Minimal counter-example generation for SPIN

Paul Gastin[1] and Pierre Moro[2]

[1] LSV, ENS Cachan & CNRS
61, Av. du Prés. Wilson, F-94235 Cachan Cedex, France,
Paul.Gastin@lsv.ens-cachan.fr
[2] LIAFA, Univ. Paris 7
2 place Jussieu, F-75251 Paris Cedex 05, France
moro@liafa.jussieu.fr

Abstract. In this paper, we propose an algorithm to compute a counter-example of minimal size to some property in a finite state program, using the same programmation constraints than SPIN. This algorithm uses nested Breadth-first searches guided by priority queues. This algorithm works in quadratic time and is linear in memory,

1 Introduction

Model-checking is used to prove correctness of properties of hardware and software systems. When the program is incorrect, locating errors is important to provide hints on how to correct either the system or the property to be checked. Model checkers usually exhibit counter-examples, that is, faulty execution traces of the system. The simpler the counter-example is, the easier it will be to locate, understand and fix the error. A counter-example may mean that the abstraction of the system (formalized as the model) is to coarse; several techniques allow to refine it, guided by the counter-example found by the model-checker. The refinement stage can be done manually or automatically, but since even the automatic computation of refinements can be very expensive, it is very important to compute *small* counter-examples (ideally of minimal size) in case the property is not satisfied.

It is well-known that verifying whether a finite state system $\mathcal{M}$ satisfies an LTL property φ is equivalent to testing whether a Büchi automaton $\mathcal{A} = \mathcal{A}_{\mathcal{M}} \cap \mathcal{A}_{\neg\varphi}$ has no accepting run, where $\mathcal{A}_{\mathcal{M}}$ is a Kripke structure describing the system and $\mathcal{A}_{\neg\varphi}$ is a Büchi automaton describing executions that violate φ. It is easy, in theory, to determine whether a Büchi automaton has at least one accepting run. Since there is only a finite number of accepting states, this problem is indeed equivalent to finding a reachable accepting state and a loop around it. A counter-example to φ in $\mathcal{M}$ can then be given as a path $\rho = \rho_1\rho_2$ in the Büchi automaton, where ρ_1 is a simple (loop-free) path from the initial state to an accepting state, and ρ_2 is a simple loop around this accepting state (see Figure 1). Our goal is to find short counter-examples. The first trivial remark is that we can reduce the length of a counter-example if we do not insist on the

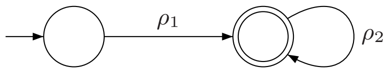

Fig. 1. An accepting path in a Büchi automaton

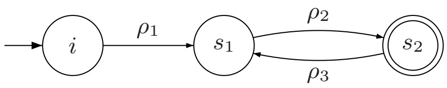

Fig. 2. An accepting path in a Büchi automaton

fact that the loop starts from an accepting state. Hence, we consider counter-examples of the form $\rho = \rho_1\rho_2\rho_3$ where $\rho_1\rho_2$ is a path from the initial state to an accepting state, and $\rho_2\rho_3$ is a simple loop (see Figure 2).

A minimal counter-example can then be defined as a path of this form, such that the length of ρ is minimal.

A minimal counter-example can of course be computed in polynomial time using minimal paths algorithms based on breadth first searchs (BFS). Since the model of the system frequently comes from several components working concurrently, the resulting Büchi automaton to be checked for emptiness may be huge. Therefore, memory is a *critical resource* and, for instance, we cannot afford to store the minimal distances between all pairs of states. Actually, even linear space may be a problem if the constant is too high. In tools like SPIN, only one integer and a few bits per state are stored for the computation of a "small" counter-example (it is well-known that SPIN does not compute a *minimal* counter-example). The aim of this paper is to give a polynomial time algorithm for computing a minimal counter-example using no more memory than SPIN does.

There exists several algorithms [CVWY91,HPY96,GMZ04,SE05] to check a Büchi automaton for emptiness and to construct a counter-example when the language is nonempty. All these algorithms use nested depth first search (DFS) and therefore they cannot be easily adapted to compute a minimal counter-example. It is also possible to use Tarjan like algorithms to find a counter-example, see e.g., [VG03].

