Simple Linear-Time Algorithms for Minimal Fixed Points*
(Extended Abstract)

Xinxin Liu and Scott A. Smolka

Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794 USA
{xinxin,sas}@cs.sunysb.edu

Abstract. We present global and local algorithms for evaluating minimal fixed points of dependency graphs, a general problem in fixed-point computation and model checking. Our algorithms run in linear-time, matching the complexity of the best existing algorithms for similar problems, and are simple to understand. The main novelty of our global algorithm is that it does not use the counter and "reverse list" data structures commonly found in existing linear-time global algorithms. This distinction plays an essential role in allowing us to easily derive our local algorithm from our global one. Our local algorithm is distinguished from existing linear-time local algorithms by a combination of its simplicity and suitability for direct implementation.
We also provide linear-time reductions from the problems of computing minimal and maximal fixed points in Boolean graphs to the problem of minimal fixed-point evaluation in dependency graphs. This establishes dependency graphs as a suitable framework in which to express and compute alternation-free fixed points.
Finally, we relate HORNSAT, the problem of Horn formula satisfiability, to the problem of minimal fixed-point evaluation in dependency graphs. In particular, we present straightforward, linear-time reductions between these problems for both directions of reducibility. As a result, we derive a linear-time local algorithm for HORNSAT, the first of its kind as far as we are aware.

1 Introduction

Model checking [CE81, QS82, CES86] is a verification technique aimed at determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [CW96].

Model checking has spurred interest in evaluating *fixed points*, as these can be used to express basic system properties, such as safety and liveness. Probably, the most canonical temporal logic for expressing fixed-point properties of systems is the

* Research supported in part by NSF grants CCR-9505562 and CCR-9705998, and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

modal mu-calculus [Pra81, Koz83], which makes explicit use of the dual fixed-point operators μ (least or minimal fixed point) and ν (greatest or maximal fixed point).

Model-checking algorithms come in two types: global and local. In a *global* algorithm, the entire transition system representing the system to be analyzed is constructed in advance of the model-checking computation; this can sometimes lead to exceedingly large memory requirements due to the *state explosion* problem. An alternative is *local* model checking [Lar88, SW91, Lar92], in which the state space is constructed incrementally, as the model-checking computation proceeds. An advantage of local model checking is that pruning is often possible: how much of the state space actually has to be explored depends on how much of it turns out to be relevant to establishing satisfaction of the formula to be verified.

In this paper, we present linear-time algorithms for global and local evaluation of minimal fixed points of *Dependency Graphs*, an abstract framework for fixed-point computation. Dependency graphs are a special case of the Partitioned Dependency Graphs (PDGs) presented in [LRS98]. We later provide linear-time reductions from both the problems of evaluating maximal and minimal fixed points of Boolean Graphs [And94] to that of evaluating minimal fixed points of dependency graphs. Thus, the generality of the problem of minimal fixed-point evaluation in dependency graphs subsumes that of many other problems of maximal and minimal fixed-point computation, such as those found in Boolean graphs [And94], Boolean equation systems [VL94], the alternation-free modal mu-calculus, and the alternation-free equational mu-calculus [CKS92, BC96].

Our algorithms are very simple in nature, perhaps deceptively so, and this is partially a consequence of the abstractness of the dependency graph framework. We describe our algorithms both at an abstract level of dependency-graph computation, and at a lower level in terms of arrays and lists, suitable for direct implementation.

Several linear-time global [CS93, AC88, And94] and local [And94, VL94] algorithms for similar fixed-point computation problems have previously appeared in the literature. The global algorithms in [CS93, AC88, And94] all use counters and "reverse list" data structures. In terms of the minimal fixed point of a Boolean equation system, the counter of a variable v keeps track of the number of remaining zero-valued variables in the right-hand side of v's conjunctive defining equation, while a reverse list records, for each variable v, the equation right-hand sides in which v appears. Reverse lists enable fast access to counters when modification is required. Counters and reverse lists are first seen in an algorithm for deciding the satisfiability of Horn Formulae [DG84].

A main novelty of our global algorithm is that it does not use counters and reverse lists. This distinction plays an essential role in allowing us to easily derive our local algorithm from our global one.

Our local algorithm is distinguished from existing linear-time local algorithms as follows. The local algorithm of [VL94] is expressed at an abstract level only; no implementation-level description is provided and the details of such an implementation, particularly one that is linear-time, are not obvious. The local algorithm of [And94] is considerably more complex than our local algorithm, involving mutually recursive calls between procedures *visit* ("visit a node") and *fwtn* ("find a witness").

