

set up non-certifying and certifying planarity demo. Let the non-certifying demo run during introduction

Certifying Algorithms

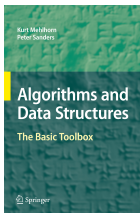
Algorithmics meets Software Engineering

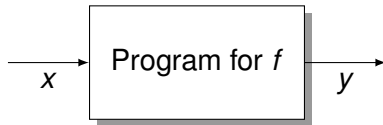
Kurt Mehlhorn



mpi max planck institut
informatik

- problem definition and certifying algorithms
- examples of certifying algorithms
 - testing bipartiteness
 - matchings in graphs
 - planarity testing
 - convex hulls
 - further examples
- advantages of certifying algorithms
- universality
- formal verification and certifying algorithms
- summary





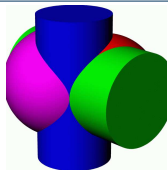
- A user feeds x to the program, the program returns y .
- How can the user be sure that, indeed,

$$y = f(x)?$$

The user has no way to know.

Warning Examples

- LEDA 2.0 planarity test was incorrect
- Rhino3d (a CAD systems) fails to compute correct intersection of two cylinders and two spheres
- CPLEX (a linear programming solver) fails on benchmark problem *etamacro*.
- Mathematica 4.2 (a mathematics systems) fails to solve a small integer linear program



```
In[1] := ConstrainedMin[ x , {x==1,x==2} , {x} ]
```

```
Out[1] = {2, {x->2}}
```

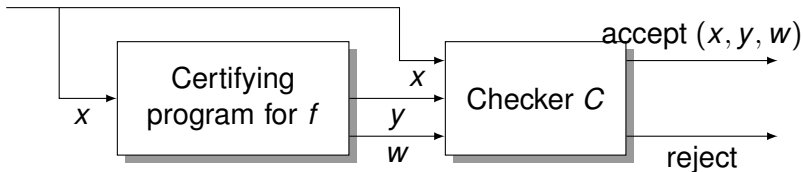
```
In[1] := ConstrainedMax[ x , {x==1,x==2} , {x} ]
```

```
ConstrainedMax::"lpsub": "The problem is unbounded."
```

```
Out[2] = {Infinity, {x -> Indeterminate}}
```

Programs must justify (prove) their answers
in a way
that is easily checked by their users.

A Certifying Program for a Function f



- On input x , a **certifying program** returns the function value y and a certificate (witness) w
- w proves $y = f(x)$ even to a dummy,
- and there is a simple program C , the **checker**, that verifies the validity of the proof.

A First Example: Testing Bipartiteness of Graphs

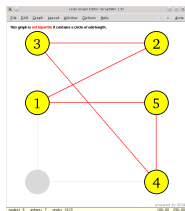
A graph is **bipartite** if its vertices can be colored black and white such that the endpoints of each edge have distinct colors.



Conventional algorithm outputs YES or NO

Certifying Algorithm outputs

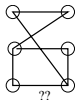
- a two-coloring in the YES-case
- an odd cycle in the NO-case



Remark: simple modification of the standard algorithm suffices

A First Example: Testing Bipartiteness of Graphs

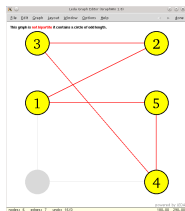
A graph is **bipartite** if its vertices can be colored black and white such that the endpoints of each edge have distinct colors.



Conventional algorithm outputs YES or NO

Certifying Algorithm outputs

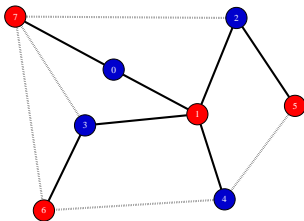
- a two-coloring in the YES-case
- an odd cycle in the NO-case



