Minimal Counterexample 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. We propose an algorithm to compute a counterexample of minimal size to some property in a finite state program, using the same space constraints than SPIN. This algorithm uses nested breadth-first searches guided by a priority queue. It works in time $\mathcal{O}(n^2 \log n)$ 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 counterexamples, that is, faulty execution traces of the system [CV03]. The simpler the counterexample is, the easier it will be to locate, understand and fix the error. A counterexample may mean that the abstraction of the system (formalized as the model) is too coarse; several techniques allow to refine it, guided by the counterexample 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* counterexamples (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 counterexample 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 counterexamples. The first trivial remark is that we can reduce the length of a counterexample if we do not insist on the fact

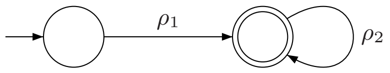

Fig. 1. An accepting path in a Büchi automaton

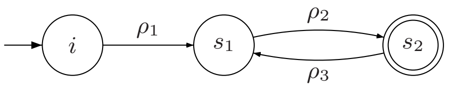

Fig. 2. An accepting path in a Büchi automaton

that the loop starts from an accepting state. Hence, we consider counterexamples 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 counterexample can then be defined as a path of this form, such that the length of ρ is minimal.

A minimal counterexample can of course be computed in polynomial time using minimal paths algorithms based on breadth first searches (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" counterexample (it is well-known that SPIN does not compute a *minimal* counterexample). The aim of this paper is to give a polynomial time algorithm for computing a minimal counterexample using no more memory than SPIN does, i.e., one integer and a few flags per state.

There exists several algorithms [CVWY92, HPY96, GMZ04, SE05] to check a Büchi automaton for emptiness and to construct a counterexample 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 counterexample. It is also possible to use Tarjan like algorithms to find a counterexample, see e.g. [Cou99, VG04].

In [GMZ04], an algorithm computing a minimal counterexample 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.

In [HK06], the authors propose an algorithm based on interleaved BFSs. They use three integers and some bits per state, which is more than SPIN does. Moreover, they need to explore the edges badkwards which would be difficult in practice with SPIN.

Our contribution is the following:

- We propose a polynomial time algorithm to compute a counterexample of minimal size. This algorithm only uses forward edges and does not use more memory than SPIN does when trying to reduce the size of counterexamples,

i.e., one integer and some bits per state. It is based on a BFS which is driven by a priority queue and can also be seen as several BFSs interleaved.
- We improve this algorithm with several optimizations.

Note that, we do not address the problem of finding the smallest counterexample, given an LTL property and a finite system. We only focus in this paper on the problem 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.

In the case of symbolic model checking, the problem is slightly different. In particular, in [SB05], the authors show that classical techniques for checking LTL properties (without past) give the smallest counterexample.

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 counterexample, and prove its correctness. We also present an algorithm to recover the trace of a counterexample 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 conclude in Section 5.

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 *counterexample* an accepting path.

2.1 Space Constraints

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 uses one integer and a few bits per state when reducing the size of a counterexample. 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 Counterexample

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

The main algorithm is presented in Section 3.5. It uses several algorithms that are presented first.

Actually, instead of computing directly a counterexample $\rho_1\rho_2\rho_3$ 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 counterexample can be reconstructed from s_1 and s_2.

3.1 Reconstructing the Counterexample

Let $\rho_1\rho_2\rho_3$ be a minimal counterexample (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 counterexample.

If states are stored in an hash table as usual, one can recover the trace of the counterexample using a BFS algorithm [CSRL01] that stores, when 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.

But if one wants to use bit-state hashing techniques [WL93, Hol98], one cannot generate the trace using the backward pointer technique. Since all informations about a state are not stored in the hash table, once a state is removed from the queue, the only remaining informations for this state are the one stored in the hash table, i.e., some flags and depth informations. A pointer to this memory location does not give complete information about the state.

We propose a simple algorithm to reconstruct the counterexample, when pointer to fathers cannot be used, e.g., 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 minimal distance from r. Once r' is reached, the shortest path is stored in the DFS stack. The description is given in Algorithm 1.