In this paper, we also clarify the relationship between minimal fixed-point evaluation of dependency graphs and HORNSAT, the problem of Horn formula satisfiability (see, for example, [HW74]). In particular, we present straightforward, linear-time reductions between these problems for both directions of reducibility. As a result, we derive a linear-time local algorithm for HORNSAT, the first of its kind as far as we are aware.

Relatedly, Shukla et al. [SHIR96] consider the reduction of model checking in the alternation-free modal mu-calculus to HORNSAT, but not the other direction. They further mention that they "have developed HORNSAT-based methods to capture the tableau based local model checking in [Cle90] and [SW91]," but provide no details. In any event, a local algorithm for HORNSAT based on a tableau method is unlikely to be linear-time.

In [LRS98], a local algorithm for computing fixed points of arbitrary alternation depth is presented. Although in this paper we treat the problem of computing alternation-free fixed points, which is a special case of the problem solved by the algorithm in [LRS98], the algorithm presented here is based on completely different ideas, and as a result runs faster than the algorithm of [LRS98] on instances of alternation-free fixed points (linear vs cubic). We comment more in the Conclusions about how to combine the ideas of the two algorithms, in search for faster algorithms for arbitrary fixed points.

The structure of the rest of this paper is as follows. Section 2 presents our dependency-graph framework. Our linear-time global and local algorithms are the subject of Section 3. Section 4 contains our reductions from minimal and maximal Boolean graph fixed-point computation to dependency graph evaluation, while the interreducibility results involving HORNSAT are presented in Section 5. Finally, Section 6 concludes.

2 Dependency Graphs

A *dependency graph* is a pair (V, E), where V is a set of *vertices* and $E \subseteq V \times \wp(V)$ is a set of *hyper-edges*. For a hyper-edge $e = (v, S)$ of a dependency graph G, we call v the *source* of e and S the *target set* of e.

Example 1. $G_0 = (V_0, E_0)$ is a dependency graph where $V_0 = \{u, v, w\}$, $E_0 = \{(u, \emptyset), (u, \{v, w\})(v, \{u, w\}), (w, \{u, v\})\}$. ■

Let $G = (V, E)$ be a dependency graph. An *assignment* A of G is a function from V to $\{0, 1\}$. A *post-fixed-point assignment* F of G is an assignment such that, for every $v \in V$, whenever $(v, S) \in E$ such that for all $v' \in S$ it holds that $F(v') = 1$, then $F(v) = 1$. If we take the standard partial order $\sqsubseteq$ between assignments of dependency graph G such that $A \sqsubseteq A'$ just in case $A(v) \leq A'(v)$ for all $v \in V$ (using the usual order $0 < 1$), then by a straightforward application of the Knaster-Tarski fixed-point theorem, there exists a unique minimum post-fixed-point assignment which we will write as $F^G_{\min}$.

Our aim is to find efficient algorithms which, for a given $G = (V, E)$, computes $F^G_{\min}$ or $F^G_{\min}(v)$, for some $v \in V$. The former problem is one of *global* fixed-point computation, and the latter is one of *local* fixed-point computation.

For the reader familiar with *boolean equation systems* [VL94], a dependency graph (V, E) can be viewed as a boolean equation system with $x \in V$ having the equation $x = \bigvee_{(x,S)\in E} \bigwedge_{y\in S} y$, and $F^G_{\min}$ corresponding to the minimal solution of the boolean equation system. For the dependency graph of Example 1, the boolean equation system is $u = \mathbf{true} \vee (v \wedge w), v = u \wedge w, w = u \wedge v$. Subsequently, in Section 4, we also discuss the relationship between dependency graphs and *boolean graphs* [And94].

3 Algorithms

In this section, we present our two algorithms, a global algorithm that computes $F^G_{\min}$, and a local algorithm that computes $F^G_{\min}(v)$, where G is the input dependency graph and v is a vertex of G. We first describe the global algorithm using mathematical entities of functions and sets, without considering the detailed implementation data structures. The correctness of the algorithm is easy to see from such a description. Later we will furnish the global algorithm with more efficient data structures. We then modify the global algorithm to obtain the local algorithm.

