

Formally verified incremental cycle detection

Armaël Guéneau

with J.-H. Jourdan, A. Charguéraud and F. Pottier

Formally Verified Algorithms

Can we formally verify the *functional correctness*...

Formally Verified Algorithms

Can we formally verify the *functional correctness*

and *asymptotic complexity*...

Formally Verified Algorithms

Can we formally verify the *functional correctness*

and *asymptotic complexity*

of *non-trivial* algorithms...

Formally Verified Algorithms

Can we formally verify the *functional correctness*

and *asymptotic complexity*

of *non-trivial* algorithms

with respect to concrete source code?

Previous work: time credits (1)

Previous work: interactive proofs in Separation Logic with *Time Credits*, using Coq and the CFML library.

Charguéraud and Pottier (2017) verify Tarjan's Union-Find.

- Manual accounting of credits: “union costs $4\alpha(n) + 12$ ”;
- Challenging mathematical analysis but fairly short code;

Previous work: time credits (2)

Guéneau, Charguéraud and Pottier (2018) formalize the O notation and advertise for *asymptotic* complexity specifications, e.g. “union costs $f(n)$ where $f \in O(\alpha(n))$ ”.

- Required for specifications to be modular;
- Proofs use a semi-automated *cost synthesis* mechanism;
- However, only small illustrative examples are presented.

Question: **does this approach scale?**

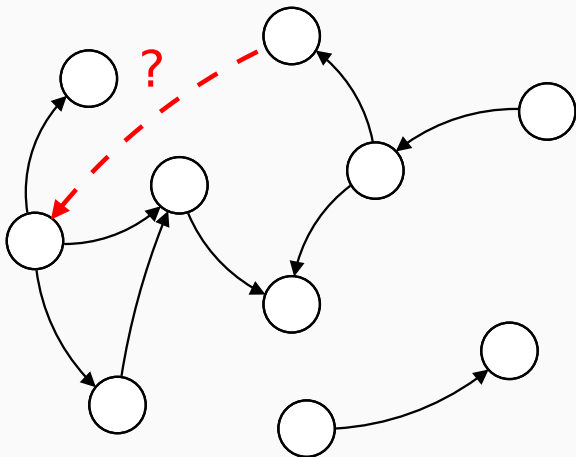
In this talk

Verification of a **state-of-the-art** incremental cycle detection algorithm due to Bender, Fineman, Gilbert and Tarjan (2016).

- non-trivial implementation (200 lines of OCaml code)
- subtle complexity analysis
- used in Coq (universe constraints) and Dune (build dependencies)

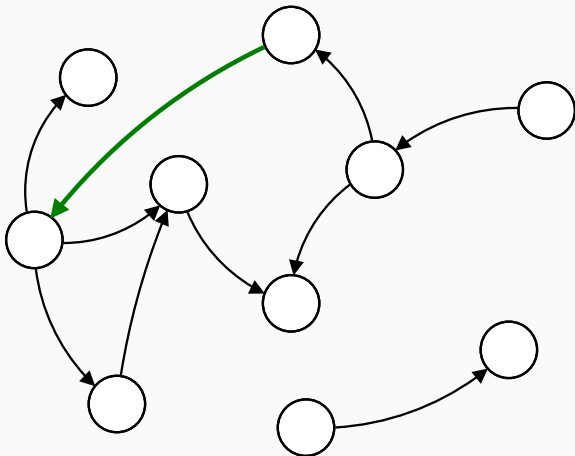
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



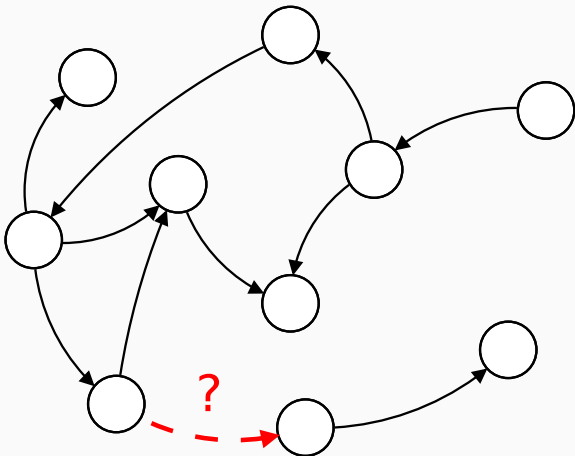
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