Note that, once the distances are computed by a BFS, a backward search in the graph starting from s and following edges for which the distance decreases until r (hence distance 0) is reached, allows to construct efficiently the shortest path from r to s. Unfortunately, backward searches cannot be used in practice

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

```
void BFS_trace (State $r$, State $r'$)
 1: Queue F;
 2: F.enqueue($r$,0); $r$.bfs_flag = true;
 3: while F $\neq \emptyset$ do
 4:    ($s$,n) = F.dequeue();
 5:    for all $s' \in E(s)$ do
 6:       if $\neg$ $s'$.bfs_flag then
 7:          F.enqueue($s'$, n+1); $s$.bfs_flag = true;
 8:          $s$.depth = n+1;
 9:       end if
10:       if $s' == r'$ then
11:          goto 15;
12:       end if
13:    end for
14: end while
15: DFS_trace($r$,$r'$);

void DFS_trace (State $s$, State $r'$)
 1: cp.push($s$,$s$.depth);; $s$.dfs_flag = true;
 2: if $s == r'$ then
 3:    exit all recursive calls of DFS_trace
 4: end if
 5: for all $s' \in E(s)$ do
 6:    if $\neg$ $s'$.dfs_flag and $s'$.depth == $s$.depth+1 then
 7:       DFS_trace($s'$,$r'$);
 8:    end if
 9: end for
10: cp.pop();
```

with SPIN since it would be hard to compute the set of predecessors of a state. Indeed, the number of potential predecessors may be use, e.g., if the state is reached by an assignment to some integer variable.

3.2 Distances from the Initial State

The first step is to compute with a BFS the distances between the initial state and each state. They correspond to the possible length of the path ρ_1 of the counterexample (see Figure 2). Moreover, we also store in a queue called `Accept`, all the accepting states that are reachable from the initial state. All this is quite standard and presented in Algorithm 2 for the sake of completeness.

3.3 Another Breadth First Search

Once Algorithm 2 has completed, we have stored in `Accept`, all reachable accepting states. We will now find the smallest counterexample going through one

Algorithm 2. A BFS to store distances from the initial state

```
Queue BFS_distance(State $i$)
 1: Queue F, Accept;
 2: F.enqueue($i$,0);
 3: $i$.depth = 0; $i$.bfs_flag = true;
 4: while (F $\neq \emptyset$) do
 5:    ($s$,n) = F.dequeue();
 6:    if ($s \in$ F) then
 7:       Accept.enqueue($s$);
 8:    end if
 9:    for all $s' \in E(s)$ do
10:       if $\neg$ $s'$.bfs_flag then
11:          $s'$.depth = n+1;
12:          F.enqueue($s'$,n+1);
13:          $s'$.bfs_flag = true;
14:       end if
15:    end for
16: end while
17: return Accept;
```

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.

We denote by `r` the current accepting state we are working on. Algorithm 3 will fill a *priority queue* (see [CSRL01][1]) 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 counterexample found so far (∞ if no counterexamples were found yet).

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

Lemma 1

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

Proof. (1) For each state, we have s.`depth` $= d(i, s)$. The property is clear when $s = r$. Now, when s' is inserted in `PQ` 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.

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

Algorithm 3. A BFS to construct the priority queue

```
Priority Queue BFS_PQ(State $r$)
1: Queue F; Priority Queue PQ;
2: F.enqueue($r$,0); $r$.bfs_flag = true;
3: if $r$.depth $<$ maxdepth then
4:   PQ.enqueue($r$, $r$.depth);
5: end if
6: while F $\neq \emptyset$ do
7:   ($s$,n) = F.dequeue();
8:   for all $s' \in E(s)$ do
9:     if $\neg$ $s'$.bfs_flag then
10:      F.enqueue($s'$, n+1)); $s'$.bfs_flag = true;
11:      if $s'$.depth + n + 1 $<$ maxdepth then
12:        PQ.enqueue($s'$, $s'$.depth + n + 1);
13:      end if
14:    end if
15:  end for
16: end while
17: return PQ;
```

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

3.4 BFS Guided by a Priority Queue

Algorithm 4 finds the smallest counterexample whose loop goes through a specified repeated state `r`. Again, our search is limited by `maxdepth` but we omit this optimization from our intuitive description. After Algorithm 3 we have in the priority queue `PQ` 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 $d^+(s, r)$ denotes the length of a shortest *nonempty* path from s to r). Note that the corresponding counterexample can then be reconstructed using Algorithm 1.

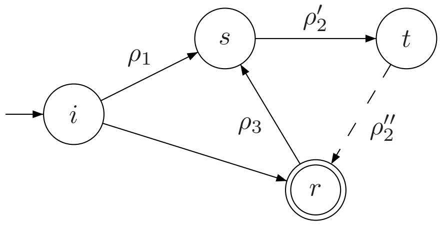

Fig. 3.

