Formal Proof and Analysis of an Incremental Cycle Detection Algorithm

Armaël Guéneau
Inria, Paris, France

Jacques-Henri Jourdan
CNRS, LRI, Univ. Paris Sud, Université Paris Saclay, France

Arthur Charguéraud
Inria & Université de Strasbourg, CNRS, ICube, Strasbourg, France

François Pottier
Inria, Paris, France

Abstract

We study a state-of-the-art incremental cycle detection algorithm due to Bender, Fineman, Gilbert, and Tarjan. We propose a simple change that allows the algorithm to be regarded as genuinely online. Then, we exploit Separation Logic with Time Credits to simultaneously verify the correctness and the worst-case amortized asymptotic complexity of the modified algorithm.

2012 ACM Subject Classification Theory of computation → Graph algorithms analysis; Theory of computation → Program verification; Software and its engineering → Correctness; Software and its engineering → Software performance

Keywords and phrases interactive deductive program verification, complexity analysis

Digital Object Identifier 10.4230/LIPIcs.ITP.2019.18

Related Version An extended version of the paper is available at `https://hal.inria.fr/hal-02167236`.

Supplement Material The Coq development is accessible at `https://gitlab.inria.fr/agueneau/incremental-cycles`, and the mechanized metatheory of Separation Logic with Time Credits is available at `https://gitlab.inria.fr/charguer/cfml2`.

1 Introduction

A good algorithm must be correct. Yet, to err is human: algorithm designers and algorithm implementors sometimes make mistakes. Although testing can detect mistakes, it cannot in general prove their absence. Thus, when high reliability is desired, algorithms should ideally be verified. A "verified algorithm" traditionally means an algorithm whose correctness has been verified: it is a package of an implementation, a specification, and a machine-checked proof that the algorithm always produces a result that the specification permits.

A growing number of verified algorithms appear in the literature. To cite just a very few examples, in the area of graph algorithms, Lammich and Neumann [27, 26] verify a generic depth-first search algorithm which, among other applications, can be used to detect a cycle in a directed graph; Lammich [25], Pottier [36], and Chen et al. [10, 9] verify various algorithms for finding the strongly connected components of a directed graph. A verified algorithm can serve as a building block in the construction of larger verified software: for instance, Esparza et al. [12] use a cycle detection algorithm as a component in a verified LTL model-checker.

However, a good algorithm must not just be correct: it must also be fast, and reliably so. Many algorithmic problems admit a simple, inefficient solution. Therefore, the art and science of algorithm design is chiefly concerned with imagining more efficient algorithms, which often

are more involved as well. Due to their increased sophistication, these algorithms are natural candidates for verification. Furthermore, because the very reason for existence of these algorithms is their alleged efficiency, not only their correctness, but also their complexity, should arguably be verified.

Following traditional practice in the algorithms literature [22, 43], we study the complexity of an algorithm based on an abstract cost model, as opposed to physical worst-case execution time. Furthermore, we wish to establish asymptotic complexity bounds, such as $O(n)$, as opposed to concrete bounds, such as $3n + 5$. While bounds on physical execution time are of interest in real-time applications, they are difficult to establish and highly dependent on the compiler, the runtime system, and the hardware. In contrast, an abstract cost model allows reasoning at the level of source code. We fix a specific model in which every function call has unit cost and every other primitive operation has zero cost. Although one could assign a nonzero cost to each primitive operation, that would make no difference in the end: an asymptotic complexity bound is independent of the costs assigned to the primitive operations, and is robust in the face of minor changes in the implementation.

In prior work, Charguéraud and Pottier [8] verify the correctness and the worst-case amortized asymptotic complexity of an OCaml implementation of the Union-Find data structure. They establish concrete bounds, such as $4\alpha(n) + 12$, as opposed to asymptotic bounds, such as $O(\alpha(n))$. This case study demonstrates that it is feasible to mechanize such a challenging complexity analysis, and that this analysis can be carried out based on actual source code, as opposed to pseudo-code or an idealized mathematical model of the data structure. Charguéraud and Pottier use CFML [6, 7], an implementation inside Coq of Separation Logic [37] with Time Credits [3, 8, 17, 18, 32]. This program logic makes it possible to simultaneously verify the correctness and the complexity of an algorithm, and allows the complexity argument to depend on properties whose validity is established as part of the correctness argument. We provide additional background in Section 2.

In subsequent work, Guéneau, Charguéraud and Pottier [17] formalize the O notation, propose a way of advertising asymptotic complexity bounds as part of Separation Logic specifications, and implement support for this approach in CFML. They present a collection of small illustrative examples, but do not carry out a challenging case study.

One major contribution of this paper is to present such a case study. We verify the correctness and worst-case amortized asymptotic complexity of an incremental cycle detection algorithm (and data structure) due to Bender, Fineman, Gilbert, and Tarjan [4, §2]. With this data structure, the complexity of building a directed graph of n vertices and m edges, while incrementally ensuring that no edge insertion creates a cycle, is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Although its implementation is relatively straightforward, its design is subtle, and it is far from obvious, by inspection of the code, that the advertised complexity bound is respected.

As a second contribution, on the algorithmic side, we simplify and enhance Bender et al.'s algorithm. To handle the insertion of a new edge, the original algorithm depends on a runtime parameter, which limits the extent of a certain backward search. This parameter influences only the algorithm's complexity, not its correctness. Bender et al. show that setting it to $\min(m^{1/2}, n^{2/3})$ throughout the execution of the algorithm allows achieving the advertised complexity. This means that, in order to run the algorithm, one must anticipate the *final* values of m and n. This seems at least awkward, or even impossible, if one wishes to use the algorithm in an online setting, where the sequence of operations is not known in advance. Instead, we propose a modified algorithm, where the extent of the backward search is limited by a value that depends only on the *current* state. The pseudocode for both algorithms appears in Figure 2; it is explained later on (Section 5). The modified algorithm has the same complexity as the original algorithm and is a genuine online algorithm. It is the one that we verify.

As a third contribution, on the methodological side, we switch from $\mathbb{N}$ to $\mathbb{Z}$ in our accounting of execution costs, and explain why this leads to a significant decrease in the number of proof obligations. In our previous work [8, 17], costs are represented as elements of $\mathbb{N}$. In this approach, at each operation of (say) unit cost in the code, one must prove that the number of execution steps performed so far is less than the number of steps advertised in the specification. This proof obligation arises because, in $\mathbb{N}$, the equality $m + (n - m) = n$ holds if and only if $m \leq n$ holds. In contrast, in $\mathbb{Z}$, this equality holds unconditionally. For this reason, representing costs as elements of $\mathbb{Z}$ can dramatically decrease the number of proof obligations (Section 3). Indeed, one must then verify just once, at the end of a function body, that the actual cost is less than or equal to the advertised cost. The switch from $\mathbb{N}$ to $\mathbb{Z}$ requires a modification of the underlying Separation Logic, for which we provide a machine-checked soundness proof.

Our verification effort has had some practical impact already. For instance, the Dune build system [41] needs an incremental cycle detection algorithm in order to reject circular build dependencies as soon as possible. For this purpose, the authors of Dune developed an implementation of Bender et al.'s original algorithm, which we recently replaced with our improved and verified algorithm [16]. Our contribution increases the trustworthiness of Dune's code base, without sacrificing its efficiency: in fact, our measurements suggest that our code can be as much as 7 times faster than the original code in a real-world scenario. As another potential application area, it is worth mentioning that the second author (Jourdan) has deployed an as-yet-unverified incremental cycle detection algorithm in the kernel of the Coq proof assistant [44], where it is used to check the satisfiability of universe constraints [39, §2]. At the time, this yielded a dramatic improvement in the overall performance of Coq's proof checker: the total time required to check the Mathematical Components library dropped from 25 to 18 minutes [23]. The algorithm deployed inside Coq is more general than the verified algorithm considered in this paper, as it also maintains strong components, as in Section 4 of Bender et al.'s paper [4]. Nevertheless, we view the present work as one step towards verifying Coq's universe inference system.