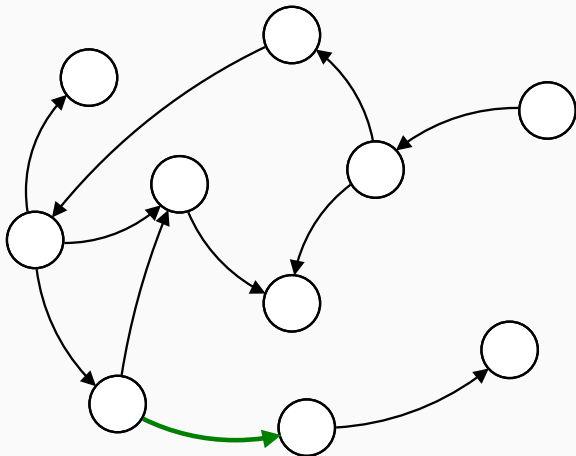
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



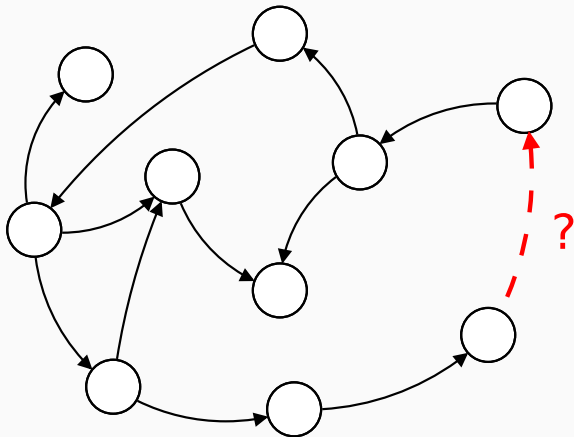
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



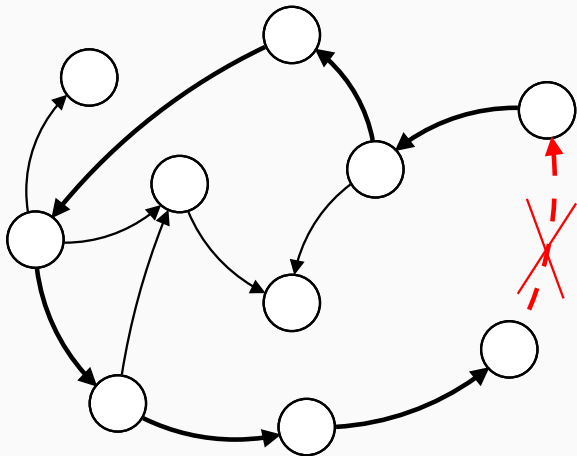
Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



Incremental cycle detection

The problem: checking for acyclicity of a dynamically constructed graph



Naive algorithm: traverse the graph at each step.
Inserting m arcs costs $O(m^2)$.

Using Bender et al.'s algorithm, inserting m arcs costs:

- $O(m\sqrt{m})$ for sparse graphs;
- $O(mn^{2/3})$ for dense graphs.

In the general case: $O(m \cdot \min(m^{1/2}, n^{2/3}))$.

Contributions

- An OCaml implementation as a standalone library;
- A machine-checked proof of both its functional correctness and amortized asymptotic complexity;
- A simple yet crucial improvement to make Bender et al.'s algorithm truly online;
- Time credits that are counted in \mathbb{Z} (instead of \mathbb{N}): this leads to significantly fewer proof obligations (!).

Minimal OCaml interface

```
type add_edge_result =  
  | EdgeAdded  
  | EdgeCreatesCycle  
  
val add_edge_or_detect_cycle :  
  graph -> vertex -> vertex ->  
  add_edge_result
```

Toplevel specification (functional correctness only)

$$\begin{aligned} \forall g G v w. \quad & \text{let } m := |\text{edges } G| \text{ in} \\ & \text{let } n := |\text{vertices } G| \text{ in} \\ & v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies \\ & \left\{ \begin{array}{l} \text{IsGraph } g G \\ (\text{add_edge_or_detect_cycle } g v w) \end{array} \right\} \\ & \left\{ \begin{array}{l} \lambda \text{res. match res with} \\ \quad | \text{EdgeAdded} \Rightarrow \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \Rightarrow [w \longrightarrow_G^* v] \end{array} \right\} \end{aligned}$$
$$\forall g G. \text{IsGraph } g G \Vdash \text{IsGraph } g G \star [\forall x. x \dashrightarrow_G^+ x]$$

Toplevel specification (functional correctness only)

$\forall g G v w.$ let $m := |\text{edges } G|$ in
let $n := |\text{vertices } G|$ in

Separation Logic with Time Credits:

- $\$n$ asserts the ownership of n time credits
- $\$n$ is a Separation Logic assertion, like $p \hookrightarrow 3$
- Each **function call** (or loop iteration) consumes $\$1$
- $\$(n + m) \equiv \$n \star \$m$
- Credits are not duplicable: $\$1 \not\Rightarrow \$1 \star \$1$