Algorithm 4. Algorithm for finding the smallest counterexample

```
(State,State,int) Prio_min(State r, Priority Queue PQ)
1: Queue G;
2: n = PQ.PrioMin();
3: while (PQ $\neq \emptyset$ or G $\neq \emptyset$) and (n + 1 < maxdepth) do
4:   /* Put in G pairs (s,s) such that s is in PQ with priority n,
     without being marked.*/
5:   while (PQ.min() == n) do
6:     (s,m) = PQ.extract_min();
7:     if $\neg$ 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() $\neq$ # do
14:    (s,t) = G.dequeue();
15:    for all t$'$ $\in E($t$)$ do
16:      if t' == r then
17:        return (s,n+1);
18:      else if $\neg$ 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,$\infty$);
```

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 PQ 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) such that $(s,n+1)$ is in PQ. When we reach state r we have found our smallest counterexample 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 popped 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

$$\forall s \qquad d(i,s) + d(r,s) + d^+(s,r) > n \tag{1}$$

$$\forall t \qquad t \text{ is marked} \vee (t,n) \in \texttt{PQ} \vee \forall s, d(i,s) + d(r,s) + d(s,t) > n \tag{2}$$

$$\forall s,t \qquad (s,t) \in G \Longrightarrow t \text{ is marked} \tag{3}$$

$$\forall s,t \qquad (s,t) \in G \Longrightarrow d(i,s) + d(r,s) + d(s,t) = n \tag{4}$$

$$\forall s,t \qquad (s,t) \in G \text{ before } \# \Longrightarrow d(i,s) + d(r,s) + d(s,t) = n \tag{5}$$

$$\forall s,t \qquad (s,t) \in G \text{ after } \# \Longrightarrow d(i,s) + d(r,s) + d(s,t) = n+1 \tag{6}$$

$$\texttt{PQ.PrioMin}() > n \tag{7}$$

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

(1) Since $\texttt{PQ.PrioMin}() = n$, 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 $\texttt{PQ.PrioMin}() = n$, we deduce using Lemma 1 that $d(i,s) + d(r,s) = n$ and $d(s,t) = 0$. Using Lemma 1 again we obtain $(t,n) = (s,n) \in \texttt{PQ}$.
(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 $(t,n) \in \texttt{PQ}$ 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): `PQ` 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 $(t', n) \notin \texttt{PQ}$ 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 $(t, n) \notin \texttt{PQ}$ 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, $(t', n + 1) \in \texttt{PQ}$.

Let s, t' satisfy the hypotheses of the claim. By (7) we know that $(t', n) \notin \texttt{PQ}$ 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 $n + 1 < \texttt{maxdepth}$ (test line 3), using Lemma 1 we obtain $(t', n + 1) = (s, n + 1) \in \texttt{PQ}$, which proves the claim.

Lemma 3. *Either* $d(i, s) + d(r, s) + d^+(s, r) \geq \texttt{maxdepth}$ *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* $d(i, s) + d(r, s) + d^+(s, r) = n + 1 < \texttt{maxdepth}$ *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). □

Algorithm 5. The complete algorithm

```
Stack Minimal_Counterexample (State $i$)
 1: Queue Accept = BFS_distance(i);
 2: maxdepth = $\infty$;
 3: while $Accept \neq \emptyset$ do
 4:    State r = Accept.dequeue();
 5:    Priority Queue PQ = BFS_PQ(r);
 6:    (s,n) = Prio_min(r, PQ)
 7:    if n < maxdepth then
 8:       $s_1$ = s; $s_2$ = r;
 9:       maxdepth = n;
10:    end if
11: end while
12: if maxdepth $< \infty$ then
13:    Stack cp;
14:    BFS_trace(i,$s_1$); BFS_trace($s_1$,$s_2$); BFS_trace($s_2$,$s_1$);
15:    return cp;
16: end if
17: return $\emptyset$;
```

3.5 Synthesis

We give now the complete algorithm which computes the smallest counterexample. 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`). In fact, these flags should be erased after each call to an algorithm, this is omitted for simplicity. The size of each queue is at most linear in the number of states.

4 Improvements

The first improvement is to use, before calling Algorithm 5, a nested-DFS algorithm such as [CVWY92, HPY96, SE05, GMZ04], or a Tarjan-like algorithm [Cou99, VG04][2]. This allows to perform a linear time search to detect whether there exists some counterexample, and in this case it can also initialize `maxdepth` to the size of the counterexample 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 counterexample has already been found and stored in a path called `cp`. Then, if an algorithm like a nested-DFS was used,

