Thus, if two states of $B$ are probabilistically bisimilar then both are in the same subset left$_a(B)$, mid$_a(B)$, or right$_a(B)$ of $B$.

We next turn to the case of a set of probabilistic states $B$, see the right-side of Figure 2. Again, we assume that the non-trivial constellation $C$ is replaced by its two non-empty subsets $B_C$ and $C \setminus B_C$. As in the previous case, although the block $B$ is stable under the constellation $C$, this may not be the case under the subsets $B_C$ and $C \setminus B_C$.

To restore stability, we now consider for all $q$, $0 \leq q \leq 1$, the sets

$$B_q = \{ u \in B \mid u(B_C) = q \}.$$

Note that, for finitely many $q \in [0,1]$, we have $B_q \neq \emptyset$. Observe that each set $B_q$ is stable under $B_C$ as by construction $u(B_C) = v(B_C) = q$ for any $u, v \in B_q$. The set $B_q$ is also stable under $C \setminus B_C$. To see this consider two states $u, v \in B_q$. As block $B \subseteq U$ is stable under constellation $C \subseteq S$, $u(C) = v(C)$. Hence, $u(C \setminus B_C) = u(C) - u(B_C) = v(C) - v(B_C) = v(C \setminus B_C)$. By Lemma 1, the new blocks $B_q$ are also stable under the other constellations in $C$.

According to Lemma 2.2, only those blocks $B$ that contain a probabilistic state $u \in B$ such that $u(B_C) > 0$ can be unstable under $B_C$ and $C \setminus B_C$. Thus, at Line 13 of Algorithm 1 we consider all those blocks $B$ and replace each of them by the non-empty subsets $B_q$, $0 \leq q \leq 1$ at Line 14 in $B$. This makes the partition stable again under all constellations in $C$, in particular under the new constellations $B_C$ and $C \setminus B_C$.

Again, it is straightforward to see that Invariants 1 and 2 are not violated by replacing the block $B$ by the blocks $B_q$. For Invariant 1, if states are probabilistically bisimilar in $B$, they remain in the same block $B_q$. For Invariant 2, as $B$ is refined, partition $B$ remains a refinement of partition $C$.

For the detailed algorithm in Section 4, it is required to group the sets $B_q$ as follows: left$_p(B) := B_0$, right$_p(B) := B_1$, and mid$_p(B) = \{ B_q \mid 0 < q < 1 \}$. This does not play a role here, but left$_p(B)$, mid$_p(B)$, and right$_p(B)$ are already indicated in Figure 2, in particular mid$_p(B) = \{ B_{1/4}, B_{1/2}, B_{3/4} \}$.

### 3.4. An Example

We provide an example to illustrate how Algorithm 1 calculates partitions.

**Example 1.** Consider the PLTS given in Figure 3. We provide a detailed account of the partitions that are obtained when calculating probabilistic bisimulation. The obtained partitions are listed in Table 1. In the lower table, nine partitions together with their constellations are listed that are generated for a run of Algorithm 1. In the upper table the blocks that occur in these partitions are defined. Observe that we put the blocks and constellations with action states and probabilistic states in different columns. This is only for clarity, as in the current partition and the current set of constellations they are joined.
Algorithms 2018, 11, 131

Table 1. The generated partitions for the PLTS of Example 1.

<table>
<thead>
<tr>
<th>Blocks of Actions States</th>
<th>Blocks of Probabilistic States</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_0 = {t_1, t_3, t_4, t_6, t_7, r_1, \ldots}$</td>
<td>$U_0 = {u_{1-6}, v_{1-5}}$</td>
</tr>
<tr>
<td>$S_1 = {s_1, \ldots}$</td>
<td>$U_1 = {u_1, u_5, v_{1-5}}$</td>
</tr>
<tr>
<td>$S_2 = {t_2, t_5, t_8, t_9}$</td>
<td>$U_2 = {u_{2, 4}}$</td>
</tr>
<tr>
<td>$S_3 = {t_{10}}$</td>
<td>$U_3 = {u_5, u_6}$</td>
</tr>
<tr>
<td>$S_4 = {s_1, s_2}$</td>
<td>$U_4 = {u_5}$</td>
</tr>
<tr>
<td>$S_5 = {s_3, s_4}$</td>
<td>$U_5 = {u_6}$</td>
</tr>
<tr>
<td>$S_6 = {s_3}$</td>
<td></td>
</tr>
<tr>
<td>$S_7 = {s_4}$</td>
<td></td>
</tr>
</tbody>
</table>

Algorithm 1 starts with four blocks of action states, $S_0$ to $S_3$, which contain the action states with no outgoing transitions and those with an outgoing transition labelled with $a$, with $b$, and with $c$, respectively. In the algorithm, all probabilistic states are initially collected in block $U_0$. There are two constellations, viz. $S_0 \cup S_1 \cup S_2 \cup S_3$ and $U_0$. These initial partitions are listed in Row 0 of the lower part of Table 1.

Since the constellation with action states is non-trivial we split it, rather arbitrarily, in $S_0$ and $S_1 \cup S_2 \cup S_3$. The block $U_0$ is not stable under $S_0$ and $S_1 \cup S_2 \cup S_3$ and is split in $U_1 = \{u_1, u_3, v_{1-5}\}$, $U_2 = \{u_{2, 4}\}$ and $U_3 = \{u_5, u_6\}$. This is because we have $u[S_0] = 1$ for $u$ equal to $u_1$, $u_3$, and $v_{1}$ to $v_5$; we have $u[S_0] = \frac{1}{2}$ for $u$ equal to $u_2$ and $u_4$; we have $u[S_0] = 0$ and $u[S_0] = 0$. The resulting partitions are listed at Row 1 in Table 1.

For the second iteration, we consider the non-trivial constellation $S_1 \cup S_2 \cup S_3$ and split it into $S_1$ and $S_2 \cup S_3$. Note, the action states $s_1$ to $s_4$ in $S_1$ do not have incoming transitions. Consequently, for all $u \in U_1$, we have $u[S_1] = 0$; for all $u \in U_2$ we have $u[S_1] = 0$; for all $u \in U_3$ we have $u[S_1] = 0$. Thus, all blocks of probabilistic states are stable under $S_1$ and $S_2 \cup S_3$. Hence, no block is split.

In the third iteration, we split the non-trivial constellation $S_2 \cup S_3$ into $S_2$ and $S_3$. For all, $u \in U_1$ we have $u[S_2] = 0$. Thus, $U_1$ is stable under $S_2$ and $S_3$. For $U_2$, the probabilistic states $u_2$ and $u_4$ agree on the value $\frac{1}{2}$ for $S_2$, hence for $S_3$ too. Thus, $U_2$ is stable as well. However, for $u_5$ and $u_6$ in $U_3$ we have $u_5[S_2] = 1$ and $u_6[S_2] = \frac{1}{2}$. Therefore, $U_1$ needs to be split in $U_4 = \{u_5\}$ and $U_5 = \{u_6\}$.