$\forall g G. \text{ISGraph } g \ G \Vdash \text{ISGraph } g \ G \star [\forall x. x \mapsto_G x]$

Toplevel specification (correctness and complexity)

$$\begin{aligned} \forall g G v w. \quad & \text{let } m := |\text{edges } G| \text{ in} \\ & \text{let } n := |\text{vertices } G| \text{ in} \\ & v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies \\ & \left\{ \text{IsGraph } g G \star \$(\dots) \right\} \\ & (\text{add_edge_or_detect_cycle } g v w) \\ & \left\{ \begin{array}{l} \lambda \text{res. match res with} \\ \quad | \text{EdgeAdded} \implies \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \implies [w \longrightarrow_G^* v] \end{array} \right\} \end{aligned}$$
$$\forall g G. \text{IsGraph } g G \Vdash \text{IsGraph } g G \star [\forall x. x \dashrightarrow_G^+ x]$$

Toplevel specification (correctness and complexity)

$$\begin{aligned} \forall g G v w. \quad & \text{let } m := |\text{edges } G| \text{ in} \\ & \text{let } n := |\text{vertices } G| \text{ in} \\ & v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies \\ & \left\{ \begin{array}{l} \text{IsGraph } g G \star \$(\psi(m+1, n) - \psi(m, n)) \\ (\text{add_edge_or_detect_cycle } g v w) \end{array} \right\} \\ & \left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \implies \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \implies [w \xrightarrow*_G v] \end{array} \right\} \end{aligned}$$

$$\forall g G. \text{IsGraph } g G \Vdash \text{IsGraph } g G \star [\forall x. x \dashrightarrow_G^+ x]$$

$$\psi \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$$

Using the specification

<code>let g = create_graph () in</code>	
<code>add_vertex g 1;</code>	$\$(\psi(0, 1) - \psi(0, 0))$
<code>...</code>	
<code>add_vertex g n;</code>	$\$(\psi(0, n) - \psi(0, n - 1))$
<code>add_edge_or_detect_cycle g 1 2;</code>	$\$(\psi(1, n) - \psi(0, n))$
<code>add_edge_or_detect_cycle g 2 3;</code>	$\$(\psi(2, n) - \psi(1, n))$
<code>...</code>	
<code>add_edge_or_detect_cycle g (m-1) m;</code>	$\$(\psi(m, n) - \psi(m - 1, n))$

Total cost: $\psi(m, n) - \psi(0, 0)$

Using the specification

let g = create_graph () in	
add_vertex g 1;	$\$(\psi(0, 1) - \psi(0, 0))$
...	
add_vertex g n;	$\$(\psi(0, n) - \psi(0, n - 1))$
add_edge_or_detect_cycle g 1 2;	$\$(\psi(1, n) - \psi(0, n))$
add_edge_or_detect_cycle g 2 3;	$\$(\psi(2, n) - \psi(1, n))$
...	
add_edge_or_detect_cycle g (m-1) m;	$\$(\psi(m, n) - \psi(m - 1, n))$

Total cost: $\psi(m, n) - \psi(0, 0) \in O(m \cdot \min(m^{1/2}, n^{2/3}) + n)$

Bender et al.'s algorithm in action

Demo

IsGraph's hidden potential

$\forall g G v w.$

let $m, n := |\text{edges } G|, |\text{vertices } G|$ in

$v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$

$\left\{ \text{IsGraph } g G \star \$(\psi(m+1, n) - \psi(m, n)) \right\}$

$(\text{add_edge_or_detect_cycle } g v w)$

$\left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \Rightarrow \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \Rightarrow [w \xrightarrow*_G v] \end{array} \right\}$

IsGraph's hidden potential

$\forall g G v w.$

let $m, n := |\text{edges } G|, |\text{vertices } G|$ in

$v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$

$\left\{ \text{IsGraph } g G \star \$(\psi(m+1, n) - \psi(m, n)) \right\}$

$(\text{add_edge_or_detect_cycle } g v w)$

$\left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \Rightarrow \text{IsGraph } g (G + (v, w)) \\ \quad | \text{EdgeCreatesCycle} \Rightarrow [w \xrightarrow*_G v] \end{array} \right\}$

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

$\text{Inv } G L I := (\forall x. x \dashrightarrow_G^+ x) \wedge \dots$

IsGraph's hidden potential

$\forall g G L M I v w.$

let $m, n := |\text{edges } G|, |\text{vertices } G|$ in

$v, w \in \text{vertices } G \wedge (v, w) \notin \text{edges } G \implies$

$\left\{ \begin{array}{l} \text{IsRawGraph } g G L M I \star [\text{Inv } G L I] \star \mathcal{S}\phi(G, L) \\ \star \mathcal{S}(\psi(m+1, n) - \psi(m, n)) \end{array} \right\}$