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

Algorithm 6. A BFS to store distances from the initial state

```
Queue BFS_distance(State $i$)
 1: Queue F, Accept;
 2: F.enqueue($i$,0);
 3: $i$.depth = 0; $i$.bfs_flag = true;
 4: maxdepth = size(cp); n = 0; saved = 0
 5: while (F $\neq \emptyset$) $\wedge$ (n < maxdepth) do
 6:    ($s$,n) = F.dequeue();
 7:    if ($s \in$ F) then
 8:       Accept.enqueue($s$);
 9:    end if
10:    for all $s' \in E(s)$ do
11:       if $s'$.color != black and $\neg$ $s'$.bfs_flag then
12:          $s'$.depth = n+1;
13:          F.enqueue($s'$,n+1);
14:          $s'$.bfs_flag = true;
15:       end if
16:    end for
17:    if $s$.color == blue and $s$.is_in_cp and depth($s$,cp) - n > saved then
18:       saved = depth($s$,cp) - n;
19:       maxdepth = size(cp) - saved;
20:    end if
21: end while
22: return Accept;
```

one knows if a state is on the head of the counterexample (it will be `blue` (see [SE05, GMZ04] for more information on the `blue` flag[3]) and in the current stack). Algorithm 2 computes the minimal distances between the initial state and all the states. So for each state that belongs to the head of the counterexample `cp`, one can compare its distance from the initial state in the 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.

Looking for counterexample in Algorithm 3
If a successor of a state is also the current accepting state, then we have found a counterexample (and it has the form of Figure 1). Since we know its length we can update `maxdepth` (see lines 19-21 in Algorithm 7).

Limiting the state space in Algorithm 3
We can also add a condition in the body of the loop saying that we are looking for counterexamples for which the loop size is at most `maxdepth` (see lines 9-11 in Algorithm 7).

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

Algorithm 7. A BFS to construct the priority queue

```
Priority Queue BFS_PQ(State $r$)
 1: Queue F; Priority Queue PQ;
 2: F.enqueue($r$,0); $r$.bfs_flag = true;
 3: if $r$.depth < maxdepth then
 4:    PQ.enqueue($r$, $r$.depth);
 5: end if
 6: loop = false;
 7: while $F \neq \emptyset$ do
 8:    ($s$,n) = F.dequeue();
 9:    if n + 1 $\geq$ maxdepth then
10:       break;
11:    end if
12:    for all $s' \in E(s)$ do
13:       if $\neg$ $s'$.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:             PQ.enqueue($s'$, $s'$.depth + n + 1);
17:          end if
18:          loop = loop $\vee$ ($s'$ == $r$);
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 PQ;
27: else
28:    return $\emptyset$
29: end if
```

Call to Algorithm 4 iff a smaller counterexample may exist
There is also in Algorithm 7, a local boolean named `loop`, which records if there exists an accepting path into the limited 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 `PQ`
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 `PQ` pairs $(s, d(i,s) + d(r,s))$ which are then used by Algorithm 4 to find some state which minimizes the quantity above.

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 PQ. This is prevented by the additional constraint on line 15 of Algorithm 7.

Note that this only saves some memory in the priority queue PQ. 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 PQ, when this pair is extracted from PQ 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 proposed an algorithm to compute the smallest counterexample of a property represented by a Büchi automaton. We have presented a set of improvements that can immediately be used to get a more efficient algorithm.

Our algorithm has nice properties. First, it can find all smallest counterexamples 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 for bounded explicit model checking, setting the maxdepth variable to some value. The algorithm properties ensure that it will found the smallest counterexample passing through the state space bounded by the maxdepth value. This is not the case for classical nested-DFS algorithms which fail to answer properly for some graph configurations (depending on the ordering for the visit).

References

[Cou99] Couvreur, J.M.: On-the-fly verification of linear temporal logic. In: FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Berlin Heidelberg New York (1999)

[CSRL01] Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education (2001)

[CV03] Clarke, E.M., Veith, H.: Counterexamples revisited: Principles, algorithms, applications. In: Verification: Theory and Practice. LNCS, vol. 2772, pp. 208–224. Springer, Heidelberg (2003)

[CVWY92] Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design 1(2/3), 275–288 (1992)

[GMZ04] Gastin, P., Moro, P., Zeitoun, M.: Minimization of counterexample in SPIN. In: Proc. of SPIN'04. LNCS, vol. 2989, pp. 92–108. Springer, Berlin Heidelberg New York (2004)

[HK06] Hansen, H., Kervinen, A.: Minimal counterexamples in O(n log n) memory and O(n^2) time. In: Proc. of ACDC'06, pp. 133–142. IEEE Computer Society Press, Los Alamitos, CA, USA (2006)

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

[HPY96] Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Proc. of SPIN'96. American Mathematical Society (1996)

[SB05] Schuppan, V., Biere, A.: Shortest counterexamples for symbolic model checking of LTL with past. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 493–509. Springer, Heidelberg (2005)

[SE05] Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)

[VG04] Valmari, A., Geldenhuys, J.: Tarjan's algorithm makes on-the-fly LTL verification more efficient. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 205–219. Springer, Heidelberg (2004)

[WL93] Wolper, P., Leroy, D.: Reliable hashing without collosion detection. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 59–70. Springer, Heidelberg (1993)