Article

An efficient algorithm to determine probabilistic bisimulation

J.F. Groote [1,*], H.J. Rivera Verduzco [2] and E.P. de Vink [3]

[1] J.F.Groote@tue.nl
[2] H.J.Rivera.Verduzco@student.tue.nl
[3] evink@win.tue.nl
* Correspondence: j.f.groote@tue.nl; Tel.: +31-40-2475003

Academic Editor: name
Version August 20, 2018 submitted to Algorithms

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 & Majster-Cederbaum (Journal of Computer and System Sciences 60:187–231, 2000). Also 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 [20], Larsen and Skou propose the notion of probabilistic bisimulation. Although described for deterministic transition systems, the same notion is very suitable for probabilistic transition systems with nondeterminism [22,23], so-called PLTSs, too. 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, 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.

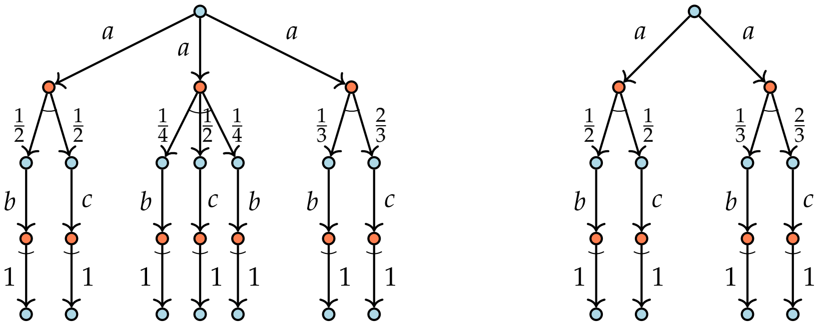

Figure 1. Two probabilistically bisimilar nondeterministic transition systems.

In [3], Baier et al. give an algorithm for probabilistic bisimulation for PLTSs, thus dealing both with probabilistic and nondeterministic choice, of time complexity $O(mn(\log m + \log n))$ and space 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 of [3] 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 $O\left((m_a + m_p)\log n_p + m_p \log n_a\right)$ and space complexity $O\left(m_a + m_p\right)$, where n_a 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_a 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 come near. 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 of [3], both in time as well as in space complexity.

Like 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 [21] 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 [27], 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 [3,9]. Actually, our algorithm uses a simplification of the algorithm of [27] since the calculation of so-called *majority candidates* can be avoided, too.

We implemented both the new algorithm and the algorithm from [3]. 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 [19], 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 [3]. 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 of [3] 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 [7,11].

Related work. Probabilistic bisimulation preserves logic equivalence for PCTL [14]. In [18], Katoen c.s. report up to logarithmic state space reduction obtained by probabilistic bisimulation minimisation for DTMCs. Quotienting modulo probabilistic bisimulation is based on the algorithm of [9]. In the same vein, Dehnert et al. propose symbolic probabilistic bisimulation minimisation to reduce computation time for model checking PCTL in a setting for DTMCs [8], 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 [24].

For CTMCs, Hillston et al. propose the notion of contextual lumpability based on lumpable bisimulation in [16]. Their reduction technique uses the Valmari-Franceschinis algorithm for Markov chain lumping mentioned earlier. Crafa and Renzato [6] characterise 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 of [3]. A similar result applies to the coalgebraic approach to partition refinement in [10] that yields a general bisimulation decision procedure, that can be instantiated with probabilistic system types.

Probabilistic simulation for PLTSs has been treated in [3], too. In [28] maximum flow techniques are proposed to improve the complexity. Zhang and Jansen present in [29] a space-efficient algorithm based on partition refinement for simulation between probabilistic automata, which improves upon the algorithm for simulation by Crafa and Renzato in [6] for concrete experiments taken from the PRISM benchmark suite. A polynomial algorithm, essentially cubic, for deciding weak and branching probabilistic bisimulation by Turrini and Hermanns, recasting the algorithm of [5], is presented in [25].

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 the subsequent section 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 $f : S \to [0,1]$ 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 $s/R = \{\, t \in S \mid sRt \,\}$ and, for a subset T of S, we define $T/R = \{\, s \in S \mid \exists t \in T \colon sRt \,\}$. 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 π of S induces an equivalence relation R_π on S, by $sR_\pi t$ iff $s, t \in B$ for some block B of π. A partition π is called a *refinement* of a partition ϱ iff each block of π is a subset of a block of ϱ. Hence, each block in ϱ is a disjoint union of blocks from π.

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

Definition 2.1 (Probabilistic labeled transition system). A *probabilistic labeled transition system* (PLTS) for a set of actions *Act* is a pair $\mathcal{A} = (\, S, \to)$ where

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

It is common to write $s \xrightarrow{a} f$ for $\langle\, s, a, f \,\rangle \in \to$. For $s \in S$, $a \in Act$, 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 $\overset{a}{\nrightarrow} F$ 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 $\{\, u_f \mid \exists s \in S, a \in Act \colon s \xrightarrow{a} f \,\}$, with typical element u. Note that, since $\to$ is finite, U is also finite. We also use the notation $s \xrightarrow{a} u_f$ if $s \xrightarrow{a} f$ 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 $u_f \mapsto_p s$ for $0 < p \leqslant 1$ and $s \in S$ iff $u_f(s) = p$. In order 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, $m_a = |{\to}|$ 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 $m_p \geqslant n_p$ as every distribution has support of at least size 1.

The following definition for probabilistic bisimulation stems from [20].

Definition 2.2 (Probabilistic bisimulation). Consider a PLTS $\mathcal{A} = (\, S, \to)$. 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 sRt and $s \xrightarrow{a} f$, for some action $a \in Act$ 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 sRt, 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 [15] for a proof.

3. A partition refinement algorithm for probabilistic bisimulation (outline)

Many efficient algorithms for standard bisimulation calculate partitions of states [12,17,21]. 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 *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 3.1 (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 $a \in Act$ 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 $a \in Act$.
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 3.2. 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 $a \in Act$ either both satisfy $s \xrightarrow{a} C$ and $t \xrightarrow{a} C$, or neither does. So, B' is stable under C. □

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 3.3. 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 $C \backslash C'$ 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 $B \backslash B'$.

Proof. We only provide the proof for the first part of this lemma. If $s, t \in B$, then both $s \overset{a}{\nrightarrow} C'$ and $t \overset{a}{\nrightarrow} C'$ by assumption. So, B is stable under C' with respect to a. Furthermore, B is stable under $C \backslash C'$: Suppose $s, t \in B$ and $s \xrightarrow{a} C \backslash C'$. So, $s \xrightarrow{a} C$. As B is stable under C, $t \xrightarrow{a} C$, and by assumption $t \overset{a}{\nrightarrow} C'$. Therefore, $t \xrightarrow{a} C \backslash C'$. Suppose $s \overset{a}{\nrightarrow} C \backslash C'$. Then also $s \overset{a}{\nrightarrow} C$. As B is stable under C, $t \overset{a}{\nrightarrow} C$ and hence, $t \overset{a}{\nrightarrow} C \backslash C'$. □

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

Proof. By the first condition of Definition 3.1 and stability of all blocks in $\mathcal{B}$ we have that either $B \subseteq S$ or $B \subseteq U$, for each block $B \in \mathcal{B}$. We write $s\mathcal{B}t$ iff $s, t \in B$ for some $B \in \mathcal{B}$. Note that used in this way $\mathcal{B}$ is an equivalence relation on S.