$(\text{add_edge_or_detect_cycle } g v w)$

$\left\{ \begin{array}{l} \lambda \text{ res. match res with} \\ \quad | \text{EdgeAdded} \implies \text{let } G' := G + (v, w) \text{ in } \exists L' M' I'. \\ \quad \quad \text{IsRawGraph } g G' L' M' I' \star [\text{Inv } G' L' I'] \star \mathcal{S}\phi(G', L') \\ \quad | \text{EdgeCreatesCycle} \implies [w \longrightarrow_G^* v] \end{array} \right\}$

A very informal sketch of the complexity analysis

Current level enumeration: directly $O(\psi(m + 1, n) - \psi(m, n))$

Levels update:

- increases the level of edges
- decreases ϕ (i.e. releases time credits)
- $O(1)$ amortized (!)

Adding the new edge:

- increases ϕ (i.e. needs to provide time credits)
- potential for the new edge: $O(\psi(m + 1, n) - \psi(m, n))$

Complexity invariants depend on concrete code

We must give a definition for ϕ and ψ :

$$\phi(G, L) := C \cdot \sum_{(u,v) \in G} (\sqrt{m} - L(u))$$

$$\psi(m, n) := C' \cdot (m\sqrt{m} + m + n + 1)$$

...for some constants C and C' which we must define.

NB: “ $\psi(m, n) := O(m\sqrt{m} + n)$ ” does not make sense!

C and C' closely depend on details of the implementation.

We *do not want* to write them by hand in the proof!

Robust complexity proofs using abstract constants

The solution relies on our mechanisms for *cost synthesis* and *deferring proof obligations*.

Proof sketch of `update_levels`'s specification:

$$\exists C. \forall g w l. \{ \$ (C \cdot (\dots)) \star \dots \} \text{update_levels } g w l \{ \dots \}$$

- Defer choosing a value for C ;
- Cost synthesis yields obligations of the form “ $C \geq \text{cost_foo} + \text{cost_bar} + \dots$ ”: defer them;
- Automatically deduce a suitable value for C .

Then, ϕ is defined using “the” C from the specification.

Time Credits in \mathbb{Z}

Originally, Time Credits are counted in \mathbb{N} :

$$\begin{aligned} \$0 &\equiv \text{emp} \\ \forall m n \in \mathbb{N}. \quad \$(m + n) &\equiv \$m \star \$n \\ \forall n \in \mathbb{N}. \quad \$n &\Vdash \text{emp} \end{aligned}$$

We work in a variant of SL with credits counted in \mathbb{Z} :

$$\begin{aligned} \$0 &\equiv \text{emp} \\ \forall m n \in \mathbb{Z}. \quad \$(m + n) &\equiv \$m \star \$n \\ \forall n \in \mathbb{Z}. \quad \$n \star [n \geq 0] &\Vdash \text{emp} \end{aligned}$$

Corollary: $\forall n \in \mathbb{Z}. \text{emp} \equiv \$n \star \$(-n)$

Time Credits in \mathbb{Z} enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =  
  match l with  
  | x :: xs when x <> 0 -> walk xs  
  | _ -> l
```


Time Credits in \mathbb{Z} enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =  
  match l with  
  | x :: xs when x <> 0 -> walk xs  
  | _ -> l
```

“Pay-per-use” pricing scheme:

$$\forall l. \{\text{emp}\} \text{ walk } l \{ \lambda l'. \$(|l'| - |l|) \star [\text{suffix } l' l \wedge l' \neq l] \}$$

Time Credits in \mathbb{Z} enable simpler specifications (& invariants)

```
let rec walk (l: int list): int list =  
  match l with  
  | x :: xs when x <> 0 -> walk xs  
  | _ -> l
```

“Pay-per-use” pricing scheme:

$$\forall l. \{\text{emp}\} \text{ walk } l \{ \lambda l'. \$(|l'| - |l|) \star [\text{suffix } l' l \wedge l' \neq l] \}$$

“Fixed-rate” pricing scheme:

$$\forall l. \{\$|l|\} \text{ walk } l \{ \lambda l'. \$|l'| \star [\text{suffix } l' l \wedge l' \neq l] \}$$

Summary

- We improve and verify a state-of-the-art algorithm;
- SL with (Possibly Negative) Time Credits is powerful; it allows writing rich and modular specifications;
- Our code is already useful: integrated into Dune, bringing a 7x performance improvement (!);
- Our cost synthesis and deferring mechanisms allow manageable proofs at scale.

More in the paper and my (upcoming) PhD dissertation.

