Stuttering equivalence is too slow!

David N. Jansen†  Jeroen J. A. Keiren*†

†Open University of the Netherlands, School of Computer Science, Netherlands
*Radboud Universiteit, Institute for Computing and Information Sciences, Nijmegen, Netherlands

Abstract
Groote and Wijs recently described an algorithm for deciding stuttering equivalence and branching bisimulation equivalence, acclaimed to run in \(O(m \log n)\) time. Unfortunately, the algorithm does not meet the acclaimed running time. In this paper, we present two counterexamples where the algorithms uses \(\Omega(md)\) time. In order to analyse the problem we present pseudocode of the algorithm, and indicate the time that can be spent on each part of the algorithm in order to meet the desired bound. We also propose fixes to the algorithm such that it indeed runs in \(O(m \log n)\) time.

1 Introduction
It has long been an open problem whether the algorithm by Groote and Vaandrager [2] for computing stuttering equivalence [1] and branching bisimulation [5] was optimal. Recently, Groote and Wijs [4, 3], presented an improvement. They describe an algorithm for deciding stuttering equivalence in time \(O(m \log n)\) and space \(O(m)\), where \(n\) is the number of states, and \(m\) the number of transitions of the Kripke structure at hand, with \(m \geq n\). This is an improvement over the previous running time of \(O(mn)\).

Unfortunately, the algorithm [3] falls short of the stated goal. In this paper we introduce two counterexamples where the algorithm will use more time than \(O(m \log n)\), namely \(\Omega(md)\), where \(d\) is the maximal outdegree of a state in the Kripke structure.

Since the original description of the algorithm relies heavily on auxiliary data structures and pointers in order to ensure that all information is available quickly when needed, without making the data structures any bigger than strictly necessary, the problem with the algorithm is hard to detect in this original description. We therefore first present our understanding of the algorithm as presented in [3] by giving a high-level description in pseudocode, which leaves out as much of the detailed data structures as possible. This also allows us to assign time budgets to its parts, that are satisfied locally and that together allow us to determine the overall time bound. Subsequently, we identify two problems in the original algorithm, by giving counterexamples that lead to running times higher than the desired bound of \(O(m \log n)\). For each of these problems, we indicate how to fix the algorithm, so that ultimately we can confirm the main result of [3] that stuttering equivalence and branching bisimulation can be computed in \(O(m \log n)\) time and \(O(m)\) space.

Throughout this paper we assume the reader is familiar with the definitions of Kripke structure and (divergence-blind) stuttering equivalence and with the auxiliary notions introduced in [3]. The note is best read while having a copy of [3] within reach for reference. We focus our analysis on deciding stuttering equivalence for Kripke structures. The results carry over directly to the computation of branching bisimulation.
Algorithm 1 The main algorithm for stuttering equivalence. Closely follows [3]

1.1 Initialise all temporary data. \( \mathcal{O}(m \log n) \)

1.2 while there is a nontrivial constellation do

1.3 Choose a nontrivial constellation \( S_p \) and a splitter \( S_p \subset S_p \) that is small (i.e., \( |S_p| \leq \frac{1}{2}|S_p| \)). \( \mathcal{O}(1) \) per splitter \( S_p \)

1.4 Create a new constellation \( \text{New} \) and move \( S_p \) from \( S_p \) to \( \text{New} \).

1.5 for all \( s \in S_p \) do

1.5.1 Find predecessors of \( S_p \) (except \( S_p \) itself)

1.6 for all \( s' \in \text{in} \tau(s) \) do

1.7 Mark the block of \( s' \) as refinable. \( \mathcal{O}(|\text{in} \tau(s)|) \)

1.8 Mark \( s' \) as predecessor of \( S_p \).

1.9 Register that \( s' \rightarrow s \) goes to \( \text{New} \) (instead of \( S_p \)).

1.10 end for

1.11 end for

1.12 Prepare \( S_p \) to be refined (i.e., mark states, register transitions). \( \mathcal{O}(|\text{out}(S_p)|) \)

1.13 begin {Stabilise the partition again:}

1.14 for all refinable blocks \( \text{Ref} \) do

1.15 Result := TRY_SPLIT(\( \text{Ref} \), \( \text{New} \), marked states \( \in \text{Ref} \),

1.16 unmarked bottom states \( \in \text{Ref} \))