At this point, all constellations with actions states are trivial, so at iteration 4 we turn to the non-trivial constellation of probabilistic states $U_1 \cup U_2 \cup U_4 \cup U_5$ and split it into $U_1$ and $U_2 \cup U_4 \cup U_5$. Block $S_0$ is stable since each of its states has no transitions at all. Block $S_1$ is not stable: $s_1 \rightarrow U_1$ and $s_2 \rightarrow U_1 \cup U_4 \cup U_5$, but $s_3, s_4 \rightarrow U_1$ and $s_3, s_4 \rightarrow U_2 \cup U_4 \cup U_5$. Thus, $S_1$ needs to be split into $S_4 = \{s_1, s_2\}$ and $S_5 = \{s_3, s_4\}$. Block $S_2$ is stable since its states have only $b$-transitions into $U_1$. Block $S_3$ is a singleton and therefore cannot be split.

The following iteration, iteration 5, sets $U_2$ and $U_4 \cup U_5$ apart as constellations. Again, in absence of transitions, block $S_0$ is stable under $U_2$ and $U_4 \cup U_5$. The same holds for $S_2$ that has only $b$-transitions into $U_0$. Block $S_3$ can be ignored. For $S_4$, both $s_1$ and $s_2$ have an $a$-transition into $U_2$ as their only transition. Hence, block $S_4$ is stable. Similarly, $S_5$ is stable, as its states $s_3$ and $s_4$ both have an
According to the Stability Property, Lemma 3, the corresponding equivalence relation is a probabilistic equivalence relation. All probabilistic states have value 0 for both \( u \) and \( v \). This requires time linear in the number of considered action transitions. Note that the number of probabilistic states \( u \) and \( v \) is needed.

Returning to constellations of actions states, at Iteration 7, we split \( S_4 \cup S_6 \cup S_7 \) over \( S_4 \) and \( S_6 \cup S_7 \). All probabilistic states have value 0 for both \( S_4 \) and \( S_6 \cup S_7 \), hence no split of probabilistic blocks is needed.

This is similar in Iteration 8, where the non-trivial constellation \( S_6 \cup S_7 \) is split, and none of the blocks become unstable. Now, all constellations are trivial and the algorithm terminates. According to the Stability Property, Lemma 3, the corresponding equivalence relation is a probabilistic bisimulation. Thus, the final partition is \( \{ S_0, S_2, S_3, S_4, S_6, S_7, U_1, U_2, U_4, U_5 \} \). Moreover, the deadlock states \( t_1, t_3, t_4, t_6, t_7 \) and \( r_1 \) to \( r_5 \) are probabilistically bisimilar, the states \( t_2, t_5, t_8, t_9 \) that have only a \( b \)-transition into a Dirac distribution to deadlock are probabilistically bisimilar, the states \( s_1 \) and \( s_2 \) are probabilistically bisimilar (which is clear when identifying states \( t_7 \) and \( t_8 \)), whereas the remaining action states \( s_3, s_4 \) and \( t_{10} \) have no probabilistically bisimilar counterpart. For the probabilistic states the states \( u_1, u_3 \) and \( v_1 \) to \( v_3 \) are identified by probabilistic bisimulation. This also holds for the probabilistic states \( u_2 \) and \( u_4 \). Probabilistic states \( u_5 \) and \( u_6 \) each have no probabilistically bisimilar counterpart.

4. A Partition-Refinement Algorithm for Probabilistic Bisimulation (Detailed)

Algorithm 1 gives an outline but leaves many details implicit. The detailed refinement-partition algorithm is presented in this section as Algorithm 2. It has the same structure as Algorithm 1, but in this section we focus on how to efficiently calculate whether and how blocks must be split, and how this split is actually carried out. We first explain grouping of action transitions per action, next we introduce various data structures that are used by the algorithm, subsequently we explain how the algorithm is working line-by-line, and finally we give an account of its complexity.

4.1. Grouping Action Transitions per Action Label

To obtain the complexity bound of our algorithm, it is essential that we can group action transitions by actions linearly in the number of transitions. Grouping means that the action transitions with the same action occur consecutively in this ordering. It is not necessary that the transitions are ordered according to some overall ordering.

We assume that \(|\text{Act}| \leq m_a\) and that the actions in \( \text{Act} \) are consecutively numbered. Recall, \( m_a \) denotes the number of transitions \( s \overset{a}{\rightarrow} u \). These assumptions are easily satisfied, by removing those actions in \( \text{Act} \) that are not used in transitions and by sorting and numbering the remaining action labels. Sorting these actions adds a negligible \( O(|\text{Act}| \log |\text{Act}|) \leq O(m_a \log m_a) \).

Grouping transitions is performed by an array of buckets indexed with actions. All transitions are put in the appropriate bucket in constant time exploiting actions being numbered. Furthermore, all buckets that contain transitions are linked together. When all transitions are in the buckets, a straightforward traversal of all linked buckets provides the transitions in a grouped order. This requires time linear in the number of considered action transitions. Note that the number of buckets is equal to \(|\text{Act}| \leq m_a\) and, therefore, the buckets do not require more than linear memory.

4.2. Data Structures

We give a concise overview of the concrete data structures in the algorithm for states, transitions, blocks, and constellations. We list the names of the fields in these data structures in a programming vein to keep a close link with the actual implementation.
The chosen data structures are not particularly optimised. Exploiting ideas from [6,24,25] to store states, blocks, and constellations, usage of time and memory can be further reduced. All data structures come in two flavours, one related to actions and the other related to probabilities. We treat them simultaneously and only mention their differences when appropriate.

4.2.1. Global

In the detailed algorithm, there are arrays containing transitions, actions, blocks and constellations. There is a stack of non-trivial constellations to identify in constant time which constellation must be investigated in the main loop. Furthermore, there is an array containing the variables state_to_constellation_cnt, which are explained below.

For all action transitions \( s \xrightarrow{a} u \), it is maintained how many action transitions there are labelled with the same action \( a \), and that go from \( s \) to the constellation \( C \) containing \( u \). This value is called state_to_constellation_cnt for this transition. The value is required to efficiently split probabilistic blocks (the idea of using such variables stems from [5]). For each state \( s \), constellation \( C \), and action \( a \) there is one instance of state_to_constellation_cnt stored in a global array. Each transition \( s \xrightarrow{a} u \) contains a reference called state_to_constellation_cnt_ptr to the appropriate value in this array. See Figure 3 for a graphical illustration with a constellation \( C \) of probabilistic states and blocks \( B_1 \) and \( B_2 \) of action states. The purpose of this construction is that state_to_constellation_cnt can be changed by one operation for all transitions from the same state with the same action to the same constellation, simultaneously.

