Computing Bisimulation-Based Comparisons

Linh Anh Nguyen

Institute of Informatics, University of Warsaw
Banacha 2, 02-097 Warsaw, Poland
nguyen@mimuw.edu.pl

Abstract. By using the idea of Henzinger et al. for computing the similarity relation, we give an efficient algorithm, with complexity $O((m+n)n)$, for computing the largest bisimulation-based auto-comparison and the directed similarity relation of a labeled graph for the setting without counting successors, where m is the number of edges and n is the number of vertices. Moreover, we provide the first algorithm with a polynomial time complexity, $O((m+n)^2n^2)$, for computing such relations but for the setting with counting successors (like the case with graded modalities in modal logics and qualified number restrictions in description logics).

1 Introduction

In a transition system, bisimilarity between states is an equivalence relation defined inductively as follows: x and x' are bisimilar if they have the same label, each transition from x to a state y can be simulated by a transition from x' to a state y' that is bisimilar to y, and vice versa (i.e., each transition from x' to a state y' can be simulated by a transition from x to a state y that is bisimilar to y') [5]. In modal logic, a Kripke model can be treated as a transition system, and the characterization of bisimilarity is as follows: two states are bisimilar iff they cannot be distinguished by any formula. The "only if" implication is called the invariance and does not require additional conditions, while the whole assertion is called the Hennessy-Milner property and holds for modally saturated models (including finitely-branching models) [10, 5, 1].

Simulation between states in a transition system is a pre-order defined inductively as follows: x' simulates x if they have the same label and, for each transition from x to a state y, there exists a transition from x' to a state y' that simulates y. Notice the lack of the backward direction. Similarity is an equivalence relation defined as follows: x and x' are similar if x simulates x' and vice versa. In modal logic, the characterization of simulation is as follows: x' simulates x iff x' satisfies all existential formulas (in negation normal form) satisfied by x (see, e.g., [1]). The "only if" implication is a kind of preservation and does not require additional conditions, while the whole assertion can also be called the Hennessy-Milner property and holds for modally saturated models.

Now, consider the pre-order $\lesssim$ between states in a transition system defined inductively as follows, $x \lesssim x'$ if: the label of x is a subset of the label of x';

for each transition from x to a state y, there exists a transition from x' to a state y' such that $y \lesssim y'$; and conversely, for each transition from x' to a state y', there exists a transition from x to a state y such that $y \lesssim y'$. In the context of description logic, such a relation is called the largest bisimulation-based auto-comparison of the considered interpretation [4, 3]. The relation $\simeq$, defined by $x \simeq x'$ iff $x \lesssim x'$ and $x' \lesssim x$, is an equivalence relation that can be called the directed similarity relation.[1] The characterization of $\lesssim$ is as follows: $x \lesssim x'$ iff x' satisfies all semi-positive formulas satisfied by x. This was proved for modally saturated interpretations in some description logics [4, 3]. The "only if" implication was proved earlier for some modal logics [8].

The above mentioned notions and their characterizations can be formulated appropriately for different kinds of transition systems and different variants or extensions of modal logic. For example, when transitions are labeled, the transitions used in each of the mentioned conditions should have the same label. This corresponds to the case of multimodal logics and description logics. Extensions of the basic description logic $\mathcal{ALC}$ may allow additional concept constructors, which may require more conditions for bisimulation or bisimulation-based comparison [3]. Some of such constructors are qualified number restrictions, which correspond to graded modalities in graded modal logics. In this work, the case with these constructors is called the setting with counting successors, and in this case, we use the symbol $\lesssim_c$ instead of $\lesssim$ (the latter is used for the setting without counting successors).

In this work, we consider bisimulation-based comparison and directed similarity, formulated for (finite) labeled graphs, and the objective is to provide efficient algorithms for computing the largest bisimulation-based auto-comparison (and hence also the directed similarity relation) of a labeled graph for two settings, with or without counting successors.