Remark: simple modification of the standard algorithm suffices

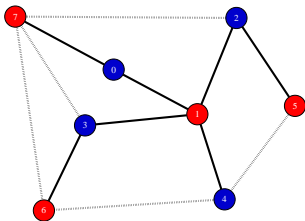
Bipartite Graphs: An Algorithm

- construct a spanning tree of G
- use it to color the vertices with colors **red** and **blue**
- check for all non-tree edges: do endpoints have distinct colors?
- if yes, the graph is bipartite and the coloring proves it
- if no, declare the graph non-bipartite: Let $e = \{u, v\}$ be a non-tree edge with equal colored endpoints
 - e together with the tree path from u to v is an odd cycle
 - tree path has even length since u and v have the same color



Bipartite Graphs: An Algorithm

- construct a spanning tree of G
- use it to color the vertices with colors **red** and **blue**
- check for all non-tree edges: do endpoints have distinct colors?
- if yes, the graph is bipartite and the coloring proves it
- if no, declare the graph non-bipartite: Let $e = \{u, v\}$ be a **non-tree edge with equal colored endpoints**
 - e together with the tree path from u to v is an odd cycle
 - tree path has even length since u and v have the same color



- **I do not claim** that I invented the concept; it is an old concept
 - al-Kwarizmi: multiplication
 - extended Euclid: gcd
 - primal-dual algorithms in combinatorial optimization
 - Blum et al.: Programs that check their work
- **I do claim** that Näher and I were the first (1995) to adopt the concept as the design principle for a large library project:
LEDA
(Library of Efficient Data Types and Algorithms)
- Kratsch/McConnell/M/Spinrad (SODA 2003) coin name
- McConnell/M/Näher/Schweitzer (2010): 80 page survey

How I got interested?

- till '83: only theoretical work in algorithms and complexity
- '83 – '89: participation in a project on VLSI design:
implementation work proceeds very slowly
- since '89: LEDA, library of efficient data types and algorithms
- many implementations incorrect
- '95: adopt exact computation paradigm (computational geometry) and certifying algorithms as design principles
- '95 – '99: make textbook algs certifying, reimplementing of library, LEDA book
- since '00: additional certifying algorithms
- '10: 80 page survey paper
- since '12: formal verification of checkers



Planarity Testing
Maximum Cardinality Matchings
Further Examples



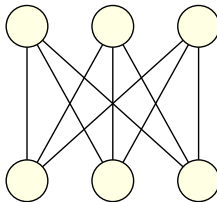
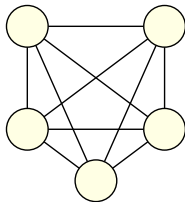
Example II: Planarity Testing

- Given a graph G , decide whether it is planar
- Tarjan (76): planarity can be tested in linear time
- A story and a demo
- Combinatorial planar embedding is a witness for planarity

Chiba et al (85): planar embedding of a planar G in linear time

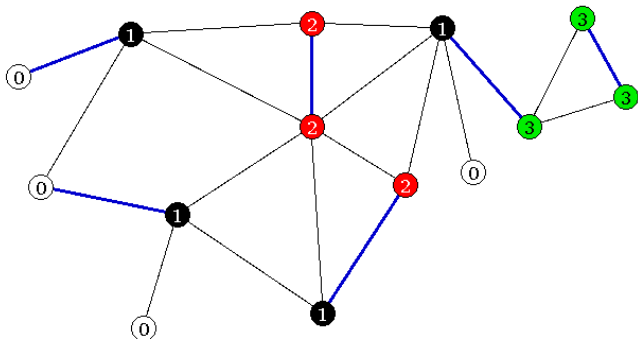
- Kuratowski subgraph is a witness for non-planarity

Hundack/M/Näher (97): Kuratowski subgraph of non-planar G in linear time, LEDAbook, Chapter 9



Example III: Maximum Cardinality Matchings

- A matching M is a set of edges no two of which share an endpoint



- The blue edges form a matching of maximum cardinality; this is non-obvious as two vertices are unmatched.
- A conventional algorithm outputs the set of blue edges.

Edmonds' Theorem: Let M be a matching in a graph G and let ℓ be a labelling of the vertices with non-negative integers such that for each edge $e = (u, v)$ either $\ell(u) = \ell(v) \geq 2$ or $1 \in \{\ell(u), \ell(v)\}$. Then

$$|M| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

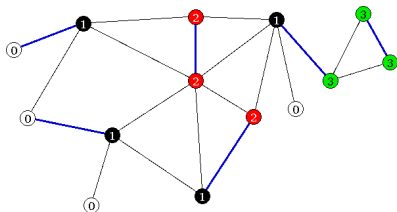
where n_i is the number of vertices labelled i .