![Figure 3. Transitions with state_to_constellation_cnt stored in a global array.](image)

4.2.2. Transition

Each transition consists of the fields from, label and to. Here, from and to refer to an action/probabilistic state, and label is the action label or probabilistic label of the transition. The action labels are consecutive numbers; the probabilistic labels are exact fractions. Action transitions also contain a reference state_to_constellation_cnt_ptr to the variable state_to_constellation_cnt as indicated above.

4.2.3. State

Each action state and probabilistic state contains a list of incoming transitions and a reference to the block in which the state resides. For intermediate calculations, each state contains a boolean mark_state which is used to indicate that a state has been marked. Each action state also contains two more variables for temporary use. When deciding whether blocks need to be split, the
variable residual_transition_cnt indicates how many residual transitions there are to blocks $C \setminus B_C$ when splitting takes place by a block $B_C$. The variable transition_cnt_ptr is used to let the variable state_to_constellation_cnt_ptr for an action transition point to a new instance of state_to_constellation_cnt when this transitions is moved to a new block. In probabilistic states, there is the temporary variable cumulative_prob used to calculate the total probability to reach a block under splitting.

4.2.4. Block

Blocks contain an indication of the constellation in which it occurs, a list of the states contained in the block including the size of this list, and a list of transitions ending in this block. For blocks of action states, this list of transitions is grouped by action label, i.e., transitions with the same action label are a consecutive sublist. For temporary use, there is also a variable to indicate that the block is marked. This marking contains exactly the information that the functions $aMark$ and $pMark$, discussed below, provide for blocks of action states and blocks of probabilistic states, respectively.

4.2.5. Constellation

Finally, constellations contain a list of the blocks in the constellation as well as the cumulative number of states contained in all blocks in this constellation.

4.3. Explanation of the Detailed Algorithm

Algorithm 1 focuses on how, by refining partitions and sets of constellations, probabilistic bisimulation can be calculated. In Algorithm 2, we stress the details of carrying out concrete refinement steps to realise the required time bound. As already indicated, the overall structure of both algorithms is the same.

The initial Lines 2 and 3 of Algorithm 2 are the same as those of Algorithm 1. In Line 3, the partition $B$ is set to contain one block with all probabilistic states and a number of blocks of action states, grouped per common outgoing action labels. Thus, two action states are in the same block initially if their menu, i.e., the set of actions for which there is a transition, is identical. This initial partition $B$ is calculated using a simple partition refinement algorithm on outgoing transitions of states. This operation is linear in the number of outgoing action transitions when using grouping of transitions as explained in Section 4.1.

At Line 4, the incoming transitions are ordered on actions as indicated in Section 4.1. At Line 5, an array with one instance of state_to_constellation_cnt for each action label is made where each instance contains the number of action transitions that contain that action label. The reference state_to_constellation_cnt for each action transition is set to refer to the appropriate instance in this array. This is done by simply traversing all transitions $s \xrightarrow{a} u$ grouped by action labels and incrementing the appropriate entry in the array containing all state_to_constellation_cnt variables. The appropriate entry can be found using the temporary variable transition_cnt_ptr associated to state $s$. If no entry for state_to_constellation_cnt exists yet, the variable transition_cnt_ptr belonging to $s$ is null and an appropriate entry must be created.

In Line 6, selecting a non-trivial constellation is straightforward, as a stack of non-trivial constellations is maintained. Initially, this stack contains $C = \{S, U\}$. To obtain the required time complexity, we select $B_C$ such that $|B_C| \leq \frac{1}{2}|C|$ in Line 7. This is done in constant time as we know the number of states in $C$. Hence, either the first or second block $B$ of constellation $C$ satisfies that $|B| \leq \frac{1}{2}|C|$ (for if the first block contains more than half the states the second one cannot). We replace the constellation $C$ by $B_C$ and $C \setminus B_C$ in $C$, see Line 8, and put the constellation $C \setminus B_C$ on the stack of non-trivial constellations if it is non-trivial.

From Line 9 to Line 19, the partition $B$ is refined to restore the invariants, especially Invariant 3. This is done by first marking the blocks (Line 11 and Line 16) such that it is clear how they must be split, and by subsequently splitting the blocks (Lines 12–14, and Lines 17–19). Both operations are described in the next two subsections.
4.4. Marking

Given a constellation $C$ that contains a block $B_C$ and in the case of an action transition, an action $a$, we need to know which blocks need to be split in what way. This is calculated using the functions $a\text{Mark}(B, C, B_C, a)$ and $p\text{Mark}(B, C, B_C)$. The first one is for marking blocks with respect to action transitions, the second for marking blocks with respect to probabilities.

Both functions yield a five-tuple $\langle B, left, mid, right, and large \rangle$. Here, $B \subseteq B$ is a set of blocks that may have to be split and left, mid, and right are functions that together for each block $B \in B$ provide the sets into which $B$ must be partitioned. The set large($B$) is the largest set among them. For every set $B'$ in which $B$ must be partitioned, except for large($B$), it holds that $|B'| \leq \frac{1}{2}|B|$. To obtain the complexity bound, we only move such small blocks out of $B$, i.e., those blocks not equal to large($B$).

We note that sets in left($B$), mid($B$) and right($B$) can be empty. Such sets can be ignored. It is also possible that there is only one non-empty set being equal to $B$ itself. In this case, $B$ is stable under $B_C$ and $C \setminus B_C$. Furthermore, it is equal to large($B$) and therefore $B$ is kept intact.

We now concentrate on the function $a\text{Mark}(B, C, B_C, a)$ with a partition $B$, a constellation $C$, a block $B_C$ contained in $C$, and an action $a$. In this situation, $C$ is a non-trivial constellation of probabilistic states. Since $C$ contains probabilistic states only, including transitions for states in $B_C$ are action transitions. The situation is depicted in Figure 2, at the left. The call $a\text{Mark}(B, C, B_C, a)$ returns the tuple $\langle B_a, left_a, mid_a, right_a, large_a \rangle$ defined as follows.

\[
B_a = \{ B \in B | \exists s \in B: s \xrightarrow{a} B_C \}
\]

and, for each $B \in B_a$,

- left$_{a}(B) = \{ s \in B | s \xrightarrow{a} B_C \land s \xrightarrow{a} C \setminus B_C \}$,
- mid$_{a}(B) = \{ s \in B | s \xrightarrow{a} B_C \land s \xrightarrow{a} C \setminus B_C \}$,
- right$_{a}(B) = \{ s \in B | s \xrightarrow{a} B_C \land s \xrightarrow{a} C \setminus B_C \}$, and
- large$_{a}(B) = \text{the largest set among } left_{a}(B), mid_{a}(B), \text{and } right_{a}(B).