3.1 The Global Algorithm

The high-level description of the global algorithm is given in Figure 1. The algorithm takes as input a dependency graph $G = (V, E)$ and computes an assignment A that is an approximation of $F^G_{\min}$ and such that upon termination $A = F^G_{\min}$.

```
W := E;
for v ∈ V do
    A(v) := 0; D(v) := ∅;
od
while W ≠ ∅ do
      let e = (v, S) ∈ W;
      W := W − {e};
      if ∀v' ∈ S. A(v') = 1 then
                                  A(v) := 1; W := W ∪ D(v);
                            else
                                  let v' ∈ S & A(v') = 0;
                                  D(v') := D(v') ∪ {e};
      fi
od
```

Fig. 1. Algorithm Global1

Besides A, the algorithm also maintains a set W containing all of the hyper-edges waiting to be processed, and a function $D : V \to \wp(E)$ that, for each $v \in V$, records in $D(v)$ the hyper-edges which have been processed under the assumption that $A(v) = 0$. Here we adopt the common idea of using A to approximate $F^G_{\min}$ from below. Thus, in the initialization phase, $A(v)$ is set to 0, $D(v)$ to $\emptyset$, for every $v \in V$, and W to E, the hyper-edge set of G. The main part of the algorithm consists of a single while-loop that repeatedly processes the hyper-edges remaining in W. The algorithm terminates upon completion of the while-loop.

The intuition behind the processing of each $e = (v, S) \in W$ in the body of the while-loop is as follows. There are two cases. If $A(v') = 1$ for all $v' \in S$ (a special case is $e = (v, \emptyset)$), then since A approximates $F^G_{\min}$, which is a post-fixed point assignment of G, $A(v)$ is now forced to be updated to 1. At the same time, each $e' \in D(v)$ must be re-processed since the assumption of $A(v) = 0$ no longer holds. The second case is that there exists a $v' \in S$ such that $A(v') = 0$. In this case $A(v)$ will not be forced to become 1 and we add e to $D(v')$ to indicate that e has not been used to force $A(v)$ to be 1 assuming $A(v') = 0$. As such, if $A(v')$ is later forced to be 1, then all $e \in D(v')$ depending on the now false assumption can be re-processed.

Lemma 1. *The while-loop of algorithm Global1 has the following invariant properties:*

1. $A \sqsubseteq F^G_{\min}$;
2. *For all* $v \in V$ *with* $A(v) = 0$, *whenever* $(v, S) \in E$ *then either* $(v, S) \in W$ *or* $(v, S) \in D(v')$ *for some* $v' \in S$ *with* $A(v') = 0$.

Proof Upon entering the while-loop the first invariant property holds trivially. The only place at which A is updated in the while-loop is the *true* branch of the if-statement. In this case, $A(v)$ is changed from 0 to 1, for some $v \in V$, and the condition is that there exists $(v, S) \in E$ such that $\forall v' \in S.\ A(v') = 1$. Assume that the invariant property holds before the update; we are left to show that $A(v) \leq F^G_{\min}(v)$ after the update. By the condition of the update and the inductive assumption, $\forall v' \in S.\ F^G_{\min}(v') = 1$, and since $F^G_{\min}$ is a post-fixed-point assignment, $F^G_{\min}(v) = 1$. Thus, after the update, the invariant property is maintained. The second invariant property is easy to verify. ■

Theorem 2. *Upon termination of running algorithm Global1 on a given dependency graph* $G = (V, E)$, *it holds that* $F^G_{\min} = A$.

Proof By 1. of Lemma 1, it certainly holds that $A \sqsubseteq F^G_{\min}$ upon termination. To conclude the proof we show that A is a post-fixed-point assignment of G upon termination, so $A \sqsupseteq F^G_{\min}$ since $F^G_{\min}$ is the minimum post-fixed-point assignment of G. Suppose $(v, S) \in E$ with $\forall v' \in S.\ A(v') = 1$. We show that $A(v) = 1$ and hence A is a post-fixed-point assignment. If $A(v) = 0$, then by 2. of Lemma 1 (coupled with the fact that $W = \emptyset$ upon termination), there exists $v' \in S$ with $A(v') = 0$. Contradiction. ■

```
W := nil;
for k = 1 to n do
    A[k] := 0; D[k] := nil; Load(W, H, k);
od
while W ≠ nil do
      (k, l) := hd(W); W := tl(W);
      if A[k] = 0 then
                        while A[hd(l)] = 1 & l ≠ nil do l := tl(l); od
                        if l = nil then
                                          A[k] := 1; W := W@D[k];
                                    else
                                          D[hd(l)] := (k, tl(l)) :: D[hd(l)];
                        fi
      fi
od
proc Load(W, H, k) ≡
  l := H[k];
  while l ≠ nil do
        W := (k, hd(l)) :: W; l := tl(l)
  od
end
```

Fig. 2. Algorithm Global2

We now describe the implementation of our global algorithm using concrete data structures. We first discuss the representation of the input dependency graph $G = (V, E)$. Suppose G has n nodes, then to represent V we can use the first n positive integers $\{1, \ldots, n\}$, and to represent E we can use an array H with indexing set $\{1, \ldots, n\}$ such that for each $k \in \{1, \ldots, n\}$, $H[k]$ lists the hyper-edges of G having k as the source. Since the source has been fixed to k, to list the hyper-edges in $H[k]$ we only need to list the target sets, which are again represented by lists. Thus $H[k]$ is a list of lists, with each sublist corresponding to a target set of a hyper-edge of the form (k, S). Now W is represented as a list of hyper-edges, with each hyper-edge represented as a pair consisting of a number and a list of numbers.