The related work is as follows. The Paige-Tarjan algorithms for computing bisimilarity [9] are currently the most efficient ones, with complexity $O((m+n)n)$, where m is the number of transitions and n the number of states. They were originally formulated for graphs w.r.t. both the settings with or without counting successors, but can be reformulated or extended for other contexts (like transition systems, Kripke models, or interpretations in description logics). It exploits the idea of Hopcroft's automaton minimization algorithm [7], but makes a generalization to deal with nondeterministic finite automata instead of deterministic ones. The currently most efficient algorithms, with complexity $O((m+n)n)$, for computing the similarity relation were first provided by Bloom and Paige [2] and by Henzinger et al. [6]. The former was formulated for transition systems and the latter was formulated for graphs. They are both devoted to the setting without counting successors. Divroodi [3] provided a simple algorithm for computing the largest bisimulation-based auto-comparison of an interpretation in a number of description logics.[2] It is not efficient, having a high

[1] The term "bisimulation-based comparison" of [4, 3] is a synonym of the term "directed simulation" of [8], and the term "directed similarity" is named in this spirit.

[2] It is like Algorithm 1 given on page 5 for the setting without counting successors.

polynomial time complexity for the case without counting successors, and an exponential time complexity for the case with counting successors (i.e., qualified number restrictions).

In this work, by using the idea of Henzinger et al. [6] for computing the similarity relation, we give an efficient algorithm, with complexity $O((m+n)n)$, for computing the largest bisimulation-based auto-comparison and the directed similarity relation of a labeled graph for the setting without counting successors. Moreover, we provide the first algorithm with a polynomial time complexity, $O((m+n)^2n^2)$, for computing such relations but for the setting with counting successors.

The rest of this paper is structured as follows. Section 2 formally introduces some notions. In Sections 3 and 4, we present our algorithms for the settings with or without counting successors, respectively, justify their correctness and analyze their complexity. Section 5 concludes this work.

2 Bisimulation-Based Comparisons

A *(finite) labeled graph* is a tuple $G = \langle V, E, A, L\rangle$, where V is a finite set of vertices, $E \subseteq V^2$ is a set of edges, A is a finite set of labels, and L is a function that maps each vertex to a subset of A.

From now on, let $G = \langle V, E, A, L\rangle$ be a given labeled graph.

For a binary relation R, we write $R(x, y)$ to denote $\langle x, y\rangle \in R$. For $x \in V$, we denote $post(x) = \{y \in V \mid E(x, y)\}$ and $pre(x) = \{y \in V \mid E(y, x)\}$.

A relation $Z \subseteq V^2$ is called a *bisimulation-based auto-comparison of G without counting successors* if it satisfies the following conditions for every $x, y, x', y' \in V$:

$$Z(x, x') \Rightarrow L(x) \subseteq L(x') \tag{1}$$

$$Z(x, x') \wedge y \in post(x) \Rightarrow \exists y' \in post(x')\ Z(y, y') \tag{2}$$

$$Z(x, x') \wedge y' \in post(x') \Rightarrow \exists y \in post(x)\ Z(y, y'). \tag{3}$$

A relation $Z \subseteq V^2$ is called a *bisimulation-based auto-comparison of G with counting successors* if it satisfies (1) and, for every $x, x' \in V$,

$$\begin{array}{l}\text{if } Z(x, x') \text{ holds, then there exists a bijection}\\ h : post(x) \to post(x') \text{ such that } h \subseteq Z.\end{array} \tag{4}$$

We write $x \lesssim x'$ (resp. $x \lesssim_c x'$) to denote that there exists a bisimulation-based auto-comparison Z of G without (resp. with) counting successors such that $Z(x, x')$ holds. Observe that both $\lesssim$ and $\lesssim_c$ are pre-orders, and $\lesssim_c$ is stronger than $\lesssim$ (i.e., the former is a subset of the latter). It can be shown that the relation $\lesssim$ (resp. $\lesssim_c$) is the largest (w.r.t. $\subseteq$) bisimulation-based auto-comparison of G without (resp. with) counting successors.