We calculate $B_a$ by traversing the list of all transitions with action $a$ going into $B_C$ and adding each block containing any source state of these transitions to $B_a$. The blocks in $B_a$ are the only blocks that may be unstable under $B_C$ and $C \setminus B_C$ with respect to $a$ (Lemma 2).

The for loop at Line 10 iterates over all actions. As the incoming transitions into block $B_C$ are grouped per action, all incoming transitions with the same action can easily be processed together, while the total processing time is linear in the number of incoming transitions. However, note that calculating $B_a$ is based on partition $B$, while $B$ is refined at Line 14. Thus, the calculation of $B_a$ for different actions $a$ can be based on repeatedly refined partitions $B$.

Next, we discuss how to construct the blocks left$_{a}(B)$, mid$_{a}(B)$, and right$_{a}(B)$. While traversing $a$-labelled transitions into $B_C$, all action states in a block $B$ with an $a$-transition into $B_C$ are marked and (temporarily) moved into left$_{a}(B)$. The remaining states in block $B$ form the subset right$_{a}(B)$. We keep track of the number of states in a block. Thus, we can easily maintain the size of right$_{a}(B)$.

To find out which states now in left$_{a}(B)$ must be transferred to mid$_{a}(B)$, the variables state_to_constellation_cnt are used. Recall that these variables record for each transition $s \xrightarrow{a} u$, with $u \in S$, how many transitions $s \xrightarrow{a} v$ there are to states $v \in C$. These variables are initialised in Line 5 of Algorithm 2. When the first state is moved to left$_{a}(B)$, we copy the value of state_to_constellation_cnt of transition $s \xrightarrow{a} u$ to the variable residual_transition_cnt belonging to state $s$ of the transition, subtracted by one. The number residual_transition_cnt indicates how many unvisited $a$-transitions are left from the state $s$ into $C$. Every time an $a$-transition is visited of which the source state is already in left$_{a}(B)$, we decrease residual_transition_cnt of the source state by one again. If all $a$-transitions into $B_C$ have been visited, the number residual_transition_cnt of a state $s$ indicates how many transitions labelled $a$ go from $s$ into $C \setminus B_C$. 


Subsequently, we traverse the states in left \(_a(B)\). If a state \(s\) has a non-zero residual transition \(cnt\), we know that there are \(a\)-transitions from \(s\) to both \(B_C\) and \(C_\backslash B_C\). Therefore, we move state \(s\) into mid \(_a(B)\). Otherwise, all transitions from \(s\) with action \(a\) go to \(B_C\) and \(s\) must remain in left \(_a(B)\).

While moving states into left \(_a(B)\) and mid \(_a(B)\), we also keep track of the sizes of these sets. Hence, it is easy to indicate in large \(_a(B)\) which set is the largest.

We calculate \(pMark(B, C, B_C)\) in a slightly different manner than \(aMark\). In particular, we have \(mid_p : B \rightarrow 2^{\mid B\mid}\), i.e., \(mid_p(B)\) is a set of blocks. This indicates that the block \(B\) can be partitioned in many sets, contrary to the situation with action blocks where \(B\) could be split in at most three blocks. The situation is depicted in Figure 2 at the right. The five-tuple that \(pMark\) returns has the following components:

\[
B_p = \{ B \in B \mid \exists u \in B : u[B_C] > 0 \}
\]

and, for each \(B \in B_p\),

\[
\begin{align*}
left_p(B) &= \{ u \in B \mid u[B_C] = 1 \}, \\
mid_p(B) &= \{ \{ u \in B \mid u[B_C] = q \} \mid q \in (0, 1) \}, \\
right_p(B) &= \{ u \in B \mid u[B_C] = 0 \}, \text{ and} \\
large_p(B) &= \text{the largest set from } \{ \left_p(B) \} \cup \mid_p(B) \cup \{ \right_p(B)\}. 
\end{align*}
\]

The above is obtained by traversing through all incoming probabilistic transitions in \(B_C\). Whenever there is a state \(u\) in a block \(B\) such that \(u \rightarrow_p B_C\), one of the following cases applies:

- If \(B\) is not in \(B_p\) yet, it is added now. The variable cumulative prob in state \(u\) is set to \(p\), and \(u\) is (temporarily) moved from \(B\) to \(left_p(B)\).
- If \(B\) is already in \(B_p\), then the probability \(p\) is added to cumulative prob of state \(u\).

After the traversal of all incoming probabilistic transitions into \(B_C\), the variable cumulative prob of \(u\) contains \(u[B_C]\), i.e., the probability to reach \(B_C\) from the state \(u\).

Those states that are left in \(B\) form the set \(right_p(B)\). We know the number of states in \(right_p(B)\) by keeping track how many states were moved to \(left_p(B)\). Next, the states temporarily stored in \(left_p(B)\) must be distributed over \(left_p(B)\) and \(mid_p\). First, all states with \(\text{cumulative prob} < 1\) are moved into some set \(M\) such that \(left_p(B)\) contains exactly the states with \(\text{cumulative prob} = 1\). Then, the states in \(M\) are sorted on their value for \(\text{cumulative prob}\) such that it is easy to move all states with the same \(\text{cumulative prob}\) into separate sets in \(mid_p(B)\). In Figure 2, at the right, the set \(mid_p(B)\) consists of three sets, corresponding to the probabilities \(q = \frac{1}{2}, q = \frac{1}{2}\) and \(q = \frac{2}{3}\) to reach \(B_C\). Note that all processing steps mentioned require time proportional to the number of incoming probabilistic transitions in \(B_C\), except for the time to sort. In the complexity analysis below, it is explained that the cumulative sorting time is bounded by \(O(m_p \cdot \log n_p)\).

By traversing the sets of states in \(left_p(B)\) and \(mid_p(B)\) once more, we can determine which set among \(left_p(B)\), \(right_p(B)\), and the set of sets \(mid_p(B)\) contains the largest number of probabilistic states. This set is reported in \(large_p(B)\).

4.5. Splitting

In Lines 14 and 19 of Algorithm 2, a block \(B'\) is moved out of the existing block \(B\). By the marking procedure, either \(aMark\) or \(pMark\), the states involved are already put in separate lists and are moved in constant time to the new block \(B'\).