The functions A and D in the algorithm can now be implemented as two arrays of length n, with A having elements from $\{0, 1\}$ and D having elements of lists of hyper-edges. The refined algorithm, Global2, is given in Figure 2. Here we use some common notations for list operations. For a list l, $hd(l), tl(l)$ are the head and tail of l, respectively. Also $a :: l$ is a list with head a and tail l, and $l@l'$ is the concatenation of two lists l and l'.

Compared with Global1, there are some minor changes in Global2. W here is initialized by repeatedly executing procedure *Load*. Also, there is an extra test to see whether $A[k] = 0$ before processing each hyper-edge (k, l) in W, since if $A[k]$ is already 1 (set as the result of processing other hyper-edges), then there is no need to process (k, l). Finally, the testing of the condition whether $\forall v' \in S.\ A(v') = 1$ is implemented by a combination of while- and if-statements; it is easy to see why this is a faithful implementation of this test.

Since Global1 is just a high-level presentation of the algorithm, without detail to the data structures, it is only meaningful to study the precise complexity of the algorithm presented as Global2. For a dependency graph $G = (V, E)$, let $|G|$ be the size of G defined by $|G| = |V| + \sum_{(v,S)\in E}(|S| + 1)$ where $|V|$ and $|S|$ denote, as usual, the size of the sets V and S, respectively.

Theorem 3. *When executed on input dependency graph $G = (V, E)$, algorithm Global2 takes $O(|G|)$ time.*

Proof The execution of Global2 can be clearly divided into two phases: the *initialization phase*, consisting of the assignment of nil to W and the for-loop, and the *iteration phase*, consisting of the main while-loop. For the initialization phase, the assignment to W takes constant time and it is not difficult to see that the for-loop takes $O(|G|)$ time. Next we show that the iteration phase also takes $O(|G|)$ time.

Let $M = n + \sum_{(k,l)\in W}(len(l) + 1) + \sum_{1\le i\le n, A[i]=0}\sum_{(k,l)\in D[i]}(len(l) + 1)$, where $len(l)$ is the length of the list l, and for a list l we write $a \in l$ to mean that a is an element of list l. Clearly at the beginning of the main while-loop, it holds that $\sum_{(k,l)\in W}(len(l) + 1) = \sum_{(v,S)\in E}(|S| + 1)$ and $D[i] =$ nil for all $1 \le i \le n$, so $M = |V| + \sum_{(v,S)\in E}(|S| + 1)$. That is, M has the initial value of $|G|$ upon entering the iteration phase. Suppose that during one iteration, i.e. one execution of the body of the main while-loop, the inner while-loop takes t iterations. Note that after each such single iteration, M decreases by at least $1 + t$, while the time spent on this iteration is $c + t * d$ where c, d are some constant natural numbers. Thus, the execution time spent on the main while-loop is proportional to the decrease in M. Since M has an initial value of $|G|$ and lower bound of 0, the total execution time of the main while-loop is $O(|G|)$. ∎

In the Section 4 we will show that any instance of a usual maximal or minimal fixed-point problem in the form of a *boolean graph* can be reduced to a dependency graph of linear size in linear time. Thus the complexity measure given by Theorem 3 matches the complexity of the best global linear algorithms based on boolean graphs and boolean equation systems.

3.2 The Local Algorithm

In many model checking applications, the answer we are looking for is just the value of $F^G_{\min}$ at a given node $v \in V$, not the entire $F^G_{\min}$. In this case, of course, we can still run the global algorithm to obtain $F^G_{\min}$ and then look up the value at v. Local model checking, however, is likely to be a more efficient strategy. Fortunately, the global algorithm presented above can be easily made into a local one. The key observation is that there is no need to assume that all hyper-edges will have to be examined in deciding the value at a particular node. So there is no need to include all the hyper-edges in W in the beginning of the algorithm. Suppose that we want to know $F^G_{\min}(v_0)$ for $v_0 \in V$. Then W will initially include all hyper-edges having v_0 as the source, and will expand later when there is the need to investigate other nodes.