In summary, the main contributions of this paper are:

- A simple yet crucial improvement to Bender et al.'s incremental cycle detection algorithm, making it a genuine online algorithm;
- An implementation of it in OCaml as a self-contained, reusable data structure;
- A machine-checked proof of the functional correctness and worst-case amortized asymptotic complexity of this implementation.
- The discovery of the nonobvious fact that counting time credits in $\mathbb{Z}$ leads to significantly fewer proof obligations, together with a study of the metatheory of Separation Logic with Time Credits in $\mathbb{Z}$ and support for it in CFML.

Our code and proofs are available online (Supplement Material). Our methodology is modular: at the end of the day, the verified data structure is equipped with a succinct specification (Figure 1) which is intended to serve as the sole reference when verifying a client of the algorithm. We believe that this case study illustrates the great power and versatility of our approach, and we claim that this approach is generally applicable to many other nontrivial data structures and algorithms.

2 Separation Logic with Time Credits

Hoare Logic [19] allows verifying the correctness of an imperative algorithm by using *assertions* to describe the state of the program. Separation Logic [37] improves modularity by employing assertions that describe only a fragment of the state and at the same time assert the unique

ownership of this fragment. In general, a Separation Logic assertion claims the ownership of certain *resources*, and (at the same time) describes the current state of these resources. A heap fragment is an example of a resource.

Separation Logic with Time Credits [3, 8, 32] is a simple extension of Separation Logic in which "a permission to perform one computation step" is also a resource, known as a *credit*. The assertion \$1 represents the unique ownership of one credit. The logic enforces the rule that every function call consumes one credit. Credits do not exist at runtime; they appear only in assertions, such as pre- and postconditions, loop invariants, and data structure invariants. For instance, the Separation Logic triple:

$$\forall\, g\, G.\ \{\, \mathrm{IsGraph}\, g\ G \ *\ \$(3\,|\mathrm{edges}\ G| + 5)\,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\ G\,\}$$

can be read as follows. If initially g is a runtime representation of the graph G and if $3m + 5$ credits are at hand, where m is the number of edges of G, then the function call $\mathit{dfs}(g)$ executes safely and terminates; after this call, g remains a valid representation of G, and no credits remain.

In the *dfs* example, assuming that the assertion $\mathrm{IsGraph}\, g\ G$ is credit-free (which means, roughly, that this assertion definitely does not own any credits), the precondition guarantees the availability of $3m + 5$ credits (and no more), and no credits remain in the postcondition. So, this triple guarantees that the execution of $\mathit{dfs}(g)$ involves at most $3m + 5$ computation steps. Later on in this paper (Section 8), we define $\mathrm{IsGraph}\, g\ G$ in such a way that it is *not* credit-free: its definition involves a nonnegative number of credits. If that were the case in the above example, then $3m + 5$ would have to be interpreted as an amortized bound. Amortization is discussed in greater depth in the next section (Section 4).

Admittedly, $3m + 5$ is too low-level a bound: it would be preferable to state that the cost of $\mathit{dfs}(g)$ is $O(m)$, a more abstract and more robust specification. Following Guéneau et al. [17], this can be expressed in the following style:

$$\begin{aligned}\exists(f : \mathbb{Z} \to \mathbb{Z}).\qquad & \text{nonnegative}\ f\ \wedge\ \text{monotonic}\ f\ \wedge\ f \preceq_{\mathbb{Z}} \lambda m.m \\ & \wedge\ \forall\, g\, G.\ \{\, \mathrm{IsGraph}\, g\ G\ *\ \$\, f(|\mathrm{edges}\ G|)\,\}\ \mathit{dfs}(g)\ \{\, \mathrm{IsGraph}\, g\ G\,\}\end{aligned}$$

The concrete function $\lambda m.(3m + 5)$ is no longer visible; it has been abstracted away under the name f. The specification states that f is nonnegative ($\forall m.\ f(m) \geq 0$), monotonic ($\forall m m'.\ m \leq m' \Rightarrow f(m) \leq f(m')$), and dominated by the function $\lambda m.m$, which means that f grows linearly.

The soundness of Separation Logic with Time Credits stems from the fact that a credit cannot be spent twice. Technically, the soundness metatheorem for Separation Logic with Time Credits guarantees that, for every valid Hoare triple, the following inequality holds:

$$\text{credits in precondition}\ \geq\ \text{steps taken}\ +\ \text{credits in postcondition}.$$

This type of metatheorem is proved by Charguéraud and Pottier [8, §3] and by Mével et al. [32] for Separation Logics with nonnegative credits.

The CFML tool can be viewed as an implementation of Separation Logic with Time Credits for OCaml inside Coq. CFML enables reasoning in forward style. The user inspects the source code, step by step. At each step, she is allowed to visualize and manipulate a description of the current program state in the form of a Separation Logic formula. This formula not only describes the current heap, but also indicates how many time credits are currently available. Guéneau et al. [17, §5, §6] describe the deduction rules of the logic and the manner in which they are applied.

3 Negative Time Credits

In the original presentations of Separation Logic with Time Credits [3, 8, 17, 18], credits are counted in $\mathbb{N}$. This seems natural because $\$n$ is interpreted as a permission to take n steps of computation, and a number of execution steps is never a negative value.

In this setting, credits are affine, that is, it is sound to discard them: the law $\$n \vdash \mathit{true}$ holds. The law $\$(m+n) \equiv \$m * \$n$ holds for every $m, n \in \mathbb{N}$. This splitting law is used when one wishes to spend a subset of the credits at hand. Yet, in practice, the law that is most often needed is a slightly different formulation. Indeed, if n credits are at hand and if one wishes to step over an operation whose cost is m, the appropriate law is $\$n \equiv \$(n-m) * \$m$, which holds only under the side condition $m \leq n$. (This is subtraction in $\mathbb{N}$, so $m > n$ implies $n - m = 0$.)

This side condition gives rise to a proof obligation, and these proof obligations tend to accumulate. If n credits are initially at hand and if one wishes to step over a sequence of k operations whose costs are $m_1, m_2, \ldots, m_k$, then k proof obligations arise: $n - m_1 \geq 0$, $n - m_1 - m_2 \geq 0$, and so on, until $n - m_1 - m_2 - \ldots - m_k \geq 0$. In fact, these proof obligations are redundant: the last one alone implies all of the previous ones. Unfortunately, in an interactive proof assistant such as Coq, it is not easy to take advantage of this fact and present only the last proof obligation to the user. Furthermore, in the proof of Bender et al.'s algorithm, we have encountered a more complex situation where, instead of looking at a straight-line sequence of k operations, one is looking at a loop, whose body is a sequence of operations, and which itself is followed with another sequence of operations. In this situation, proving that the very last proof obligation implies all previous obligations may be possible in principle, but requires a nontrivial strengthening of the loop invariant, which we would rather avoid, if at all possible!

To avoid this accumulation, in this paper, we work in a variant of Separation Logic where Time Credits are counted in $\mathbb{Z}$. Its basic laws are as follows:

$\$0 \equiv \mathit{true}$	zero credit is equivalent to nothing at all
$\$(m+n) \equiv \$m * \$n$	credits are additive
$\$n * [n \geq 0] \vdash \mathit{true}$	nonnegative credits are affine; negative credits are not

Quite remarkably, in the second law, there is no side condition. In particular, this law implies $\$0 \equiv \$n * \$(-n)$, which creates positive credit out of thin air, but creates negative credit at the same time. As put by Tarjan [42], "we can allow borrowing of credits, as long as any debt incurred is eventually paid off". In the third law, the side condition $n \geq 0$ guarantees that a debt cannot be forgotten. Without this requirement, the logic would be unsound, as the second and third laws together would imply $\$0 \vdash \$1$.

Because the second law has no side condition, stepping over a sequence of k operations whose costs are $m_1, m_2, \ldots, m_k$ gives rise to no proof obligation at all. At the end of the sequence, $n - m_1 - m_2 - \ldots - m_k$ credits remain, which the user typically wishes to discard. This is done by applying the third law, giving rise to just one proof obligation: $n - m_1 - m_2 - \ldots - m_k \geq 0$. In summary, switching from $\mathbb{N}$ to $\mathbb{Z}$ greatly reduces the number of proof obligations that appear about credits.