Blocks contain lists of incoming transitions. When moving the states to a new block, the incoming transitions are moved by traversing the incoming transitions of each moved state, removing them from the list of incoming transitions of the old block and inserting them in the same list for the new block. There is a complication, namely that incoming action transitions must be grouped by action labels. This is done separately for the transitions moved to \(B'\) as explained in Section 4.1 and this is
linear in the number of transitions being moved. When removing incoming action transitions from the old block \( B \), the ordering of the transitions is maintained. Thus, the grouping of incoming action transitions into \( B \) remains intact without requiring extra work.

When moving action states to a new block we also need to adapt the variable \( \text{state_to_constellation}_\text{cnt} \) for each action transition \( s \overset{a}{\rightarrow} C \) with state \( s \in B \). Observe that this only needs to be done if there are some \( a \)-transitions to \( B_C \) and some to \( C \setminus B_C \), which means that \( s \in \text{mid}_a(B) \). In that case \( \text{residual}_\text{transition}_\text{cnt} \) for state \( s \) is larger than 0.

This is accomplished by traversing all incoming transitions \( s \overset{a}{\rightarrow} u \) into \( B_C \) one extra time. If \( \text{residual}_\text{transition}_\text{cnt} \) for \( s \) is larger than 0 we need to replace the \( \text{state_to_constellation}_\text{cnt} \) for this transition \( s \overset{a}{\rightarrow} u \) by the value of \( \text{state_to_constellation}_\text{cnt} - \text{residual}_\text{transition}_\text{cnt} \) of \( s \). For all non-visited transitions \( s \overset{a}{\rightarrow} u' \) where \( u' \in C \setminus B_C \), the value of \( \text{state_to_constellation}_\text{cnt} \) must be set to \( \text{residual}_\text{transition}_\text{cnt} \) of \( s \).

This is where we use that \( \text{state_to_constellation}_\text{cnt} \) is actually referred to by the pointer \( \text{state_to_constellation}_\text{cnt}_\text{ptr} \) (see Figure 3). When traversing the first transition of the form \( s \overset{a}{\rightarrow} u \) with \( u \in B_C \) such that \( \text{residual}_\text{transition}_\text{cnt} \) for \( s \) is larger than 0, a new entry in the array containing the variables \( \text{state_to_constellation}_\text{cnt} \) is constructed containing the value \( \text{state_to_constellation}_\text{cnt} - \text{residual}_\text{transition}_\text{cnt} \) and the auxiliary variable \( \text{transition}_\text{cnt}_\text{ptr} \) is used to point to this entry. At the same time, the value in old entry in this array for \( \text{state_to_constellation}_\text{cnt} \) is replaced by the value \( \text{residual}_\text{transition}_\text{cnt} \) of state \( s \). In this way, the values of \( \text{state_to_constellation}_\text{cnt} \) of all transitions labelled with \( a \) from \( s \) to \( C \setminus B_C \) are updated in constant time, i.e., without visiting the transitions that are not moved. For all transitions \( s \overset{a}{\rightarrow} u' \) with \( u' \in B_C \), the variable \( \text{state_to_constellation}_\text{cnt}_\text{ptr} \) is made to refer the new entry in the array.

### 4.6. Complexity Analysis

The complexity of the algorithm is determined below. Recall that \( n_a \) and \( n_p \) are the number of action states and probabilistic states, respectively, while \( m_a \) is the number of action transitions and \( m_p \) is the cumulative size of the supports of the distributions.

**Theorem 2.** The total time complexity of the algorithm is \( O((m_a + m_p) \log n_p + (m_p + n_a) \log n_a) \) and the space complexity is \( O(m_a + m_p + n_a) \).

**Proof.** In Algorithm 2, the cost of each computation step is indicated. The initialisation of the algorithm at Lines 2–5 is linear in \( n_a, n_p \) and \( m_a \). At Line 3, calculating \( \{S_A \mid A \subseteq \text{Act}\} \) can be done by iteratively splitting \( S \) using the outgoing transitions grouped per action label. This is linear in the number of action transitions. At Line 4, grouping the incoming transitions per action is also linear as argued in Section 4.1.

The while loop at Line 6 is executed for each \( B_C \subseteq C \) where \( |B_C| \leq \frac{1}{2}|C| \). As \( B_C \) becomes a constellation itself, each state can only be part of this splitting step \( \log_2(n_a) \) times and \( \log_2(n_p) \) times, respectively. The steps in Lines 10–13 and Lines 16–18 require steps proportional to the number of incoming action transitions and probabilistic transitions, respectively, in \( B_C \), apart from a sorting penalty which we treat separately below. The cumulative complexity of this part is therefore \( O(m_a \log n_p + m_p \log n_a) \).

At Lines 14 and 19, the states in \( B' \) are moved to a new block. This requires to group the incoming action transitions in a block \( B' \) per action, which can be done in time linear in the number of these transitions. Block \( B' \) is not the largest block of \( B \) considered and therefore \( |B'| \leq \frac{1}{2}|B| \). Hence, each state can only be \( \log_2(n_p) \) or \( \log_2(n_a) \) times be involved in the operation to move to a new block. Hence, the total time to be attributed to moving is \( O((m_a + n_p) \log n_p + (m_p + n_a) \log n_a) \).

While marking, probabilistic states in \( \text{mid}_p(B) \) need to be sorted. An ingenious argument by Valmari and Franceschinis [6] shows that this will at most contribute \( O(m_p \log n_p) \) to the total complexity: Let \( K \) be the total number of times sorting takes place. Assume, for \( 1 \leq i \leq K \), that
the total number of distributions in $mid_p(B)$ when sorting it for the $i$-th time is $k_i$. Clearly, $k_i \leq n_p$. Each time a distribution in $mid_p(B)$ is involved in sorting, the number of reachable constellations with non-zero probability from this distribution is increased by one. Before sorting it could reach $C$, and after sorting it can reach both new constellations $B_C$ and $C \setminus B_C$ with non-zero probability. Note that this does not hold for the states in $left_p(B)$ and $right_p(B)$, and this is the reason why we have to treat them separately. In particular, to obtain complexity $O(m_p \log n_p)$, it is not allowed to involve the states in $left_p(B)$ and $right_p(B)$ in the sorting process as shown by an example in [6]. Due to the increased number of reachable constellations, the total number of times a probabilistic state can be involved in sorting is bounded by the size of the distribution. In other words, $\sum_{i=1}^{K} k_i \leq m_p$. Hence, the total time that is required by sorting is bounded as follows:

$$O\left(\sum_{i=1}^{K} k_i \log k_i \right) \leq O\left(\sum_{i=1}^{K} k_i \log n_p \right) \leq O\left(m_p \log n_p \right).$$

Adding up the complexities leads to the conclusion that the total complexity of the algorithm is $O\left((m_a + m_p + n_p) \log n_p + (m_p + n_a) \log n_a \right)$. As $m_p \geq n_p$, the stated time complexity in the theorem follows.