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

Thus the stable partition $\mathcal{B}$ induces an equivalence relation that satisfies the conditions for a probabilistic bisimulation of Definition 2.2, as was to be shown. □

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 $\mathcal{B}$, in this particular case containing both action states and probabilistic states, which are treated differently. In addition, following the approach of Paige and Tarjan [21], we maintain a coarser partition $\mathcal{C}$, which we call the set of *constellations*. Each constellation in partition $\mathcal{C}$ is a union of one or more blocks of $\mathcal{B}$, thus $\mathcal{B}$ is a refinement of $\mathcal{C}$. A constellation $C \in \mathcal{C}$ that consists of exactly one block in $\mathcal{B}$ is called *trivial*. We refine partitions $\mathcal{B}$ and $\mathcal{C}$ until $\mathcal{C}$ only contains trivial constellations (see line 5 of Algorithm 1).

Among other, we preserve the invariant that the blocks in partition $\mathcal{B}$ are always stable under partition $\mathcal{C}$. If all constellations in $\mathcal{C}$ are trivial, then the partitions $\mathcal{B}$ and $\mathcal{C}$ coincide. Hence, the blocks in $\mathcal{B}$ are stable under itself, and according to Lemma 3.4 we have found a probabilistic bisimulation. Our algorithm works by iteratively refining the set of constellations $\mathcal{C}$. When refining $\mathcal{C}$ we must also refine $\mathcal{B}$ to preserve the above mentioned invariant.

Since the set of states of a PLTS is finite (cf. Definition 2.1) refinement of the partitions $\mathcal{B}$ and $\mathcal{C}$ cannot be repeated indefinitely. So, 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 this is only possible if all states are not bisimilar. In practice, the main loop of the algorithm stops well before reaching that point.

The algorithm maintains the following three invariants:

Algorithm 1 Abstract partition refinement algorithm for probabilistic bisimulation

```
 1: function PARTITION-REFINEMENT
 2: $\mathcal{C} := \{ S, U \}$
 3: $\mathcal{B} := \{ U \} \cup \{ S_A \mid A \subseteq Act \}$
 4:       where $S_A = \{ s \in S \mid \forall a \in A\, \exists u \in U \colon s \xrightarrow{a} u \}$
 5: while $\mathcal{C}$ contains a non-trivial constellation $C$ do
 6:     choose block $B_C$ from $\mathcal{B}$ in $C$
 7:     replace in $\mathcal{C}$ 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 $\mathcal{B}$ unstable under $B_C$ or $C \backslash B_C$ do
10:             refine $\mathcal{B}$ by splitting $B$ into blocks of states with the same actions into $B_C$ and $C \backslash B_C$
11:     else
12:         for all blocks $B$ of probabilistic states in $\mathcal{B}$ unstable under $B_C$ do
13:             refine $\mathcal{B}$ by splitting $B$ into blocks of states with equal probabilities into $B_C$
14: return $\mathcal{B}$
```

Invariant 1. Probabilistic bisimilarity $\simeq_p$ is a refinement of $\mathcal{B}$.

Invariant 2. Partition $\mathcal{B}$ is a refinement of partition $\mathcal{C}$.

Invariant 3. Partition $\mathcal{B}$ is stable under the set of constellations $\mathcal{C}$ (mentioned already above).

Invariant 1 states that if two action states or two probabilistic states are probabilistically bisimilar, then they are in the same block of partition $\mathcal{B}$. Thus, the partition-refinement algorithm will not separate states if they are bisimilar. By Invariant 2 we have that, at the end and at the start of each iteration, each constellation in $\mathcal{C}$ is a union of blocks in $\mathcal{B}$. Invariant 3 says that blocks in partition $\mathcal{B}$ cannot be split by blocks in constellation $\mathcal{C}$.

In lines 2 and 3 of Algorithm 1 the set of constellation and the initial partition are set such that the invariants hold. All probabilistic states are put in one block, and all action states with exactly the same actions labelling outgoing transitions are also put together in blocks. (Note the universal quantification over all actions a in A for the set comprehension at line 4 in order to ensure that only maximal blocks are included in $\mathcal{B}$ for it being a partition indeed.) The set of constellations contains two constellations namely one with all action states, and one with all probabilistic states. It is straightforward to see that Invariants 1 and 2 hold. Invariant 3 is valid because all transitions from action states go to probabilistic states and vice versa.

Invariants 1 to 3 guarantee correctness of Algorithm 1. I.e., from the invariants it follows that upon termination, when all constellations have become trivial, the computed partition $\mathcal{B}$ identifies probabilistically bisimilar action states and probabilistically bisimilar probabilistic states.

Theorem 3.5. Consider the partition $\mathcal{B}$ resulting from Algorithm 1. We find that (i) two action states are in the same block of $\mathcal{B}$ iff they are probabilistically bisimilar, and (ii) two probabilistic states are in the same block of $\mathcal{B}$ iff they are probabilistically bisimilar.

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

By the Stability Property of Lemma 3.4, we have that $\mathcal{B}$ is a probabilistic bisimulation on S. It follows that two action states in the same block of $\mathcal{B}$ are probabilistically bisimilar. Reversely, by Invariant 1, probabilistically bisimilar action states are in the same block of $\mathcal{B}$. Thus, $\simeq_p$ and $\mathcal{B}$ coincide on S. In other words two action states are in the same block of $\mathcal{B}$ iff they are probabilistically bisimilar.

To compare $\simeq_p$ and the relation $\mathcal{B}$ on U, choose probabilistic states $u, v \in U$ such that $u\, \mathcal{B}\, v$. So, u and v are in the same block of $\mathcal{B}$. By stability of block B for $\mathcal{B}$ it follows that $u[B'] = v[B']$, for each block $B' \subseteq S$. Since $\simeq_p$ and $\mathcal{B}$ coincide on S this implies $u[B'] = v[B']$ for all $B' \in S/{\simeq_p}$. Thus, we have $u \simeq_p v$. Reversely, if $u \simeq_p v$, we have $u, v \in B$ for some block B of $\mathcal{B}$ by Invariant 1. So, two probabilistic states are in the same block of $\mathcal{B}$ 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. But 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 $\mathcal{C}$ by the smaller constellations B_C and $C \backslash B_C$ (line 7). This preserves Invariants 1 and 2, but Invariant 3 may be violated as stability under B_C or $C \backslash B_C$ (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 $C \backslash B_C$ 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 $C \backslash B_C$, i.e. $u[B_C] \neq v[B_C]$ and $u[C \backslash B_C] \neq v[C \backslash B_C]$. Therefore, in the remainder of the body of Algorithm 1 the blocks that are unstable under B_C and $C \backslash B_C$ 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 12 and 13). 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.