⇒ <https://gitlab.inria.fr/agueneau/incremental-cycles>

Producing the right answer is good.

Producing the right answer is good.

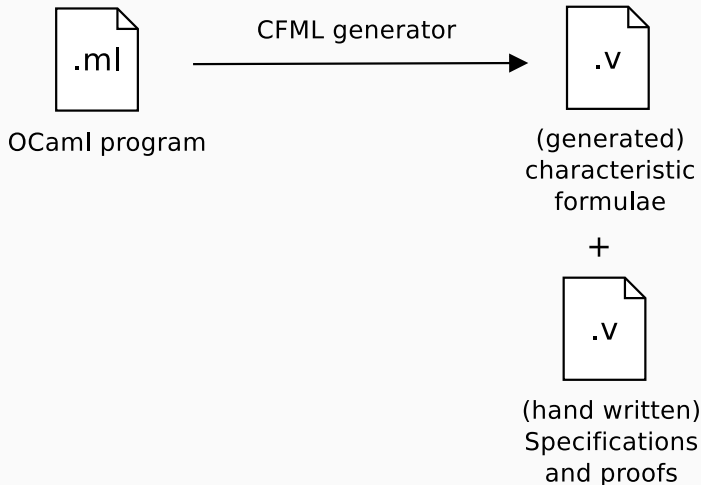
Producing the right answer **at the right time** is better.

Producing the right answer is good.

Producing the right answer **at the right time** is better.

Don't promise—just **prove** it!

Program verification framework: Coq and (extended) CFML



Example specifications using time credits

Complexity specification using explicit time credits:

$$\forall g G. \{ \text{IsGraph } g \ G \star \$ (3 |\text{edges } G| + 5) \} \text{dfs}(g) \{ \text{IsGraph } g \ G \}$$

Asymptotic complexity specification:

$$\exists (f : \mathbb{Z} \rightarrow \mathbb{Z}).$$

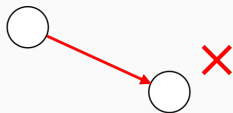
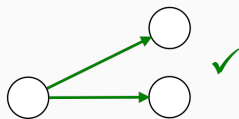
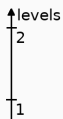
$$f \in O_{\mathbb{Z}}(\lambda m.m)$$

$$\wedge \forall g G. \{ \text{IsGraph } g \ G \star \$ f(|\text{edges } G|) \} \text{dfs}(g) \{ \text{IsGraph } g \ G \}$$

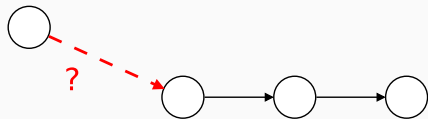
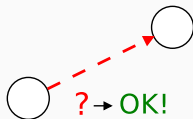
Idea 1: Levels

Each vertex v is given a level $L(v)$.

Invariant: $v \rightarrow_G w \implies L(v) \leq L(w)$



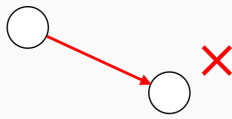
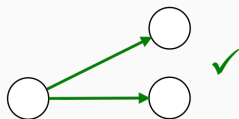
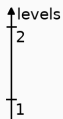
Levels can accelerate the search, but need to be maintained:



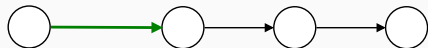
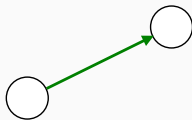
Idea 1: Levels

Each vertex v is given a level $L(v)$.

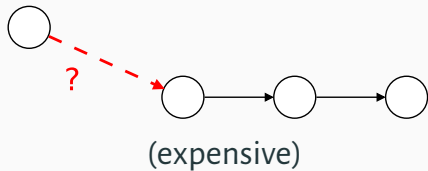
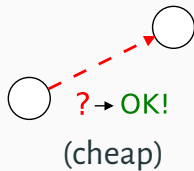
Invariant: $v \rightarrow_G w \implies L(v) \leq L(w)$



Levels can accelerate the search, but need to be maintained:



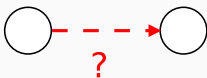
Idea 1 (bis): Tradeoff on the number of levels



- Too many levels: the expensive case triggers often, outweighing the cheap case
- Too few levels: similar to the naive algorithm, insufficient benefit out of the cheap case

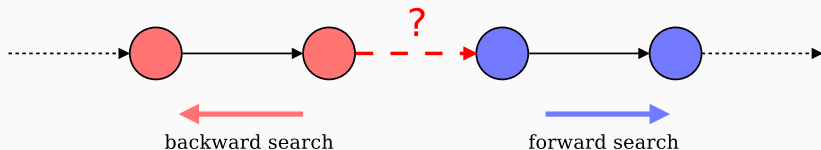
Idea 1 (ter): Tradeoff on the number of levels

Why do we gain anything?