The space complexity follows as all data structures are linear in the number of transitions and states. As $n_p \leq m_p$, this complexity can be stated as $O\left(m_a + m_p + n_a\right)$. □

Note that it is reasonable that the number of probabilistic transitions $m_p$ is at least equal to the number of action states $n_a - 1$ as otherwise there are unreachable action states. This allows formulating our complexity more compactly.

**Corollary 1.** Algorithm 2 has time complexity $O\left((m_a + m_p) \log n_p + m_p \log n_a\right)$ and space complexity $O\left(m_a + m_p\right)$ if all action states are reachable.

The only other algorithm to determine probabilistic bisimilarity for PLTS is by Baier, Engelen and Majster-Cederbaum [4]. The algorithm uses extended ordered binary trees and is claimed to have a complexity of $O(n_a n_p |Act|)$ where $n$ is the number of transitions (including distributions) and $n$ the number of action states. For a fair comparison, we reconstructed their complexity in terms of $n_a$, $n_p$, $m_a$ and $m_p$. Their space complexity is $O\left(n_a n_p |Act|\right)$ and the time complexity is $O\left(m_a n_a \log n_a + n_a n_p \log n_p + m_p^2 n_p^2\right)$. The last part $n_p^2 n_p$ is not mentioned in the analysis in [4]. It is due to taking the time into account for “inserting $Pre(\alpha, \mu_i)$ into $\nu.states$” (see page 208 of [4]) for the version of ordered balanced trees used, and we believe it to be forgotten [26].

This complexity is not easily comparable to ours. We make two reasonable assumptions to facilitate comparison. The first assumption is that the number of action transitions is equal to the number of distributions: $m_a = n_p$. As second assumption, we use that $\log n_p$ and $\log n_a$ only differ by a constant.

In the rare case that the support of distributions is large, i.e., if all or nearly all action states have a positive probability in each distribution, then $m_p$ is equal or close to $n_a n_p$. In this case our space complexity becomes $O\left(n_a n_p\right)$ and our time complexity is $O\left(n_a n_p \log n_p\right)$, which is comparable mutatis mutandis to the complexity in [4]. However, in the more common case where the support of distributions is limited by some constant $c$, i.e., $m_p \leq c n_p$, we can simplify the space and time complexities to those in the following Table 2.

Table 2. Space and time complexity of the GRV algorithm and the BEM algorithm.

<table>
<thead>
<tr>
<th></th>
<th>GRV (this article)</th>
<th>BEM [4]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space complexity</td>
<td>$O(n_p)$</td>
<td>$O\left(n_a n_p</td>
</tr>
<tr>
<td>Time complexity</td>
<td>$O(n_p \log n_a)$</td>
<td>$O\left(n_a n_p \log n_a + n_p^2 n_p\right)$</td>
</tr>
</tbody>
</table>
In the table the underlined part stems from the extra time needed for insertions. It is clear that if the assumptions mentioned are satisfied, the complexity of the present algorithm stands out well. This is confirmed in the next section where we report on the performance on a number of benchmarks of implementations of both algorithms.

5. Benchmarks

Both our algorithm, below referred to as GRV, and the reference algorithm by Baier, Engelen and Majster-Cederbaum [4], for which we use the abbreviation BEM, have been implemented in C++ as part of the mCRL2 toolset [9,10] (www.mcrl2.org). This toolset is available under a Boost license which means that the source code is open and available without restriction to be inspected or used. In the implementation of BEM, some of the operations are not carried out exactly as prescribed in [4] for reasons of practicality.

We have extensively tested the correctness of the implementation of the new algorithm by applying it to millions of randomly generated PLTSs, and comparing the results to those of the implementation of the BEM algorithm. This is not done because we doubt the correctness of the algorithm, but because we want to be sure that all the details of our implementation are right.

We experimentally compared the performance of both implementations. All experiments have been performed on a relatively dated machine running Fedora 12 with INTEL XEON E5520 2.27 GHz CPUs and 1TB RAM. For the probabilities exact rational number arithmetic is used which is much more time consuming than floating point arithmetic. The reported runtimes do not include the time to read the input PLTS and write the output.

Our first experimental question regards the growth of the practical complexity of the BEM and GRV algorithm when concrete probabilistic transition systems grow in size. To get an impression of this, we considered the so-called “ant on a grid” puzzle published in the New York Times [27,28]. In this puzzle, an ant sits on a square grid. When it reaches the leftmost or rightmost position on the grid it dies. When it reaches the upper or lower position of the grid it is free and lives happily ever after. On any remaining position, the ant chooses with equal probability to go to a neighbouring position on the grid. The question is what the probabilities for the ant are to die and stay alive, given an initial position on the grid.

The specification in probabilistic mCRL2 of the ant-on-a-grid is given in Figure 4, where the dimensions of the grid are \( \max_x \) and \( \max_y \), and the initial position is given by \( i_x \) and \( i_y \).

```
sort Direction = struct up | down | right | left;
proc X(x : N, y : N) =
  (x ≈ 1 ∨ x ≈ max_x) → dead X(x,y) ⊗
  (y ≈ 1 ∨ y ≈ max_y) → live X(x,y) ⊗
  (dist d : Direction[1/4].
    (d ≈ up)  → step X(x+1,y) ⊗
    (d ≈ down) → step X(x-1,y) ⊗
    (d ≈ right) → step X(x,y+1) ⊗
    (d ≈ left) → step X(x,y-1) ));
init X(i_x,i_y);
```

Figure 4. The specification of ant-on-a-grid in mCRL2.

The actions dead, live and step indicate that the ant is dead, stays alive and makes a step. The process expression \( p \cdot q \) stands for sequential composition and \( p + q \) represents the choice in behaviour. The notations \( c \rightarrow p \) and \( c \rightarrow p \cdot q \) are the if-then and if-then-else of mCRL2. The curly equal sign (\( ≈ \)) in conditions stands for equality applied to data expressions. The expression \( \text{dist } d \text{: Direction[1/4]} \) means that each direction \( d \) is chosen with probability \( \frac{1}{4} \). From this description, PLTSs are generated that are used as input for the probabilistic bisimulation reduction tools.
Figure 5 depicts the runtime results of a set of experiments when increasing the total number of states of the ant on the grid model. At the left are the results when running the BEM algorithm, whereas the results for the GRV algorithm are shown at the right. Note that the x-axis only depicts the number of action states. This figure indicates that the practical running times of both algorithms are pretty much in line with the theoretical complexity. This is in agreement with our findings on other examples as well. Furthermore, it should be noted that the difference in performance is dramatic. The largest example that our implementation of the BEM algorithm can handle within a timeout of 5 h requires approximately 10,000 s compared to 2 s for GRV. The particular example regards a PLTS of $6.4 \times 10^5$ action states. The graphs clearly indicate that the difference grows when the probabilistic transition systems get larger.