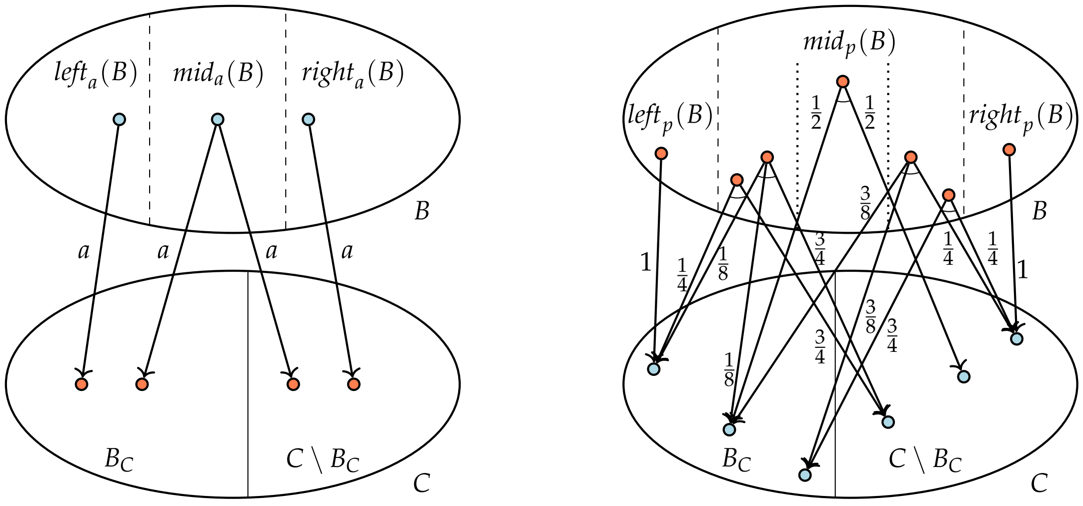

Figure 2. Splitting a non-stable block B into *left*, *middle* and *right*.

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 $\mathcal{B}$, and we distinguish two non-empty subsets of C, the block B_C on its own and the remaining blocks together in $C \backslash B_C$. As B is stable under C, the block B can only be unstable under B_C or $C \backslash B_C$ if there is an action $a \in Act$ and a state $s \in B$ such that $s \xrightarrow{a} B_C$ (Lemma 3.3.1). So, 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:

$$\begin{aligned} \mathit{left}_a(B) &= \{\, s \in B \mid s \xrightarrow{a} B_C \wedge s \overset{a}{\nrightarrow} C \backslash B_C \,\}, \\ \mathit{mid}_a(B) &= \{\, s \in B \mid s \xrightarrow{a} B_C \wedge s \xrightarrow{a} C \backslash B_C \,\}, \text{ and} \\ \mathit{right}_a(B) &= \{\, s \in B \mid s \overset{a}{\nrightarrow} B_C \wedge s \xrightarrow{a} C \backslash B_C \,\}. \end{aligned}$$

Note that the remaining set $\{\, s \in B \mid s \overset{a}{\nrightarrow} B_C \wedge s \overset{a}{\nrightarrow} C \backslash B_C \,\}$ must be empty; if not, this would imply that there is some action state t such that $t \overset{a}{\nrightarrow} C$. But 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 $\mathit{left}_a(B)$, $\mathit{mid}_a(B)$, $\mathit{right}_a(B)$ are stable under C is immediate. As subsets of stable sets are also stable (Lemma 3.2) and B is stable all other configurations of $\mathcal{C}$, the sets $\mathit{left}_a(B)$, $\mathit{mid}_a(B)$, $\mathit{right}_a(B)$ are stable under all other configurations of $\mathcal{C}$ too.

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

Lines 9 and 10 can now be read as follows. For all $a \in \mathit{Act}$ 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 $\mathcal{B}$ by $\{\mathit{left}_a(B), \mathit{mid}_a(B), \mathit{right}_a(B)\} \setminus \emptyset$ to restore stability under B_C and $C \backslash B_C$.

Invariants 1 and 2 are preserved by splitting B. For Invariant 2 this is trivial by construction. For Invariant 1, note that the states in different blocks among $\mathit{left}_a(B)$, $\mathit{mid}_a(B)$, $\mathit{right}_a(B)$ cannot be probabilistically bisimilar as they have unique transitions to states B_C and $C \backslash B_C$ 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 $\mathit{left}_a(B)$, $\mathit{mid}_a(B)$, or $\mathit{right}_a(B)$ 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 $C \backslash B_C$. 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 $C \backslash B_C$.

To restore stability we now consider for all q, $0 \leqslant q \leqslant 1$, 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 $C \backslash B_C$. 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, $u[C \backslash B_C] = u[C] - u[B_C] = v[C] - v[B_C] = v[C \backslash B_C]$. By Lemma 3.2 the new blocks B_q are also stable under the other constellations in $\mathcal{C}$.

According to Lemma 3.3.2 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 $C \backslash B_C$. So, at line 12 of Algorithm 1 we consider all those blocks B and replace each of them by the non-empty subsets B_q, $0 \leqslant q \leqslant 1$ at line 13 in $\mathcal{B}$. This makes the partition stable again under all constellations in $\mathcal{C}$, in particular under the new constellations B_C and $C \backslash B_C$.

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 $\mathcal{B}$ remains a refinement of partition $\mathcal{C}$.

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

3.4. An example

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

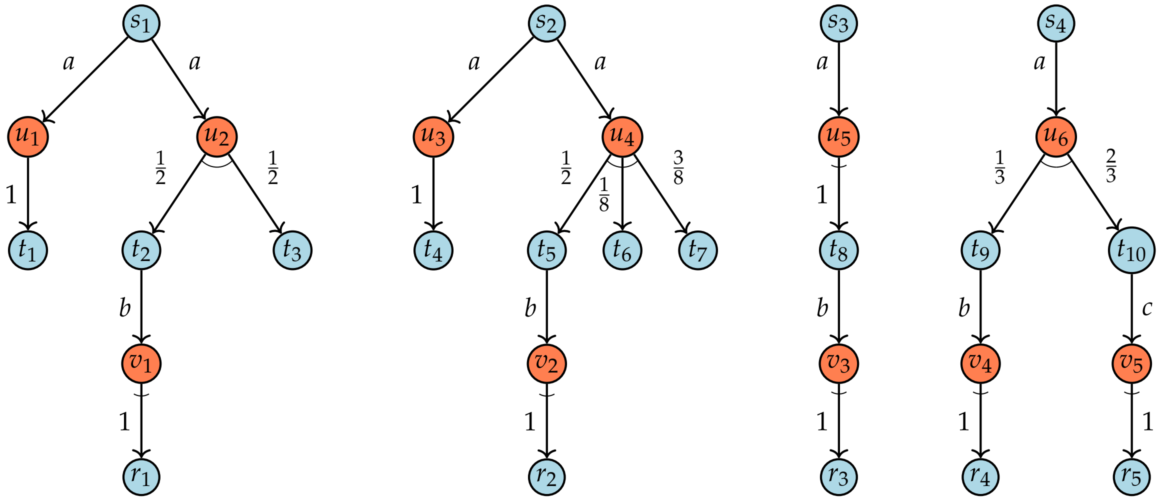

Figure 3. A PLTS used to illustrate the calculation of partitions in Example 3.6.