A secondary benefit of this switch is to reduce the number of conversions between $\mathbb{N}$ and $\mathbb{Z}$ that must be inserted in specifications and proofs. Indeed, we model OCaml's signed integers as mathematical integers in $\mathbb{Z}$. (We currently ignore the mismatch between OCaml's limited-precision integers and ideal integers. It should ideally be taken into account, but this is orthogonal to the topic of this paper.)

InitGraph
$\exists k.\ \{\$k\}\ \texttt{init\_graph()}\ \{\lambda g.\mathrm{IsGraph}\ g\ \varnothing\}$

DisposeGraph
$\forall g\, G.\ \mathrm{IsGraph}\ g\ G \Vdash \mathit{true}$

AddVertex
$\forall g\, G\, v.$
$\mathrm{let}\ m, n := |\mathrm{edges}\ G|, |\mathrm{vertices}\ G|\ \mathrm{in}$
$v \notin \mathrm{vertices}\ G \implies$
$\left\{ \begin{array}{l} \mathrm{IsGraph}\ g\ G\ * \\ \$(\psi\,(m, n+1) - \psi\,(m, n)) \end{array} \right\}$
$(\texttt{add\_vertex}\ g\ v)$
$\left\{ \lambda().\ \mathrm{IsGraph}\ g\ (G + v) \right\}$

AddEdge
$\forall g\, G\, v\, w.$
$\mathrm{let}\ m, n := |\mathrm{edges}\ G|, |\mathrm{vertices}\ G|\ \mathrm{in}$
$v, w \in \mathrm{vertices}\ G \wedge (v, w) \notin \mathrm{edges}\ G \implies$
$\left\{ \begin{array}{l} \mathrm{IsGraph}\ g\ G\ * \\ \$(\psi\,(m+1, n) - \psi\,(m, n)) \end{array} \right\}$
$(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w)$
$\left\{ \begin{array}{l} \lambda \mathit{res}.\ \mathrm{match}\ \mathit{res}\ \mathrm{with} \\ \quad \mid \mathrm{Ok} \Rightarrow \mathrm{IsGraph}\ g\ (G + (v, w)) \\ \quad \mid \mathrm{Cycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\}$

Acyclicity
$\forall g\, G.\ \mathrm{IsGraph}\ g\ G \Vdash$
$\quad \mathrm{IsGraph}\ g\ G\ *\ [\forall x.\ x \not\longrightarrow_G^+ x]$

Complexity
$\mathrm{nonnegative}\ \psi\ \wedge\ \mathrm{monotonic}\ \psi\ \wedge$
$\psi \preceq_{\mathbb{Z}\times\mathbb{Z}} \lambda(m, n).\ (m \cdot \min(m^{1/2}, n^{2/3}) + n)$

Figure 1 Specification of an incremental cycle detection algorithm.

Because negative time credits are not affine, it is not the case here that every assertion is affine, as in Iris [24] or in earlier versions of CFML. Affine and non-affine assertions must now be distinguished: a points-to assertion, which describes a heap-allocated object, remains affine; the assertion $\$n$ is affine if and only if n is nonnegative; an abstract assertion, such as $\mathrm{IsGraph}\ g\ G$, may or may not be affine, depending on the definition of IsGraph. (Here, it is in fact affine; see §4 and DisposeGraph in Figure 1.) We have adapted CFML so as to support this distinction.

From a metatheoretical perspective, the introduction of negative time credits requires adapting the proof of soundness of Separation Logic with Time Credits. We have successfully updated our pre-existing Coq proof of this result [8]; an updated proof is available online (Supplement Material).

4 Specification of the Algorithm

The interface for an incremental cycle detection algorithm consists of three public operations: `init_graph`, which creates a fresh empty graph, `add_vertex`, which adds a vertex, and `add_edge_or_detect_cycle`, which either adds an edge or report that this edge cannot be added because it would create a cycle.

Figure 1 shows a formal specification for an incremental cycle detection algorithm. It consists of six statements. InitGraph, AddVertex, and AddEdge are Separation Logic triples: they assign pre- and postconditions to the three public operations. DisposeGraph and Acyclicity are Separation Logic entailments. The last statement, Complexity, provides a complexity bound. It is the only statement that is specific to the algorithm discussed in this paper. Indeed, the first five statements form a generic specification, which any incremental cycle detection algorithm could satisfy.

The six statements in the specification share two variables, namely IsGraph and ψ. These variables are implicitly existentially quantified in front of the specification: a user of the algorithm must treat them as abstract.

The predicate IsGraph is an *abstract representation predicate*, a standard notion in Separation Logic [35]. It is parameterized with a memory location g and with a mathematical graph G. The assertion IsGraph g G means that a well-formed data structure, which represents the mathematical graph G, exists at address g in memory. At the same time, this assertion denotes the unique ownership of this data structure.

Because this is Separation Logic with Time Credits, the assertion IsGraph g G can also represent the ownership of a certain number of credits. For example, for the specific algorithm considered in this paper, we later define IsGraph g G as $\exists L.\ \$\phi(G, L) * \ldots$ (Section 8), where ϕ is a suitable potential function [42]. ϕ is parameterized by the graph G and by a map L of vertices to integer levels. Intuitively, this means that $\phi(G, L)$ credits are stored in the data structure. These details are hidden from the user: ϕ does not appear in Figure 1. Yet, the fact that IsGraph g G can involve credits means that the user must read AddVertex and AddEdge as amortized specifications [42]: the actual cost of a single `add_vertex` or `add_edge_or_detect_cycle` operation is not directly related to the number of credits that explicitly appear in the precondition of this operation.

The function ψ has type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$. In short, $\psi(m, n)$ is meant to represent the advertised cost of a sequence of n vertex creation and m edge creation operations. In other words, it is the number of credits that one must pay in order to create n vertices and m edges. This informal claim is explained later on in this section.

InitGraph states that the function call `init_graph()` creates a valid data structure, which represents the empty graph $\varnothing$, and returns its address g. Its cost is k, where k is an unspecified constant; in other words, its complexity is $O(1)$.

DisposeGraph states that the assertion IsGraph g G is affine: that is, it is permitted to forget about the existence of a valid graph data structure. By publishing this statement, we guarantee that we are not hiding a debt inside the abstract predicate IsGraph. Indeed, to prove that DisposeGraph holds, we must verify that the potential $\phi(G, L)$ is nonnegative (Section 3).

AddVertex states that `add_vertex` requires a valid data structure, described by the assertion IsGraph g G, and returns a valid data structure, described by IsGraph g $(G + v)$. (We write $G + v$ for the result of extending the mathematical graph G with a new vertex v and $G + (v, w)$ for the result of extending G with a new edge from v to w.) In addition, `add_vertex` requires $\psi(m, n + 1) - \psi(m, n)$ credits. These credits are not returned: they do not appear in the postcondition. They either are actually consumed or become stored inside the data structure for later use. Thus, one can think of $\psi(m, n + 1) - \psi(m, n)$ as the amortized cost of `add_vertex`.

Similarly, AddEdge states that the cost of `add_edge_or_detect_cycle` is $\psi(m+1, n) - \psi(m, n)$. This operation returns either Ok, in which case the graph has been successfully extended with a new edge from v to w, or Cycle, in which case this new edge cannot be added, because there already is a path in G from w to v. (The proposition $w \longrightarrow_G^* v$ appears within square brackets, which convert an ordinary proposition to a Separation Logic assertion.) In the latter case, the data structure is invalidated: the assertion IsGraph g G is not returned. Thus, in that case, no further operations on the graph are allowed.

By combining the first four statements in Figure 1, a client can verify that a call to `init_graph`, followed with an arbitrary interleaving of n calls to `add_vertex` and m successful calls to `add_edge_or_detect_cycle`, satisfies the specification $\{\$(k + \psi(m, n))\}\ \ldots\ \{true\}$, where k is the cost of `init_graph`. Indeed, the cumulated cost of the calls to `add_vertex` and `add_edge_or_detect_cycle` forms a telescopic sum that adds up to $\psi(m, n) - \psi(0, 0)$, which itself is bounded by $\psi(m, n)$.

