Set-Up

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

Certifying Algorithms

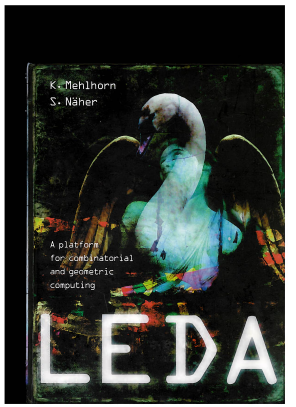
Algorithmics meets Software Engineering

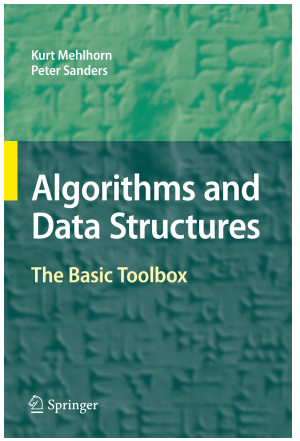
Kurt Mehlhorn

max planck institut
informatik

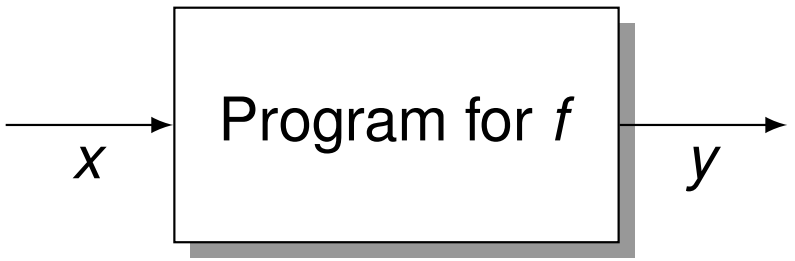
Outline of Talk

- 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





The Problem

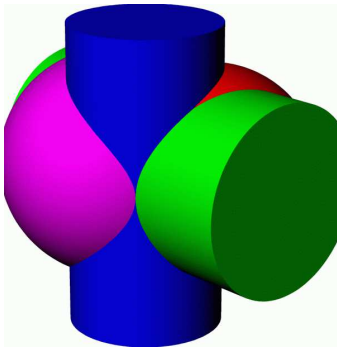


- A user feeds x to the program, the program returns y.
- How can the user be sure that, indeed,

$$y = f(x)?$$

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

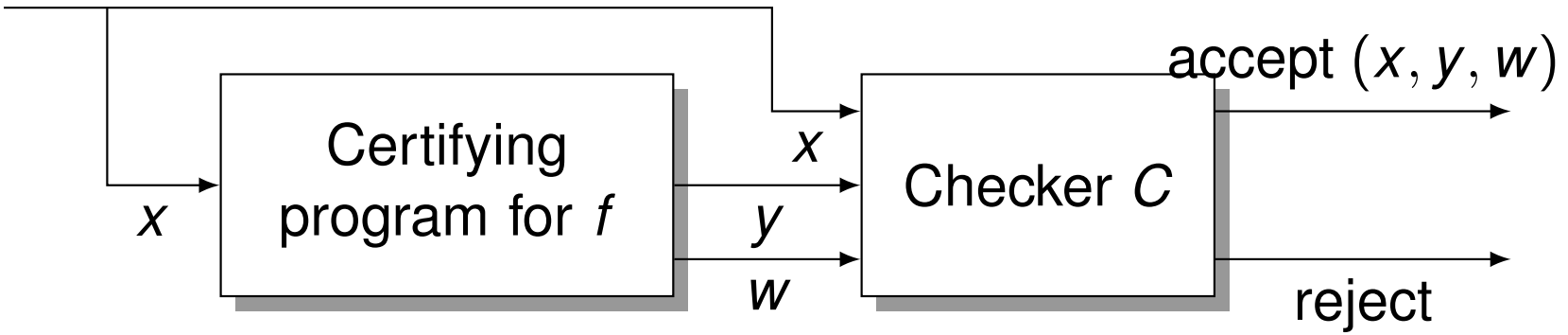
```
In[1] := ConstrainedMin[ x , {x==1,x==2} , {x} ]
Out[1] = {2, {x->2}}

In[1] := ConstrainedMax[ x , {x==1,x==2} , {x} ]
ConstrainedMax::"lpsub": "The problem is unbounded."
Out[2] = {Infinity, {x -> Indeterminate}}
```

The Proposal

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

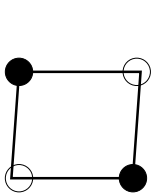
A Certifying Program for a Function f

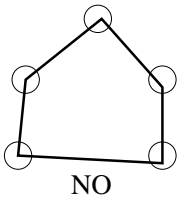


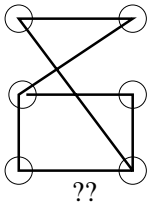
- On input x, a certifying program returns
  the function value y and a certificate (witness) w
- w proves $y = f(x)$ even to a dummy,
- and there is a simple program C, the checker, that verifies the validity of the proof.