1.17 TRY_SPLIT’(Result, \( S_p \), states \( \in \text{Result} \) with a transition to \( S_p \),

1.18 bottom states \( \in \text{Result} \) without transition to \( S_p \))

1.19 POSTPROCESSNEWBOTTOM()

1.20 end for

1.21 end

1.22 Destroy all temporary data (i.e., markings of states and blocks). \( \mathcal{O}(|\text{in}(S_p)|+|\text{out}(S_p)|) \)

1.23 end while

2 Pseudocode for Groote/Wijs 2016

We first present the main part of the algorithm from [3] as we understand it in terms of pseudocode, while separating out a routine TRY_SPLIT that tries to split a refinable block into states that can reach the splitter (called red states) and those that cannot (called blue states). There is a small difference between the calls to TRY_SPLIT in lines 1.15 and 1.16, which we will explain later. The high-level structure of the algorithm is presented in Algorithm 1. It maintains the invariant:

Invariant 1. The current blocks are stable with respect to the constellations, i.e., all states in a block can reach the same constellations through a (weak) transition.

A nontrivial constellation, i.e., a constellation containing multiple blocks, indicates that the current blocks are not yet stable with respect to themselves. The main loop separates a block \( S_p \) from a nontrivial constellation \( S_p \), moving it to a new constellation \( \text{New} \), and then restores the invariant by refining blocks with respect to these new constellations, if needed. The latter is done in TRY_SPLIT.

In the algorithms we assigned a time budget to some steps to facilitate the analysis of the algorithms’ complexity. We generally use the abbreviations \( |\text{in}(B)| = \sum_{s \in B} \max\{1,|\text{in}(s)|\} \) and \( |\text{out}(B)| = \sum_{s \in B} |\text{out}(s)| \) (note that in the latter case, it is not necessary to take the maximum, as the transition relation is total, i.e., \( |\text{out}(s)| > 0 \) for all \( s \in B \)). Similarly, \( \text{in}_r(s) \) are the inert incoming transitions and \( \text{out}_r(s) \) the inert outgoing transitions of \( s \).

2.1 Splitting blocks

The most important new idea from [3] is used in the step when a block is actually being refined. They start to find both the red and the blue states, spending the same amount of work on either part, until it becomes clear which one is the smaller. In other words, they use the idea “Process the smaller half” not only when looking for splitters, but also when refining blocks. The work
Algorithm 2 Refine a block into red and blue states, called in Line 1.15. Slightly improved

\begin{algorithm}
\begin{algorithmic}[1]
\Function{TrySplit}{Ref, Sp, Red, Blue}
\State Try to refine block \textit{Ref}, depending on whether states have transitions to the splitter constellation \textit{Sp}. \textit{Red} contains all states in \textit{Ref} with a strong transition to \textit{Sp}, and \textit{Blue} contains all bottom states in \textit{Ref} without transition to \textit{Sp}.
\Begin
\While{\textit{Red} contains unvisited states}
\ForAll{\textit{s} ∈ \textit{Red}}
\If{\textit{nottoblue}(\textit{s}) undefined}
\State \textit{nottoblue}(\textit{s}) := |\textit{out}(\textit{s})|\endIf
\If{\textit{nottoblue}(\textit{s}) = 0}
\State \textit{Blue} := \textit{Blue} \cup \{\textit{s}\}\EndIf
\EndFor
\EndWhile
\End
\While{\textit{Blue} contains unvisited states}
\State Choose an unvisited \textit{s} ∈ \textit{Blue}.
\State Mark \textit{s} as visited.
\ForAll{\textit{s′} ∈ \textit{in}(\textit{s}) \setminus \textit{New}}
\If{\textit{notred}(\textit{s′}) undefined}
\State \textit{notred}(\textit{s′}) := |\textit{in}(\textit{s′})|\endIf
\If{\textit{notred}(\textit{s′}) = 0}
\State \textit{Red} := \textit{Red} \cup \{\textit{s′}\}\EndIf
\EndFor
\EndWhile
\End
\State \textbf{Result} := \textit{Ref}
\End
\Function{Register}{\textit{s} ∈ \textit{Result}}
\ForAll{\textit{s′} ∈ \textit{out}(\textit{s}) \setminus \textit{New}}
\State Register that \textit{(Result, constellation of \textit{s′})} needs postprocessing.
\State Register that \textit{s} can reach the constellation of \textit{s′}.
\EndFor
\End
\State \Return{\textit{Result}}
\end{algorithmic}
\end{algorithm}