To insert a new edge from v to w and detect potential cycles:
- If $L(v) < L(w)$, insert the edge (v, w), declare success, and exit
- Perform a backward search:
  - start from v
  - follow an edge (backward) only if its source vertex x satisfies $L(x) = L(v)$
  - if w is reached, declare failure and exit
  - if F edges have been traversed, interrupt the backward search
    - → in Bender et al.'s algorithm, F is a constant Δ
    - → in our algorithm, F is $L(v)$
- If the backward search was not interrupted, then:
  - if $L(w) = L(v)$, insert the edge (v, w), declare success, and exit
  - otherwise set $L(w)$ to $L(v)$
- If the backward search was interrupted, then set $L(w)$ to $L(v) + 1$
- Perform a forward search:
  - start from w
  - upon reaching a vertex x:
    - if x was visited during the backward search, declare failure and exit
    - if $L(x) \geq L(w)$, do not traverse through x
    - if $L(x) < L(w)$, set $L(x)$ to $L(w)$ and traverse x
- Finally, insert the edge (v, w), declare success, and exit

Figure 2 Pseudocode for Bender et al.'s algorithm and for our improved algorithm.

Since Separation Logic with Time Credits is sound, the triple $\{\$(k+\psi(m,n))\}\ \ldots\ \{\mathit{true}\}$ implies that the actual worst-case cost of the sequence of operations is $k + \psi(m, n)$. This confirms our earlier informal claim that $\psi(m, n)$ represents the cost of creating n vertices and m edges.

Acyclicity states that, from the Separation Logic assertion IsGraph g G, the user can deduce that G is acyclic. In other words, as long as the data structure remains in a valid state, the graph G remains acyclic.

Although the exact definition of ψ is not exposed, Complexity provides an asymptotic bound: $\psi(m, n) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$. Technically, the relation $\preceq_{\mathbb{Z}\times\mathbb{Z}}$ is a domination relation between functions of type $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ [17]. Our complexity bound thus matches the one published by Bender et al. [4].

5 Overview of the Algorithm

We provide pseudocode for Bender et al.'s algorithm [4, §2] and for our improved algorithm in Figure 2. The only difference between the two algorithms is the manner in which a certain internal parameter, named F, is set. The value of F influences the complexity of the algorithm, not its correctness.

When the user requests the creation of an edge from v to w, finding out whether this operation would create a cycle amounts to determining whether a path already exists from w to v. A naïve algorithm could search for such a path by performing a forward search, starting from w and attempting to reach v.

One key feature of Bender et al.'s algorithm is that a positive integer level $L(v)$ is associated with every vertex v, and the following invariant is maintained: L forms a pseudo-topological numbering. That is, "no edge goes down": if there is an edge from v to w, then $L(v) \leq L(w)$ holds. The presence of levels can be exploited to accelerate a search: for

instance, during a forward search whose purpose is to reach the vertex v, any vertex whose level is greater than that of v can be disregarded. The price to pay is that the invariant must be maintained: when a new edge is inserted, the levels of some vertices must be adjusted.

A second key feature of Bender et al.'s algorithm is that it not only performs a forward search, but begins with a backward search that is both *restricted* and *bounded*. It is restricted in the sense that it searches only one level of the graph: starting from v, it follows only *horizontal* edges, that is, edges whose endpoints are both at the same level. Therefore, all of the vertices that it discovers are at level $L(v)$. It is bounded in the sense that it is interrupted, even if incomplete, after it has processed a predetermined number of edges, denoted by the letter F in Figure 2.

A third key characteristic of Bender et al.'s algorithm is the manner in which levels are updated so as to maintain the invariant when a new edge is inserted. Bender et al. adopt the policy that the level of a vertex can never decrease. Thus, when an edge from v to w is inserted, all of the vertices that are accessible from w must be promoted to a level that is at least the level of v. In principle, there are many ways of doing so. Bender et al. proceed as follows: if the backward search was not interrupted, then w and its descendants are promoted to the level of v; otherwise, they are promoted to the next level, $L(v) + 1$. In the latter case, $L(v) + 1$ is possibly a new level. We see that such a new level can be created only if the backward search has not completed, that is, only if there exist at least F edges at level $L(v)$. In short, a new level may be created only if the previous level contains sufficiently many edges. This mechanism is used to control the number of levels.

The last key aspect of Bender et al.'s algorithm is the choice of F. On the one hand, as F increases, backward searches become more expensive, as each backward search processes up to F edges. On the other hand, as F decreases, forward searches become more expensive. Indeed, a smaller value of F leads to the creation of a larger number of levels, and (as explained later) the total cost of the forward searches is proportional to the number of levels.

Bender et al. set F to a constant Δ, defined as $\min(m^{1/2}, n^{2/3})$ throughout the execution of the algorithm, where m and n are upper bounds on the *final* numbers of edges and vertices in the graph. As explained earlier (Section 1), though, it seems preferable to set F to a value that does not depend on such upper bounds, as they may not be known ahead of time. In our modified algorithm, F stands for $L(v)$, where v is the source of the edge that is being inserted. This value depends only on the *current* state of the data structure, so our algorithm is truly online. We prove that it has the same complexity as Bender et al.'s original algorithm, namely $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

6 Informal Complexity Analysis

We now present an informal complexity analysis of Bender et al.'s original algorithm. In this algorithm, the parameter F is fixed: it remains constant throughout the execution of the algorithm. Under this hypothesis, the following invariant holds: for every level k except the highest level, there exist at least F horizontal edges at level k (edges whose endpoints are both at level k). A proof is given in Appendix A of the extended version.

From this invariant, one can derive two upper bounds on the number of levels. Let K denote the number of nonterminal levels. First, the invariant implies $m \geq KF$, therefore $K \leq m/F$. Furthermore, for each nonterminal level k, the vertices at level k form a subgraph with at least F edges, which therefore must have at least $\sqrt{F}$ vertices. In other words, at every nonterminal level, there are at least $\sqrt{F}$ vertices. This implies $n \geq K\sqrt{F}$, therefore $K \leq n/\sqrt{F}$.

Let us estimate the algorithm's complexity. Consider a sequence of n vertex creation and m edge creation operations. The cost of one backward search is $O(F)$, as it traverses at most F edges. Because each edge insertion triggers one such search, the total cost of the backward searches is $O(mF)$. The forward search traverses an edge if and only if this edge's source vertex is promoted to a higher level. Therefore, the cost of a forward search is linear in the number of edges whose source vertex is thus promoted. Because there are m edges and because a vertex can be promoted at most K times, the total cost of the forward searches is $O(mK)$. In summary, the cost of this sequence of operations is $O(mF + mK)$.

By combining this result with the two bounds on K obtained above, one finds that the complexity of the algorithm is $O(m \cdot (F + \min(m/F, n/\sqrt{F}))$. A mathematical analysis (Appendix A of the extended version) shows that setting F to Δ, where Δ is defined as $\min(m^{1/2}, n^{2/3})$, leads to the asymptotic bound $O(m \cdot \min(m^{1/2}, n^{2/3}))$. This completes our informal analysis of Bender et al.'s original algorithm.

In our modified algorithm, in contrast, F is not a constant. Instead, each edge insertion operation has its own value of F: indeed, we let F stand for $L(v)$, where v is the source vertex of the edge that is being inserted. We are able to establish the following invariant: for every level k except the highest level, there exist at least k horizontal edges at level k. This subsequently allows us to establish a bound on the number of levels: we prove that $L(v)$ is bounded by a quantity that is asymptotically equivalent to Δ.

7 Implementation

Our OCaml code, shown in Figure 3, relies on auxiliary operations whose implementation belongs in a lower layer. We do not prescribe how they should be implemented and what data structures they should rely upon; instead, we provide a specification for each of them, and prove that our algorithm is correct, regardless of which implementation choices are made. We provide and verify one concrete implementation, so as to guarantee that our requirements can be met.

