Certifying Algorithms

R.M. McConnel, K. Mehlhorn, S. Näher, P. Schweitzer

Presented By: Joe Simons
Certifying Algorithm

Definition:

Produces a certificate along with output makes it "easy" to verify the output
Compute $y = f(x)$

input $x$, output $y$

Program for $f$

(might have bugs)
Compute $y = f(x)$

input $x$, output $y$

(might have bugs)

Have to trust algorithm and implementation
Compute $y = f(x)$

input $x$, output $y$, certificate $w$

Certifying program for $f$

Checker $C$

accept $y$

reject

(might have bugs)
Compute $y = f(x)$

input $x$, output $y$, certificate $w$

Certifying program for $f$

Checker $C$

accept $y$

reject

(might have bugs)

Only have to trust Checker
Compute $y = f(x)$

input $x$, output $y$, certificate $w$

Certifying program for $f$

Checker $C$

accept $y$

reject

(might have bugs)

Only have to trust Checker simple to compute
Motivation

Library of Efficient Data structures and Algorithms (LEDA)
Motivation

Library of Efficient Data structures and Algorithms (LEDA)

1991: added Hopcroft and Tarjan planarity testing algorithm
Motivation

Library of Efficient Data structures and Algorithms (LEDА)

1991: added Hopcroft and Tarjan planarity testing algorithm

1993: discover planar graph that LEDА declares non-planar
Motivation

Library of Efficient Data structures and Algorithms (LEDAs)

1991: added Hopcroft and Tarjan planarity testing algorithm

1993: discover planar graph that LEDA declares non-planar

several days of debugging to fix implementation
Examples of program failures

In 2000, Gutwenger and Mutzel implement the 1973 Hopcroft and Tarjan algorithm for deciding triconnectivity in graphs.
Examples of program failures

In 2000, Gutwenger and Mutzel implement the 1973 Hopcroft and Tarjan algorithm for deciding triconnectivity in graphs.

Find that algorithm reports false positives and provide correction.
Examples of program failures

Mathematica 4.2 fails to solve

a small integer linear program
Examples of program failures

Pentium chip with bug in hardware for division
Examples of program failures

Linear program solvers

Geometric software

etc.
Complex questions like

“is this graph planar?”

deserve more than a yes-no answer
Complex questions like

“is this graph planar?”

deserve more than a yes-no answer

a program should justify its answers in a way
that is easily checked by the user of the program
if graph is planar, should give proof
   (e.g. LEDA gave a planar drawing)

if not planar, program should also prove it
Kuratowski’s theorem

A graph is planar iff it does not contain a subdivision of $K_5$ or $K_{3,3}$ as a subgraph.
Kuratowski’s theorem

A graph is planar iff it does not contain a subdivision of $K_5$ or $K_{3,3}$ as a subgraph.

Subdivision: edges may be split by new vertices.
Kuratowski’s theorem

A graph is planar iff it does not contain a subdivision of $K_5$ or $K_{3,3}$ as a subgraph.
Kuratowski’s theorem

a graph is planar iff it does not contain a subdivision of $K_5$ or $K_{3,3}$ as a subgraph
User doesn’t care that

a Kuratowski subgraph

always exists in a non-planar graph
User doesn’t care that

a Kuratowski subgraph

always exists in a non-planar graph

Only that

subdivision of $K_5$ or $K_{3,3}$

$\Rightarrow$ non-planar graph
Simple proof (sketch):

Assume $K_5$ or $K_{3,3}$ is planar.

$\Rightarrow$ Contradiction with Euler’s formula

$$V - E + F = 2$$
Non-planarity certificate:

sequence of edges in Kuratowski subgraph

Subdivision of $K_5$:

10 disjoint paths

sharing 5 endpoints
Non-planarity certificate:

sequence of edges in Kuratowski subgraph

Subdivision of $K_5$:

10 disjoint paths

sharing 5 endpoints

Easy for checker to verify
Non-planarity certificate:
sequence of edges in Kuratowski subgraph

Subdivision of $K_5$:
10 disjoint paths
sharing 5 endpoints

Easy for checker to verify

Subdivision $K_{3,3}$ similarly easy
Planarity certificate:

obvious choice: planar drawing

but hard to verify in linear time
Planarity certificate:

obvious choice: planar drawing

but hard to verify in linear time

better choice:

combinatorial planar embedding

(similar to doubly-connected edge list)
Planarity certificate:

obvious choice: planar drawing

but hard to verify in linear time

better choice:

combinatorial planar embedding

(similar to doubly-connected edge list)