\begin{tabular}{ll}
\textit{whenever} |\textit{Blue}| > \frac{1}{2}|\textit{Ref}| \textbf{then} & \textit{whenever} |\textit{Red}| > \frac{1}{2}|\textit{Ref}| \textbf{then} \\
\State Stop this process. & \State Stop this process. \\
\textbf{end whenever} & \textbf{end whenever} \\
\end{tabular}
Algorithm 3 Refine as required by new bottom states, called in Line 1.17

3.1 function PostprocessNewBottom()
3.2 while there is a pair \((\hat{\hat{B}}, B)\) that needs postprocessing do
3.3 Choose a pair \((\hat{\hat{B}}, B)\) that needs postprocessing.
3.4 Delete \((\hat{\hat{B}}, B)\) from the pairs that need postprocessing.
3.5 if some, but not all new bottom states can reach \(B\) then
3.6 \(\text{TRYSplit}'(\hat{\hat{B}}, B, \text{states} \in \hat{\hat{B}}\text{ with a transition to } B,\)
3.7 \(\text{new bottom states} \in \hat{\hat{B}}\text{ without transition to } B)\)
3.8 \(\text{for all constellations } C\text{ that } \text{New} \text{ can reach do}\)
3.9 \(\text{if } (\hat{\hat{B}}, C)\text{ still needs postprocessing then}\)
3.10 \(\text{Register that } (\text{New}, C)\text{ needs postprocessing.}\)
3.11 end if
3.12 end for
3.13 end if
3.14 end while
3.15 Destroy all temporary data.
3.16 return

*New is the new block created in \(\text{TRYSplit}'\), Line 3.6*

spent on the refinement can then be bounded: state \(s\) is involved in such a refinement at most \(O(\log n)\) times. Upon every such refinement, at most \(O(\|\text{in}(s)\| + \|\text{out}(s)\|)\log n)\) time is spent for state \(s\). So, overall \(O(\|\text{in}(s)\| + \|\text{out}(s)\|)\log n)\) time is spent on state \(s\). Summing over all states then gives the desired time complexity \(O(m \log n)\).

Note that this may require detailed bookkeeping of the amount of work. One may balance the work by using an auxiliary variable, which stores the amount of work done on the red states minus the amount of work done on the blue states. Every time a state or transition is checked (i.e. every time the loop in Lines 2.7 or 2.10 is entered), the balance is increased or reduced by 1.

We deviate from \[3\] slightly: \textit{ibid.} uses a priority queue to keep track of \(\text{nottoblue}\), but actually nothing queue-like is needed, as the order of states is irrelevant for the correctness, time and memory bounds. Note that the data structure should allow to test for membership in time \(O(1)\).

They use the priority only to store the value \(\text{nottoblue}(s')\) and check whether this value is defined by a test for membership in the priority queue. We propose instead to set \(\text{nottoblue}(s')\) to some special value (e.g. 0) to indicate that it was not yet calculated. We also need a list or similar of all states whose \(\text{nottoblue}(s')\) is defined, to destroy the temporary data later.