For brevity, we do not give the specifications of these auxiliary operations. Instead, we list them and briefly describe what they are supposed to do. Each of them is required to have constant time complexity.

To update the graph, the algorithm requires the ability to create new vertices and new edges (`create_vertex` and `add_edge`). To avoid creating duplicate edges, it must be able to test the equality of two vertices (`vertex_eq`).

The backward search requires the ability to efficiently enumerate the horizontal incoming edges of a vertex (`get_incoming`). The collection of horizontal incoming edges of a vertex y is updated during a forward search. It is reset when the level of y is increased (`clear_incoming`). An edge is added to it when a horizontal edge from x to y is traversed (`add_incoming`). The backward search also requires the ability to generate a fresh mark (`new_mark`), to mark a vertex (`set_mark`), and to test whether a vertex is marked (`is_marked`). These marks are consulted also during the forward search.

The forward search requires the ability to efficiently enumerate the outgoing edges of a vertex (`get_outgoing`). It also reads and updates the level of certain vertices (`get_level`, `set_level`).

Several choices arise in the implementation of graph search. First, the frontier can be either implicit, if the search is formulated as a recursive function, or represented as an explicit data structure. We choose the latter approach, as it lends itself better to the implementation of an interruptible search. Second, one must choose between an imperative style, where the

```
let rec visit_backward g target mark fuel stack =
  match stack with
  | [] -> VisitBackwardCompleted
  | x :: stack ->
    let (stack, fuel), interrupted = interruptible_fold (fun y (stack, fuel) ->
      if fuel = 0 then Break (stack, -1)
      else if vertex_eq y target then Break (stack, fuel)
      else if is_marked g y mark then Continue (stack, fuel - 1)
      else (set_mark g y mark; Continue (y :: stack, fuel - 1))
    ) (get_incoming g x) (stack, fuel) in
    if interrupted
    then if fuel = -1 then VisitBackwardInterrupted else VisitBackwardCyclic
    else visit_backward g target mark fuel stack

let backward_search g v w fuel =
  let mark = new_mark g in
  let v_level = get_level g v in
  set_mark g v mark;
  match visit_backward g w mark fuel [v] with
  | VisitBackwardCyclic -> BackwardCyclic
  | VisitBackwardInterrupted -> BackwardForward (v_level + 1, mark)
  | VisitBackwardCompleted -> if get_level g w = v_level
                              then BackwardAcyclic
                              else BackwardForward (v_level, mark)

let rec visit_forward g new_level mark stack =
  match stack with
  | [] -> ForwardCompleted
  | x :: stack ->
    let stack, interrupted = interruptible_fold (fun y stack ->
      if is_marked g y mark then Break stack
      else
        let y_level = get_level g y in
        if y_level < new_level then begin
          set_level g y new_level;
          clear_incoming g y;
          add_incoming g y x;
          Continue (y :: stack)
        end else if y_level = new_level then begin
          add_incoming g y x;
          Continue stack
        end else Continue stack
    ) (get_outgoing g x) stack in
    if interrupted then ForwardCyclic
    else visit_forward g new_level mark stack

let forward_search g w new_w_level mark =
  clear_incoming g w;
  set_level g w new_w_level;
  visit_forward g new_w_level mark [w]

let add_edge_or_detect_cycle (g : graph) (v : vertex) (w : vertex) =
  let succeed () = add_edge g v w; Ok in
  if vertex_eq v w then Cycle
  else if get_level g w > get_level g v then succeed ()
  else match backward_search g v w (get_level g v) with
    | BackwardCyclic -> Cycle
    | BackwardAcyclic -> succeed ()
    | BackwardForward (new_level, mark) ->
        match forward_search g w new_level mark with
        | ForwardCyclic -> Cycle
        | ForwardCompleted -> succeed ()
```

Figure 3 OCaml implementation of the verified incremental cycle detection algorithm.

frontier is represented as a mutable data structure and the code is structured in terms of "while" loops and "break" and "continue" instructions, and a functional style, where the frontier is an immutable data structure and the code is organized in terms of tail-recursive functions or higher-order loop combinators. Because OCaml does not have "break" and "continue", we choose the latter style.

The function `visit_backward`, for instance, can be thought of as two nested loops. The outer loop is encoded via a tail call to `visit_backward` itself. This loop runs until the stack is exhausted or the inner loop is interrupted. The inner loop is implemented via the loop combinator `interruptible_fold`, a functional-style encoding of a "for" loop whose body may choose between interrupting the loop (`Break`) and continuing (`Continue`). This inner loop iterates over the horizontal incoming edges of the vertex x. It is interrupted when a cycle is detected or when the variable `fuel`, whose initial value corresponds to F (Section 5), reaches zero.

The main public entry point of the algorithm is `add_edge_or_detect_cycle`, whose specification was presented in Figure 1. The other two public functions, `init_graph` and `add_vertex`, are trivial; they are not shown.

8 Data Structure Invariants

As explained earlier (Section 4), the specification of the algorithm refers to two variables, IsGraph and ψ, which must be regarded as abstract by a client. Figure 4 gives their formal definitions. The assertion IsGraph g G captures both the invariants required for functional correctness and those required for the complexity analysis. It is a conjunction of three conjuncts, which we describe in turn.

The conjunct IsRawGraph g G L M I asserts that there is a data structure at address g in memory, claims the unique ownership of this data structure, and summarizes the information that is recorded in this structure. The parameters G, L, M, I together form a logical model of this data structure: G is a mathematical graph; L is a map of vertices to integer levels; M is a map of vertices to integer marks; and I is a map of vertices to sets of vertices, describing horizontal incoming edges. The parameters L, M and I are existentially quantified in the definition of IsGraph, indicating that they are internal data whose existence is not exposed to the user.

The second conjunct, [Inv G L I], is a pure proposition that relates the graph G with the maps L and I. Its definition appears next in Figure 4. Anticipating on the fact that we sometimes need a relaxed invariant, we actually define a more general predicate InvExcept E G L I, where E is a set of "exceptions", that is, a set of vertices where certain properties are allowed *not* to hold. Instantiating E with the empty set $\emptyset$ yields Inv G L I.

The proposition InvExcept E G L I is a conjunction of five properties. The first four capture functional correctness invariants: the graph G is acyclic, every vertex has positive level, L forms a pseudo-topological numbering of G, and the sets of horizontal incoming edges represented by I are accurate with respect to G and L. The last property plays a crucial role in the complexity analysis (Section 6). It asserts that "every vertex has enough coaccessible edges at the previous level": for every vertex x at level $k+1$, there must be at least k horizontal edges at level k from which x is accessible. The vertices in the set E may disobey this property, which is temporarily broken during a forward search.

The last conjunct in the definition of IsGraph is $\$\phi(G, L)$. This is a *potential* [42], a number of credits that have been received from the user (through calls to `add_vertex` and `add_edge_or_detect_cycle`) and not yet spent. $\phi(G, L)$ is defined as $C \cdot (\text{net } G\ L)$. The

$$\text{IsGraph}\, g\, G := \exists L\, M\, I.\ \text{IsRawGraph}\, g\, G\, L\, M\, I * [\text{Inv}\, G\, L\, I] * \$\phi(G, L)$$

$$\text{Inv}\, G\, L\, I := \text{InvExcept}\, \emptyset\, G\, L\, I$$

$$\text{InvExcept}\, E\, G\, L\, I := \begin{cases} \textit{acyclicity:} & \forall x.\quad x \not\longrightarrow_G^+ x \\ \textit{positive levels:} & \forall x.\quad L(x) \geq 1 \\ \textit{pseudo-topological numbering:} & \forall x\, y.\ x \longrightarrow_G y \implies L(x) \leq L(y) \\ \textit{horizontal incoming edges:} & \forall x\, y.\ x \in I(y) \iff x \longrightarrow_G y \ \wedge\ L(x) = L(y) \\ \textit{replete levels:} & \forall x.\quad x \in E \vee \text{enoughEdgesBelow}\, G\, L\, x \end{cases}$$