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.


YES

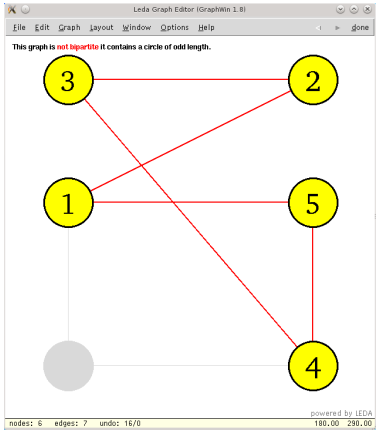

NO


??

Conventional algorithm outputs YES or NO

Certifying Algorithm outputs

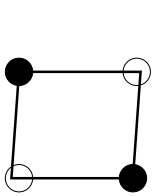
- a two-coloring in the YES-case
- an odd cycle in the NO-case

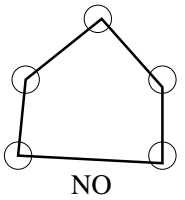


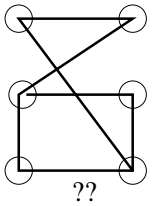
Remark: simple modification of the standard algorithm suffices

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.



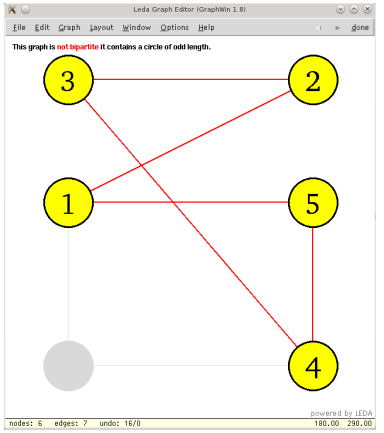




Conventional algorithm outputs YES or NO

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

History

- 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, reimplementation of library, LEDA book
- since ’00: additional certifying algorithms
- ’10: 80 page survey paper
- since ’12: formal verification of checkers