In [GMZ04], an algorithm computing a minimal counter-example is presented. As far as the memory is concerned, this algorithm is as efficient as SPIN. However, it is still based on DFSs and its time complexity is exponential.

Our contribution is the following:

– We propose a polynomial time algorithm to compute a counter-example of minimal size. This algorithm does not use more memory than SPIN does with option `-i` when trying to reduce the size of counter-examples.
– We improve this algorithm with several optimizations.

One can notice that the problem of finding the smallest counter-example, given an LTL property and a finite system, is an NP-complete problem [KSF06].

However, we only focus in this paper on the probleme of finding a minimal accepting path in a Büchi automaton representing the product of the model and the negation of the property to be checked.

The paper is organized as follows. We first recall some notations and the development context in the Section 2. Then we present in Section 3 an algorithm that computes a minimal counter-example, and proove its correctness. We also present an algorithm to recover the trace of a counter-example when only the states s_1 and s_2 are known (see Figure 2). This is needed when using bit-state hashing techniques. In Section 4, we propose several optimizations in order to obtain a more efficient algorithm. We finally conclude.

2 Context and notations

Let $\mathcal{A} = (S, E, i, F)$ be a Büchi automaton where S is a finite set of states, $E \subseteq S \times S$ is the transition relation, $i \in S$ is the initial state and $F \subseteq S$ is the set of accepting states. Usually transitions are labeled with actions but since these labels are irrelevant for the emptiness problem, they are ignored in this paper. In pictures, the initial state is marked with an ingoing edge and accepting states are doubly circled.

Recall that a path in an automaton is a sequence of states $s_1 s_2 \cdots s_k$ such that for all $i = 1, \ldots, k-1$ there is a transition from s_i to s_{i+1}. We denote by $d(r, s)$ the *distance* between r and s, that is the length of a minimal path from r to s. Note that $d(r, s) = 0$ if $r = s$ and $d(r, s) = \infty$ if s is not reachable from r. A loop is a path $s_1 s_2 \cdots s_k$ with $k > 1$ and $s_k = s_1$. A path $s_1 s_2 \cdots s_k$ is *simple* if $s_i \neq s_j$ for all $i \neq j$. A loop $s_1 s_2 \cdots s_k$ is a *cycle* if $s_1 s_2 \cdots s_{k-1}$ is a simple path. A loop (resp. a cycle) is *accepting* if it contains an accepting state. Finally, an *accepting path* is of the form $\gamma = i \cdots s_k \cdots s_{k+\ell}$ where $i \cdots s_{k+\ell-1}$ is a simple path and $s_k \cdots s_{k+\ell}$ is an accepting cycle. We call $i \cdots s_k$ the *head* of γ. Note that an accepting path starts in the initial state. We also call *counter-example* an accepting path.

2.1 development context

When checking for emptiness a Büchi automaton that arises from a model and the negation of an LTL formula, we often run out of memory. Hence, it is crucial to use as little memory as possible. This is why SPIN only use one integer and a few bits per state when reducing the size of a counter-example. Our aim is to use no more memory than SPIN does. Since we want to compute shortest paths we will use BFS and store some distances. The memory constraint implies that only one distance per state can be stored at any given time of the algorithm.

3 An algorithm to find the smallest counter-example

We will describe an algorithm to compute a minimal counter-example. We do not include any optimization in this section. Section 4 will describe the improvements yielding an efficient algorithm that can be implemented.

Algorithm 1 An algorithm to generate a shortest path from r to r'

```
void BFS_trace (State r, State r')
Queue F;
F.enqueue(r,0); r.bfs_flag = true;
while F ≠ ∅ do
  (s,n) = F.dequeue();
  for all s' ∈ E(s) do
    if ¬ s'.bfs_flag then
      F.enqueue(s', n+1); s.bfs_flag = true;
      s.depth = n+1;
    end if
    if s' == r' then
      goto 15;
    end if
  end for
end while
DFS_trace(r,r');

void DFS_trace (State s, State r')
cp.push(s,s.depth);
if s == r' then
  exit all recursive calls of DFS_trace
end if
for all s' ∈ E(s) do
  if s'.depth == s.depth+1 then
    DFS_trace(s',r');
  end if
end for
cp.pop();
```

Actually, instead of computing directly a counter-example $\rho_1\rho_2\rho_3$ as described in Figure 2, we will only compute the key-states s_1 and s_2 so that ρ_2 is a path from s_1 to s_2. The next section shows how the counter-example can be reconstructed from s_1 and s_2.