$$\text{enoughEdgesBelow}\, G\, L\, x := |\text{coaccEdgesAtLevel}\, G\, L\, k\, x| \geq k \quad \text{where } k = L(x) - 1$$

$$\text{coaccEdgesAtLevel}\, G\, L\, k\, x := \{\, (y, z) \mid y \longrightarrow_G z \longrightarrow_G^* x \ \wedge\ L(y) = L(z) = k \,\}$$

$$\left.\begin{array}{lcl} \phi(G, L) & := & C \cdot (\text{net}\, G\, L) \\ \text{net}\, G\, L & := & \text{received}\, m\, n - \text{spent}\, G\, L \end{array}\right\} \text{where } m = |\text{edges}\, G| \text{ and } n = |\text{vertices}\, G|$$

$$\text{spent}\, G\, L := \sum_{(u,v)\, \in\, \text{edges}\, G} L(u)$$

$$\text{received}\, m\, n := m \cdot (\text{maxLevel}\, m\, n + 1)$$

$$\text{maxLevel}\, m\, n := \min(\lceil (2m)^{1/2} \rceil, \lfloor (\tfrac{3}{2} n)^{2/3} \rfloor) + 1$$

$$\psi(m, n) := C' \cdot (\text{received}\, m\, n + m + n)$$

Figure 4 Definitions of IsGraph and ψ, with auxiliary definitions.

constant C is derived from the code; its exact value is in principle known, but irrelevant, so we refer to it only by name. The quantity "net G L" is defined as the difference between "received m n", an amount that has been received, and "spent G L", an amount that has been spent. "net G L" can also be understood as a sum over all edges of a per-edge amount, which for each edge (u, v) is "maxLevel m $n - L(u)$". This is a difference between "maxLevel m n", which one can prove is an upper bound on the current level of every vertex, and $L(u)$, the current level of the vertex u. This difference can be intuitively understood as the number of times the edge (u, v) might be traversed in the future by a forward search, due to a promotion of its source vertex u.

We have reviewed the three conjuncts that form IsGraph g G. There remains to define ψ, which also appears in the public specification (Figure 1). Recall that $\psi(m, n)$ denotes the number of credits that we request from the user during a sequence of m edge additions and n vertex additions. Up to another known-but-irrelevant constant factor C', it is defined as "$m + n +$ received m n", that is, a constant amount per operation plus a sufficient amount to justify that $\phi(m, n)$ credits are at hand, as claimed by the invariant IsGraph g G. It is easy to check, by inspection of the last few definitions in Figure 4, that Complexity is satisfied, that is, $\psi(m, n)$ is $O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$.

The public function `add_edge_or_detect_cycle` expects a graph g and two vertices v and w. Its public specification has been presented earlier (Figure 1). The top part of Figure 5 shows the same specification, where IsGraph (01) and ψ (02) have been unfolded. This shows that we receive time credits from two different sources: the potential of the data structure, on the one hand, and the credits supplied by the user for this operation, on the other hand.

$$\forall g\ G\ L\ M\ I\ v\ w.\ \text{let } m := |\text{edges } G| \text{ and } n := |\text{vertices } G| \text{ in}$$
$$v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$$

$$\left\{ \begin{array}{ll} \text{IsRawGraph } g\ G\ L\ M\ I\ *\ [\text{Inv } G\ L\ I]\ *\ \$\phi(G, L)\ * & (01) \\ \$(C' \cdot (\text{received } (m+1)\ n - \text{received } m\ n + 1)) & (02) \end{array} \right\}$$
$$(\texttt{add\_edge\_or\_detect\_cycle}\ g\ v\ w)$$

$$\left\{ \begin{array}{l} \lambda res.\ \text{match } res \text{ with} \\ \quad \mid \text{Ok} \Rightarrow\ \text{let } G' := G + (v, w) \text{ in } \exists L'\ M'\ I'. \\ \qquad\qquad \text{IsRawGraph } g\ G'\ L'\ M'\ I'\ *\ [\text{Inv } G'\ L'\ I']\ *\ \$\phi(G', L') \\ \quad \mid \text{Cycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\}$$

Figure 5 Specifications for edge creation, after unfolding of the representation predicate.

9 Specifications for the Algorithm's Main Functions

The specifications of the two search functions, `backward_search` and `forward_search`, appear in Figure 6. They capture the algorithm's key internal invariants and spell out exactly what each search achieves and how its cost is accounted for.

The function `backward_search` expects a nonnegative integer *fuel*, which represents the maximum number of edges that the backward search is allowed to process. In addition, it expects a graph g and two distinct vertices v and w which must satisfy $L(w) \leq L(v)$. (If that is not the case, an edge from v to w can be inserted immediately.) The graph must be in a valid state (03). The specification requires $A \cdot \mathit{fuel} + B$ credits to be provided (04), for some known-but-irrelevant constants A and B. Indeed, the cost of a backward search is linear in the number of edges that are processed, therefore linear in *fuel*.

This function returns either BackwardCyclic, BackwardAcyclic, or a value of the form BackwardForward(k, *mark*). The first line in the postcondition (05) asserts that the graph remains valid and changes only in that some marks are updated: M changes to M'.

The remainder of the postcondition depends on the function's return value, *res*. If it is BackwardCyclic, then there exists a path in G from w to v (06). If it is BackwardAcyclic, then v and w are at the same level and there is no path from w to v (07). In this case, no forward search is needed. If it is of the form BackwardForward(k, *mark*), then a forward search is required.

In the latter case, the integer k is the level to which the vertex w and its descendants should be promoted during the subsequent forward search. The value *mark* is the mark that was used by this backward search; the subsequent forward search uses this mark to recognize vertices reached by the backward search. The postcondition asserts that the vertex v is marked, whereas w is not (08), since it has not been reached. Moreover, every marked vertex lies at the same level as v and is an ancestor of v (09). Finally, one of the following two cases holds. In the first case, w must be promoted to the level of v and currently lies below the level of v (10) and the backward search is complete, that is, every ancestor of v that lies at the level of v is marked (11). In the second case, w must be promoted to level $L(v) + 1$ and there exist at least *fuel* horizontal edges at the level of v from which v can be reached (12).

The function `forward_search` expects the graph g, the target vertex w, the level k to which w and its descendants should be promoted, and the mark *mark* used by the backward search. The vertex w must be at a level less than k and must be unmarked. The graph must be in a valid state (13). The forward search requires a constant amount of credits B'. Furthermore, it requires access to the potential $\$\phi(G, L)$, which is used to pay for edge processing costs.

$$
\begin{array}{l}
\forall \mathit{fuel}\ g\ G\ L\ M\ I\ v\ w. \\
\mathit{fuel} \geq 0 \ \wedge\ v, w \in \text{vertices}\ G \ \wedge\ v \neq w \ \wedge\ L(w) \leq L(v) \implies \\
\left\{ \begin{array}{ll}
\text{IsRawGraph}\ g\ G\ L\ M\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (03) \\
\$(A \cdot \mathit{fuel} + B) & (04)
\end{array} \right\} \\
(\texttt{backward\_search}\ g\ v\ w\ \mathit{fuel}) \\
\left\{ \begin{array}{ll}
\lambda \mathit{res}.\ \exists M'. & \\
\quad \text{IsRawGraph}\ g\ G\ L\ M'\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (05) \\
\quad [\text{match}\ \mathit{res}\ \text{with} & \\
\quad \mid \text{BackwardCyclic} \Rightarrow w \longrightarrow_G^* v & (06) \\
\quad \mid \text{BackwardAcyclic} \Rightarrow L(v) = L(w) \wedge w \not\longrightarrow_G^* v & (07) \\
\quad \mid \text{BackwardForward}(k, \mathit{mark}) \Rightarrow & \\
\qquad M'\ v = \mathit{mark} \wedge M'\ w \neq \mathit{mark} \wedge & (08) \\
\qquad (\forall x.\ M'\ x = \mathit{mark} \implies L(x) = L(v) \wedge x \longrightarrow_G^* v) \wedge & (09) \\
\qquad (\quad (k = L(v) \wedge L(w) < L(v)\ \wedge & (10) \\
\qquad\quad \forall x.\ L(x) = L(v) \wedge x \longrightarrow_G^* v \implies M'\ x = \mathit{mark}) & (11) \\
\qquad \vee\ (k = L(v) + 1\ \wedge\ \mathit{fuel} \leq |\text{coaccEdgesAtLevel}\ G\ L\ (L(v))\ v|))] & (12)
\end{array} \right\}
\end{array}
$$