Maximum Cardinality Matching: A Certifying Alg

Edmonds' Theorem: Let M be a matching in a graph G and let ℓ be a labelling of the vertices with non-negative integers such that for each edge $e = (u, v)$ either $\ell(u) = \ell(v) \geq 2$ or $1 \in \{\ell(u), \ell(v)\}$. Then

$$|M| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

where n_i is the number of vertices labelled i .



- $|M| = 6$
- $n_1 = 4, n_2 = 3, n_3 = 3.$
- no matching has more than

$$4 + \lfloor 3/2 \rfloor + \lfloor 3/2 \rfloor = 6$$

edges.

Maximum Cardinality Matching: A Certifying Alg

Edmonds' Theorem: Let M be a matching in a graph G and let ℓ be a labelling of the vertices with non-negative integers such that for each edge $e = (u, v)$ either $\ell(u) = \ell(v) \geq 2$ or $1 \in \{\ell(u), \ell(v)\}$. Then

$$|M| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

where n_i is the number of vertices labelled i .

- Let M_1 be the edges in M having at least one endpoint labelled 1 and, for $i \geq 2$, let M_i be the edges in M having both endpoints labelled i .
- $M = M_1 \cup M_2 \cup M_3 \cup \dots$
- $|M_1| \leq n_1$ and $|M_i| \leq n_i/2$ for $i \geq 2$.



Further Examples

- biconnectivity, strong connectivity, flows, . . . ,
- Convex Hulls
- Schmidt, Mehlhorn/Neumann/Schmidt: Three-Connectivity of Graphs
- Georgiadis/Tarjan: Dominators in Digraphs
- Wang: Arrangements of Algebraic Curves
- Mehlhorn/Sagraloff/Wang: Root Isolation for Real Polynomials
- Althaus/Dumitriu: Certifying feasibility and objective value of linear programs
- Hauenstein/Sottile: alphaCertified: certifying solutions to polynomial systems
- Cook et al: Traveling Salesman Tours
- Cheung/Gleixner/Steffy: Verifying Integer Programming Results



The Advantages of Certifying Algorithms

- Certifying algs can be tested on
 - **any** input
 - and not just on inputs for which the result is known.
- Certifying algorithms are reliable:
 - Either give the correct answer
 - or notice that they have erred \Rightarrow confinement of error
- Computation as a service
 - There is no need to understand the program, understanding the witness property and the checking program suffices.
 - One may even keep the program secret and only publish the checker



The Advantages of Certifying Algorithms

- Certifying algs can be tested on
 - **any** input
 - and not just on inputs for which the result is known.
- Certifying algorithms are reliable:
 - Either give the correct answer
 - or notice that they have erred \Rightarrow confinement of error
- Computation as a service
 - There is no need to understand the program, understanding the witness property and the checking program suffices.
 - One may even keep the program secret and only publish the checker



The Advantages of Certifying Algorithms

- Certifying algs can be tested on
 - **any** input
 - and not just on inputs for which the result is known.
- Certifying algorithms are reliable:
 - Either give the correct answer
 - or notice that they have erred \Rightarrow confinement of error
- Computation as a service
 - There is no need to understand the program, understanding the witness property and the checking program suffices.
 - One may even keep the program secret and only publish the checker



- General techniques
 - Linear programming duality
 - Characterization theorems
 - Program composition
- Probabilistic programs and checkers
- Reactive Systems (data structures)
- does apply to problems in NP (and beyond), e.g., SAT
 - output a satisfying assignment of satisfiable inputs
 - output a resolution proof for unsatisfiability otherwise

- Does every problem have a certifying algorithm? Can every program be converted into a certifying one?
 - I know 100+ certifying algorithms, see survey by McConnell/M/Näher/Schweitzer (CS Review), in particular, all text-book algorithms can be made certifying
 - most programs in LEDA are certifying, and
 - Thm: Every deterministic program can be made certifying without asymptotic loss of efficiency
- (at least in principle)

I still believe that the opposite should be true; however, for every formalization that I tried, I could prove the theorem.