3.1 Reconstructing the counter-example

Let $\rho_1\rho_2\rho_3$ be a minimal counter-example (see Figure 2). Assume that only the states s_1 and s_2 that are at the beginning and the end of ρ_2 are known. The problem is to reconstruct the counter-example.

If states are stored in an hash table as usual, one can recover the trace of the counter-example using BFS algorithms [CSRL01] that store each time a state is visited for the first time, a pointer to its father. It then suffices to apply this BFS from the initial state i to s_1 to generate ρ_1, then to apply it from s_1 to s_2 to generate ρ_2 and finally to apply it once more from s_2 to s_1 to generate ρ_3.

Algorithm 2 A BFS to store distances from the initial state

```
Queue BFS_distance(State i)
Queue F, Accept;
F.enqueue(i,0);
i.depth = 0; i.bfs_flag = true;
while (F ≠ ∅)  do
  (s,n) = F.dequeue();
  if (s ∈ F) then
    Accept.enqueue(s);
  end if
  for all s' ∈ E(s) do
    if ¬ s'.bfs_flag then
      s'.depth = n+1;
      F.enqueue(s',n+1);
      s'.bfs_flag = true;
    end if
  end for
end while
return Accept;
```

But if one wants to use bit-state hashing techniques [Hol98,WL93], one cannot generate the trace using a backward pointer technique. Once a state is removed from the queue of the BFS, then somehow the state is lost.

We propose a simple algorithm to reconstruct the counter-example, when bit-state hashing techniques are used. Since we know states i, s_1 and s_2 we only need to compute a shortest path between a pair (r, r') of states. We first use a BFS to store $d(r, s)$ for each state visited until r' is reached. Then we use a DFS starting from r, that visits a successor s' of a state s iff its distance to r is $d(r, s)+1$. This condition enforces the DFS to visit states in the order implied by their minimimal distance from r. Once r' is reached, the shortest path is stored in the DFS stack. The description is given in Algorithm 1.

3.2 Distances from the initial state

The first step is to compute with a BFS the distances beetween the initial state and each state. They correspond to the possible length of the path ρ_1 of the counter-example (see Figure 2). Moreover, we also store in a queue called `Accept`, all the accepting states that are reachable from the initial state. See Algorithm 2.

3.3 Another Breadth First search

Once Algorithm 2 has completed, we have stored in `Accept`, all reachables accepting states. We will now find the smallest counter-example going through one of these states, and we will repeat this operation for each accepting state. Note that, since we used a queue to store accepting states, we will start with the accepting state which is the closest to the initial state.

Algorithm 3 A BFS to construct the priority queue

```
Priority Queue BFS_PF(State r)
1: Queue F; Priority Queue PF;
2: F.enqueue(r,0); r.bfs_flag = true;
3: if r.depth < maxdepth then
4:   PF.enqueue(r, r.depth);
5: end if
6: while F ≠ ∅ do
7:   (s,n) = F.dequeue();
8:   for all s' ∈ E(s) do
9:     if ¬ s'.bfs_flag then
10:       F.enqueue(s', n+1)); s'.bfs_flag = true;
11:       if s'.depth + n + 1 < maxdepth then
12:         PF.enqueue(s', s'.depth + n + 1);
13:       end if
14:     end if
15:   end for
16: end while
17: return PF;
```

We denote by `r` the current accepting state we are working on. Algorithm 3 will fill a *priority queue* (see [CSRL01][3]) with the set of states reachable from `r`. The priority that will be associated with a state s will be $d(i,s)+d(r,s)$, i.e., $|\rho_1|+|\rho_3|$ in the sense of the Figure 2. We already know $d(i,s)$ from Algorithm 2. This information is stored as the s.`depth`. To fill the priority queue, we perform another BFS starting from `r` that visits all states reachable from `r`. We use a global variable `maxdepth` that contains the size of the smallest counter-example found so far (∞ if no counter-examples were already found).