We write $x \simeq x'$ to denote that $x \lesssim x'$ and $x' \lesssim x$. Similarly, we write $x \simeq_c x'$ to denote that $x \lesssim_c x'$ and $x' \lesssim_c x$. We call $\simeq$ (resp. $\simeq_c$) the *directed similarity relation of G without (resp. with) counting successors*. Both $\simeq$ and $\simeq_c$ are equivalence relations.

3 The Case without Counting Successors

In this section, we present three algorithms for computing the largest bisimulation-based auto-comparison of a given labeled graph $G = \langle V, E, A, L\rangle$ in the setting without counting successors. The first two algorithms, SchematicComparison and RefinedComparison, are used to help to understand the last one, EfficientComparison. The idea of the EfficientComparison algorithm and the explanation via two simpler algorithms follow from the ones by Henzinger et al. for computing simulations [6].

All the three mentioned algorithms compute the sets $leq(v)$ and $geq(v)$ for $v \in V$, where $leq(v) = \{u \in V \mid u \lesssim v\}$ and $geq(v) = \{u \in V \mid v \lesssim u\}$ with $\lesssim$ being the relation defined in Section 2. For the explanations given in the current section, however, by $\lesssim$ we denote the following relation, which depends on and changes together with leq:

$$\{\langle u, v\rangle \in V^2 \mid u \in leq(v)\}. \tag{5}$$

The SchematicComparison algorithm (on page 5) initializes the sets $leq(v)$ and $geq(v)$, for $v \in V$, to satisfy the condition (1) with Z replaced by $\lesssim$. Then, while the condition (2) (resp. (3)) with Z replaced by $\lesssim$ and x, y, x' (resp. x', y', x) replaced by u, v, w is not satisfied, it updates the mappings leq and geq appropriately in order to delete the pair $\langle u, w\rangle$ (resp. $\langle w, u\rangle$) from $\lesssim$. It is easy to see that at the end the relation $\lesssim$ specified by (5) is the largest bisimulation-based auto-comparison of G without counting successors. That is, the SchematicComparison algorithm is correct.

The RefinedComparison algorithm (on page 5) refines the SchematicComparison algorithm by using two additional mappings $prevGeq$ and $prevLeq$, which approximate the mappings geq and leq, respectively. The mappings geq and leq are refined by reducing the sets $geq(v)$ and $leq(v)$, for $v \in V$, in each iteration of the "repeat" loop. The set $prevGeq(v)$ (resp. $prevLeq(v)$) stores the value of $geq(v)$ (resp. $leq(v)$) at some earlier iteration and is a superset of $geq(v)$ (resp. $leq(v)$). The refinement is done by making updates only for $w \in pre(prevGeq(v)) \setminus pre(geq(v))$ (resp. $w \in pre(prevLeq(v)) \setminus pre(leq(v))$) instead of for $w \in pre(geq(v))$ (resp. $pre(leq(v))$) as in the SchematicComparison algorithm. It is straightforward to check that the conditions I1, I2, I3 listed in the RefinedComparison algorithm are invariants of the "repeat" loop. When the algorithm terminates, we have $geq(v) = prevGeq(v)$ and $leq(v) = prevLeq(v)$ for all $v \in V$, and hence the relation $\lesssim$ specified by (5) satisfies the conditions (1)–(3) and is a bisimulation-based auto-comparison of G without counting successors. It is the largest one because the deletions of elements from the sets $geq(v)$ and $leq(v)$, for $v \in V$, are appropriate and done only when needed. Therefore, the RefinedComparison algorithm is correct.

The EfficientComparison algorithm (on page 6) modifies the RefinedComparison algorithm in that instead of maintaining the sets $prevGeq(v)$ and $prevLeq(v)$, for $v \in V$, it maintains the sets

$$removeGeq(v) = pre(prevGeq(v)) \setminus pre(geq(v)), \tag{6}$$

Algorithm 1: SchematicComparison

input : a labeled graph $G = \langle V, E, A, L \rangle$.
output : for each $v \in V$, the sets $geq(v)$ and $leq(v)$.