Example 3.6. 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, 9 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.

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 line 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 $U_1 = \{u_1, u_3, v_{1-5}\}$, $U_2 = \{u_2, u_4\}$ 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 $u_5[S_0] = 0$ and $u_6[S_0] = 0$. The resulting partitions are listed at line 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 $u_6[S_2] = \frac{1}{3}$. 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: $s_1, s_2 \xrightarrow{a} U_1$ and $s_1, s_2 \xrightarrow{a} U_2 \cup U_4 \cup U_5$, but $s_3, s_4 \overset{a}{\nrightarrow} U_1$ and $s_3, s_4 \xrightarrow{a} U_2 \cup U_4 \cup U_5$. 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 a-transition into $U_4 \cup U_5$ and no other transitions. All in all, in this iteration no blocks require splitting to restore Invariant 3.

Next, at iteration 6, we split non-trivial constellation $U_4 \cup U_5$ into U_4 and U_5. For S_0, S_2, S_3 and S_4 we conclude stability in the same way as in the previous iteration. However, now we have for $s_3, s_4 \in S_5$ on the one hand $s_3 \xrightarrow{a} U_4$ and $s_3 \not\xrightarrow{a} U_5$, but on the other hand $s_4 \not\xrightarrow{a} U_4$ and $s_4 \xrightarrow{a} U_5$. Hence, S_5 needs to be split, yielding the singletons $S_6 = \{s_3\}$ and $S_7 = \{s_4\}$.

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 3.4, 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_5 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.

Table 1. The generated partitions for the PLTS of Example 3.6

blocks of actions states	blocks of probabilistic states
$S_0 = \{t_1, t_3, t_4, t_6, t_7, r_{1-5}\}$	$U_0 = \{u_{1-6}, v_{1-5}\}$
$S_1 = \{s_{1-4}\}$	$U_1 = \{u_1, u_3, v_{1-5}\}$
$S_2 = \{t_2, t_5, t_8, t_9\}$	$U_2 = \{u_2, u_4\}$
$S_3 = \{t_{10}\}$	$U_3 = \{u_5, u_6\}$
$S_4 = \{s_1, s_2\}$	$U_4 = \{u_5\}$
$S_5 = \{s_3, s_4\}$	$U_5 = \{u_6\}$
$S_6 = \{s_3\}$	
$S_7 = \{s_4\}$	

	$\mathcal{B}$		$\mathcal{C}$	
0	S_0, S_1, S_2, S_3	U_0	$S_0 \cup S_1 \cup S_2 \cup S_3$	U_0
1	S_0, S_1, S_2, S_3	U_1, U_2, U_3	$S_0,\ S_1 \cup S_2 \cup S_3$	$U_1 \cup U_2 \cup U_3$
2	S_0, S_1, S_2, S_3	U_1, U_2, U_3	$S_0,\ S_1,\ S_2 \cup S_3$	$U_1 \cup U_2 \cup U_3$
3	S_0, S_1, S_2, S_3	U_1, U_2, U_4, U_5	$S_0,\ S_1,\ S_2,\ S_3$	$U_1 \cup U_2 \cup U_4 \cup U_5$
4	S_0, S_2, S_3, S_4, S_5	U_1, U_2, U_4, U_5	$S_0,\ S_2,\ S_3,\ S_4 \cup S_5$	$U_1,\ U_2 \cup U_4 \cup U_5$
5	S_0, S_2, S_3, S_4, S_5	U_1, U_2, U_4, U_5	$S_0,\ S_2,\ S_3,\ S_4 \cup S_5$	$U_1,\ U_2,\ U_4 \cup U_5$
6	$S_0, S_2, S_3, S_4, S_6, S_7$	U_1, U_2, U_4, U_5	$S_0,\ S_2,\ S_3,\ S_4 \cup S_6 \cup S_7$	$U_1,\ U_2,\ U_4,\ U_5$
7	$S_0, S_2, S_3, S_4, S_6, S_7$	U_1, U_2, U_4, U_5	$S_0,\ S_2,\ S_3,\ S_4,\ S_6 \cup S_7$	$U_1,\ U_2,\ U_4,\ U_5$
8	$S_0, S_2, S_3, S_4, S_6, S_7$	U_1, U_2, U_4, U_5	$S_0,\ S_2,\ S_3,\ S_4,\ S_6,\ S_7$	$U_1,\ U_2,\ U_4,\ U_5$

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

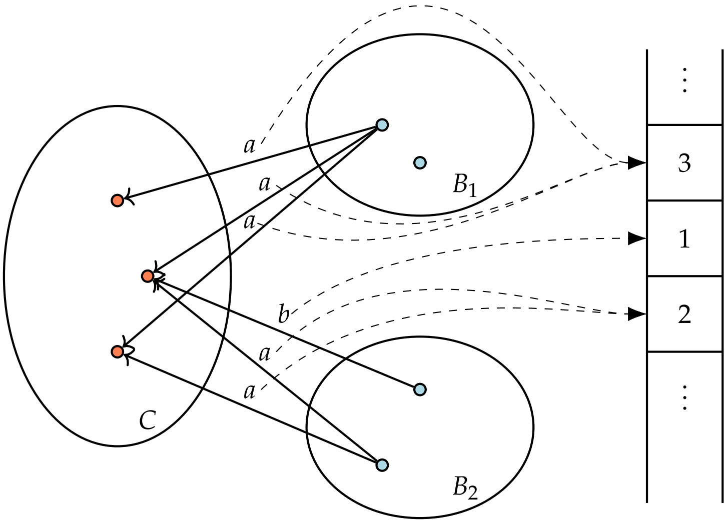

Figure 4. Transitions with *state_to_constellation_cnt* stored in a global array.

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

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 $|Act| \leqslant m_a$ 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 [12,26,27] 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.

Global. In the detailed algorithm there are arrays containing transitions, actions, blocks as well as 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 [21]). 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 4 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.

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.

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 $C \backslash B_C$ 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.

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.