Once Algorithm 3 has been performed, we have in the priority queue `PF` the states reachable from r ordered according to $d(i,s)+d(r,s)$. We will use this information to find the smallest counter-example passing through `r`.

Lemma 1.

1. *For all* $(s,n) \in \texttt{PF}$*, we have* $n = d(i,s)+d(r,s) < \texttt{maxdepth}$.
2. *For all state* s*, if* $d(i,s)+d(r,s) < \texttt{maxdepth}$ *then* $(s, d(i,s)+d(r,s)) \in \texttt{PF}$.

Proof. (1) For each state, we have $s.\texttt{depth} = d(i,s)$. The property is clear when $s = r$. Now, when s' is inserted in `PF` at line `12`, we have $n+1 = d(r,s')$ by classical properties of the BFS. Since this is guarded by the test in line `11`, the result follows.

(2) If $s = r$ then line `4` is executed and we get the result. Let now s' be such that $d(i,s')+d(r,s') < \texttt{maxdepth}$. Since $d(r,s') < \texttt{maxdepth}$ we deduce that $d(r,s') < \infty$ and s' is reachable from r. Hence s' will be considered and lines

[3] There are different implementations for a priority queue (binary heap, binomial heap, Fibonacci heap). They all give the same (theoretical) complexity for our purpose

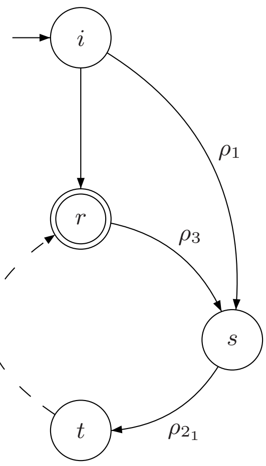

Fig. 3.

11-13 will be executed with s'. Since s'.depth $= d(i, s')$ and $n + 1 = d(r, s')$ we deduce from the hypothesis that $(s', d(i, s') + d(r, s'))$ is inserted in PF. □

3.4 BFS guided by a priority queue

Algorithm 4 finds the smallest counter-example whose loop goes through a specified repeated state r. Again, our search is limited by maxdepth but we omit this optimization from our intuitive descritption. After Algorithm 3 we have in the priority queue PF all pairs (s, n) with $n = d(i, s) + d(r, s)$ (Lemma 1). The aim is to find a state s such that $d(i, s) + d(r, s) + d^+(s, r)$ is minimal (here $d^+(s, r)$ denote the length of a shortest *nonempty* path from s to r). Note that the corresponding counter-example can then be reconstructed using Algorithm 1.

The idea is to use simultaneous (interleaved) BFSs. We begin with a BFS starting from some state s with $d(i, s)+d(r, s)$ minimal. Assume we have reached a state t (see Figure 3). If $d(i, s) + d(r, s) + d(s, t)$ is smaller than the minimal priority in PF then we continue the BFS from state t. If, on the other hand, there is some state s' with $d(i, s') + d(r, s') < d(i, s) + d(r, s) + d(s, t)$ then we start a new BFS from state s' instead. We use a single queue G for all the interleaved BFSs. In this queue, we store pairs (s, t) since, when we eventually reach r, we need to know from which state s we started with.

The algorithm proceeds in rounds (separated by # in the queue G). In the initialization phase, we put in G all pairs (s, s) with $n = d(i, s)+d(r, s)$ minimal. Then we consider all successors t' of states t such that (s, t) is in G for some s. The "rank" of these states t' is $n + 1$ and we add (s, t') to G for the next round if t' has not yet been reached. We also add for the next round the pairs (s, s)