foreach $v \in V$ **do**
  $geq(v) := \{u \in V \mid L(v) \subseteq L(u)\}$;
  $leq(v) := \{u \in V \mid L(u) \subseteq L(v)\}$;
repeat
  // **assert:** I0: $\forall u, w \in V\ w \in geq(u) \leftrightarrow u \in leq(w)$
  if *there are* $u, v, w \in V$ *such that* $v \in post(u)$, $w \in geq(u)$ *and* $post(w) \cap geq(v) = \emptyset$ **then**
    $geq(u) := geq(u) \setminus \{w\}$, $leq(w) := leq(w) \setminus \{u\}$;
  if *there are* $u, v, w \in V$ *such that* $v \in post(u)$, $w \in leq(u)$ *and* $post(w) \cap leq(v) = \emptyset$ **then**
    $leq(u) := leq(u) \setminus \{w\}$, $geq(w) := geq(w) \setminus \{u\}$;
until *no change occurred during the last iteration*;

Algorithm 2: RefinedComparison

foreach $v \in V$ **do**
  $prevGeq(v) := V$, $prevLeq(v) := V$;
  if $post(v) = \emptyset$ **then**
    $geq(v) := \{u \in V \mid L(v) \subseteq L(u)$ and $post(u) = \emptyset\}$;
    $leq(v) := \{u \in V \mid L(u) \subseteq L(v)$ and $post(u) = \emptyset\}$;
  else
    $geq(v) := \{u \in V \mid L(v) \subseteq L(u)$ and $post(u) \neq \emptyset\}$;
    $leq(v) := \{u \in V \mid L(u) \subseteq L(v)$ and $post(u) \neq \emptyset\}$;
repeat
  // **assert:**
  // I1: $\forall v \in V\ geq(v) \subseteq prevGeq(v) \wedge leq(v) \subseteq prevLeq(v)$
  // I2: $\forall u, v, w \in V\ v \in post(u) \wedge w \in geq(u) \rightarrow post(w) \cap prevGeq(v) \neq \emptyset$
  // I3: $\forall u, v, w \in V\ v \in post(u) \wedge w \in leq(u) \rightarrow post(w) \cap prevLeq(v) \neq \emptyset$
  if *there exists* $v \in V$ *such that* $geq(v) \neq prevGeq(v)$ **then**
    foreach $u \in pre(v)$ *and* $w \in pre(prevGeq(v)) \setminus pre(geq(v))$ **do**
      if $w \in geq(u)$ **then**
        $geq(u) := geq(u) \setminus \{w\}$, $leq(w) := leq(w) \setminus \{u\}$;
    $prevGeq(v) := geq(v)$;
  if *there exists* $v \in V$ *such that* $leq(v) \neq prevLeq(v)$ **then**
    foreach $u \in pre(v)$ *and* $w \in pre(prevLeq(v)) \setminus pre(leq(v))$ **do**
      if $w \in leq(u)$ **then**
        $leq(u) := leq(u) \setminus \{w\}$, $geq(w) := geq(w) \setminus \{u\}$;
    $prevLeq(v) := leq(v)$;
until *no change occurred during the last iteration*;