Adding a horizontal edge: the search for a cycle can be restricted *to this level*.

Idea 2: Two-way Search



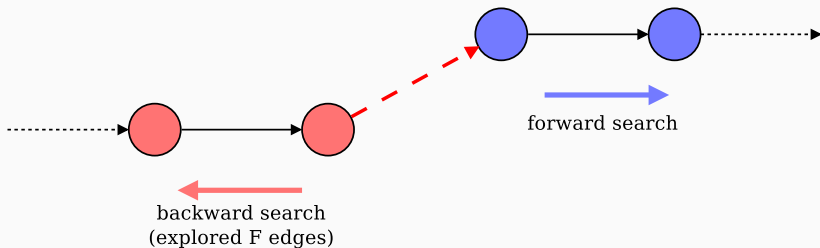
The backward search is:

- *restricted* to the same level
- *bounded* by a predetermined number of edges F

The forward search restores the invariant on levels as it goes.

Idea 3: when do new levels get created?

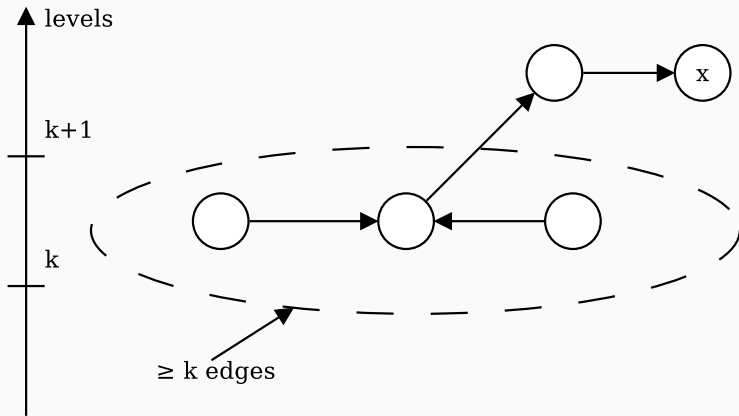
If the backward search explores all F edges...



then nodes are moved to a higher level during the forward search.

Main complexity invariant: levels are “replete”

For every node x at level $k + 1$ there are at least k edges at level k from which x can be reached.

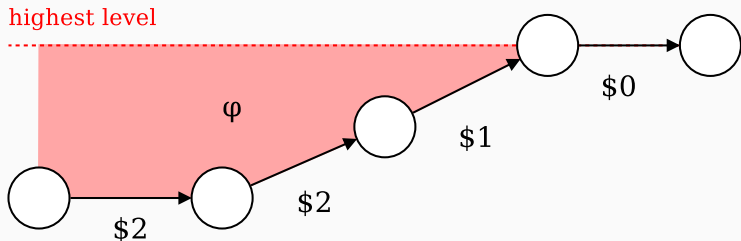


Corollary: there are at least k edges at level k .

The graph potential ϕ

The potential ϕ stores Time Credits for edges depending on their current level (lower level = more credits).

Credits are received at each edge insertion, and spent when *raising* nodes.



$$\phi(G, L) := C \cdot \sum_{(u,v) \in G} (\text{highest_level } G L - L(u))$$

Forward traversal economics

- Traversing an edge (u, v) costs 1
- Raising v releases $\text{card}(\{w \mid (v, w) \in G\})$ from ϕ (this pays for exploring all the successors of v)
- The *stack* holds credits for the next edges to explore

The traversal stack contains credits representing the “working capital” of the traversal.

$\text{out}(v) := \text{card}(\{w \mid (v, w) \in G\})$

$|stack| := \sum_{v \in stack} \text{out}(v)$

```
let rec visit_forward g new_level visited stack =  
  match stack with  
  | [] -> ()  
  | u :: stack ->  
    let stack = List.fold_left (fun stack v ->  
      ...  
      set_level g v new_level;  
      v :: stack  
    ) stack (get_outgoing g u) in  
  visit_forward g new_level visited stack
```

$out(v) := \text{card}(\{w \mid (v, w) \in G\})$

$|stack| := \sum_{v \in stack} out(v)$

```

       $\varphi(G, L)$ 
       $|stack|$ 
let rec visit_forward g new_level visited stack =
  match stack with
  | [] -> ()
  | u :: stack ->  $out(u) + |stack|$ 
    let stack = List.fold_left (fun stack v ->
      ...  $|stack|$ 
      set_level g v new_level;
      v :: stack  $out(v) + |stack|$ 
    ) stack (get_outgoing g u) in
  visit_forward g new_level visited stack
```

Proof methodology, in practice

In practice, credit counts involve multiplicative constants:

$$\begin{aligned}\phi(G, L) &:= C \cdot \sum_{(u,v) \in G} (\text{highest_level } G L - L(u)) \\ |stack| &:= C' \cdot \sum_{v \in stack} \text{out}(v)\end{aligned}$$

$\exists C'' . 0 \leq C'' \wedge \forall g \text{ nl vs stack } \dots$

$\{ \$C'' \star \$|stack| \star \dots \} \text{ visit_forward } g \text{ nl vs stack } \{ \lambda(). \dots \}$

C, C' and C'' depend on specifics of the implementation.