### 2.2 New bottom states

While refining a block, it may happen that some states become bottom states because all their inert transitions become transitions from a red state to a blue state and therefore are no longer inert. Here, we also added a slight improvement (Lines 2.23–2.30): we only look for new bottom states in \(\text{Red}\).

It may happen that a new bottom state can no longer reach all the constellations that were reachable from the original bottom states. To repair Invariant \(\text{II}\), we may have to split some new bottom states off the block. Lines 2.28–2.29 prepare some temporary data for these splits.

The further splitting itself is shown in Algorithm 3. The basic idea is to check, for each constellation that is reachable from some new bottom state, whether the block has to be split. Of course, as soon as a block is split, both parts have to be checked for the remaining reachable constellations. The algorithm in \[3\] constructs, for each block \(B\) and constellation \(B\), a list \(S_B \subseteq B\) of states that can reach \(B\) (see Line 2.33). These lists are used in Line 3.6 to decide which states are blue, namely the new bottom states that are not in \(S_B\). The time budget \(O(\|\text{out}(\text{NBot})\|)\) can be met if the list \(S_B\) follows the same order as the list \(\text{NBot}\).

\(^1\)Note that priority queues typically have longer access times.
Algorithm [3] generally follows [3], except in Lines [5.7, 5.11] where we tried to find a formulation that fits in the time budget.

3 Counterexamples to time complexity

The algorithm contains two problems that cause it to be too slow. First, the second call to TrySplit, here referred to as TrySplit′, takes too long. We give a counterexample, and improve TrySplit′ to improve the complexity to the required bound. As indicated above, also the postprocessing of new bottom states as carried out in [3] is too slow. In section 3.3 we analyse the problem in the original algorithm; the proposed improvement has already been incorporated in Algorithm [3].

3.1 TrySplit′ is too slow

In the call to TrySplit′ (in Line 1.16), the initial set of red states is given implicitly, through a list of transitions. As a consequence, the test whether the potentially blue state s′ has a non-inert transition to Sp (this is why we require s′ /∈ Red in Line 2.10) in the variant TrySplit′ is executed in a different way compared to the one in TrySplit. Groote and Wijs [3] add this test just before Line 2.12. If s′ is marked (i.e. it has a transition to Sp), they can access one of their many auxiliary variables, but otherwise, “it can be checked by walking over the transitions s′ → s'' ∈ s'.Ttgt” (obviously, this is meant instead of the original “… ∈ s.Ttgt” – see Section 5.3, item 1.(b).ii.A.second bullet.first dash of [3]). So they execute a loop to verify out(s′) ∩ Sp = ∅, namely:

for all s'' ∈ out(s′) do
  if s'' ∈ Sp \ Ref then continue to Line 2.10
end for

This loop makes their algorithm slower than promised: the test uses time O(|out(s′)|), but should take at most O(1). Our first counterexample illustrates this time budget overrun.

Assume that the partition shown in Figure 1 has been reached. Then, we refine Sp: we select Sp, find its weak predecessors (the whole block Ref, so nothing is refined) and the weak predecessors of Sp\Sp (the right half of block Ref). Note that we do the latter without walking over the states in Sp\Sp: it is ok to spend O(|in(Sp)|) time here. We also save time by calculating the complement of the weak predecessors, i.e. the states in New, because it is smaller than Ref \ New: we are allowed to spend an additional O(|in(New)| + |out(New)|) time on this task.

As the algorithm looks through the predecessors of all states, it considers the state for inclusion in New. This happens k times (once for each transition to a state). The cited passage then requires that we check each time whether some immediate successor of a state is in Sp\Sp. If the transitions to the states are checked before the state, we have to check Θ(d) non-inert transitions to different constellations (plus possibly some inert ones), and in the end we find that the state should not be considered further. The problem is that we spend (much) time on a transition to New from a state that is possibly in New but in the end not actually in New. The part of the algorithm in Section 5.3, 1.(b).ii.A.second bullet, is only allowed to spend O(1) time on each such transition.