Algorithm 3: EfficientComparison

```
1  foreach v ∈ V do
       // prevGeq(v) := V ,   prevLeq(v) := V
2      if post(v) = ∅ then
3          geq(v) := {u ∈ V | L(v) ⊆ L(u) and post(u) = ∅};
4          leq(v) := {u ∈ V | L(u) ⊆ L(v) and post(u) = ∅};
5      else
6          geq(v) := {u ∈ V | L(v) ⊆ L(u) and post(u) ≠ ∅};
7          leq(v) := {u ∈ V | L(u) ⊆ L(v) and post(u) ≠ ∅};
8      removeGeq(v) := pre(V) \ pre(geq(v));
9      removeLeq(v) := pre(V) \ pre(leq(v));
10 repeat
       // assert:
       // I4: ∀v ∈ V removeGeq(v) = pre(prevGeq(v)) \ pre(geq(v))
       // I5: ∀v ∈ V removeLeq(v) = pre(prevLeq(v)) \ pre(leq(v))
11     if there exists v ∈ V such that removeGeq(v) ≠ ∅ then
12         foreach u ∈ pre(v) and w ∈ removeGeq(v) do
13             if w ∈ geq(u) then
14                 geq(u) := geq(u) \ {w},  leq(w) := leq(w) \ {u};
15                 foreach w′ ∈ pre(w) do
16                     if post(w′) ∩ geq(u) = ∅ then
17                         removeGeq(u) := removeGeq(u) ∪ {w′};
18                 foreach u′ ∈ pre(u) do
19                     if post(u′) ∩ leq(w) = ∅ then
20                         removeLeq(w) := removeLeq(w) ∪ {u′};
           // prevGeq(v) := geq(v)
21         removeGeq(v) := ∅;
22     if there exists v ∈ V such that removeLeq(v) ≠ ∅ then
23         foreach u ∈ pre(v) and w ∈ removeLeq(v) do
24             if w ∈ leq(u) then
25                 leq(u) := leq(u) \ {w},  geq(w) := geq(w) \ {u};
26                 foreach w′ ∈ pre(w) do
27                     if post(w′) ∩ leq(u) = ∅ then
28                         removeLeq(u) := removeLeq(u) ∪ {w′};
29                 foreach u′ ∈ pre(u) do
30                     if post(u′) ∩ geq(w) = ∅ then
31                         removeGeq(w) := removeGeq(w) ∪ {u′};
           // prevLeq(v) := leq(v)
32         removeLeq(v) := ∅;
33 until no change occurred during the last iteration;
```

$$removeLeq(v) = pre(prevLeq(v)) \setminus pre(leq(v)). \tag{7}$$

It can be checked that the equivalences (6) and (7) are invariants of the "repeat" loop if the sets $prevGeq(v)$ and $prevLeq(v)$ are computed as in the RefinedComparison algorithm by uncommenting the corresponding lines. Thus, the EfficientComparison algorithm reflects the RefinedComparison algorithm and is correct.

We use the same idea and technique of [6] for analyzing the complexity of the EfficientComparison algorithm. Let $n = |V|$, $m = |E|$ and assume that $|A|$ is a constant. We also assume that the algorithm is modified by using and maintaining two arrays $countPostGeq[1..n, 1..n]$ and $countPostLeq[1..n, 1..n]$ of natural numbers such that $countPostGeq[w', u] = |post(w') \cap geq(u)|$ and $countPostLeq[w', u] = |post(w') \cap leq(u)|$ for all vertices w' and u from V. These arrays are initialized in time $O((m + n)n)$. Whenever a vertex w is removed from $geq(u)$ (resp. $leq(u)$), the counters $countPostGeq[w', u]$ (resp. $countPostLeq[w', u]$) are decremented for all predecessors w' of w. The costs of these decrements is absorbed in the cost of the "if" statements at the lines 16, 19, 27, 30 of the algorithm. Using the arrays $countPostGeq$ and $countPostLeq$, the test $post(w') \cap geq(u) = \emptyset$ at the line 16 and the similar ones at the lines 19, 27, 30 of those "if" statements can be executed in constant time (e.g., by checking if $countPostGeq[w', u] = 0$ for the case of the line 16).

The initialization of $geq(v)$ and $leq(v)$ for all $v \in V$ requires time $O(n^2)$. The initialization of $removeGeq(v)$ and $removeLeq(v)$ for all $v \in V$ requires time $O((m + n)n)$. Given two vertices v and w, if the test $w \in removeGeq(v)$ at the line 12 is positive in iteration i of the "repeat" loop, then that test is negative in all the iterations $j > i$. This is due to the invariant I4 and that

- in all the iterations, $w \in removeGeq(v)$ implies that $w \notin pre(geq(v))$,
- the value of $prevGeq(v)$ in all the iterations $j > i$ is a subset of the value of $geq(v)$ in the iteration i.