We develop tactics to make the proofs independent from their exact expression, and avoid writing it explicitly by hand.

Time Credits in \mathbb{N} and redundant proof obligations

Starting with $\$n$ then paying for operations with costs m_1, m_2, \dots, m_k produces redundant proof obligations:

$\$n$

pay $\$m_1$

$$\rightsquigarrow n - m_1 \geq 0$$

$\$(n - m_1)$

pay $\$m_2$

$$\rightsquigarrow n - m_1 - m_2 \geq 0$$

\dots

$\$(n - m_1 - m_2 - \dots - m_{k-1})$

pay $\$m_k$

$$\rightsquigarrow n - m_1 - m_2 - \dots - m_k \geq 0$$

Time Credits in \mathbb{Z} eliminate redundant proof obligations

Paying for a sequence of operations produces a single final proof obligation:

$\$n$

pay $\$m_1$

\rightsquigarrow no proof obligation

$\$(n - m_1)$

pay $\$m_2$

\rightsquigarrow no proof obligation

...

$\$(n - m_1 - \dots - m_{k-1})$

pay $\$m_k$

\rightsquigarrow no proof obligation

discard $\$(n - m_1 - \dots - m_k)$ $\rightsquigarrow n - m_1 - \dots - m_k \geq 0$

This also allows for *simpler* loop invariants and specifications.

Pre/Post-condition duality

With integer time credits, these two specifications are equivalent (using the frame rule):

$$\{\$n\} \text{ f } n \{ \lambda(). \text{emp} \}$$

$$\{\text{emp}\} \text{ f } n \{ \lambda(). \$(-n) \}$$

Bonus: returning negative credits allow the complexity to depend on the result of the function! Example:

$$\{\text{emp}\} \text{ collatz_stopping_time } n \{ \lambda i. \$(-i) \}$$

Interaction with loops

From the proof of the forward traversal:

```
//  $\phi(G, L) \star [\text{Inv } G \ L \ I]$ 
List.fold_left ... (fun ... ->
  //  $\exists L'. \phi(G, L')$ 
  [extract credits from  $\phi(G, L')$ ]
  ...
)
//  $\phi(G, L'') \star [\text{Inv } G \ L'' \ I'']$ 
```

(Difficult) Lemma: $\forall G \ L \ I. \text{Inv } G \ L \ I \implies \phi(G, L) \geq 0$

Time Credits in \mathbb{N} would require a nontrivial strengthening of the loop invariant.

Walk

```
let rec walk (a: int array) (i: int): int =  
  if i < Array.length a && a.(i) <> 0 then walk a (i+1)  
  else i+1
```

Walk

```
let rec walk (a: int array) (i: int): int =  
  if i < Array.length a && a.(i) <> 0 then walk a (i+1)  
  else i+1
```

$\forall a \ i \ A. 0 \leq i \leq |A| \implies$

$\{a \rightsquigarrow \text{Array } A\} \text{ walk } a \ i \ \{\lambda j. a \rightsquigarrow \text{Array } A \star \$(i - j) \star [i < j \leq |A|]\}$

Walk

```
let rec walk (a: int array) (i: int): int =  
  if i < Array.length a && a.(i) <> 0 then walk a (i+1)  
  else i+1
```

$$\forall a \ i \ A. 0 \leq i \leq |A| \implies$$
$$\{a \rightsquigarrow \text{Array } A\} \text{ walk } a \ i \ \{\lambda j. a \rightsquigarrow \text{Array } A \star \$(i - j) \star [i < j \leq |A|]\}$$
$$\forall a \ i \ A. 0 \leq i \leq |A| \implies$$
$$\{a \rightsquigarrow \text{Array } A \star \$(|A| - i)\}$$
$$\text{ walk } a \ i$$
$$\{\lambda j. a \rightsquigarrow \text{Array } A \star \$(|A| - j) \star [i < j \leq |A|]\}$$

Interruptible Iteration

```
let rec interruptible_iter f l =  
  match l with  
  | [] -> true  
  | x :: l' -> f x && interruptible_iter f l'
```