Examples

Planarity Testing
Maximum Cardinality Matchings
Further Examples

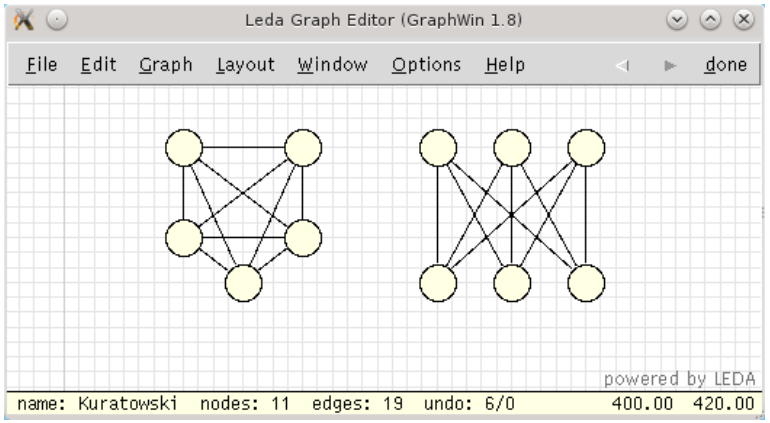
Example II: Planarity Testing

- 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

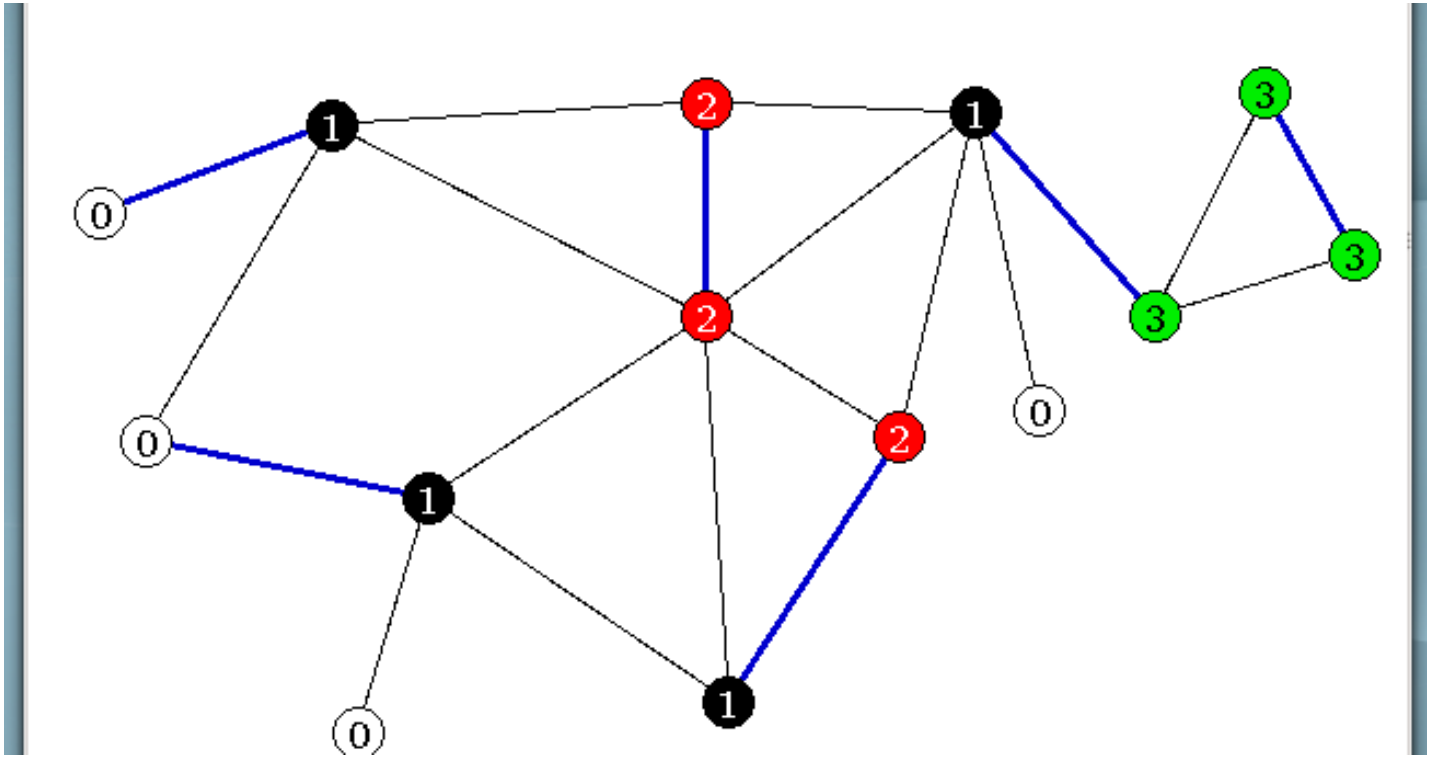
- Kuratowski subgraph is a witness for non-planarity

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.

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

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

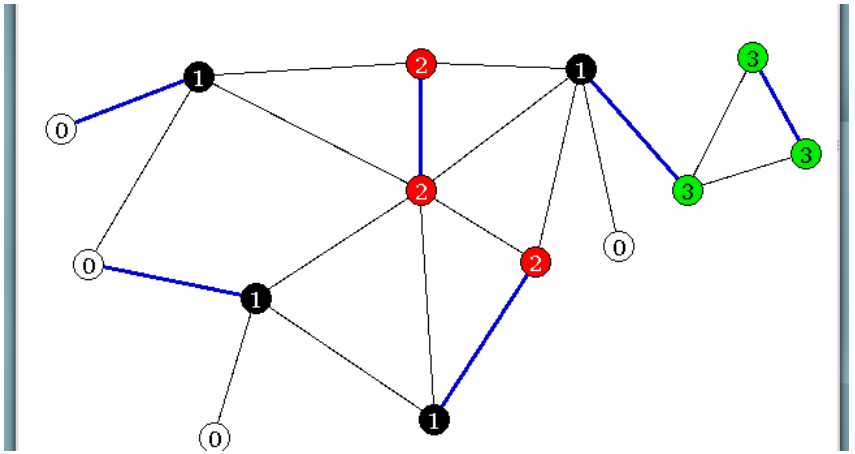
where n_i is the number of vertices labelled i.

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.