It follows that the test $w \in geq(u)$ at the line 13 is executed $\Sigma_v \Sigma_w |pre(v)| = O((m+n)n)$ times. This test is positive at most once for every w and u, because after a positive test w is removed from $geq(u)$ and never put back. Thus, the body of the "if" statement at the line 13 contributes time $\Sigma_w \Sigma_u (1 + |pre(w)| + |pre(u)|) = O((m + n)n)$. This implies that the "if" statement at the line 11 contributes time $O((m + n)n)$. Similarly, the "if" statement at the line 22 contributes time $O((m + n)n)$. Summing up, the (modified) EfficientComparison algorithm runs in time $O((m + n)n)$. We arrive at:

Theorem 3.1. *Given a labeled graph G with n vertices and m edges, the largest bisimulation-based auto-comparison of G and the directed similarity relation of G in the setting without counting successors can be computed in time $O((m+n)n)$.*

4 The Case with Counting Successors

In this section, we present the ComparisonWithCountingSuccessors algorithm (on page 8) for computing the largest bisimulation-based auto-comparison of

a given labeled graph $G = \langle V, E, A, L\rangle$ in the setting with counting successors. It uses the following data structures:

- $leq \subseteq V^2$,
- $remove \subseteq V^2$,
- $f : V^3 \to V$ is a partial mapping,
- $g : V^3 \to V$ is a partial mapping.

Algorithm 4: ComparisonWithCountingSuccessors

```
   input  : a labeled graph G = ⟨V, E, A, L⟩.
   output : the relation leq.
 1 set leq, remove to empty sets and f, g to empty mappings;
 2 foreach u, u' ∈ V do
 3     if L(u) ⊆ L(u') and |post(u)| = |post(u')| then
 4         leq := leq ∪ {⟨u, u'⟩},  W := post(u');
 5         foreach v ∈ post(u) do
 6             extract v' from W,  f(u, v, u') := v',  g(u', v', u) := v;
 7     else remove := remove ∪ {⟨u, u'⟩};
 8 while remove ≠ ∅ do
 9     extract (v, v') from remove;
10     foreach u ∈ pre(v) and u' ∈ pre(v') such that f(u, v, u') = v' do
11         undefine f(u, v, u') and g(u', v', u);
12         set g' to the empty mapping;
13         S := {v},  T' := ∅;
14         repeat
15             S' := {w' ∈ post(u') | ∃w ∈ S ⟨w, w'⟩ ∈ leq} \ T';
16             foreach w' ∈ S' do
17                 set g'(w') to an element w of S such that ⟨w, w'⟩ ∈ leq;
18             S := {g(u', w', u) | w' ∈ S' and w' ≠ v'},  T' := T' ∪ S';
19         until S' = ∅ or v' ∈ S';
20         if S' = ∅ then
21             delete the pair ⟨u, u'⟩ from leq;
22             undefine f(u, w, u') and g(u', w', u) for any w, w';
23             remove := remove ∪ {⟨u, u'⟩};
24             break;
25         w' := v';
26         repeat
27             w := g'(w'),  w'' := f(u, w, u');
28             f(u, w, u') := w',  g(u', w', u) := w,  w' := w'';
29         until w = v;
```

The variable *leq* will keep the relation to be computed $\lesssim_c$. As an invariant, *leq* is a superset of $\lesssim_c$. A pair $\langle u, u'\rangle$ is deleted from *leq* and inserted into the set

remove only when we already know that $u \not\lesssim_c u'$. At the beginning (in the steps 1–7), the relation *leq* is initialized to $\{\langle u, u' \rangle \in V^2 \mid L(u) \subseteq L(u') \wedge |post(u)| = |post(u')|\}$ and the relation *remove* is initialized to $V^2 \setminus leq$. Then, during the (main) "while" loop, we refine the relation *leq* by extracting and processing each pair $\langle v, v' \rangle$ from the relation *remove*. As invariants of that main loop, for all $u, w, u', w' \in V$,

$$leq \cap remove = \emptyset, \tag{8}$$

$$f(u, w, u') = w' \Rightarrow \langle u, w \rangle \in E \wedge \langle u, u' \rangle \in leq \wedge \langle u', w' \rangle \in E \wedge \langle w, w' \rangle \in leq \cup remove, \tag{9}$$

$$g(u', w', u) = w \Leftrightarrow f(u, w, u') = w', \tag{10}$$

$$\text{if } leq(u, u') \text{ holds, then the function } \lambda w \in post(u).f(u, w, u') \text{ is well-defined and is a bijection between } post(u) \text{ and } post(u') \text{ and } \{\langle w, w' \rangle \mid w \in post(u), w' = f(u, w, u')\} \subseteq leq \cup remove. \tag{11}$$