The local algorithm is presented in Figure 52 at the abstract level of sets and dependency-graph hyper-edges. It takes as input a dependency graph $G = (V, E)$,

```
for v ∈ V do A(v) := ⊥; od
A(v0) = 0; D(v0) := ∅; W := {e ∈ E | e = (v0, S)};
while W ≠ ∅ do
    let e = (v, S) ∈ W;
    W := W − {e};
    case of
        ∀v' ∈ S.A(v') = 1: A(v) := 1; W := W ∪ D(v);
        v' ∈ S.A(v') = 0: D(v') := D(v') ∪ {e};
        v' ∈ S.A(v') = ⊥: A(v') := 0; D(v') := ∅;
                          W := W ∪ {e' ∈ E | e' = (v', S)};
    esac
od
```

Fig. 3. Algorithm Local1

with $v_0 \in V$ the vertex whose minimal fixed-point value we are interested in computing. We assume that the execution of the case-statement is such that the conditions are tested in the order they are written, until one of them is satisfied. The corresponding statement is then executed, and only one of the cases is executed. We use $\perp$ to represent value *unknown*; thus, A is unknown everywhere initially. Whenever a node v is being investigated, it is assumed that $A(v) = 0$, and later it may be forced to 1. Also, at the same time, all hyper-edges having v as the source are added into W.

The processing of hyper-edges in W is very much like in the global algorithm, except that now we have a third case, in which the node under examination can neither be evaluated to 1 nor to 0, and the only way to proceed is to investigate a new node that has not been investigated yet. This happens in the last case of the case-statement. The rigorous proof of the correctness of algorithm Local1 is provided by the following lemma and theorem.

Lemma 4. *The while-loop of algorithm Local1 has the following invariant properties:*

1. *For all $v \in V$, if $A(v) = 1$ then $F^G_{\min}(v) = 1$;*
2. *For all $v \in V$ with $A(v) = 0$, whenever $(v, S) \in E$ then either $(v, S) \in W$ or $(v, S) \in D(v')$ for some $v' \in S$ with $A(v') = 0$.*

Proof Similar to the proof of Lemma 1. ∎

Theorem 5. *Upon termination of running algorithm Local1 on a given dependency graph $G = (V, E)$, for all $v \in V$, if $A(v) = 0$ then $F^G_{\min}(v) = 0$, and if $A(v) = 1$ then $F^G_{\min}(v) = 1$.*

Proof By 1. of Lemma 4, it certainly holds that if $A(v) = 1$ then $F^G_{\min}(v) = 1$ after termination. To show that if $A(v) = 0$ then $F^G_{\min}(v) = 0$, we construct an assignment B of G such that $B(v) = 0$ just in case $A(v) = 0$ (thus $B(v) = 1$ if $A(v) = 1$ or $A(v) = \perp$). We show in the following that B is a post fixed point of G,

so $F^G_{\min} \sqsubseteq B$, and if $A(v) = 0$ then $B(v) = 0$ and $F^G_{\min}(v) = 0$. Suppose $(v, S) \in E$ with $\forall v' \in S.B(v') = 1$. We will show that $B(v) = 1$ so B is a post-fixed-point assignment. If $B(v) = 0$ then $A(v) = 0$, since in this case $W = \emptyset$. By 2. of Lemma 4, there exists $v' \in S$ with $A(v') = 0$. Contradiction. ■

The implementation of the local algorithm with array and list data structures is given in Figure 4. It is not difficult to see that the total execution time of the local algorithm cannot exceed that of the global algorithm, and, hence, Local2 still takes $O(|G|)$ time in the worst case.

```
for k = 1 to n do A[k] := ⊥; od
A[m] := 0; D[m] := nil; W := nil; Load(W, H, m);
while W ≠ nil do
      (k, l) := hd(W); W := tl(W);
      if A[k] = 0 then
          while A[hd(l)] = 1 & l ≠ nil do l := tl(l); od
          case of
            l = nil: A[k] := 1; W := W@D[k];
            A[hd(l)] = 0 : D[hd(l)] := (k, tl(l)) :: D[hd(l)];
            A[hd(l)] = ⊥ : A[hd(l)] := 0; D[hd(l)] := nil;  Load(W, H, hd(l));
          esac
      fi
od
```

Fig. 4. Algorithm Local2

4 Boolean Graphs and Dependency Graphs

In [And94], *Boolean graphs* were introduced as a general framework for model checking problems. A boolean graph is a triple (V, E, L) where V is a set of vertices, $E \subseteq V \times V$ a set of edges, and $L \to \{\vee, \wedge\}$ is a function labeling the vertices as disjunctive or conjunctive. We say $v \in V$ is a *disjunctive node* (*conjunctive node*) if $L(v) = \vee$ ($L(v) = \wedge$).