Algorithm 4 Algorithm for finding the smallest counter-example

```
(State,State,int) Prio_min(State r, Priority Queue PF)
1: Queue G;
2: n = PF.PrioMin();
3: while (PF ≠ ∅ or G ≠ ∅) and (n + 1 < maxdepth) do
4:   /* Put in G pairs (s,s) such that s is in PF with priority n,
     without being marked.*/
5:   while (PF.min() == n) do
6:     (s,m) = PF.extract_min();
7:     if ¬ s.marked then
8:       G.enqueue(s,s);
9:       s.marked = true;
10:     end if
11:   end while
12:   G.enqueue(#);
13:   while G.head() ≠ # do
14:     (s,t) = G.dequeue();
15:     for all t′ ∈ E(t) do
16:       if t’ == r then
17:         return (s,n+1);
18:       else if ¬ t’.marked then
19:         G.enqueue(s,t’);
20:         t’.marked = true;
21:       end if
22:     end for
23:   end while
24:   G.dequeue(); /* symbol # */
25:   n++;
26: end while
27: return (r,∞);
```

such that $(s, n+1)$ is in `PF`. When we reach state r we have found our smallest counter-example whose loop goes through r.

Lemma 2. *Invariant for Algorithm 4: there is exactly one # in* `G` *between lines* 13-23 *and there is no # in* `G` *outside lines* 12-24*.*

Proof. At the beginning of the algorithm, `G` is empty. We insert a # in the queue at line 12 and no # is inserted or deleted between lines 13-23. Hence, the # inserted at line 12 is poped at line 24. The result follows. □

The invariants for the loops of Algorithm 4 are given by the following table

Invariants for loop 3 : (1, 2, 3, 4)
Invariants for loop 5 : (1, 2, 3, 4)
Invariants for loop 13 : (2, 3, 5, 6, 7)

where

$$\begin{aligned}
&\forall s && d(i,s) + d(r,s) + d^+(s,r) > n && (1)\\
&\forall t && t \text{ is marked} \vee (t,n) \in \mathtt{PF} \vee \forall s, d(i,s) + d(r,s) + d(s,t) > n && (2)\\
&\forall s,t && (s,t) \in G \Longrightarrow t \text{ is marked} && (3)\\
&\forall s,t && (s,t) \in G \Longrightarrow d(i,s) + d(r,s) + d(s,t) = n && (4)\\
&\forall s,t && (s,t) \in G \text{ before } \# \Longrightarrow d(i,s) + d(r,s) + d(s,t) = n && (5)\\
&\forall s,t && (s,t) \in G \text{ after } \# \Longrightarrow d(i,s) + d(r,s) + d(s,t) = n+1 && (6)\\
& && \mathtt{PF}.\mathtt{PrioMin}() > n && (7)
\end{aligned}$$

Loop 3. We first show that (1, 2, 3, 4) hold initially for loop 3, i.e., after line 2:

(1) Since $\mathtt{PF}.\mathtt{PrioMin}() = n$, we deduce from Lemma 1 that $d(i,s) + d(r,s) \geq n$ for all s. The result follows since $d^+(s,r) > 0$.

(2) Assume that $d(i,s)+d(r,s)+d(s,t) \leq n$ for some s. Since $\mathtt{PF}.\mathtt{PrioMin}() = n$, we deduce using Lemma 1 that $d(i,s) + d(r,s) = n$ and $d(s,t) = 0$. Using Lemma 1 again we obtain $(t,n) = (s,n) \in \mathtt{PF}$.

(3, 4) Holds trivially since G is empty.

Loop 5. Assuming that (1, 2, 3, 4) are invariants for loop 3, we obtain immediately that (1, 2, 3, 4) hold initially for loop 5. We show that they are preserved by the execution of lines (6-10):

(1) Clear since n is unchanged.

(2) If t is marked or $(t,n) \in \mathtt{PF}$ before line 6 then the same holds after line 10. Moreover n is unchanged in this loop hence the third part of (2) is also invariant.

(3) Clear since whenever a pair (s,s) is inserted in G at line 8 then s is marked at line 9 .

(4) When a pair (s,s) is inserted in G at line 8 then we have $d(i,s)+d(r,s) = n$ by Lemma 1.

Loop 13. First, note that (2) and (7) hold after line 11 and are invariants by lines (12-24): PF and n remain unchanged in the body of loop 13 and once a state is marked, it remains so forever.

Also, (3) holds after line 11 and when a pair (s,t') is inserted in G at line 19 then t' is marked at the next line. Hence, (3) is preserved by the execution of lines (14-22).