- Does every problem have a certifying algorithm? Can every program be converted into a certifying one?
 - I know 100+ certifying algorithms, see survey by McConnell/M/Näher/Schweitzer (CS Review), in particular, all text-book algorithms can be made certifying
 - most programs in LEDA are certifying, and
 - Thm: Every deterministic program can be made certifying without asymptotic loss of efficiency
- (at least in principle)

I still believe that the opposite should be true; however, for every formalization that I tried, I could prove the theorem.

- Does every problem have a certifying algorithm? Can every program be converted into a certifying one?
- I know 100+ certifying algorithms, see survey by McConnell/M/Näher/Schweitzer (CS Review), in particular, all text-book algorithms can be made certifying
- most programs in LEDA are certifying, and
- Thm: Every deterministic program can be made certifying without asymptotic loss of efficiency

(at least in principle)

I still believe that the opposite should be true; however, for every formalization that I tried, I could prove the theorem.

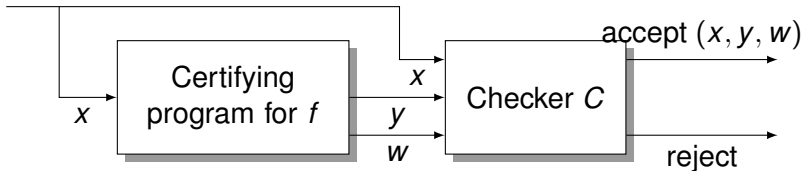
- Does every problem have a certifying algorithm? Can every program be converted into a certifying one?
- I know 100+ certifying algorithms, see survey by McConnell/M/Näher/Schweitzer (CS Review), in particular, all text-book algorithms can be made certifying
- most programs in LEDA are certifying, and
- **Thm: Every deterministic program can be made certifying without asymptotic loss of efficiency**
(at least in principle)

I still believe that the opposite should be true; however, for every formalization that I tried, I could prove the theorem.

- Does every problem have a certifying algorithm? Can every program be converted into a certifying one?
- I know 100+ certifying algorithms, see survey by McConnell/M/Näher/Schweitzer (CS Review), in particular, all text-book algorithms can be made certifying
- most programs in LEDA are certifying, and
- **Thm: Every deterministic program can be made certifying without asymptotic loss of efficiency**
(at least in principle)

I still believe that the opposite should be true; however, for every formalization that I tried, I could prove the theorem.

A Certifying Program for a Function f



- On input x , a **certifying program** returns the function value y and a certificate (witness) w
- w proves $y = f(x)$ even to a dummy,
- and there is a simple program C , the **checker**, that verifies the validity of the proof.

The Maximum Cardinality Matching Checker

Edmonds' Theorem: Let M be a matching in a graph $G = (V, E)$ and let $\ell : V \rightarrow \mathbb{N}$ such that for each edge $e = (u, v)$ of G either $\ell(u) = \ell(v) \geq 2$ or $1 \in \{\ell(u), \ell(v)\}$. Then

$$|M| \leq n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor,$$

where n_i is the number of vertices labelled i .

The Checker Program has input G , M , and ℓ :

- checks that $M \subseteq E$,
- checks that M is a matching,
- checks that ℓ satisfies the hypothesis of the theorem, and
- checks that $|M| = n_1 + \sum_{i \geq 2} \lfloor n_i/2 \rfloor$

set $c[v] = 0$ for all $v \in V$;

for all $e = (u, v) \in M$: increment $c[u]$ and $c[v]$;

if some counter reaches 2, M is not a matching.



Who Checks the Checker?

How can we be sure that the checker programs are correct?

My answer up to 2011: Because they are so simple.

Because we can prove their correctness in a formal system

Isabelle/HOL

Nipkow/Paulson

- formal mathematics
- proof are machine-checked
- only kernel needs to be trusted



Who Checks the Checker?

How can we be sure that the checker programs are correct?

My answer up to 2011: Because they are so simple.