Let $G = (V, E, L)$ be a boolean graph. An *assignment* A of G is a function from V to $\{0, 1\}$. A *pre-fixed-point assignment* F of G is an assignment such that for every disjunctive node $v \in V$, if $F(v) = 1$ then there exists $(v, v') \in E$ such that $F(v') = 1$, and for every conjunctive node $v \in V$, if $F(v) = 1$ then $F(v') = 1$ for all $(v, v') \in E$. A *post-fixed-point assignment* F of G is an assignment such that for every disjunctive node $v \in V$, if there exists $(v, v') \in E$ such that $F(v') = 1$ then $F(v) = 1$, and for every conjunctive node $v \in V$, if $F(v') = 1$ for all $(v, v') \in E$ then $F(v) = 1$. A *fixed-point assignment* F of G is both a pre-fixed point assignment and a post-fixed point assignment.

If we take the standard partial order $\sqsubseteq$ between assignments of boolean graphs G such that $A \sqsubseteq A'$ just in case $A(v) \leq A'(v)$ for all $v \in V$ (using the usual order $0 < 1$), then according to the Knaster-Tarski fixed-point theorem, for a given boolean graph G, there exists a unique maximum pre-fixed-point assignment, which we will write as $F^G_{\max}$, and also a unique minimum post-fixed point-assignment, which we will write as $F^G_{\min}$, and both of them are fixed points.

It turns out that many fixed-point computation problems in model checking, especially the problem of model checking the alternation-free modal mu-calculus, can be reduced to a series of problems involving the computation of $F^G_{\max}$ or $F^G_{\min}$ on some boolean graph G. In this section we present direct linear time reductions from both maximal and minimal fixed-point evaluation of boolean graphs to the problem of minimal fixed-point evaluation on dependency graphs. Thus our algorithms presented in the previous section represent both local and global linear-time algorithms for alternation-free fixed-point evaluation.

Theorem 6. *Let $G = (V, E, L)$ be a boolean graph, and $G_1 = (V, E_1)$ be the dependency graph having the same vertex set as G, and whose set of hyper-edges E_1 is the minimum set satisfying the following:*

1. *each $(v, v') \in E$ with $L(v) = \vee$ induces a hyper-edge $(v, \{v'\}) \in E_1$;*
2. *each $v \in V$ with $L(v) = \wedge$ induces a hyper-edge $(v, \{v' \mid (v, v') \in E\}) \in E_1$.*

With G_1 thus constructed, it holds that $F^G_{\min} = F^{G_1}_{\min}$.

Proof It is easy to see that an assignment of G (which must also be an assignment of G_1) is a post-fixed-point assignment of G if and only if it is at the same time a post-fixed-point assignment of G_1. Thus $F^G_{\min}$ is a post-fixed-point assignment of G_1. Since $F^{G_1}_{\min}$ is the minimum post-fixed-point assignment of G_1, $F^{G_1}_{\min} \sqsubseteq F^G_{\min}$. Similarly, to show $F^G_{\min} \sqsubseteq F^{G_1}_{\min}$ just note that $F^{G_1}_{\min}$ is a post-fixed-point assignment of G, and $F^G_{\min}$ is the minimum post-fixed-point assignment of G. The two inequality give us $F^G_{\min} = F^{G_1}_{\min}$. ∎

For an assignment F of a (dependency or boolean) graph G, we write $\neg F$ for the assignment of G obtained under componentwise-complement of F.

Theorem 7. *Let $G = (V, E, L)$ be a boolean graph, and $G_0 = (V, E_0)$ be the dependency graph having the same vertex set as G, and whose set of hyper-edges E_0 is the minimum set satisfying the following:*

1. *each $(v, v') \in E$ with $L(v) = \wedge$ induces a hyper-edge $(v, \{v'\}) \in E_0$;*
2. *each $v \in V$ with $L(v) = \vee$ induces a hyper-edge $(v, \{v' \mid (v, v') \in E\}) \in E_0$.*

With G_0 thus constructed, it holds that $F^G_{\max} = \neg F^{G_0}_{\min}$.