$$
\begin{array}{l}
\forall g\ G\ L\ M\ I\ w\ k\ \mathit{mark}. \\
w \in \text{vertices}\ G \ \wedge\ L(w) < k \ \wedge\ M\ w \neq \mathit{mark} \implies \\
\left\{ \begin{array}{ll}
\text{IsRawGraph}\ g\ G\ L\ M\ I\ *\ [\text{Inv}\ G\ L\ I]\ * & (13) \\
\$(B' + \phi(G, L)) & (14)
\end{array} \right\} \\
(\texttt{forward\_search}\ g\ w\ k\ \mathit{mark}) \\
\left\{ \begin{array}{ll}
\lambda \mathit{res}.\ \exists L'\ I'. & \\
\quad \text{IsRawGraph}\ g\ G\ L'\ M\ I'\ * & (15) \\
\quad [L'(w) = k \wedge (\forall x.\ L'(x) = L(x) \vee w \longrightarrow_G^* x)]\ * & (16) \\
\quad \text{match}\ \mathit{res}\ \text{with} & \\
\quad \mid \text{ForwardCyclic} \Rightarrow [\exists x.\ M\ x = \mathit{mark} \wedge w \longrightarrow_G^* x] & (17) \\
\quad \mid \text{ForwardCompleted} \Rightarrow & \\
\qquad \$\phi(G, L')\ * & (18) \\
\qquad [(\forall x\ y.\ L(x) < k \wedge w \longrightarrow_G^* x \longrightarrow_G y \implies M\ y \neq \mathit{mark}) \wedge & (19) \\
\qquad\ \text{InvExcept}\ \{x \mid w \longrightarrow_G^* x \wedge L'(x) = k\}\ G\ L'\ I'] & (20)
\end{array} \right\}
\end{array}
$$

Figure 6 Specifications for the main two auxiliary functions.

This function returns either ForwardCyclic or ForwardCompleted. It affects the low-level graph data structure by updating certain levels and certain sets of horizontal incoming edges: L and I are changed to L' and I' (15). The vertex w is promoted to level k, and a vertex x can be promoted only if it is a descendant of w (16).

If the return value is ForwardCyclic, then, according to the postcondition, there exists a vertex x that is accessible from w and that has been marked by the backward search (17). This implies that there is a path from w through x to v. Thus, adding an edge from v to w would create a cycle. In this case, the data structure invariant is lost.

If the return value is ForwardCompleted, then, according to the postcondition, $\phi(G, L')$ credits are returned (18). This is precisely the potential of the data structure in its new state. Furthermore, two logical propositions hold. First (19), the forward search has not

encountered a marked vertex: for every edge (x, y) that is accessible from w, where x is at level less than k, the vertex y is unmarked. (This implies that there is no path from w to v.) Second (20), the invariant Inv $G\ L'\ I'$ is satisfied *except* for the fact that the property of "replete levels" (Figure 4) may be violated at descendants of w whose level is now k. Fortunately, this proposition (20), combined with a few other facts that are known to hold at the end of the forward search, implies Inv $G'\ L'\ I'$, where G' stands for $G + (v, w)$. In other words, at the end of the forward search, all levels and all sets of horizontal incoming edges are consistent with the mathematical graph G', where the edge (v, w) exists. Thus, after this edge is effectively created in memory by the call `add_edge g v w`, all is well: we have both IsRawGraph $g\ G'\ L'\ M'\ I'$ and Inv $G'\ L'\ I'$, so `add_edge_or_detect_cycle` satisfies its postcondition, under the form shown in Figure 6.

10 Related Work

Neither interactive program verification nor Separation Logic with Time Credits are new (Section 1). Outside the realm of Separation Logic, several researchers present machine-checked complexity analyses, carried out in interactive proof assistants. Van der Weegen and McKinna [46] study the average-case complexity of Quicksort, represented in Coq as a monadic program. The monad is used to introduce both nondeterminism and comparison-counting. Danielsson [11] implements a *Thunk* monad in Agda and uses it to reason about the amortized complexity of data structures that involve delayed computation and memoization. McCarthy et al. [30] present a monad that allows the time complexity of a Coq computation to be expressed in its type. Nipkow [33] proposes machine-checked amortized complexity analyses of several data structures in Isabelle/HOL. The code is manually transformed into a cost function.

Several mostly-automated program verification systems can verify complexity bounds. Madhavan et al. [29] present such a system, which can deal with programs that involve memoization, and is able to infer some of the constants that appear in user-supplied complexity bounds. Srikanth et al. [40] propose an automated verifier for user-supplied complexity bounds that involve polynomials, exponentials, and logarithms. When a bound is not met, a counter-example can be produced. Such automated tools are inherently limited in the scope of programs that they can handle. For instance, the algorithm considered in the present paper appears to be far beyond reach of any of these fully automated tools.

There is also a vast body of work on fully-automated inference of complexity bounds, beginning with Wegbreit [47] and continuing with more recent papers and tools [15, 14, 2, 13, 20]. Carbonneaux et al.'s analysis produces certificates whose validity can be checked by Coq [5]. It is possible in principle to express these certificates as derivations in Separation Logic with Time Credits. This opens the door to provably-safe combinations of automated and interactive tools.

Finally, there is a rich literature on static and dynamic analyses that aim at detecting performance anomalies [34, 31, 21, 28, 45].

Early work on the verification of garbage collection algorithms includes, in some form, the verification of a graph traversal. For example, Russinoff [38] uses the Boyer-Moore theorem prover to verify Ben Ari's incremental garbage collector, which employs a two-color scheme. In more recent work, specifically focused on the verification of graph algorithms, Lammich [25], Pottier [36], and Chen et al. [10, 9] verify various algorithms for finding the strongly connected components of a directed graph. In particular, Chen et al. [9] repeat a single proof using Why3, Coq and Isabelle. None of these works include a verification of asymptotic complexity.

11 Conclusion

In this paper, we have used a powerful program logic to simultaneously verify the correctness and complexity of an actual implementation of a state-of-the-art incremental cycle detection algorithm. Although neither interactive program verification nor Separation Logic with Time Credits are new, there are still relatively few examples of applying this simultaneous-verification approach to nontrivial algorithms or data structures. We hope we have demonstrated that this approach is indeed viable, and can be applied to a wide range of algorithms, including ones that involve mutable state, dynamic memory allocation, higher-order functions, and amortization.

As a technical contribution, whereas all previous works use credits in $\mathbb{N}$, we use credits in $\mathbb{Z}$ and allow negative credits to exist temporarily. We explain in Section 3 why this is safe and convenient.

Following Guéneau et al. [17], our public specification exposes an asymptotic complexity bound: no literal constants appear in it. We remark, however, that it is often difficult to use something that resembles the O notation in specifications and proofs. Indeed, in its simplest form, a use of this notation in a mathematical statement $S[O(g)]$ can be understood as an occurrence of a variable f that is existentially quantified at the beginning of the statement: $\exists f.\ (f \preceq g) \wedge S[f]$. An example of such a statement was given earlier (Section 2). Here, f denotes an unknown function, which is dominated by the function g. The definition of the domination relation $\preceq$ involves further quantifiers [17]. In the analysis of a complex algorithm or data structure, however, it is often the case that an existential quantifier must be hoisted very high, so that its scope encompasses not just a single statement, but possibly a group of definitions, statements, and proofs. The present paper shows several instances of this phenomenon. In the public specification (Figure 1), the cost function ψ must be existentially quantified at the outermost level. In the definition of the data structure invariant (Figure 4) and in the proofs that involve this invariant, several constants appear, such as C and C', which must be defined beforehand. Thus, even if one could formally use $S[O(g)]$ as syntactic sugar for $\exists f.\ (f \preceq g) \wedge S[f]$, we fear that one might not be able to use this sugar very often, because a lot of mathematical work is carried out under the existential quantifier, in a context where f must be explicitly referred to by name. That said, novel ways of understanding the O notation may permit further progress; Affeldt et al. [1] make interesting steps in such a direction.