![Graph showing runtime results for the ant-on-a-grid puzzle](image)

Figure 5. Scaling of runtime results for the ant-on-a-grid puzzle

To further understand the practical usability of the GRV algorithm, we applied it to a number of benchmarks taken from the PRISM Benchmark Suite (www.prismmodelchecker.org/benchmarks/) and the mCRL2 toolset (www.mcrl2.org/). The tests taken from PRISM were first translated into mCRL2 code to generate the corresponding PLTSs.

Table 3 collects the results for the experiments conducted. The ant$_N$M_grid examples refer to the ant-on-a-grid puzzle for an $N$ by $M$ grid with the ant initially placed at the approximate center of the grid. The models airplane$_N$ are instances of an airplane ticket problem using $N$ seats. In the airplane ticket problem, $N$ passengers enter a plane. The first passenger lost his boarding pass and therefore takes a random seat. Each subsequent passenger will take his own seat unless it is already taken, in which case he randomly selects an empty seat as well. The intriguing question is to determine the probability that the last passenger will have his or her own seat (see [28] for a more detailed account).

The following three benchmarks stem from PRISM: The brp$_N$MAX models are instances of the bounded retransmission protocol when transmitting $N$ packages and bounding the number of retransmissions to $MAX$. The self$_stabilisation_N$ and shared_coin$_N$K are extensions of the self stabilisation protocol and the shared coin protocol, respectively. For the self stabilisation protocol, $N$ processes are involved in the protocol, each holding a token initially. The shared coin protocol is modelled using $N$ processes and setting the threshold to decide head or tail to $K$.

Finally, the random$_N$ tests are randomly generated PLTSs with $N$ action states. All the models are available in the mCRL2 toolset.

At the left of Table 3, the characteristics for each PLTS are given: the number of action states ($n_a$), the number of action transitions ($m_a$), the number of distributions ($n_p$), and the cumulative support of the distributions ($m_p$). The symbol “K” is an indicator for 1000 states. The same characteristics for the minimised PLTS are also provided. Furthermore, the runtime for minimising the probabilistic transition system in seconds as well as the required memory in megabytes are indicated for both algorithms. As mentioned earlier, we limited the runtime to 5 h.
Table 3. Runtime (in s) and memory use (in MB) results for the reference algorithm (BEM) and the GRV algorithm.