Proof It is easy to see that if F is a pre-fixed-point assignment of G then $\neg F$ is a post-fixed-point assignment of G_0, and if F is a post-fixed-point assignment of G_0 then $\neg F$ is a pre-fixed-point assignment of G. So $\neg F^G_{\max}$ is a post-fixed-point assignment of G_0. Since $F^{G_0}_{\min}$ is the minimum post-fixed-point assignment of G_0, $F^{G_0}_{\min} \sqsubseteq \neg F^G_{\max}$, and so $F^G_{\max} \sqsubseteq \neg F^{G_0}_{\min}$. On the other hand, $\neg F^{G_0}_{\min}$ is a pre-fixed-point assignment of G. Since $F^G_{\max}$ is the maximum pre-fixed-point assignment of G, $\neg F^{G_0}_{\min} \sqsubseteq F^G_{\max}$. Hence $F^G_{\max} = \neg F^{G_0}_{\min}$. ∎

5 Horn Formulae and Dependency Graphs

A *literal* is either a propositional letter P (a positive literal) or the negation $\neg P$ of a propositional letter P (a negative literal). A *basic Horn formula* is a disjunction of literals, with at most one positive literal. A basic Horn formula will also be called a Horn clause, or simply a clause. A *Horn formula* is a conjunction of basic Horn formulae. A Horn formula is satisfiable if there is a boolean assignment under which the Horn formula evaluates to true.

[DG84] is the earliest work to present linear-time algorithms for HORNSAT, the problem of Horn formula satisfiability. Shukla et al. [SHIR96] demonstrated effective reductions from model checking in the alternation-free modal mu-calculus and related problems to HORNSAT, thereby establishing an early link between HORNSAT and fixed-point computation. Interestingly, the idea of counter and "reverse list" data structures found in linear-time global model checking algorithms such as [CS93, AC88, And94] was already present in [DG84]. This suggests closer connection between the structure of the two problems.

In this section, we present reductions between these problems for both directions of reducibility. The reductions are straightforward, and should help further clarify the connections between model-checking-based fixed-point computation and HORNSAT. The reductions are linear-time with suitable representations of the problem instances. As a result, we derive a linear-time local algorithm for HORNSAT, the first of its kind as far as we are aware.

Theorem 8. *Let $G = (V, E)$ be a dependency graph, $v_0 \in V$, $\{P_v \mid v \in V\}$ a set of propositional letters in one-to-one correspondence with vertices in V, and $H_G^{v_0}$ the following Horn formula:*

$$H_G^{v_0} = \bigwedge_{(v_0,S)\in E} \Big(\bigvee_{v'\in S} \neg P_{v'}\Big) \wedge \bigwedge_{(v,S)\in E, v\neq v_0} \Big(P_v \vee \bigvee_{v'\in S} \neg P_{v'}\Big).$$

Then $F^G_{\min}(v_0) = 0$ if and only if $H_G^{v_0}$ is satisfiable.

Proof Let A be an assignment of $H_G^{v_0}$ under which $H_G^{v_0}$ evaluates to 1. From A we construct an assignment F of G such that $F(v_0) = 0$ and for all other $v \in V$, $F(v) = A(P_v)$. It is easy to verify that F is a post-fixed-point assignment of G, hence $F^G_{\min}(v_0) \leq F(v_0) = 0$. For the other direction, from $F^G_{\min}$ we construct an assignment A for $H_G^{v_0}$ such that for all $v \in V$, $A(P_v) = F^G_{\min}(v)$. Since $F^G_{\min}$ is a post-fixed-point assignment, it is easy to see that under A, $H_G^{v_0}$ evaluates to $\bigwedge_{(v_0,S)\in E}(\bigvee_{v'\in S} \neg F^G_{\min}(v'))$ which should be 1 when $F^G_{\min}(v_0) = 0$ (otherwise there exists $(v_0, S_0) \in E$ such that $\forall v' \in S_0$. $F^G_{\min}(v') = 1$, which contradicts $F^G_{\min}(v_0) = 0$ and $F^G_{\min}$ being a post-fixed-point assignment). ∎

Theorem 9. *Let H be a Horn formula, and $G_H = (V, E)$ a dependency graph, where V consists of all the propositional letters occurring in H plus a special node v_0, and E contains exactly the hyper-edges obtained by one of the follow rules:*

1. *for each Horn clause C of the form $\neg P_1 \vee \ldots \vee \neg P_m$ without a positive literal, there is a hyper-edge $(v_0, \{P_1, \ldots, P_m\}) \in E$;*
2. *for each Horn clause C of the form $P_0 \vee \neg P_1 \vee \ldots \vee \neg P_m$, there is a hyper-edge $(P_0, \{P_1, \ldots, P_m\}) \in E$ (a special case here is a Horn clause P_0 which corresponds to a hyper-edge $(P_0, \emptyset) \in E$).*

Then H is satisfiable if and only if $F_{\min}^{G_H}(v_0) = 0$.