- $|M| = 6$
- $m_1 = 4$, $m_2 = 3$, $m_3 = 3$.
- 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.
- $M = M_1 \cup M_2 \cup M_3 \cup \ldots$
- $|M_1| \leq n_1$ 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
- Dictionaries

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

Odds and Ends

- 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
  - ouput a resolution proof for unsatisfiability otherwise

Universality

- 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.

Universality

- 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.

Universality

- 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.

Universality

- 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.

Universality

- 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.

The Maximum Cardinality Matching Checker

Edmonds' Theorem: Let M be a matching in a graph $G = (V, E)$ and let $\ell : V \to \mathbb{N}$ 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

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

```
definition disjoint-edges :: (α, β) pre-graph ⇒ β ⇒ β ⇒ bool where
  disjoint-edges G e1 e2 = (
    start G e1 ≠ start G e2 ∧ start G e1 ≠ target G e2 ∧
    target G e1 ≠ start G e2 ∧ target G e1 ≠ target G e2)

definition matching :: (α, β) pre-graph ⇒ β set ⇒ bool where
  matching G M = (
    M ⊆ edges G ∧
    (∀e1 ∈ M. ∀e2 ∈ M. e1 ≠ e2 ⟶ disjoint-edges G e1 e2))

definition edge-as-set :: β ⇒ α set where
  edge-as-set e ≡ {tail G e, head G e}

lemma matching_disjointness:
  assumes matching G M
  assumes e1 ∈ M   assumes e2 ∈ M  assumes e1 ≠ e2
  shows  edge-as-set e1 ∩ edge-as-set e2 = {}
  using assms
  by (auto simp add: edge-as-set_def disjoint-edges_def matching_def)
```

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 is 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 is 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 Verification: Summary

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

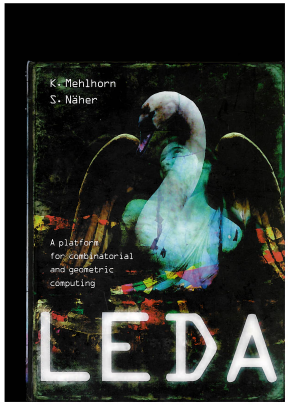
Alkassar/Böhme/M/Rizkallah: Verification of Certifying Computations, JAR 2014

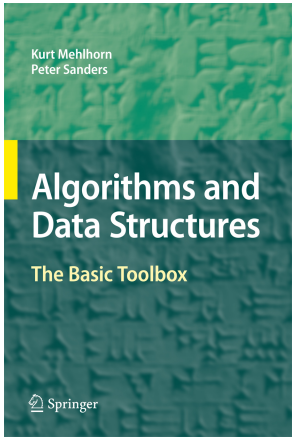
Noshinski/Rizkallah/M: Verification of Certifying Computations through AutoCorres and Simpl, NASA Formal Methods Symposium 2014

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.

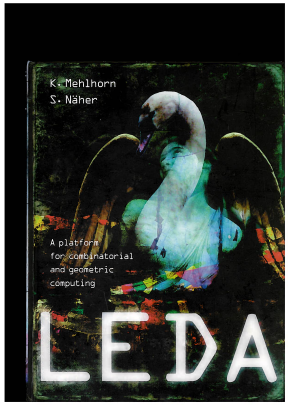
When you design your next algorithm, make it certifying.

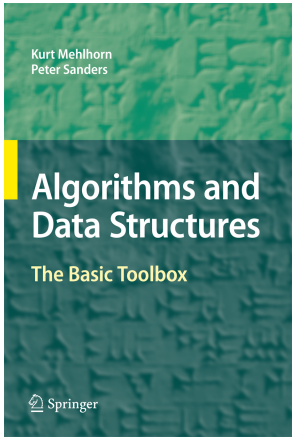




Summary

- **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.

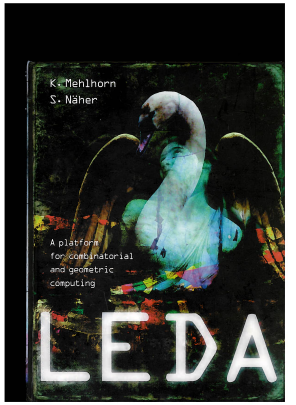


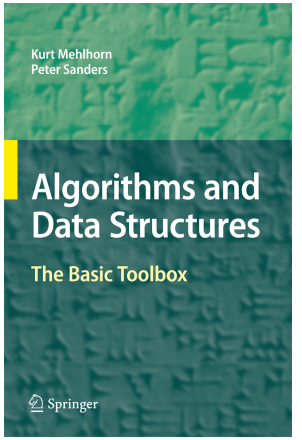


When you design your next algorithm, make it certifying.