Checker verifies against Euler’s formula
Testing if a graph is bipartite

2-coloring

odd cycle
Testing if a graph is bipartite

2-coloring

Certifies bipartness

Odd cycle

Certifies non-bipartness
Testing if a graph is bipartite

2-coloring

Certifies bipartness

Odd cycle

Certifies non-bipartness
Testing if a graph is bipartite

2-coloring

Certifies bipartness

odd cycle

Certifies non-bipartness
Testing if a graph is bipartite

2-coloring

Certifies bipartness

Odd cycle

Certifies non-bipartness
Testing if a graph is bipartite

2-coloring

Certifies bipartness

odd cycle

Certifies non-bipartness
Testing if a graph is bipartite

2-coloring

Certifies bipartness

odd cycle

Certifies non-bipartness
Checking certificate

Don’t need to know

\[ \text{non-bipartite} \Rightarrow \text{odd cycle} \]

only that

\[ \text{odd cycle} \Rightarrow \text{non-bipartite} \]

Simpler to \textit{verify} an odd cycle

than to \textit{search} for one
Algorithm

1) Choose node; color it red; declare unfinished.
Algorithm

2) While unfinished nodes:
   declare a node finished
   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:
   
   declare a node finished
   
   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:
   declare a node finished
   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:

   declare a node finished

   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:
   declare a node finished
   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:

declare a node finished

declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:

   declare a node finished

   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:
   
   declare a node finished

   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:
   
   declare a node finished

   declare neighbors unfinished and color them
Algorithm

2) While unfinished nodes:
   declare a node finished
   declare neighbors unfinished and color them
Algorithm

3) Iterate and Check edges

if endpoints are same color,

follow arrows to find odd cycle
If Bipartite

Algorithm returns 2-coloring

Checker verifies coloring is valid

If Not Bipartite

Algorithm returns odd cycle

Checker verifies cycle is valid
GCD\((a, b)\)

if \( b = 0 \), return \( a \);

return GCD\((b, a \mod b)\);

Euclid’s algorithm is non-certifying
**EGCD**\((a, b)\)

if \(b = 0\), return \((a, 1, 0)\);

let \((g, y, x) = \text{EGCD}(b, a \mod b)\);

return \((g, x, y - \lfloor a/b \rfloor)\)

Extended Euclid’s algorithm is certifying
EGCD\((a, b)\)

if \(b = 0\), return \((a, 1, 0)\);

let \((g, y, x) = \text{EGCD}(b, a \mod b)\);

return \((g, x, y - \lfloor a/b \rfloor)\)

returns \(g = \text{GCD}(a, b)\)

also computes \(x\) and \(y\) such that \(g = ax + by\)
Lemma

Let $a, b, g \in \mathbb{N}$, and $a, b$ not both zero.

Let $g = ax + yb$ for integers $x, y$

If $g$ divides $a$ and $b$, then $g = \gcd(a, b)$.
Proof: If $g$ divides $a$ and $b$, then $g = \gcd(a, b)$. 
Proof: If $g$ divides $a$ and $b$, then $g = \gcd(a, b)$.

Let $d$ divide $a$ and $b$, and $g = ax + yb$
Proof: If $g$ divides $a$ and $b$, then $g = \gcd(a, b)$.

Let $d$ divide $a$ and $b$, and $g = ax + yb$

$$g = xd \frac{a}{d} + yd \frac{b}{d} = d \left( x \frac{a}{d} + y \frac{b}{d} \right)$$
Proof: If $g$ divides $a$ and $b$, then $g = \gcd(a, b)$.

Let $d$ divide $a$ and $b$, and $g = ax + yb$

$$g = xd\frac{a}{d} + yd\frac{b}{d} = d \left( x\frac{a}{d} + y\frac{b}{d} \right)$$

$\Rightarrow d$ divides $g$

$\Rightarrow \gcd(a, b)$ divides $g$
Proof: If $g$ divides $a$ and $b$, then $g = \gcd(a, b)$.

Let $d$ divide $a$ and $b$, and $g = ax + yb$

$$g = xd \frac{a}{d} + yd \frac{b}{d} = d \left( x \frac{a}{d} + y \frac{b}{d} \right)$$

$\Rightarrow d$ divides $g$

$\Rightarrow \gcd(a, b)$ divides $g$

$g$ divides $a, b \Rightarrow g$ divides $\gcd(a, b)$
Proof: If $g$ divides $a$ and $b$, then $g = \gcd(a, b)$.