Now, since (4) holds after line 11 then (5, 6) hold after line 12 (by Lemma 2 there are no # in G except from lines (13-23) where there is exactly one # in G). Equation (5) is clearly preserved by lines (14-22) since new pairs are inserted in G after #.

It remains to show that (6) is preserved by lines (14-22). Consider the pair (s,t') inserted in G at line 19. By (5) we have $d(i,s) + d(r,s) + d(s,t) = n$. Since $t' \in E(t)$, we get $d(t,t') \leq 1$ and we deduce that $d(i,s)+d(r,s)+d(s,t') \leq n+1$. Now, t' was not marked (line 18) and $(t',n) \notin \mathtt{PF}$ by (7). We deduce from (2) that $d(i,s)+d(r,s)+d(s,t')>n$. Therefore, $d(i,s)+d(r,s)+d(s,t')=n+1$ and (6) still holds after the insertion of (s,t') in `G`.

Loop 3 continued. Finally, we have to show that (1, 2, 3, 4) still hold after line 25. We know that after line 23, the first element in `G` is # and that (2, 3, 6) hold. We deduce immediately that (3, 4) hold after line 25.

We consider (1), so assume that $d(i,s)+d(r,s)+d^+(s,r)=n+1$ for some s. Let t be such that $r\in E(t)$ and $d^+(s,r)=d(s,t)+1$. Then, we deduce that $d(i,s)+d(r,s)+d(s,t)=n$. Now, after line 11 we have $(t,n)\notin$ `PF` by (7). We deduce from (2) that t is marked. Let s' be such that $(s',t)\in G$. Since $r\in E(t)$ we deduce that line 17 will be executed before the end of loop 13. Therefore, if line 24 is reached, this means that $d(i,s)+d(r,s)+d^+(s,r)>n+1$ for all s. We deduce that (1) still holds after line 25 (if reached).

It remains to show that (2) still holds after line 25. This is a direct consequence of the following:

Claim. Assume that after line 23 there are s,t' such that t' is not marked and $d(i,s)+d(r,s)+d(s,t')\leq n+1$. Then, $(t',n+1)\in$ `PF`.

Let s,t' satisfy the hypotheses of the claim. By (7) we know that $(t',n)\notin$ `PF` hence, by (2), we get $d(i,s)+d(r,s)+d(s,t')>n$. Therefore, $d(i,s)+d(r,s)+d(s,t')=n+1$. We prove that $t'=s$ by contradiction. So assume that $t'\neq s$. Then $d(s,t')>0$ and there exists t such that $d(s,t')=d(s,t)+1$ and $t'\in E(t)$. We obtain $d(i,s)+d(r,s)+d(s,t)=n$. We deduce that t was already marked before line 12 by (7, 2). Therefore, there exists s' such that (s',t) has been inserted in `G` before line 12 (maybe in some previous execution of the body of loop 3). Therefore, after line 23, all successors of t have already been considered and must be marked. This is a contradiction with $t'\in E(t)$ and t' is not marked. Therefore, $t'=s$ and we have $d(i,s)+d(r,s)=n+1$. Since $n+1<$ `maxdepth` (test line 3), using Lemma 1 we obtain $(t',n+1)=(s,n+1)\in$ `PF`, which proves the claim.

Lemma 3. *Either $d(i,s)+d(r,s)+d^+(s,r)\geq$ `maxdepth` for all state s and Algorithm 4 exits at line 27, or Algorithm 4 exits at line 17 with a pair $(s,n+1)$ such that $d(i,s)+d(r,s)+d^+(s,r)=n+1<$ `maxdepth` and for all state s' we have $d(i,s')+d(r,s')+d^+(s',r)>n$.*

Proof. Follows easily from the invariants, in particular (1) and (5). □

3.5 Synthesis

We give now the complete algorithm which computes the smallest counter-example. This algorithm works in time $\mathcal{O}(|E|\cdot|F|\cdot\log(|S|))$, the factor $\log(|S|)$ is due to the operations on the priority queue. The algorithm works in linear space. More precisely, for each state we store an integer (depth field) and a few bits (`bfs_flag` or `marked`). The size of each queue is at most linear in the number of states.