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.

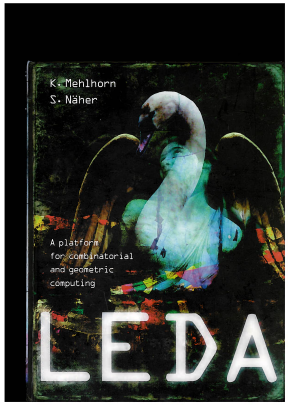


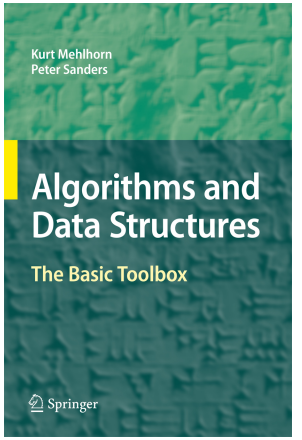


When you design your next algorithm, make it certifying.

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.

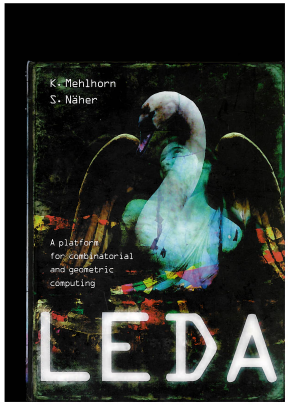


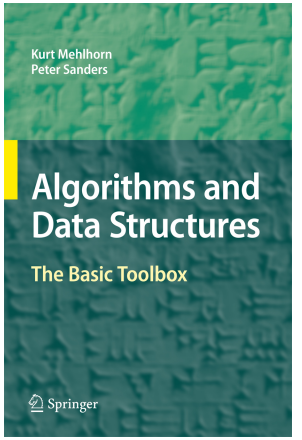


When you design your next algorithm,
make it certifying.

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.

Certifying Algs for 3-Connectivity

Kurt Mehlhorn
Adrian Neumann
Jens Schmidt

max planck institut
informatik

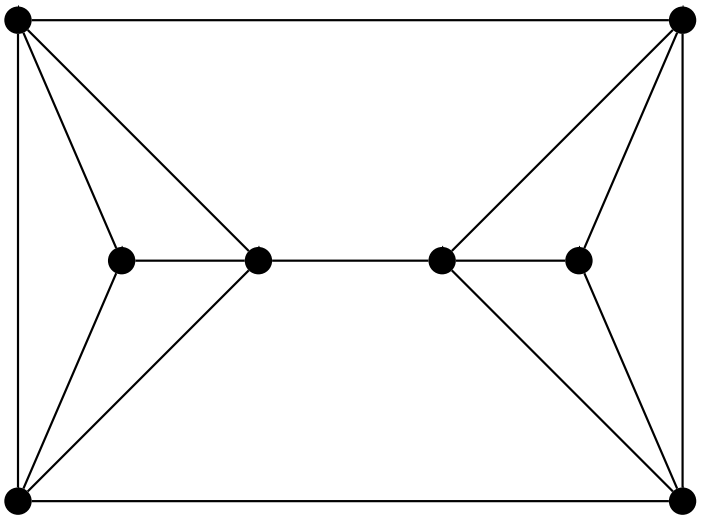
11. November 2014

k-Connectivity

A (multi-)graph is k-edge-connected if removal of any $k-1$ edges does not disconnect it.

A (multi-)graph is k-vertex-connected if removal of any $k-1$ vertices does not disconnect it.

today's talk: certifying algorithms for 3-connectivity



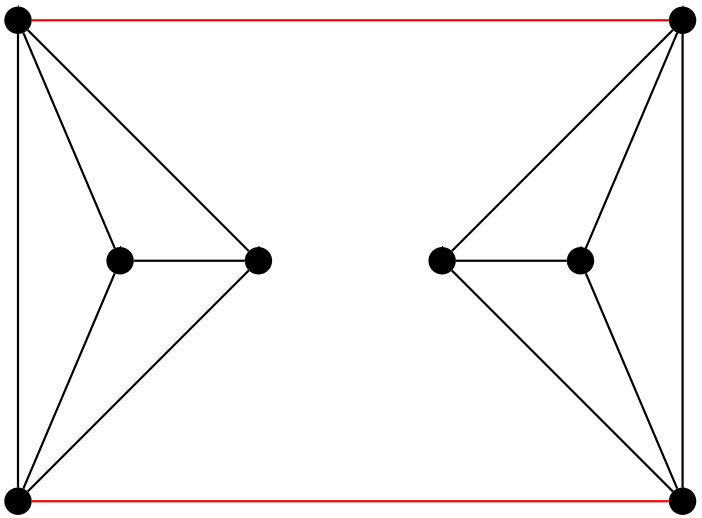
3-edge- and 3-vertex connected

k-Connectivity

A (multi-)graph is k-edge-connected if removal of any $k-1$ edges does not disconnect it.