Overall, observe that the number of states and the number of transitions both are O(k + d). So, the algorithm is allowed to spend O((k + d) log(k + d)) time, but it spends Ω(kd) time. Variants of this Kripke structure with several copies of the state show that the checks can cost O(md) time.

Note that we are not allowed to concentrate on the red states (the weak predecessors of Sp\Sp, the states and state) themselves instead of the complement, as this set is larger.

\footnote{In particular, [3] does not define notobule( ). An ad-hoc solution would be to set it to some value > d, but that does not help if there are multiple copies of .}
Lemma 2. Refining in Lines 1.16 and 3.6 as described in [3] has a worst-case time complexity of \( \Omega(md) \).

We tried (in vain) to find a recursive counterexample, which should increase the time complexity to \( \Omega(md \log n) \), but every of our ideas led to a counterexample with so many additional transitions that it still fit the bound of Lemma 2.

3.2 A faster TrySplit'

We propose to solve this problem as follows: Execute the slow test at the latest possible moment, namely immediately before a state is inserted in Blue. This is shown in Algorithm 4. Here, we also present the formal parameters list according to the implicit representation of the red and blue states: a set \( \text{FromRed} \) of transitions from red states, a set \( \text{MaybeBlue} \) of possibly blue bottom states with a predicate \( \text{isBlueTest} \) that indicates which bottom states are actually blue. The overall time budget is still met: If the red states are the smaller part, then \( \text{FromRed} \) is a subset of \( \text{out}(\text{New}) \). If the blue states are the smaller part, then states in \( \text{MaybeBlue} \setminus \text{New} \) are marked (i.e. they have a transition to \( \text{Sp} \), see Line 1.8); we are allowed to walk over them one more time.

When the test is executed in Line 4.24, all inert transitions of \( s' \) point to blue states. If \( s' \) has a non-inert transition to \( \text{Sp} \), it is actually a red state, and in particular, a red new bottom state. (It may happen that we do not find all new bottom states here; therefore, we still have to execute Line 4.33.) As every state becomes a bottom state at most once, we are allowed to spend time \( \mathcal{O}(|\text{in}(s')| + |\text{out}(s')|) \log n \) then, which is abundant. If no such transition to \( \text{Sp} \) is found, \( s' \) is a blue state and we have to account for the time differently. It is \( \mathcal{O}(|\text{out}(s')|) \) per unmarked blue state \( s' \). Every time \( s' \) becomes an unmarked blue state, the test is executed exactly once, which fits in the general bound per time that \( s' \) is involved in a refinement.
Algorithm 4 Refine a block w.r.t. $Sp \setminus Sp$ (corrected), called in Line 1.16

4.1 function TrySplit(Ref, Sp, FromRed, MaybeBlue, isBlueTest)
4.2 {Try to refine block Ref, depending on whether states have transitions to the splitter constellation Sp. FromRed contains all transitions from Ref to Sp, MaybeBlue contains all bottom states that may be initially blue states, isBlueTest is a predicate that determines whether a candidate in MaybeBlue is definitely blue.}

4.3 begin {Spend the same amount of work on either process:}
4.4 Blue := ∅
4.5 whenever |Blue| > \(\frac{1}{2}|Ref|\) then
4.6 Stop this process.
4.7 end whenever
4.8 while Blue or MaybeBlue contain unvisited states do
4.9 Choose an unvisited $s \in Blue$ or $s \in MaybeBlue$.
4.10 Mark $s$ as visited.
4.11 if $s \notin Blue$ then
4.12 if $\neg isBlueTest(s)$ then
4.13 continue to Line 4.8
4.14 end if
4.15 Blue := Blue $\cup \{s\}$
4.16 end if
4.17 for all $s' \in in_\tau(s) \setminus Red$ do
4.18 if notnotblue($s'$) undefined then
4.19 notnotblue($s'$) := $|out_\tau(s')|$ \(\bigO(1)\)
4.20 end if
4.21 if notnotblue($s'$) $\neq 0$ then
4.22 if out\$_\tau(s') \cap Sp = ∅$ then
4.23 if $\neg isBlueTest(s)$ then
4.24 Blue := Blue $\cup \{s'\}$ \(\bigO(1)\)
4.25 end if
4.26 end if
4.27 end for
4.28 end while
4.29 Stop the other process.
4.30 Move Blue to a new block New.
4.31 Destroy all temporary data.
4.32 Find new non-inert transitions and bottom states (as Lines 2.23–2.30).
4.33 end
4.34 Register new bottom states for postprocessing (as Lines 2.33–2.38).
4.35 return