Processing a pair $\langle v, v' \rangle$ extracted from the set *remove* is done as follows. For each $u \in pre(v)$ and $u' \in pre(v')$ such that $f(u, v, u') = v'$, we want to repair the values of the data structures so that the above invariants still hold. We first undefine $f(u, v, u')$ and $g(u', v', u)$ (at the line 11). Figure 1 illustrates the "repeat" loop at the lines 14–19 and its initialization at the line 13: after the initialization, $S = \{v\}$ and $T' = \emptyset$; after the first iteration, $S' = S'_1$, $S = S_1$ and $T' = S'_1$; after the second iteration, $S' = S'_2$, $S = S_2$ and $T' = S'_1 \cup S'_2$; observe that $|S_1| = |S'_1|$ and $|S_2| = |S'_2|$. Since S' is disjoint with the value of T' in the previous iteration and T' is monotonically extended by S', the loop will terminate with $S' = \emptyset$ or $v' \in S'$.

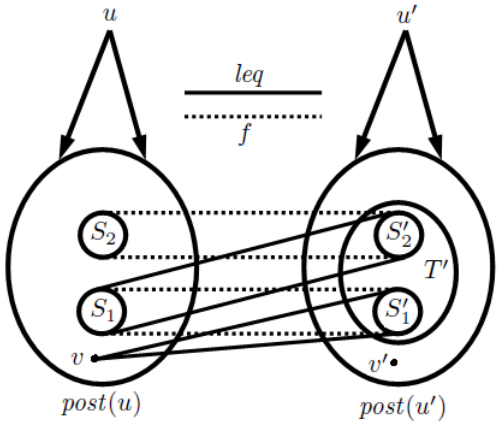

Fig. 1. An illustration for the steps 14–19 of Algorithm 4.

In the case $S' = \emptyset$, for $T = \{v\} \cup \{w \in post(u) \mid \exists w' \in T'\ f(u, w, u') = w\}$, we have that $|T| = |T'| + 1$ and $T' = \{w' \in post(u') \mid \exists w \in T\ \langle w, w' \rangle \in leq\}$ (to see this, one can use Figure 1 to help the imagination), and as a consequence, $u \not\lesssim_c u'$, because otherwise the condition (4) with Z replaced by $\lesssim_c$ cannot hold (recall that leq is a superset of $\lesssim_c$). So, in the case $S' = \emptyset$, we delete the pair $\langle u, u' \rangle$ from the relation leq, add it to the relation $remove$, and modify the mappings f and g appropriately.

In the case $v' \in S'$, the mappings f and g are repaired at the steps 25–29, using the mapping g' initialized at the step 12 and updated at the step 17. This is illustrated in Figure 2, where the dotted arrows from $post(u)$ to $post(u')$ represent some pairs $\langle w, f(u, w, u') \rangle$ such that $f(u, w, u') \in T'$ w.r.t. the old (i.e., previous) value of f, and the normal arrows from $post(u')$ to $post(u)$ represent some pairs $\langle w', g'(w') \rangle$ such that $w' \in T'$. The repair relies on updating f so that those dotted arrows are replaced by the inverse of those normal arrows, and updating g to satisfy the invariant (10).

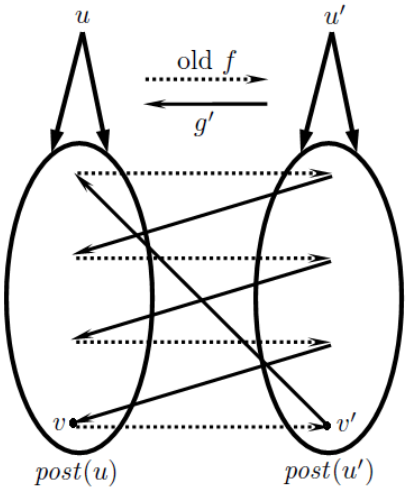

Fig. 2. An illustration for the steps 25–29 of Algorithm 4.