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 $\mathcal{B}$ 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 $\mathcal{B}$ 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

Algorithm 2 Partition refinement algorithm for probabilistic bisimulation

```
 1: function PARTITION-REFINEMENT ( $S$, $U$, $\rightarrow$ )
 2: $\mathcal{C} := \{ S, U \}$                                                    } $O(n_a + n_p)$
 3: $\mathcal{B} := \{ U \} \cup \{ S_A \mid A \subseteq Act \}$                                 } $O(n_p + n_a + m_a)$
        where $S_A = \{ s \in S \mid \forall a \in A\, \exists u \in U : s \xrightarrow{a} u \}$
 4: group the incoming action transitions in each block per label  } $O(m_a)$
 5: initialise $state\_to\_constellation\_cnt$ for each transition     } $O(m_a)$
 6: while $\mathcal{C}$ contains a non-trivial constellation $C$ do          } $\leqslant n$ iterations
 7:    choose a block $B_C$ from $\mathcal{B}$ in $C$ such that $|B_C| \leqslant \frac{1}{2}|C|$  }
 8:    split constellation $C$ into $B_C$ and $C \backslash B_C$ in $\mathcal{C}$                } $O(1)$
 9:    if $C$ contains probabilistic states then                       }
10:       for all incoming actions $a$ of states in $B_C$ do            } $\leqslant |Act|$ iterations
11:          $\langle \mathbf{B}_a, left_a, mid_a, right_a, large_a \rangle := aMark(\mathcal{B}, C, B_C, a)$  } $O$(nr of incoming $a$ transitions in $B_C$)
12:          for all blocks $B \in \mathbf{B}_a$ do                      }
13:             for all non-empty subsets $B' \subseteq B$, different from   } $O$(nr of incoming $a$ transitions in $B_C$)
                  $large_a(B)$ in $\{left(B)_a, mid_a(B), right_a(B)\}$ do  }
14:                move $B'$ out of $B$ and add $B'$ as new block to $\mathcal{B}$  } $O$(nr of incoming transitions in $B'$)
15:    else                                                          } $O$(nr of incoming prob. transitions in $B_C$)
16:       $\langle \mathbf{B}_p, left_p, mid_p, right_p, large_p \rangle := pMark(\mathcal{B}, C, B_C)$  }      plus a sorting penalty
17:       for all blocks $B \in \mathbf{B}_p$ do                          }
18:          for all non-empty sets of states $B' \subseteq B$ not equal to  } $O$(nr of incoming prob. transitions in $B_C$)
               $large_p(B)$ in $\{left_p(B)\} \cup mid_p(B) \cup \{right(B)_p\}$ do  }
19:             move $B'$ out of $B$ and add $B'$ as a new block to $\mathcal{B}$  } $O$(nr of incoming transitions in $B'$)
20: return $\mathcal{B}$
```

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 $\mathcal{C} = \{\mathcal{S}, \mathcal{U}\}$. To obtain the required time complexity, we select B_C such that $|B_C| \leqslant \frac{1}{2}|C|$ 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 $|B| \leqslant \frac{1}{2}|C|$ (for if the first block contains more than half the states the second one cannot). We replace the constellation C by B_C and $C \backslash B_C$ in $\mathcal{C}$, see line 8, and put the constellation $C \backslash B_C$ on the stack of non-trivial constellations if it is non-trivial.

From line 9 to 19 the partition $\mathcal{B}$ 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 to 14, and lines 17 to 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 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 $aMark(\mathcal{B}, C, B_C, a)$ and $pMark(\mathcal{B}, C, B_C)$. 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 $\langle \boldsymbol{B}, left, mid, right, large \rangle$. Here $\boldsymbol{B} \subseteq \mathcal{B}$ is a set of blocks that may have to be split and *left*, *mid*, *right* are functions that together for each block $B \in \boldsymbol{B}$ 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 $|B'| \leqslant \frac{1}{2}|B|$. 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 $\mathit{left}(B)$, $\mathit{mid}(B)$ and $\mathit{right}(B)$ 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 $C \backslash B_C$. Furthermore, it is equal to $\mathit{large}(B)$ and therefore B is kept intact.

We now concentrate on the function $\mathit{aMark}(\mathcal{B}, C, B_C, a)$ with a partition $\mathcal{B}$, 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, incoming transitions for states in B_C are action transitions. The situation is depicted in Figure 2, at the left. The call $\mathit{aMark}(\mathcal{B}, C, B_C, a)$ returns the tuple $\langle \boldsymbol{B}_a$, left_a, mid_a, right_a, $\mathit{large}_a \rangle$ defined as follows.

$$\begin{aligned}
\boldsymbol{B}_a &= \{\, B \in \mathcal{B} \mid \exists s \in B \colon s \xrightarrow{a} B_C \,\} \\
&\text{and, for each } B \in \boldsymbol{B}_a, \\
\mathit{left}_a(B) &= \{\, s \in B \mid s \xrightarrow{a} B_C \wedge s \overset{a}{\nrightarrow} C \backslash B_C \,\}, \\
\mathit{mid}_a(B) &= \{\, s \in B \mid s \xrightarrow{a} B_C \wedge s \xrightarrow{a} C \backslash B_C \,\}, \\
\mathit{right}_a(B) &= \{\, s \in B \mid s \overset{a}{\nrightarrow} B_C \wedge s \xrightarrow{a} C \backslash B_C \,\}, \text{ and} \\
\mathit{large}_a(B) &: \text{ the largest set among } \mathit{left}_a(B), \mathit{mid}_a(B), \text{ and } \mathit{right}_a(B).
\end{aligned}$$

We calculate $\boldsymbol{B}_a$ 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 $\boldsymbol{B}_a$. The blocks in $\boldsymbol{B}_a$ are the only blocks that may be unstable under B_C and $C \backslash B_C$ with respect to a (Lemma 3.3).

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. But note that calculating $\boldsymbol{B}_a$ is based on partition $\mathcal{B}$, while $\mathcal{B}$ is refined at line 14. Thus, the calculation of $\boldsymbol{B}_a$ for different actions a can be based on repeatedly refined partitions $\mathcal{B}$.

Next, we discuss how to construct the blocks $\mathit{left}_a(B)$, $\mathit{mid}_a(B)$, and $\mathit{right}_a(B)$. 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 $\mathit{left}_a(B)$. The remaining states in block B form the subset $\mathit{right}_a(B)$. We keep track of the number of states in a block. Thus, we can easily maintain the size of $\mathit{right}_a(B)$.

To find out which states now in $\mathit{left}_a(B)$ must be transferred to $\mathit{mid}_a(B)$, 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 $\mathit{left}_a(B)$, 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 $\mathit{left}_a(B)$, 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 $C \backslash B_C$.

Subsequently we traverse the states in $\mathit{left}_a(B)$. 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 $\mathit{mid}_a(B)$. Otherwise, all transitions from s with action a go to B_C and s must remain in $\mathit{left}_a(B)$.

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

We calculate $\mathit{pMark}(\mathcal{B}, C, B_C)$ in a slightly different manner than *aMark*. In particular, we have $\mathit{mid}_p : \boldsymbol{B} \to 2^{2^U}$, i.e., $\mathit{mid}_p(B)$ 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:

$$\begin{array}{rcl} \boldsymbol{B}_p & = & \{\, B \in \mathcal{B} \mid \exists u \in B \colon u[B_C] > 0 \,\} \\ \multicolumn{3}{l}{\text{and, for each } B \in \boldsymbol{B}_p,} \\ \mathit{left}_p(B) & = & \{\, u \in B \mid u[B_C] = 1 \,\}, \\ \mathit{mid}_p(B) & = & \{\, \{\, u \in B \mid u[B_C] = q \,\} \mid q \in \langle 0,1 \rangle \,\}, \\ \mathit{right}_p(B) & = & \{\, u \in B \mid u[B_C] = 0 \,\}, \text{ and} \\ \mathit{large}_p(B) & : & \text{the largest set from } \{\mathit{left}_p(B)\} \cup \mathit{mid}_p(B) \cup \{\mathit{right}_p(B)\}. \end{array}$$