A (multi-)graph is k-vertex-connected if removal of any $k-1$ vertices does not disconnect it.

today's talk: certifying algorithms for 3-connectivity



2-edge-connected, but not 3-edge-connected

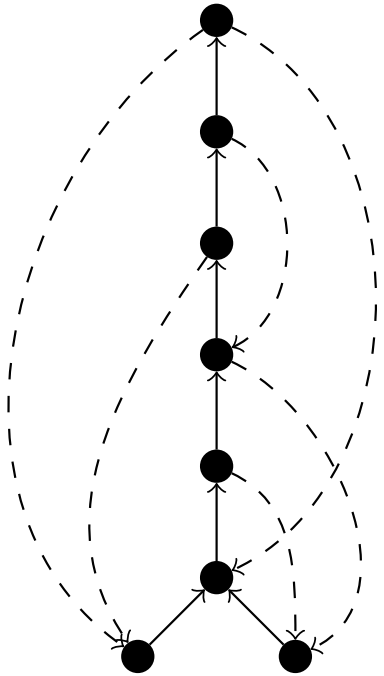
2-vertex-connected, but not 3-vertex-connected

Sources

- Kurt Mehlhorn, Adrian Neumann, Jens M. Schmidt: Certifying 3-Edge-Connectivity, available in arxive
- Jens. M. Schmidt: Contractions, Removals and Certifying 3-Connectivity in Linear Time, SIAM Journal on Computing, 2013, 494-535
- N. Linial, L. Lovász, A. Wigderson: Rubber bands, convex embeddings and graph connectivity, Combinatorica, 1988
- R. M. McConnell, K. Mehlhorn, S. Näher, P. Schweitzer: Certifying algorithms, Computer Science Review, 2011
- Alkassar, E., Böhme, S., Mehlhorn, K.,Rizkallah, C.: Verification of certifying computations, Journal of Automated Reasoning, to appear

Chain Decomposition

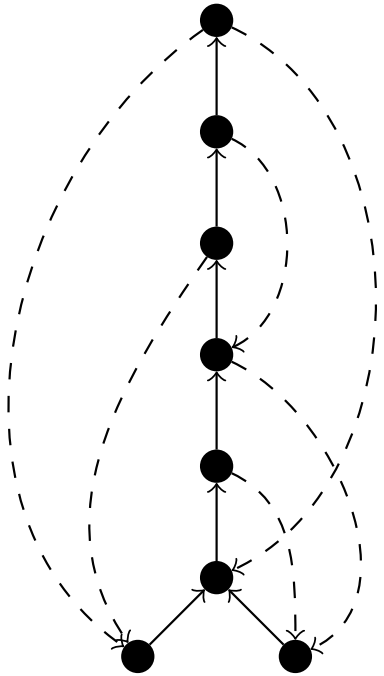
A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

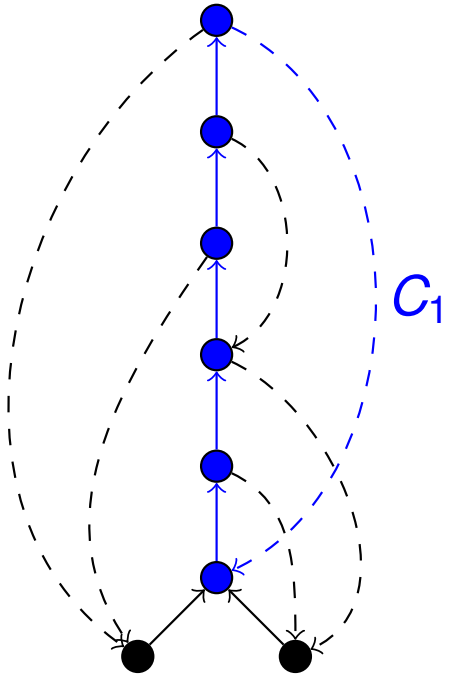
A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