Technically, it can be checked that the conditions (8)–(11) are invariants of the (main) "while" loop. When the loop terminates, we have that $remove = \emptyset$, and the invariants (9) and (11) guarantee that leq satisfies the condition (4) with Z replaced by leq. As the condition (1) with Z replaced by leq holds due to the initialization of leq, it follows that the resulting relation leq is a bisimulation-based auto-comparison of the given labeled graph G in the setting with counting successors. As each pair $\langle u, u' \rangle$ deleted from leq during the computation satisfies $u \not\lesssim_c u'$, we conclude that the resulting relation leq is the largest bisimulation-based auto-comparison of G with counting successors.

Let $n = |V|$, $m = |E|$ and assume that $|A|$ is a constant. Our complexity analysis for the ComparisonWithCountingSuccessors algorithm is straightforward. The steps 1–7 for initialization run in time $O((m + n)n)$. The body of the "foreach" loop at the line 10 runs no more than $(m + n)^2$ times totally. The "repeat" loop at the lines 14–19 runs in time $O(n^2)$. The steps 25–29 run in time $O(n)$. Summing up, the algorithm runs in time $O((m + n)^2 n^2)$. We arrive at:

Theorem 4.1. *Given a labeled graph G with n vertices and m edges, the largest bisimulation-based auto-comparison of G and the directed similarity relation of G in the setting with counting successors can be computed in time $O((m+n)^2 n^2)$.*

5 Conclusions

By using the idea of Henzinger et al. [6] for computing the similarity relation, we have given an efficient algorithm, with complexity $O((m + n)n)$, for computing the largest bisimulation-based auto-comparison and the directed similarity relation of a labeled graph for the setting without counting successors. Moreover, we have provided the first algorithm with a polynomial time complexity, $O((m + n)^2 n^2)$, for computing such relations but for the setting with counting successors. One can adapt this latter algorithm to obtain the first algorithm with a polynomial time complexity for computing the similarity relation in the setting with counting successors. Our algorithms can also be reformulated and extended for interpretations in various modal and description logics (instead of labeled graphs).

Acknowledgements

This work was done in cooperation with the project 2011/02/A/HS1/00395, which is supported by the Polish National Science Centre (NCN).

References

1. P. Blackburn, M. de Rijke, and Y. Venema. *Modal Logic*. Number 53 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2001.
2. B. Bloom and R. Paige. Transformational design and implementation of a new efficient solution to the ready simulation problem. *Sci. Comput. Program.*, 24(3):189–220, 1995.
3. A.R. Divroodi. *Bisimulation Equivalence in Description Logics and Its Applications*. PhD thesis, University of Warsaw, 2015. Available at `http://www.mimuw.edu.pl/wiadomosci/aktualnosci/doktoraty/pliki/ali_rezaei_divroodi/ad-dok.pdf`.
4. A.R. Divroodi and L.A. Nguyen. Bisimulation-based comparisons for interpretations in description logics. In *Proceedings of Description Logics'2013*, volume 1014 of *CEUR Workshop Proceedings*, pages 652–669. CEUR-WS.org, 2013.
5. M. Hennessy and R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the ACM*, 32(1):137–161, 1985.

6. M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of FOCS'1995*, pages 453–462. IEEE Computer Society, 1995.
7. J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Available at `ftp://reports.stanford.edu/pub/cstr/reports/cs/tr/71/190/CS-TR-71-190.pdf`, 1971.
8. N. Kurtonina and M. de Rijke. Simulating without negation. *J. Log. Comput.*, 7(4):501–522, 1997.
9. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.
10. J. van Benthem. *Modal Correspondence Theory.* PhD thesis, Mathematisch Instituut & Instituut voor Grondslagenonderzoek, University of Amsterdam, 1976.