Red := ∅
4.4 whenever |Red| > \(\frac{1}{2}|Ref|\) then
4.5 Stop this process.
4.6 end whenever
4.7 while Red or FromRed contain unvisited elements do
4.8 Choose an unvisited $s \in Red$ or $s \rightarrow s'' \in FromRed$.
4.9 if $s \rightarrow s''$ is chosen then
4.10 Mark $s \rightarrow s''$ as visited.
4.11 if $s$ was visited earlier then
4.12 continue to Line 4.8
4.13 end if
4.14 Red := Red $\cup \{s\}$ \(\bigO(1)\)
4.15 end if
4.16 Mark $s$ as visited.
4.17 for all $s' \in in_\tau(s)$ do
4.18 Red := Red $\cup \{s'\}$ \(\bigO(1)\)
4.19 end for
4.20 end while
4.21 Stop the other process.
4.22 Move Red to a new block New.
4.23 Destroy all temporary data.
4.24 as Lines 1.8–4.29
4.25
\[O(\frac{|in_\tau(New)|}{|out(New)|} + |MaybeBlue| + |out(NBot)|)
4.26 \]
\[O(\frac{|in_\tau(New)|}{|out(New)|} + |out(New)|)
4.27 \]
3.3 \textbf{PostprocessNewBottom is too slow}

In Algorithm \ref{alg:algorithm}, we already included an improvement in Lines \ref{line:improvement1}--\ref{line:improvement5}. If some block $\hat{B}$ is split here, it should not take longer than $O(|\text{in}(\text{New})| + |\text{out}(\text{New})|)$. The original formulation did not take this into account; it always walked over all lists $S_B$ to separate them into the part that belongs to $\text{New}$ and the part that belongs to what remains in $\hat{B}$. With our terminology, it did:

\begin{algorithm}
\begin{algorithmic}
\State \textbf{for all} constellations $B$ such that $(\hat{B}, B)$ still needs postprocessing \textbf{do}
\State \textbf{for all} new bottom states $s \in$ the original $\hat{B}$ with a transition to $B$ \textbf{do}
\If{$s \in \text{New}$}
\State Register that $(\text{New}, B)$ needs postprocessing.
\State Move $s$ from $S_B$ for $\hat{B}$ to the corresponding list for $\text{New}$.
\If{$S_B$ for $\hat{B}$ is empty}
\State Register that $(\hat{B}, B)$ no longer needs postprocessing.
\EndIf
\EndIf
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

If $\text{New}$ is much smaller than $\hat{B}$, a budget overrun may result because the loop still spends (a little) time for each state in $\hat{B} \setminus \text{New}$ with a transition to $B$. This is illustrated in our second counterexample.