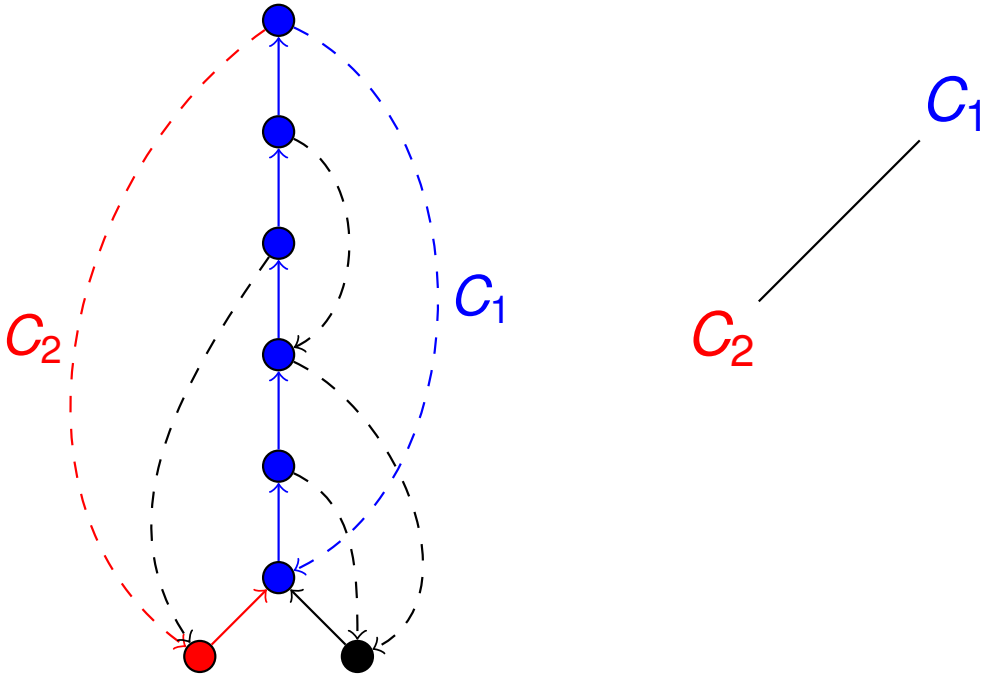


C_1

Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

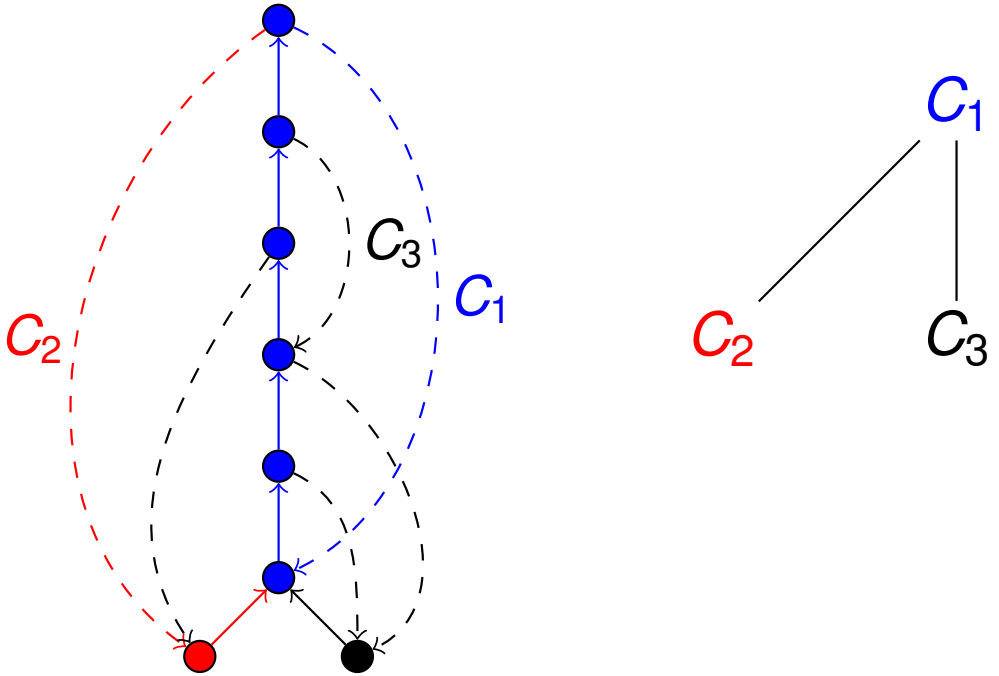
A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