Let $d$ divide $a$ and $b$, and $g = ax + yb$

$$g = xd\frac{a}{d} + yd\frac{b}{d} = d \left( x\frac{a}{d} + y\frac{b}{d} \right)$$

$\Rightarrow d$ divides $g$

$\Rightarrow \gcd(a, b)$ divides $g$

$g$ divides $a, b \Rightarrow g$ divides $\gcd(a, b)$
Algorithm:

Input: \( a, b \)

Output: \( g, x, y \)

Checker:

Verify that:

\[ g = ax + by \]

\( g \) divides \( a \) and \( b \)
Strongly certifying:

For any input $x$

Provides:
Strongly certifying:

For any input \( x \)

Provides:

Certificate that \( x \) is invalid
Strongly certifying:

For any input $x$

Provides:

Certificate that $x$ is invalid

or

Certificate that $y$ is valid output for $x$
Strongly certifying example:

Algorithm that 5-colors a planar graph

Provides:
Strongly certifying example:

Algorithm that 5-colors a planar graph

Provides:

Certificate that graph was non-planar
Strongly certifying example:

Algorithm that 5-colors a planar graph

Provides:

Certificate that graph was non-planar
or
Certificate of 5-coloring
Strongly certifying example:

Algorithm that 5-colors a planar graph

Provides:

Certificate that graph was non-planar
or
Certificate of 5-coloring

(but may 5-color a non-planar graph)
If $\leq 5$ vertices, return 5-coloring

if $m > 3n - 6$, return non-planar
If $\leq 5$ vertices, return 5-coloring

if $m > 3n - 6$, return non-planar

if $\deg(v) \leq 4$, recurse on $G - v$
If \( \leq 5 \) vertices, return 5-coloring

if \( m > 3n - 6 \), return non-planar

if \( \deg(v) \leq 4 \), recurse on \( G - v \)
If \( \leq 5 \) vertices, return 5-coloring

if \( m > 3n - 6 \), return non-planar

if \( \deg(v) \leq 4 \), recurse on \( G - v \)
If $\leq 5$ vertices, return 5-coloring

if $m > 3n - 6$, return non-planar

if $\deg(v) \leq 4$, recurse on $G - v$

(extend coloring to $G$)
If \( \leq 5 \) vertices, return 5-coloring

if \( m > 3n - 6 \), return non-planar

if \( \deg(v) \leq 4 \), recurse on \( G - v \)
If $\leq 5$ vertices, return 5-coloring

if $m > 3n - 6$, return non-planar

if $\deg(v) \leq 4$, recurse on $G - v$

if $\deg(v) = 5$

neighbors form $K_5$
If \( \leq 5 \) vertices, return 5-coloring

if \( m > 3n - 6 \), return non-planar

if \( \text{deg}(v) \leq 4 \), recurse on \( G - v \)

if \( \text{deg}(v) = 5 \)

neighbors form \( K_5 \)

or 2 non-adjacent neighbors
If \( \leq 5 \) vertices, return 5-coloring

if \( m > 3n - 6 \), return non-planar

if \( \deg(v) \leq 4 \), recurse on \( G - v \)

if \( \deg(v) = 5 \)

neighbors form \( K_5 \)

or 2 non-adjacent neighbors

remove \( v \) and "identify" neighbors
remove $v$ and “identify” neighbors
remove $v$ and “identify” neighbors

recurse
remove $v$ and “identify” neighbors

and continue coloring

recurse
Certifying: (not strongly)

For any input $x$

Provides certificate that:
Certifying: (not strongly)

For any input $x$

Provides certificate that:

$x$ is invalid
Certifying: (not strongly)

For any input $x$

Provides certificate that:

$x$ is invalid

or

$y$ is valid output for $x$
Certifying: (not strongly)

For any input $x$

Provides certificate that:

$x$ is invalid

or

$y$ is valid output for $x$

(but can’t tell which)
Certifying (not strongly) example:

Binary Search

Either:
Certifying (not strongly) example:

Binary Search

Either:

Array wasn’t sorted
Certifying (not strongly) example:

Binary Search

Either:

Array wasn’t sorted

or

Pointer to where element should be stored
Certifying (not strongly) example:

Binary Search

Either:

Array wasn’t sorted

or

Pointer to where element should be stored

(but can’t tell which)
Weakly Certifying:

For any input $x$
Weakly Certifying:

For any input $x$

Provides certificate that

$y$ is valid output for $x$
Weakly Certifying:

For any input $x$

Provides certificate that

$y$ is valid output for $x$

or

$x$ is invalid and may not halt
Weakly Certifying:

For any input $x$

Provides certificate that

$y$ is valid output for $x$

or

$x$ is invalid and may not halt

(and can’t tell which)
Weakly Certifying Example:

Naive randomized SAT solver

(for satisfiable formulas)
Weakly Certifying Example:

Naive randomized SAT solver
(for satisfiable formulas)

Guesses random assignments
Weakly Certifying Example:

Naive randomized SAT solver
(for satisfiable formulas)

Guesses random assignments

Finds valid assignment
Weakly Certifying Example:

Naive randomized SAT solver
(for satisfiable formulas)

Guesses random assignments

Finds valid assignment

or

Never Halts
Theorem:

Strongly Certifying for $f(x)$

Diagram:

- Input: $x$
- Normal: $y_0$
- Checker: $y$
- Weak: $w_1$, $w_2$
- Output: "$x$ valid?" (yes/no)

"$x$ invalid"
Apply theorem to SAT solver?