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 $u \mapsto_p B_C$, one of the following cases applies:

- If B is not in $\boldsymbol{B}_p$ yet, it is added now. The variable *cumulative_prob* in state u is set to p, and u is (temporarily) moved from B to $\mathit{left}_p(B)$.
- If B is already in $\boldsymbol{B}_p$, 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 $\mathit{right}_p(B)$. We know the number of states in $\mathit{right}_p(B)$ by keeping track how many states were moved to $\mathit{left}_p(B)$. Next, the states temporarily stored in $\mathit{left}_p(B)$ must be distributed over $\mathit{left}_p(B)$ and $\mathit{mid}_p(B)$. First, all states with $\mathit{cumulative\_prob} < 1$ are moved into some set M such that $\mathit{left}_p(B)$ contains exactly the states with $\mathit{cumulative\_prob} = 1$. Then the states in M are sorted on their value for *cumulative_prob* such that it is easy to move all states with the same *cumulative_prob* into separate sets in $\mathit{mid}_p(B)$. In Figure 2 at the right the set $\mathit{mid}_p(B)$ consists of three sets, corresponding to the probabilities $q = \frac{1}{4}$, $q = \frac{1}{2}$ and $q = \frac{3}{4}$ 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 $O\left(m_p \log n_p\right)$.

By traversing the sets of states in $\mathit{left}_p(B)$ and $\mathit{mid}_p(B)$ once more, we can determine which set among $\mathit{left}_p(B)$, $\mathit{right}_p(B)$, and the set of sets $\mathit{mid}_p(B)$ contains the largest number of probabilistic states. This set is reported in $\mathit{large}_p(B)$.

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. So, 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 *state_to_constellation_cnt* for each action transition $s \xrightarrow{a} C$ 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 $C \backslash B_C$, which means that $s \in \mathit{mid}_a(B)$. In that case *residual_transition_cnt* for state s is larger than 0.

This is accomplished by traversing all incoming transitions $s \xrightarrow{a} u$ into B_C one extra time. If *residual_transition_cnt* for s is larger than 0 we need to replace the *state_to_constellation_cnt* for this transition $s \xrightarrow{a} u$ by the value of *state_to_constellation_cnt* − *residual_transition_cnt* of s. For all non-visited transitions $s \xrightarrow{a} u'$ where $u' \in C \backslash B_C$, the value of *state_to_constellation_cnt* must be set to *residual_transition_cnt* of s.

This is where we use that *state_to_constellation_cnt* is actually referred to by the pointer *state_to_constellation_cnt_ptr* (see Figure 4). When traversing the first transition of the form $s \xrightarrow{a} u$ with $u \in B_C$ such that *residual_transition_cnt* for s is larger than 0, a new entry in the array containing the variables *state_to_constellation_cnt* is constructed containing the value *state_to_constellation_cnt* − *residual_transition_cnt* and the auxiliary variable *transition_cnt_ptr* is used to point to this entry. At the same time the value in old entry in this array for *state_to_constellation_cnt* is replaced by the value *residual_transition_cnt* of state s. In this way the values of *state_to_constellation_cnt* of all transitions labelled with a from s to $C \backslash B_C$ are updated in constant time, i.e., without visiting the transitions that are not moved. For all transitions $s \xrightarrow{a} u'$ with $u' \in B_C$, the variable *state_to_constellation_cnt_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 4.1. The total time complexity of the algorithm is $O\left((m_a + m_p)\log n_p + (m_p + n_a)\log n_a\right)$ and the space complexity is $O\left(m_a + m_p + n_a\right)$.

Proof. In Algorithm 2 the cost of each computation step is indicated. The initialisation of the algorithm at lines 2 to 5 is linear in n_a, n_p and m_a. At line 3 calculating $\{S_A \mid A \subseteq Act\}$ 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 $|B_C| \leqslant \frac{1}{2}|C|$. 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 up till 13 respectively lines 16 up till 18 require steps proportional to the number of incoming action transitions respectively probabilistic transitions in B_C, apart from a sorting penalty which we treat separately below. The cumulative complexity of this part is therefore $O\left(m_a \log n_p + m_p \log n_a\right)$.

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 $|B'| \leqslant \frac{1}{2}|B|$. 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 $O\left((m_a + n_p)\log n_p + (m_p + n_a)\log n_a\right)$.

While marking, probabilistic states in $mid_p(B)$ need to be sorted. An ingenious argument by Valmari and Franceschinis [27] shows that this will at most contribute $O\left(m_p \log n_p\right)$ to the total complexity: Let K be the total number of times sorting takes place. Assume, for $1 \leqslant i \leqslant K$, that the total number of distributions in $mid_p(B)$ when sorting it for the i-th time is k_i. Clearly, $k_i \leqslant n_p$. 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 $C \backslash B_C$ 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, in order 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 [27]. 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, $\sum_{i=1}^{K} k_i \leqslant m_p$. Hence, the total time that is required by sorting is bounded as follows:

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

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 $m_p \geqslant n_p$, 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 $n_p \leqslant m_p$, 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 to formulate our complexity more compactly.

Corollary 4.2. 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 [3]. The algorithm uses extended ordered binary trees and is claimed to have a complexity of $O(mn(\log m + \log n))$ where m 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 $O\left(m_a n_a \log n_a + n_a n_p \log n_p + n_a^2 n_p\right)$. The last part $n_a^2 n_p$ is not mentioned in the analysis in [3]. It is due to taking the time into account for 'inserting $Pre(\alpha, \mu_i)$ into *v.states*' (see page 208 of [3]) for the version of ordered balanced trees used, and we believe it to be forgotten [2].

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 of [3]. However, in the more common case where the support of distributions is limited by some constant c, i.e., $m_p \leqslant c n_p$, we can simplify the space and time complexities to those in the following table.

	GRV (this article)	BEM [3]
Space complexity	$O\left(n_p\right)$	$O\left(n_a n_p \vert Act \vert\right)$
Time complexity	$O\left(n_p \log n_a\right)$	$O\left(n_a n_p \log n_a + \underline{n_a^2 n_p}\right)$

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 by GRV, and the reference algorithm by Baier, Engelen and Majster-Cederbaum [3], for which we use the abbreviation BEM, have been implemented in C++ as part of the mCRL2 toolset [7,11][1]. This toolset is available under a Boost license which means that the

[1] See www.mcrl2.org.

```
sort   $Direction = \textbf{struct}\ up \mid down \mid right \mid left\,;$

proc   $X(x, y : \mathbb{N}) =$
           $(x \approx 1 \vee x \approx max_x) \rightarrow dead{\cdot}X(x,y) \diamond$
           $(y \approx 1 \vee y \approx max_y) \rightarrow live.X(x,y) \diamond$
           $(\ \textbf{dist}\ d : Direction[1/4]\ .$
               $((d \approx up) \rightarrow step{\cdot}X(x+1,y) +$
                $(d \approx down) \rightarrow step{\cdot}X(x-1,y) +$
                $(d \approx right) \rightarrow step{\cdot}X(x,y+1) +$
                $(d \approx left) \rightarrow step{\cdot}X(x,y-1))\ )\,;$

init   $X(i_x, i_y)\,;$
```