Algorithm 5 The complete algorithm

```
Stack Minimal_Counter-example (State i)
Accept = BFS_distance(i);
maxdepth = ∞;
while Accept ≠ ∅ do
  State r = Accept.dequeue();
  Priority Queue PF = BFS_PF(r);
  (s,n) = Prio_min(r, PF)
  if n < maxdepth then
    s1 = s; s2 = r;
    maxdepth = n;
  end if
end while
if maxdepth < ∞ then
  Stack cp;
  BFS_trace(i,s1); BFS_trace(s1,s2); BFS_trace(s2,s1);
  return cp;
end if
return ∅;
```

4 Improvements

The first improvment is to use, before calling Algorithm 5, a nested-DFS algorithm such as [CVWY91,HPY96,SE05,GMZ04], or a Tarjan-like algorithm [VG03][4]. This allows to perform a linear time search to detect whether there exists some counter-example, and in this case it can also initialize `maxdepth` to the size of the counter-example found in order to speed-up Algorithm 5.

We can further improve the computation time by applying the following optimizations.

Improving the initial value of ***`maxdepth`***

For Algorithm 2, suppose that a counter-example has already been found and stored in a path called `cp`. Then, if you have used an algorithm like a nested-DFS, you know if a state is on the head of the counter-example (it will be `blue` (see [SE05,GMZ04] for more information on the `blue` flag[5]) and in the current stack). You will compute with Algorithm 2, the minimal distance beetween the initial state and all the states. So for each state that belongs to the head of the counter-example `cp`, one can compare its distance from the initial state inthe path `cp`, and its minimal distance. Then, if the latter is smaller, one can already update the maxdepth field at this point. These modifications are described in Algorithm 6, lines 4, 11 and 17-20.

[4] In fact, a nested-DFS algorithm can also prevent revisiting some states, see the end of Algorithm 6

[5] The blue color is described in these papers, but it is common to all the nested-DFS approaches

Algorithm 6 A BFS to store distances from the initial state

```
Queue BFS_distance(State i)
Queue F, Accept;
F.enqueue(i,0);
i.depth = 0; i.bfs_flag = true;
maxdepth = size(cp); n = 0; saved = 0
while (F ≠ ∅) ∧ (n < maxdepth) do
  (s,n) = F.dequeue();
  if (s ∈ F) then
    Accept.enqueue(s);
  end if
  for all s' ∈ E(s) do
    if s'.color != black and ¬ s'.bfs_flag then
      s'.depth = n+1;
      F.enqueue(s',n+1);
      s'.bfs_flag = true;
    end if
  end for
  if s.color == blue and s.is_in_cp and depth(s,cp) - n > saved then
    saved = depth(s,cp) - n;
    maxdepth = size(cp) - saved;
  end if
end while
return Accept;
```

Looking for counter-example in Algorithm 3

If a successor of a state is also the current accepting state, then we have found a counter-example (and it has the form of Figure 1). Since we know its length we can update `maxdepth` (see lines 19-21 in Algorithm 7).

Limitate the state space in Algorithm 3

We can also add a condition in the body of the loop saying that we are looking for counter-examples for which the loop size is at most `maxdepth` (see lines 9-11 in Algorithm 7).

Call to Algorithm 4 iff a smaller counter-example may exist

There is also in Algorithm 7, a local boolean named `loop`, which records if there exists an accepting path into the limitated state space (limited by maxdepth). If this boolean `loop` is false at the end of the execution, then there are no useful loop passing through `r` and there is no need to continue the computation for this state (see lines 6, 18 and 25-29 in Algorithm 7).

Including only useful states in `PF`

Recall that we are looking for a state s for which $d(i,s) + d(r,s) + d^{+}(s,r)$ is minimal. Algorithm 3 inserts in `PF` pairs $(s, d(i,s) + d(r,s))$ which are then used by Algorithm 4 to find some state which minimizes the quantity above.