Interruptible Iteration

```
let rec interruptible_iter f l =  
  match l with  
  | [] -> true  
  | x :: l' -> f x && interruptible_iter f l'
```

Integer time credits allow for an intuitive specification:

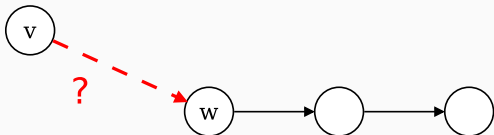
$\forall I l f.$

$$\begin{aligned} & (\forall x l'. \text{prefix } l' l \implies \{I l'\} f x \{\lambda b. I (x :: l')\}) \implies \\ & \{I [] \star \$|l|\} \\ & \text{interruptible_iter } f l \\ & \{\lambda b. \text{if } b \text{ then } I l \text{ else } \exists l' l''. I l' \star \$|l''| \star [l = l' ++ l'']\} \end{aligned}$$

Challenges

- Understanding the algorithm (!)
- (Re)inventing the complexity invariants
- Designing robust and generic invariants for (interruptible) graph traversals
- Designing Coq tactics for interactive reasoning using integer time credits

Idea 3: Policy for raising nodes to a new level



w and its descendants need to be raised to $L(v)$ or higher.

Bender et al.'s policy:

- If the backward search from v was not interrupted:
raised to $L(v)$
- Otherwise, raised to $L(v) + 1$ (possibly creating a new level).

Idea 4: choice of F

Recall: backward search is bounded to visit at most F edges.
The choice of F is crucial to get the correct complexity.

In Bender et al.:

$F = \min(m^{1/2}, n^{2/3})$, for m and n of the *final* graph
(hard to know in practice).

In our modified algorithm:

$F = L(v)$, in the *current* graph
(this makes the algorithm truly online).

Low-level Data Structure

IsRawGraph g G L M I : a SL predicate that asserts the ownership of a data structure at address g , with logical model G, L, M, I .

- G : a mathematical graph
- L : levels, as a map $\text{vertex} \rightarrow \mathbb{Z}$
- M : marks, as a map $\text{vertex} \rightarrow \text{mark}$
- I : horizontal incoming edges, a map $\text{vertex} \rightarrow \text{set vertex}$

Functional Invariant

$\text{Inv } G L I$: a pure proposition that relates G with L and I .

$\text{Inv } G L I :=$

$$\left\{ \begin{array}{ll} \text{acyclicity:} & \forall x. \quad x \not\rightarrow_G^+ x \\ \text{positive levels:} & \forall x. \quad L(x) \geq 1 \\ \text{pseudo-topological levels:} & \forall x y. \quad x \rightarrow_G y \implies L(x) \leq L(y) \\ \text{incoming edges:} & \forall x y. \quad x \in I(y) \iff x \rightarrow_G y \wedge L(x) = L(y) \\ \text{replete levels:} & \forall x. \quad \text{enough_edges_below } G L x \end{array} \right.$$

$\text{enough_edges_below } G L x :=$

$$|\text{coacc_edges_at_level } G L k x| \geq k \quad \text{where } k = L(x) - 1$$

$\text{coacc_edges_at_level } G L k x :=$

$$\{(y, z) \mid y \rightarrow_G z \rightarrow_G^* x \wedge L(y) = L(z) = k\}$$

Potential and Advertised Cost (formally)

Potential of an edge (u, v) : $\max_level\ m\ n - L(u)$.

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

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

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

$$\max_level\ m\ n := \min(\lceil (2m)^{1/2} \rceil, \lceil (\frac{3}{2}n)^{2/3} \rceil) + 1$$

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

Proof methodology

Specification excerpt for the backward traversal:

$\exists a b. 0 \leq a \wedge \forall F g v w \dots$

$\{(a \cdot F + b) \star \dots\}$ backward_search $F g v w \{\lambda res. \dots\}$

Well-behaved credits inference with integer credits

Credit synthesis requires solving heap entailments of the form:

$$\$(?c) \star \$potential \Vdash \$cost_1 \star \dots \star \$cost_n \star ?F$$

(functions returning credits makes solving these even more tricky)

Integer credits would allow turning these into:

$$\$(?c) \star \$potential \star \$(-cost_1) \star \dots \star \$(-cost_n) \Vdash ?F$$

Is this useful?...

Automation for processing synthesized cost expressions

Credit synthesis produces in the end goals of the form:

$$\begin{aligned} \exists f. \quad & \dots f \dots \\ \exists a b. \quad & \dots a \dots b \dots \end{aligned}$$

Where “...” usually:

- are complex expressions unwieldy to handle manually;
- contain symbolic expressions (abstract cost functions or constants).