Proof Similar to the proof of Theorem 8. ∎

Observe that Theorem 9 presents a reduction from the problem of satisfiability of H to the problem of whether $F_{\min}^{G_H}(v_0) = 0$, a local model checking problem. With array representations of instances of HORNSAT, it is not difficult to see that the reduction can be carried out in linear-time.

6 Conclusions

We have presented simple linear-time algorithms for minimal fixed-point evaluation in dependency graphs. The algorithms also have simple and clear proofs of correctness and complexity. We also demonstrated that dependency graphs represent a suitable framework for expressing and computing alternation-free fixed-points, an important problem in model checking. Finally, we clarified the relationship between minimal fixed-point evaluation and horn-formula satisfiability, deriving a linear-time local algorithm for HORNSAT in the process. To our knowledge, this is the first such linear-time local algorithm to appear in the literature.

In [LRS98], a local algorithm LAFP for computing fixed points of arbitrary alternation depth is presented. Like most fixed-point algorithms, LAFP computes fixed points iteratively. By using a "recovery strategy" that carefully accounts for the effects of value changes of a more dominant fixed point — thus avoiding unnecessary recomputation of fixed points nested inside it — the number of iterations required by LAFP to evaluate fixed points of an instance with size N and alternation depth ad is $O((N-1) + (\frac{N+ad}{ad})^{ad})$. Asymptotically, this matches the iteration complexity of the best existing global algorithms. However, the *total* execution time of LAFP (in which the time taken during iterations of the while-loop is taken into account) introduces an additional factor of N^2, making the total execution time worse than that of the best global algorithms. This additional factor is due to the lack of an efficient data structure that works well with the recovery strategy. As future work, we plan to extend the local algorithm presented in this paper to the the case of alternating fixed points, in the hope that the complexity matches that of the best global algorithms. We believe that this would lead to optimal algorithms for the problem.

References

[AC88] A. Arnold and P. Crubille. A linear algorithm to solve fixed-point equations on transition systems. *Information Processing Letters*, 29:57–66, September 1988.

[And94] H. R. Andersen. Model checking and boolean graphs. *Theoretical Computer Science*, 126(1), 1994.

[BC96] G. S. Bhat and R. Cleaveland. Efficient model checking via the equational μ-calculus. In E. M. Clarke, editor, *11th Annual Symposium on Logic in Computer Science (LICS '96)*, pages 304–312, New Brunswick, NJ, July 1996. Computer Society Press.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CKS92] R. Cleaveland, M. Klein, and B. Steffen. Faster model checking for the modal mu-calculus. In G.v. Bochmann and D.K. Probst, editors, *Proceedings of the Fourth International Conference on Computer Aided Verification (CAV '92)*, Vol. 663 of *Lecture Notes in Computer Science*, pages 410–422. Springer-Verlag, 1992.

[Cle90] R. Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27:725–747, 1990.

[CS93] R. Cleaveland and B. U. Steffen. A linear-time model checking algorithm for the alternation-free modal mu-calculus. *Formal Methods in System Design*, 2:121–147, 1993.

[CW96] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[DG84] W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 3:267–284, 1984.

[HW74] L. Henschen and L. Wos. Unit refutations and horn sets. *Journal of the ACM*, 21:590–605, 1974.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Lar88] K.G. Larsen. Proof systems for Hennessy–Milner logic with recursion. *Lecture Notes In Computer Science, Springer Verlag*, 299, 1988. in Proceedings of 13th Colloquium on Trees in Algebra and Programming 1988.

[Lar92] K.G. Larsen. Efficient local correctness checking. *Lecture Notes In Computer Science, Springer Verlag*, 663, 1992. in Proceedings of the 4th Workshop on Computer Aided Verification, 1992.

[LRS98] X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98), Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[Pra81] V.R. Pratt. A decidable mu-calculus. In *Proceedings of the 22nd IEEE Ann. Symp. on Foundations of Computer Science*, Nashville, Tennessee, pages 421–427, 1981.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.

[SHIR96] S. K. Shukla, H. B. Hunt III, and D. J. Rosenkrantz. HORNSAT, model checking, verification and games. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 99–110, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[SW91] C. Stirling and D. Walker. Local model checking in the modal mu-calculus. *Theoretical Computer Science*, 89(1), 1991.

[VL94] B. Vergauwen and J. Lewi. Efficient local correctness checking for single and alternating boolean equation systems. In *Proceedings of ICALP'94*, pages 304–315. LNCS 820, 1994.