Only valid input is a satisfiable formula

Validating input = Solving the problem

Error in theorem, or example?
Efficiency

Certifying Algorithm $P$ is *efficient* if

$$\text{running time of } P \text{ and Checker } C \leq \text{time of non-certifying algorithm}$$
Efficiency

Certifying Algorithm $P$ is efficient if
running time of $P$ and Checker $C$
\[ \leq \text{time of non-certifying algorithm} \]

Examples so far were efficient
Efficiency

Certifying Algorithm $P$ is efficient if
running time of $P$ and Checker $C$

$\leq$ time of non-certifying algorithm

Examples so far were efficient

No known efficient for graph 3-connectivity

best: $O(n^2)$ vs $(n)$
Certifying Algorithm

Produces a *certificate* along with output

makes it “easy” to verify the output
Certifying Algorithm

Produces a *certificate* along with output

makes it "easy" to verify the output

Checkability  Simplicity
Checkability

Witness Property:

Predicate

$\mathcal{W}(x, y, w)$
Checkability

Witness Property:

Predicate \( \mathcal{W}(x, y, w) \)

True if \( w \) proves either

\( x \) is invalid

or

\( y \) is correct output
Checkability formulations
Checkability formulations

\( \mathcal{W}(x, y, w) \) has linear time algorithm
Checkability formulations

\[ W(x, y, w) \] has linear time algorithm

has a simple logical structure
Checkability formulations

\( W(x, y, w) \) has linear time algorithm

has a simple logical structure

Correctness of program deciding \( W \)
Checkability formulations

\( W(x, y, w) \) has linear time algorithm

has a simple logical structure

Correctness of program deciding \( W \)

is obvious
Checkability formulations

$\mathcal{W}(x, y, w)$ has linear time algorithm

has a simple logical structure

Correctness of program deciding $\mathcal{W}$

is obvious

or can be formally verified
Simplicity

Simple proof of witness property
Simplicity

Simple proof of witness property

e.g. binary search find(z) returns null
Simplicity

Simple proof of witness property

e.g. binary search find(ż) returns null

successor and predecessor prove:
Simplicity

Simple proof of witness property

e.g. binary search $\text{find}(z)$ returns $\text{null}$

successor and predecessor prove:

array wasn’t sorted

or $z$ not in Array
Checkers should be so simple that
question of their correctness is not an issue

We may be able to write them in a language
with formally defined semantics
and prove them correct

(Semantics in C, C++ and Java are informal)
Advantages of Certifying Algorithms

Instance correctness

We know the output is correct for input $x$
Advantages of Certifying Algorithms

Instance correctness

We know the output is correct for input $x$

Testing all inputs

Checker verifies for every input

Not just small test cases
Advantages of Certifying Algorithms

Confinement of error

Error caught in failing module

Not spread throughout program
Advantages of Certifying Algorithms

Confinement of error

Error caught in failing module

Not spread throughout program

Verified Checkers

Can be formally verified

Efficient checker algorithm

→ less efficient language ok
Advantages of Certifying Algorithms

Black-Box Programs

Don’t need source of Program

Just source of Checker
Advantages of Certifying Algorithms

Black-Box Programs

Don’t need source of Program

Just source of Checker

Tamper-Proof

If output is wrong, Checker rejects

If Checker accepts, then output was correct

User doesn’t know / care if program was altered
Open Problems:

3-connectivity of graphs

Arrangements of algebraic curves

Shortest paths w/ obstacles

LEDA and GCAL

+ many more
Theorem:

Every deterministic program with trivial pre-condition

Has an efficient, strongly certifying counterpart
Theorem:

Every deterministic program with trivial pre-condition

Has an efficient, strongly certifying counterpart

(Assuming we have a proof in a formal system that

\[ \forall x \ P \text{ halts and gives correct output} \)
Theorem:

Every deterministic program with trivial pre-condition

Has an efficient, strongly certifying counterpart

(Assuming we have a proof in a formal system that

\[ \forall x \ P \text{ halts and gives correct output} \]

Certificate that \( x \) is invalid

or

Certificate that \( y \) is valid output for \( x \)
Theorem:

Every deterministic program with trivial pre-condition

Has an efficient, strongly certifying counterpart

(Assuming we have a proof in a formal system that

\[ \forall x \ P \text{ halts and gives correct output} \]

Certificate that \( x \) is invalid

or

Certificate that \( y \) is valid output for \( x \)
Theorem:

Every deterministic program

Has an efficient, weakly certifying counterpart
Theorem:

Every deterministic program

Has an efficient, weakly certifying counterpart

(Assuming we have a proof in a formal system that
on valid input, program gives correct output)