In future work, we would like to verify the algorithm that is used in the kernel of Coq to check the satisfiability of universe constraints. These are conjunctions of strict and non-strict ordering constraints, $x < y$ and $x \leq y$. This requires an incremental cycle detection algorithm that maintains strong components. Bender et al. [4, §5] present such an algorithm. It relies on a Union-Find data structure, whose correctness and complexity have been previously verified [8]. It is therefore tempting to re-use as much verified code as we can, without modification.

References

1 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization Techniques for Asymptotic Reasoning in Classical Analysis. *Journal of Formalized Reasoning*, 11(1):43–76, 2018.

2 Elvira Albert, Puri Arenas, Samir Genaim, German Puebla, and Damiano Zanardini. Cost analysis of object-oriented bytecode programs. *Theoretical Computer Science*, 413(1):142–159, 2012.

3 Robert Atkey. Amortised Resource Analysis with Separation Logic. *Logical Methods in Computer Science*, 7(2:17), 2011.

4 Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Transactions on Algorithms*, 12(2):14:1–14:22, 2016.

5 Quentin Carbonneaux, Jan Hoffmann, Thomas Reps, and Zhong Shao. Automated Resource Analysis with Coq Proof Objects. In *Computer Aided Verification (CAV)*, volume 10427 of *Lecture Notes in Computer Science*, pages 64–85. Springer, 2017.

6 Arthur Charguéraud. Characteristic Formulae for the Verification of Imperative Programs, 2013. Unpublished. `http://www.charguer aud.org/research/2013/cf/cf.pdf`.

7 Arthur Charguéraud. The CFML tool and library. `http://www.charguer aud.org/softs/cfml/`, 2019.

8 Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. *Journal of Automated Reasoning*, 2017.

9 Ran Chen, Cyril Cohen, Jean-Jacques Lévy, Stephan Merz, and Laurent Théry. Formal Proofs of Tarjan's Algorithm in Why3, Coq, and Isabelle. Manuscript, 2018.

10 Ran Chen and Jean-Jacques Lévy. A Semi-automatic Proof of Strong Connectivity. In *Verified Software: Theories, Tools and Experiments*, volume 10712 of *Lecture Notes in Computer Science*, pages 49–65. Springer, 2017.

11 Nils Anders Danielsson. Lightweight Semiformal Time Complexity Analysis for Purely Functional Data Structures. In *Principles of Programming Languages (POPL)*, 2008.

12 Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf, and Jan-Georg Smaus. A Fully Verified Executable LTL Model Checker. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 463–478. Springer, 2013.

13 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing Program Termination and Complexity Automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017.

14 Sumit Gulwani. SPEED: symbolic complexity bound analysis. In *Computer Aided Verification (CAV)*, volume 5643 of *Lecture Notes in Computer Science*, pages 51–62. Springer, 2009.

15 Sumit Gulwani, Krishna K. Mehra, and Trishul M. Chilimbi. SPEED: precise and efficient static estimation of program computational complexity. In *Principles of Programming Languages (POPL)*, pages 127–139, 2009.

16 Armaël Guéneau. Dune pull request #1955, March 2019.

17 Armaël Guéneau, Arthur Charguéraud, and François Pottier. A Fistful of Dollars: Formalizing Asymptotic Complexity Claims via Deductive Program Verification. In *European Symposium on Programming (ESOP)*, volume 10801 of *Lecture Notes in Computer Science*, pages 533–560. Springer, 2018.

18 Maximilian P. L. Haslbeck and Tobias Nipkow. Hoare Logics for Time Bounds: A Study in Meta Theory. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 10805 of *Lecture Notes in Computer Science*, pages 155–171. Springer, 2018.

19 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

20 Jan Hoffmann, Ankush Das, and Shu-Chun Weng. Towards automatic resource bound analysis for OCaml. In *Principles of Programming Languages (POPL)*, pages 359–373, 2017.

21 Benjamin Holland, Ganesh Ram Santhanam, Payas Awadhutkar, and Suresh Kothari. Statically-Informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities. In *Source Code Analysis and Manipulation (SCAM)*, pages 79–84, 2016.

22 John E. Hopcroft. Computer Science: The Emergence of a Discipline. *Communications of the ACM*, 30(3):198–202, 1987.

23 Jacques-Henri Jourdan. Coq pull request #89, July 2015.

24 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

25 Peter Lammich. Verified Efficient Implementation of Gabow's Strongly Connected Component Algorithm. In *Interactive Theorem Proving (ITP)*, volume 8558 of *Lecture Notes in Computer Science*, pages 325–340. Springer, 2014.

26 Peter Lammich. Refinement to Imperative/HOL. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 253–269. Springer, 2015.

27 Peter Lammich and René Neumann. A Framework for Verifying Depth-First Search Algorithms. In *Certified Programs and Proofs (CPP)*, pages 137–146, 2015.

28 Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. PerfFuzz: Automatically generating pathological inputs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 254–265, 2018.

29 Ravichandhran Madhavan, Sumith Kulal, and Viktor Kuncak. Contract-based resource verification for higher-order functions with memoization. In *Principles of Programming Languages (POPL)*, pages 330–343, 2017.

30 Jay A. McCarthy, Burke Fetscher, Max S. New, Daniel Feltey, and Robert Bruce Findler. A Coq Library for Internal Verification of Running-Times. In *Functional and Logic Programming*, volume 9613 of *Lecture Notes in Computer Science*, pages 144–162. Springer, 2016.

31 Rashmi Mudduluru and Murali Krishna Ramanathan. Efficient Flow Profiling for Detecting Performance Bugs. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 413–424, 2016.

32 Glen Mével, Jacques-Henri Jourdan, and François Pottier. Time credits and time receipts in Iris. In Luis Caires, editor, *European Symposium on Programming (ESOP)*, volume 11423 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 2019.

33 Tobias Nipkow. Amortized Complexity Verified. In *Interactive Theorem Proving (ITP)*, volume 9236 of *Lecture Notes in Computer Science*, pages 310–324. Springer, 2015.

34 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Programming Language Design and Implementation (PLDI)*, pages 369–378, 2015.

35 Matthew Parkinson and Gavin Bierman. Separation logic and abstraction. In *Principles of Programming Languages (POPL)*, pages 247–258, 2005.

36 François Pottier. Depth-First Search and Strong Connectivity in Coq. In *Journées Françaises des Langages Applicatifs (JFLA)*, 2015.

37 John C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS)*, pages 55–74, 2002.

38 David M. Russinoff. A Mechanically Verified Incremental Garbage Collector. *Formal Aspects of Computing*, 6(4):359–390, 1994. doi:10.1007/BF01211305.

39 Matthieu Sozeau and Nicolas Tabareau. Universe Polymorphism in Coq. In *Interactive Theorem Proving (ITP)*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014.

40 Akhilesh Srikanth, Burak Sahin, and William R. Harris. Complexity verification using guided theorem enumeration. In *Principles of Programming Languages (POPL)*, pages 639–652, 2017.

41 Jane Street. Dune: A composable build system, 2018.

42 Robert Endre Tarjan. Amortized Computational Complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

43 Robert Endre Tarjan. Algorithmic Design. *Communications of the ACM*, 30(3):204–212, 1987.

44 The Coq development team. *The Coq Proof Assistant*, 2019.

45 Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Synthesizing programs that expose performance bottlenecks. In *Code Generation and Optimization (CGO)*, pages 314–326, 2018.

46 Eelis van der Weegen and James McKinna. A Machine-Checked Proof of the Average-Case Complexity of Quicksort in Coq. In *Types for Proofs and Programs*, volume 5497 of *Lecture Notes in Computer Science*, pages 256–271. Springer, 2008.

47 Ben Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, 1975.