Figure 5. The specification of ant-on-a-grid in mCRL2

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 [3] 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 [1,13]. 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 5, where the dimensions of the grid are max_x and max_y, and the initial position is given by i_x and i_y. 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 $c{\rightarrow}p \diamond q$ are the if-then and if-then-else of mCRL2. The curly equal sign ($\approx$) in conditions stands for equality applied to data expressions. The expression **dist** $d{:}Direction[1/4]$ 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 6 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 five hours requires approximately 10,000 seconds compared to 2 seconds for GRV. The particular example

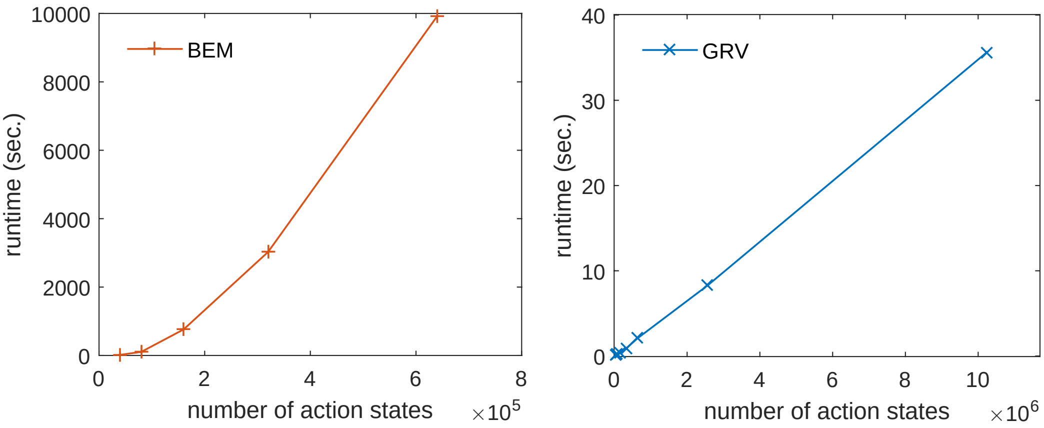

Figure 6. Scaling of runtime results for the ant-on-a-grid puzzle

regards a PLTS of 6.4×10^5 action states. The graphs clearly indicate that the difference grows when the probabilistic transition systems get larger.

In order to further understand the practical usability of the GRV algorithm, we applied it to a number of benchmarks taken from the PRISM Benchmark Suite[2] and the mCRL2 toolset[3]. The tests taken from PRISM were first translated into mCRL2 code to generate the corresponding PLTSs.

Table 2 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 its own seat (see [13] 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_stab_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 2, 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 1,000 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 hours.

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 100K action states. The one but last column of Table 2

[2] www.prismmodelchecker.org/benchmarks/
[3] www.mcrl2.org/

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 2 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 compare 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 of problem of the lost boarding pass, in [13]. However, having smaller state spaces will be beneficial anyway, 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 [4] as this appears to be required to extend this algorithm to weaker notions of equivalence [25], such as probabilistic branching bisimulation.

Author Contributions: Conceptualisation, JFG; Software and benchmarks, HJRV; Formal Analysis, JFG and EV; Original Draft Preparation, JFG; Writing, Review & Editing, EV, JFG

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

1. G. Antonick. Ant on a grid. New York Times. August 12, 2013 (http://wordplay.blogs.nytimes.com//2013/08/12/ants-2/).
2. C. Baier. Personal communication. 2018.
3. C. Baier, B. Engelen, M.E. Majster-Cederbaum. Deciding bisimilarity and similarity for probabilistic processes. Journal of Computional System Sciences 60(1):187–231, 2000.
4. E. Bandini and R. Segala. Axiomatizations for probabilistic bisimulation. In: F. Orejas, P.G. Spirakis, J. van Leeuwen (eds), Automata, Languages and Programming. ICALP 2001. Lecture Notes in Computer Science, vol 2076. pages 370-381, Springer, Berlin, Heidelberg, 2001.
5. S. Cattani and R. Segala. Decision algorithms for probabilistic bisimulation. In: L. Brim et al. (eds), Proc. 13th CONCUR, LNCS 2421, pages 371–386, Springer 2002.
6. S. Crafa and F. Ranzato. Bisimulation and simulation algorithms on probabilistic transition systems by abstract interpretation. Formal Methods in System Design 40(3):356–376, 2012.
7. S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stapper, E.P. de Vink, J.W. Wesselink, and T.A.C. Willemse. An overview of the mCRL2 toolset and its recent advances. In N. Piterman and S.A. Smolka (eds.), Proc. TACAS 2013, LNCS 7795, pages 199-213, Springer 2014.
8. C. Dehnert, J.-P. Katoen, and D. Parker. SMT-based bisimulation minimisation of Markov models. In R. Giacobazzi, J. Berdine, and I. Mastroeni (eds.), Proc. 14th. VMCAI, LNCS 7737, pages 28–47. Springer 2013.