<table>
<thead>
<tr>
<th>Model</th>
<th>$n_a$</th>
<th>$m_a$</th>
<th>$n_p$</th>
<th>$m_p$</th>
<th>$n_a$ min.</th>
<th>$n_a$ min.</th>
<th>$m_p$ min.</th>
<th>$m_p$ min.</th>
<th>time BEM</th>
<th>me. BEM</th>
<th>time GRV</th>
<th>me. GRV</th>
<th>Speed-up</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>shared_coin_2_5</td>
<td>14,096</td>
<td>28,192</td>
<td>12,891</td>
<td>14,801</td>
<td>998</td>
<td>1163</td>
<td>1479</td>
<td>5.45</td>
<td>53.57</td>
<td>0.08</td>
<td>51.34</td>
<td>68</td>
<td>1.04</td>
<td></td>
</tr>
<tr>
<td>brp_100_20</td>
<td>15,003</td>
<td>15,003</td>
<td>10,803</td>
<td>15,003</td>
<td>8504</td>
<td>8504</td>
<td>12,704</td>
<td>37.27</td>
<td>82.51</td>
<td>0.06</td>
<td>55.79</td>
<td>621</td>
<td>1.48</td>
<td></td>
</tr>
<tr>
<td>self_stab_7</td>
<td>16,130</td>
<td>56,462</td>
<td>14,337</td>
<td>31,991</td>
<td>23,995</td>
<td>23,994</td>
<td>15,998</td>
<td>491.75</td>
<td>219.01</td>
<td>0.15</td>
<td>66.88</td>
<td>3278</td>
<td>3.27</td>
<td></td>
</tr>
<tr>
<td>brp_100_40</td>
<td>29,003</td>
<td>29,003</td>
<td>20,803</td>
<td>29,003</td>
<td>16,504</td>
<td>16,504</td>
<td>24,704</td>
<td>368.67</td>
<td>176.34</td>
<td>0.22</td>
<td>65.41</td>
<td>1676</td>
<td>2.70</td>
<td></td>
</tr>
<tr>
<td>airplane_4000</td>
<td>31,991</td>
<td>31,990</td>
<td>15,998</td>
<td>31,991</td>
<td>23,995</td>
<td>23,994</td>
<td>15,998</td>
<td>491.75</td>
<td>219.01</td>
<td>0.15</td>
<td>66.88</td>
<td>3278</td>
<td>3.27</td>
<td></td>
</tr>
<tr>
<td>ant_100_100_grid</td>
<td>39,984</td>
<td>39,984</td>
<td>9997</td>
<td>39,984</td>
<td>2405</td>
<td>2405</td>
<td>2404</td>
<td>12,704</td>
<td>37.27</td>
<td>82.51</td>
<td>0.06</td>
<td>55.79</td>
<td>621</td>
<td>1.48</td>
</tr>
<tr>
<td>random_40</td>
<td>40,000</td>
<td>63,981</td>
<td>54,123</td>
<td>86,231</td>
<td>23,995</td>
<td>23,994</td>
<td>15,998</td>
<td>491.75</td>
<td>219.01</td>
<td>0.15</td>
<td>66.88</td>
<td>3278</td>
<td>3.27</td>
<td></td>
</tr>
<tr>
<td>shared_coin_2_10</td>
<td>53,736</td>
<td>107K</td>
<td>551K</td>
<td>212K</td>
<td>3938</td>
<td>7875</td>
<td>19,442</td>
<td>401.17</td>
<td>294.44</td>
<td>0.69</td>
<td>157.17</td>
<td>581</td>
<td>1.87</td>
<td></td>
</tr>
<tr>
<td>self_stab_8</td>
<td>65,026</td>
<td>260K</td>
<td>65,537</td>
<td>262K</td>
<td>6306</td>
<td>45,092</td>
<td>1766.80</td>
<td>546.21</td>
<td>111.70</td>
<td>3534</td>
<td>4.89</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>brp_200_50</td>
<td>72,003</td>
<td>72,003</td>
<td>56,462</td>
<td>72,003</td>
<td>2405</td>
<td>2405</td>
<td>2404</td>
<td>12,704</td>
<td>37.27</td>
<td>82.51</td>
<td>0.06</td>
<td>55.79</td>
<td>621</td>
<td>1.48</td>
</tr>
<tr>
<td>ant_200_100_grid</td>
<td>79,991</td>
<td>79,998</td>
<td>9997</td>
<td>79,991</td>
<td>2405</td>
<td>2405</td>
<td>2404</td>
<td>12,704</td>
<td>37.27</td>
<td>82.51</td>
<td>0.06</td>
<td>55.79</td>
<td>621</td>
<td>1.48</td>
</tr>
<tr>
<td>airplane_10000</td>
<td>79,991</td>
<td>79,998</td>
<td>9997</td>
<td>79,991</td>
<td>2405</td>
<td>2405</td>
<td>2404</td>
<td>12,704</td>
<td>37.27</td>
<td>82.51</td>
<td>0.06</td>
<td>55.79</td>
<td>621</td>
<td>1.48</td>
</tr>
<tr>
<td>random_100</td>
<td>100K</td>
<td>160K</td>
<td>134K</td>
<td>215K</td>
<td>3938</td>
<td>19,442</td>
<td>401.17</td>
<td>294.44</td>
<td>0.69</td>
<td>157.17</td>
<td>581</td>
<td>1.87</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shared_coin_2_20</td>
<td>210K</td>
<td>419K</td>
<td>185K</td>
<td>212K</td>
<td>3938</td>
<td>4583</td>
<td>5859</td>
<td>654.94</td>
<td>440.18</td>
<td>1.19</td>
<td>198.80</td>
<td>550</td>
<td>2.21</td>
<td></td>
</tr>
<tr>
<td>self_stab_9</td>
<td>262K</td>
<td>1175K</td>
<td>294K</td>
<td>1180K</td>
<td>32,806</td>
<td>65,612</td>
<td>4015.53</td>
<td>2809.32</td>
<td>3.24</td>
<td>598.14</td>
<td>1239</td>
<td>4.70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shared_coin_3_2</td>
<td>280K</td>
<td>837K</td>
<td>274K</td>
<td>5841</td>
<td>17,347</td>
<td>7722</td>
<td>1781.64</td>
<td>974.92</td>
<td>1.78</td>
<td>294.68</td>
<td>1001</td>
<td>3.31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ant_400_200_grid</td>
<td>320K</td>
<td>320K</td>
<td>79,997</td>
<td>320K</td>
<td>19,705</td>
<td>19,705</td>
<td>19,705</td>
<td>78,808</td>
<td>3026.02</td>
<td>1703.80</td>
<td>0.88</td>
<td>218.95</td>
<td>3439</td>
<td>7.78</td>
</tr>
<tr>
<td>airplane_40000</td>
<td>320K</td>
<td>320K</td>
<td>160K</td>
<td>320K</td>
<td>240K</td>
<td>240K</td>
<td>240K</td>
<td>240K</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random_400</td>
<td>400K</td>
<td>638K</td>
<td>540K</td>
<td>859K</td>
<td>293K</td>
<td>446K</td>
<td>730K</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>shared_coin_2_30</td>
<td>468K</td>
<td>936K</td>
<td>413K</td>
<td>472K</td>
<td>11,795</td>
<td>6863</td>
<td>4247.55</td>
<td>1248.44</td>
<td>2.95</td>
<td>389.13</td>
<td>823</td>
<td>3.21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ant_400_400_grid</td>
<td>640K</td>
<td>640K</td>
<td>160K</td>
<td>640K</td>
<td>39,605</td>
<td>39,605</td>
<td>39,604</td>
<td>158K</td>
<td>9917.64</td>
<td>6677.75</td>
<td>2.09</td>
<td>397.23</td>
<td>4745</td>
<td>16.31</td>
</tr>
<tr>
<td>airplane_100000</td>
<td>800K</td>
<td>800K</td>
<td>400K</td>
<td>800K</td>
<td>600K</td>
<td>600K</td>
<td>600K</td>
<td>600K</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random_800</td>
<td>800K</td>
<td>1277K</td>
<td>1079K</td>
<td>1718K</td>
<td>587K</td>
<td>1210K</td>
<td>1462K</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>brp_1000_5000</td>
<td>3510K</td>
<td>3510K</td>
<td>3510K</td>
<td>3510K</td>
<td>3510K</td>
<td>3510K</td>
<td>3510K</td>
<td>3510K</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Note:** The table includes runtime and memory usage results for various models using both the BEM and GRV algorithms, showing a comparison of performance and resource usage.
The experiments show that the GRV algorithm outperforms the reference algorithm quite substantially in all studied cases. In the case of “random_100” the difference is four orders of magnitude, despite the fact that this state space has only 100 K action states. The second last column of Table 3 lists the relative speed-up, i.e. the quotient of the time needed by BEM over the time needed by GRV, when applicable. Memory usage is comparable for both algorithms for small cases, whereas for larger examples the BEM algorithm requires up to one order of magnitude more memory than the GRV algorithm. The right-most column of Table 3 contains the relative efficiency in memory, i.e. the quotient of the memory used by BEM over the memory used by GRV, for the cases where BEM terminated before the deadline.

6. Concluding Remarks

We believe we have formulated a very efficient algorithm to determine probabilistic bisimulation. As the algorithm restricts the handling of distributions to the states in the support of the distributions, the running time of the algorithm compares favourably when the fan-out is low in the PLTS under consideration, a situation occurring frequently in practice.

Apart from deciding strong probabilistic bisimilarity, our algorithm is instrumental in the mCRL2 toolset for minimising PLTSs modulo probabilistic bisimulation. Such a reduction can be useful as a preprocessing step before applying other forms of analysis on the PLTS. Occasionally, minimisation can even simplify PLTSs such that they become suitable for visual inspection. See for example the discussion the airplane ticket problem, also known as the problem of the lost boarding pass, in [28]. However, having smaller state spaces will be beneficial regardless, as this reduces the processing time for other tools further down the analysis chain.

To fine tune the algorithm, it will be interesting in future work to investigate how to choose the non-trivial constellations C and its sub-blocks B_C optimally; their choice is now non-deterministic. Furthermore, it is interesting to refine the algorithm to probabilistic bisimulation with combined transitions [29] as this appears to be required to extend this algorithm to weaker notions of equivalence [21], such as probabilistic branching bisimulation.

Author Contributions: Conceptualisation, J.F.G.; Software and benchmarks, J.R.V.; Formal Analysis, J.F.G. and E.V.; Original Draft Preparation, J.F.G.; and Writing, Review and Editing, E.V., J.F.G.

Funding: This research received no external funding.

Acknowledgments: We thank Rodin Aarssen, Olav Bunte, and Thomas Neele for their helpful comments on a final draft of this article.

Conflicts of Interest: The authors declare no conflict of interest.

References

© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).