Because we can prove their correctness in a formal system

Isabelle/HOL

Nipkow/Paulson

- formal mathematics
- proof are machine-checked
- only kernel needs to be trusted



What do we Formally Verify and How?

- Edmonds' theorem
- Checker always halts and either rejects or accepts.
- Checker accepts a triple (G, M, ℓ) iff it satisfies the assumptions of Edmonds' theorem.

- we prove Edmonds' theorem in Isabelle
- we translate checkers from C to I-Monads with AutoCorres (NICTA)
- I-Monads is a programming language defined in Isabelle
- we prove items 2 and 3 for the resulting I-Monads program in Isabelle
- since NICTA-tools are verified, this verifies the C-code of the checker
- verification revealed that one of the checkers in LEDA was incomplete



What do we Formally Verify and How?

- Edmonds' theorem
- Checker always halts and either rejects or accepts.
- Checker accepts a triple (G, M, ℓ) iff it satisfies the assumptions of Edmonds' theorem.

- we prove Edmonds' theorem in Isabelle
- we translate checkers from C to I-Monads with AutoCorres (NICTA)
- I-Monads is a programming language defined in Isabelle
- we prove items 2 and 3 for the resulting I-Monads program in Isabelle
- since NICTA-tools are verified, this verifies the C-code of the checker
- verification revealed that one of the checkers in LEDA was incomplete



Formal Instance Correctness

If a formally verified checker accepts a triple (x, y, w) ,
we have a formal proof that y is the correct output for input x .

- a high level of trust (only Isabelle kernel needs to be trusted)
- a way to build large libraries of trusted algorithms

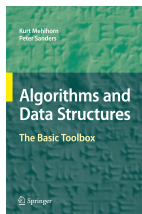
Alkassar/Böhme/M/Rizkallah: Verification of Certifying Computations,

JAR 2014

Noshinski/Rizkallah/M: Verification of Certifying Computations through AutoCorres and Simpl,
NASA Formal Methods Symposium 2014

Summary

- **Only certifying algs are good algs**
- Certifying algs have many advantages over standard algs:
 - every run is a test
 - notice when they erred
 - can be relied on without knowing code
 - are a way to computation as a service
- Formal verification of checkers and formal proof of witness property are feasible
- Most programs in the LEDA system are certifying.



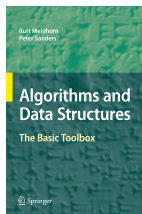
**When you design your next
algorithm,
make it certifying.**



Summary

- **Only certifying algs are good algs**
- Certifying algs have many advantages over standard algs:
 - every run is a test
 - notice when they erred
 - can be relied on without knowing code
 - are a way to computation as a service
- Formal verification of checkers and formal proof of witness property are feasible
- Most programs in the LEDA system are certifying.

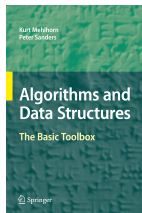
When you design your next
algorithm,
make it certifying.



Summary

- **Only certifying algs are good algs**
- Certifying algs have many advantages over standard algs:
 - every run is a test
 - notice when they erred
 - can be relied on without knowing code
 - are a way to computation as a service
- Formal verification of checkers and formal proof of witness property are feasible
- Most programs in the LEDA system are certifying.

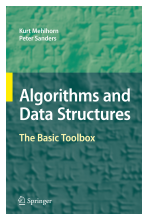
When you design your next
algorithm,
make it certifying.



Summary

- **Only certifying algs are good algs**
- Certifying algs have many advantages over standard algs:
 - every run is a test
 - notice when they erred
 - can be relied on without knowing code
 - are a way to computation as a service
- Formal verification of checkers and formal proof of witness property are feasible
- Most programs in the LEDA system are certifying.

When you design your next
algorithm,
make it certifying.



Summary

- **Only certifying algs are good algs**
- Certifying algs have many advantages over standard algs:
 - every run is a test
 - notice when they erred
 - can be relied on without knowing code
 - are a way to computation as a service
- Formal verification of checkers and formal proof of witness property are feasible
- Most programs in the LEDA system are certifying.

**When you design your next
algorithm,
make it certifying.**