Model	n_a	m_a	n_p	m_p	**min.** n_a	**min.** m_a	**min.** n_p	**min.** m_p	**time BEM**	**me. BEM**	**time GRV**	**me. GRV**	**speed-up**	**memory**
shared_coin_2_5	14,096	28,192	12,891	14,801	998	1,995	1,163	1,479	5.45	53.57	0.08	51.34	68	1.04
brp_100_20	15,003	15,003	10,803	15,003	8,504	8,504	8,504	12,704	37.27	82.51	0.06	55.79	621	1.48
self_stab_7	16,130	56,462	14,337	57,346	2,060	9,324	2,836	5,672	14.08	71.54	0.17	66.35	83	1.08
brp_100_40	29,003	29,003	20,803	29,003	16,504	16,504	16,504	24,704	368.67	176.34	0.22	65.41	1,676	2.70
airplane_4000	31,991	31,990	15,998	31,991	23,995	23,994	15,998	23,995	491.75	219.01	0.15	66.88	3,278	3.27
ant_100_100_grid	39,984	39,984	9,997	39,988	2,405	2,405	2,404	9,608	18.52	78.08	0.14	61.85	132	1.26
random_40	40,000	63,981	54,123	86,231	29,610	60,864	45,092	73,861	1,766.80	546.21	0.50	111.70	3,534	4.89
shared_coin_2_10	53,736	107K	48,131	551K	1,978	3,955	2,303	2,939	65.41	107.85	0.36	81.34	182	1.33
self_stab_8	65,026	260K	65,537	262K	6,306	32,960	9,721	19,442	401.17	294.44	0.69	157.17	581	1.87
brp_200_50	72,003	72,003	51,603	72,003	41,004	41,004	41,004	61,404	1,674.53	814.89	0.38	101.86	4,407	8.00
ant_200_100_grid	79,984	79,984	19,997	79,988	4,855	4,855	4,854	19,408	105.78	164.55	0.20	82.85	529	1.99
airplane_10000	79,991	79,990	39,998	79,991	59,995	59,994	39,998	59,995	2,805.12	1,073.63	0.37	113.83	7,581	9.43
random_100	100K	160K	134K	215K	73,607	151K	112K	183K	15,439.18	2,975.38	1.14	213.08	13,534	13.96
ant_200_200_grid	160K	160K	39,997	160K	9,805	9,805	9,804	39,208	760.17	484.78	0.41	124.84	1,854	3.88
shared_coin_2_20	210K	419K	185K	212K	3,938	7,875	4,583	5,859	654.94	440.18	1.19	198.80	550	2.21
self_stab_9	262K	1,175K	294K	1,180K	19,172	113K	32,806	65,612	4,015.53	2,809.32	3.24	598.14	1,239	4.70
shared_coin_3_2	280K	837K	274K	318K	5,841	17,347	7,722	10,068	1,781.64	974.92	1.78	294.68	1,001	3.31
ant_400_200_grid	320K	320K	79,997	320K	19,705	19,705	19,704	78,808	3,026.02	1,703.80	0.88	218.95	3,439	7.78
airplane_40000	320K	320K	160K	320K	240K	240K	160K	240K	-	-	1.34	333.17	-	-
random_400	400K	638K	540K	859K	293K	605K	446K	730K	-	-	5.11	729.84	-	-
shared_coin_2_30	468K	936K	413K	472K	5,898	11,795	6,863	8,779	2,427.55	1,248.44	2.95	389.13	823	3.21
ant_400_400_grid	640K	640K	160K	640K	39,605	39,605	39,604	158K	9,917.64	6,477.75	2.09	397.23	4,745	16.31
airplane_100000	800K	800K	400K	800K	600K	600K	400K	600K	-	-	4.22	755.948	-	-
random_800	800K	1,277K	1,079K	1,718K	587K	1,210K	893K	1,462K	-	-	11.98	1,418.79	-	-
brp_600_200	846K	846K	604K	846K	483K	483K	483K	724K	-	-	4.94	789.05	-	-
self_stab_10	1,046K	5,232K	1,311K	5,242K	58,026	383K	109K	218K	-	-	16.53	2,369.75	-	-
shared_coin_2_60	1,858K	3,716K	1,632K	1,866K	11,778	23,555	13,703	17,539	-	-	12.25	1,416.92	-	-
ant_800_800_grid	2,560K	2,560K	640K	2,560K	159K	159K	159K	636K	-	-	8.27	1,466.44	-	-
shared_coin_3_5	3,222K	9,665K	2,984K	3,437K	14,085	41,863	18,630	24,432	-	-	26.97	2,809.88	-	-
brp_1000_500	3,510K	3,510K	2,508K	3,510K	2,005K	2,005K	2,005K	3,007K	-	-	24.85	3,122.14	-	-
ant_1600_1600_grid	10,240K	10,240K	2,560K	10,240K	638K	638K	638K	2,553K	-	-	35.64	5,743.74	-	-
brp_2000_1000	14,020K	14,020K	10,016K	14,020K	8,010K	8,010K	8,010K	12,015K	-	-	115.95	12,351.47	-	-
brp_4000_2000	56,040K	56,040K	40,032K	56,040K	32,020K	32,020K	32,020K	48,030K	-	-	652.78	49,253.50	-	-

Table 2. Runtime (in sec.) and memory use (in MB) results for the reference algorithm (BEM) and the GRV algorithm

9. S. Dersavi, H. Hermanns, and H. Sanders. Optimal state-space lumping in Markov chains. Information Processing Letters 87:309–315, 2003.
10. U. Dorsch, S. Milius, L. Schröder, and T. Wissmann. Efficient coalgebraic partition refinement. In R. Meyer and U. Nestmann (eds.), Proc. 28th CONCUR, LIPIcs 85, pages 32:1–32:16, 2017.
11. J.F. Groote and M.R. Mousavi. Modeling and Analysis of Communication Systems. The MIT Press 2014. (See for the toolset www.mcrl2.org).
12. J.F. Groote, D.N. Jansen, J.J.A. Keiren, and A.J. Wijs. An $O(m \log n)$ algorithm for computing stuttering equivalence and branching bisimulation. ACM Transactions on Computational Logic 18(2):13:1–13:34, 2017.
13. J.F. Groote and E.P. de Vink. Problem solving using process algebra considered insightful. In J.-P. Katoen and R. Langerak and A. Rensink (eds.), ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday, LNCS 10500, pages 48–63. Springer 2017
14. H.A. Hansson and B. Jonsson. A logic for reasoning about time and reliability. Formal Aspects of Computing 6:512–535, 1994.
15. M. Hennessy. Exploring probabilistic bisimulations, part I. Formal Aspects of Computing 24:749–768, 2012.
16. J. Hillston, A. Marin, S. Rossi, and C. Piazza. Contextual lumpability. In A. Horváth et al., (eds), 7th international conference on Performance Evaluation Methodologies and Tools, pages 194–203. ICST/ACM, 2013.
17. P. Kannelakis and S. Smolka. CCS expressions, finite state processes and three problems of equivalence. Information and Computation 86:43–68, 1990.
18. J.-P. Katoen, T. Kemna, I. Zapreev, and D.N. Jansen. Bisimulation minimisation mostly speeds up probabilistic model checking. In O. Grumberg and M. Huth (eds.), 13th international conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 4424, pages 87-101. Springer 2007.
19. M. Kwiatkowska, G. Norman, and D. Parker. Stochastic model checking. In M. Bernardo and J. Hillston (eds.), Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation, LNCS 4486, pages 220–270. Springer 2007.
20. K.G. Larsen and A. Skou. Bisimulation through probabilistic testing. Information and Computation 94:1–28, 1991.
21. R. Paige and R.E. Tarjan. Three partition refinement algorithms. SIAM Journal of Computation 16(6):973–989, 1987.
22. R. Segala. Modeling and Verification of Randomized Distributed Real-Time Systems. PhD. thesis, Laboratory for Computer Science, MIT 1995. Available as Technical Report MIT/LCS/TR-676.
23. R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2(2):250-273.
24. L. Song, L. Zhang, H. Hermanns, and J.C. Godskesen. Incremental bisimulation abstraction refinement. ACM Transactions on Embedded Computing Systems 13(142)1–23.
25. A. Turrini and H. Hermanns. Polynomial time decision algorithms for probabilistic automata. Information and Computation 244:134–171, 2015.
26. A. Valmari. Simple bisimilarity minimization in $O(m \log n)$ time. Fundamenta Informaticae 105(3):319–339, 2010.
27. A. Valmari and G. Franceschinis. Simple $O(m \log n)$ time Markov chain lumping. In J. Esparza and R. Majumdar (eds.), Proc. 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems, LNCS 6015, pages 38–52. Springer 2010.
28. L. Zhang, H. Hermanns, F. Eisenbrand, and D.N. Jansen. Flow faster: efficient decision algorithms for probabilistic simulations. Logical Methods in Computer Science 4(4:6):1–43, 2008.
29. L. Zhang and D.N. Jansen. A space-efficient simulation algorithm on probabilistic automata. Information and Computation 249:138–159.