Assume that the partition in Figure \ref{fig:partition} has been reached. Then, we choose $Sp$ as splitter. The $\bullet$-state is a (weak) predecessor of $Sp$, but not of $Sp \setminus Sp$, and therefore is split off from the remainder of $\text{Ref}$. This turns all $\square$-states into new bottom states. First, it is registered that $(\text{Ref}, E_1), \ldots, (\text{Ref}, E_n)$ all need postprocessing. (Also $(\text{Ref}, Sp)$ needs postprocessing, but we will disregard it in the lower bound for timing.) Then, one for one, these pairs are handled. Suppose it starts with $(\text{Ref}, E_1)$. The algorithm will find that it has to split $\text{Ref}$ into two parts,
namely the \( q \)-state that is a predecessor of \( E_1 \) and the \( n - 1 \) other \( \overline{q} \)-states. Then, it walks over the \( n - 1 \) pairs \((\text{Ref}, E_i)\) that still need postprocessing and their lists \( S_{E_i} \), containing altogether \( 2 + 3 + \cdots + n \) states; from each list, it will remove the first \( \overline{q} \)-state. In total, \( \frac{1}{2}n(n - 1) - 1 \) list entries are read or removed. After that, the algorithm may handle \((\text{Ref}, E_2)\), split off one more \( \overline{q} \)-state from the rest, and walk over the \( n - 2 \) remaining pairs \((\text{Ref}, E_i)\). Here, \( \frac{1}{2}(n - 1)(n - 2) - 1 \) list entries are read or removed. For all the pairs up to \((\text{Ref}, E_n)\), the algorithm reads and finally removes \( \Theta(n^3) \) list entries.

The Kripke structure in Figure 2 has \( O(n) \) states and \( O(n^2) \) transitions. Therefore, the whole algorithm should run in time \( O(n^3 \log n) \). However, it takes at least \( \Omega(n^3) \) time. When we think of variants of this Kripke structure (e.g. with reduced outdegree of the \( \overline{q} \)-state and multiple \( \overline{q} \)-states having a transition to the same \( \overline{q} \)-state), we find that there are actually \( d \) iterations over \( \Theta(m) \) states.

**Lemma 3.** Postprocessing new bottom states as described in [3] has a worst-case time complexity of \( \Omega(md) \).

### 3.4 A faster POSTPROCESSNEWBOTTOM

The main idea for correcting the time bound was already hinted at earlier: Lines 3.7–3.11 try to distribute \( S_B \) over \textit{New} and what remains of \( \tilde{B} \) in time proportional to the outgoing transitions of \textit{New}, while keeping the order of \( S_B \) in line with the order of \textit{NBot}. This can be achieved if one distributes \( S_B \) simultaneously with distributing the states in \( \tilde{B} \) and their outgoing transitions themselves in Line 3.31. If all else fails, even constructing \( S_B \) for \textit{New} from scratch can fit the time bound \( O(|\text{out}(\text{New})|) \).

### 4 Conclusion

We showed that the algorithm of [3] for computing stuttering equivalence does not meet the acclaimed bound of \( O(m \log n) \); instead, there are examples showing it required \( \Omega(md) \) time with \( d \) the maximum outdegree in the Kripke structure.

Presentation of the algorithm in pseudocode enabled us to identify the parts of the algorithm that are responsible for the overrun of the time bound. We are convinced that, after correcting \textsc{TrySplit’} and \textsc{PostprocessNewBottom}, the time budgets (as indicated in the pseudocode) are met. Therefore, we are emboldened to confirm the main result of [3]:

**Theorem 4.** It is possible to calculate the stuttering equivalence of a Kripke structure in \( O(m \log n) \) time and \( O(m) \) memory (by using the corrected Algorithm [1]).

### References

[1] M.C. Browne, E.M. Clarke, and O. Grümberg. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(12):115–131, July 1988.

[2] J. F. Groote and F. W. Vaandrager. An efficient algorithm for branching and stuttering equivalence. In *Proc. 17th Int. Colloq. Autom. Lang. Program.*, volume 443 of LNCS, pages 626–638. Springer Berlin / Heidelberg, 1990.

[3] J. F. Groote and A. Wijs. An \( O(m \log n) \) algorithm for stuttering equivalence and branching bisimulation. arXiv e-prints 1601.01478, January 2016. \url{http://arxiv.org/abs/1601.01478}

[4] J. F. Groote and A. Wijs. An \( O(m \log n) \) algorithm for stuttering equivalence and branching bisimulation. 2016. Accepted for publication in TACAS 2016.

[5] R.J. van Glabbeek and W.P. Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM*, 43(3):555–600, 1996.