Algorithm 7 A BFS to construct the priority queue

```
Priority Queue BFS_PF(State r)
1: Queue F; Priority Queue PF;
2: F.enqueue(r,0); r.bfs_flag = true;
3: if r.depth < maxdepth then
4:   PF.enqueue(r, r.depth);
5: end if
6: loop = false;
7: while F ≠ ∅ do
8:   (s,n) = F.dequeue();
9:   if n + 1 ≥ maxdepth then
10:     break;
11:   end if
12:   for all s′ ∈ E(s) do
13:     if ¬ s′.bfs_flag then
14:       F.enqueue(s′, n+1)); s′.bfs_flag = true;
15:       if (s′.depth + n + 1 < maxdepth) and (s′.depth < s.depth) then
16:         PF.enqueue(s′, s′.depth + n + 1);
17:       end if
18:       loop = loop ∨ (s′ == r);
19:       if (s′ == r) and (s′.depth + n + 1 < maxdepth) then
20:         maxdepth = s′.depth + n + 1;
21:       end if
22:     end if
23:   end for
24: end while
25: if loop then
26:   return PF;
27: else
28:   return ∅
29: end if
```

At line 10 of Algorithm 3, we have $d(r,s) = n$, $d(r,s') = n+1$ and $s' \in E(s)$. Then, $d^+(s,r) \leq 1 + d(s',r)$. We deduce that if $d(i,s) \leq d(i,s')$ then $d(i,s)+d(r,s)+d^+(s,r) \leq d(i,s')+d(r,s')+d^+(s',r)$. Therefore, if s' minimizes this quantity, so does s and there is no need to insert s' in the priority queue PF. This is prevented by the additional constraint on line 15 of Algorithm 7.

Note that this only saves some memory in the priority queue PF. Indeed, with the notation above, we have $d(i,s) + d(r,s) < d(i,s') + d(r,s')$ (still assuming that $d(i,s) \leq d(i,s')$). Hence, even if we insert $(s', d(i,s') + d(r,s'))$ in PF, when this pair is extracted from PF at line 6 of Algorithm 4, the state s' is already marked and therefore, (s', s') is not inserted in G.

5 Conclusion

We have proprosed an algorithms to compute the smallest counter-example of a property represented by a Büchi automaton. We have presented a set of improvements that can be immediatly used for an algorithm and another version that is less efficient in time, but more efficient in memory.

Our algorithm has nice properties. First, it can find all smallest counter-examples for all accepting states, if the variable `maxdepth` is always set to ∞.

Second, the ordering of the transitions has no impact on the computation time. For nested-DFS approaches, the result can strongly depends on the order of the transitions.

Third, our algorithm can also be used as a regular algorithm for bounded model checking. This is not the case for classical nested-DFS algorithms which fail to anwser properly for some graph configurations (depending on the ordering for the visit).

References

[CSRL01] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001.

[CVWY91] C. Courcoubetis, M. Vardi, P. Wolper, and M. Yannakakis. Memory efficient algorithms for the verification of temporal properties. In *Computer-aided verification '90 (New Brunswick, NJ, 1990)*, volume 3 of *DIMACS Ser. Discrete Math. Theoret. Comput. Sci.*, pages 207–218. Amer. Math. Soc., Providence, RI, 1991.

[GMZ04] Paul Gastin, Pierre Moro, and Marc Zeitoun. Minimization of counterexample in spin. In *SPIN Workshop*, Rutgers, Piscataway, NJ, 2004. American Mathematical Society.

[Hol98] G. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):287–305, November 1998. extended and revised version of Proc. PSTV95, pp. 301-314.

[HPY96] G. Holzmann, D. Peled, and M. Yannakakis. On nested depth first search. In *Proc. Second SPIN Workshop*, Rutgers, Piscataway, NJ, 1996. American Mathematical Society.

[KSF06] O. Kupferman and S. Sheinvald-Faragy. Finding shortest witnesses to the nonemptiness of automata on infinite words. In *Proc. 17th International Conference on Concurrency Theory*, volume 4137 of *Lecture Notes in Computer Science*, pages 492–508. Springer-Verlag, 2006.

[SE05] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190, Edinburgh, UK, April 2005. Springer.

[VG03] Anti Valmari and Jaco Geldenhuys. Tarjan's algorithm makes on-the-fly ltl verification more efficient. In *Proc. of 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2003.

[WL93] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Proc. 5th International Computer Aided Verification Conference*, pages 59–70, 1993.