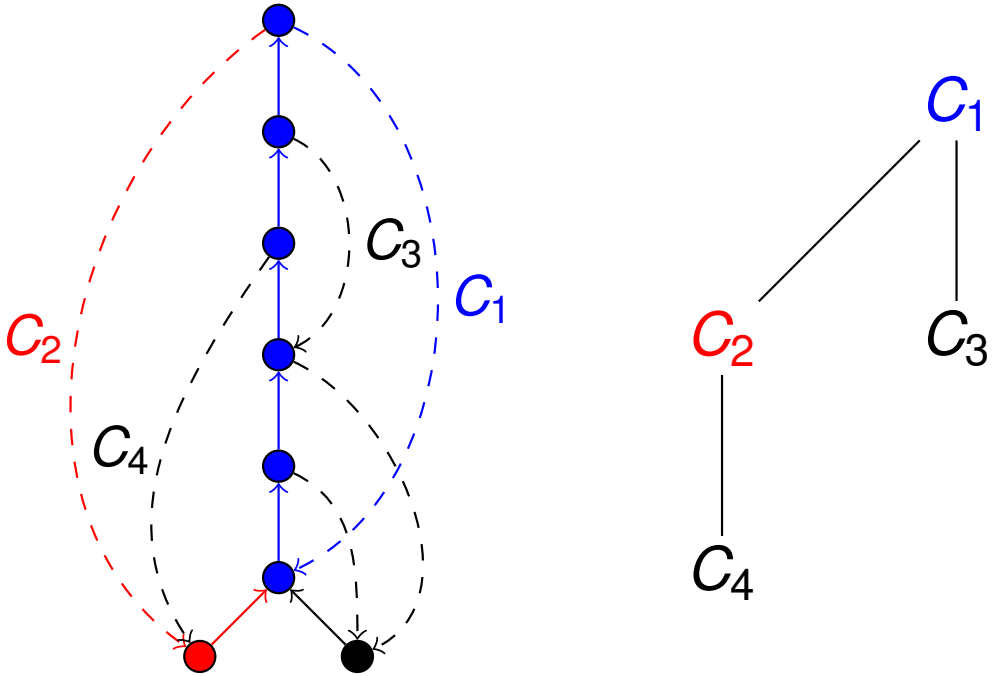
A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

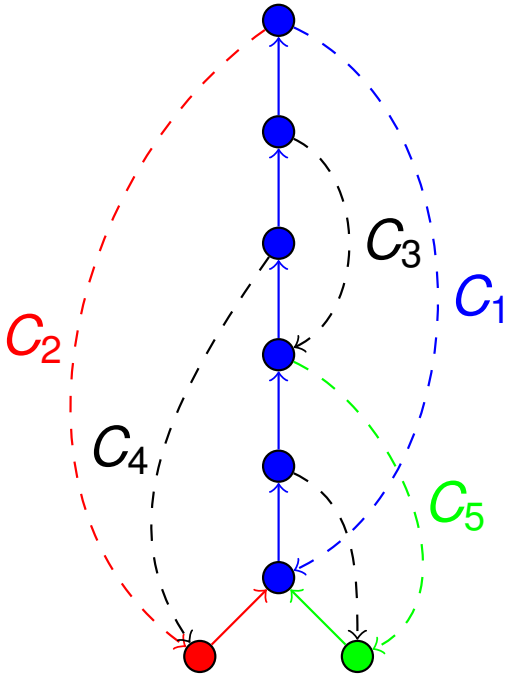
A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

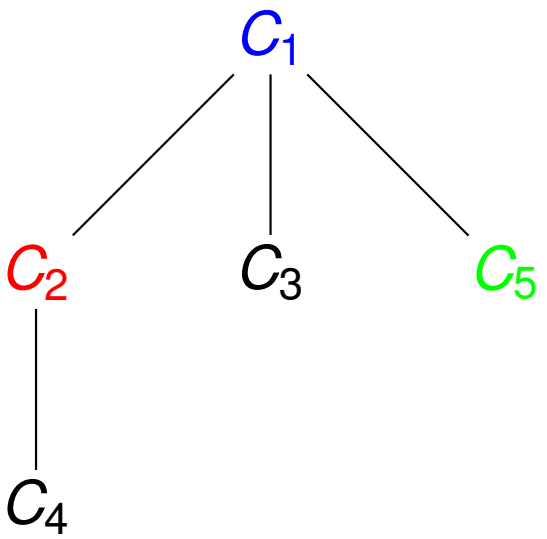


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

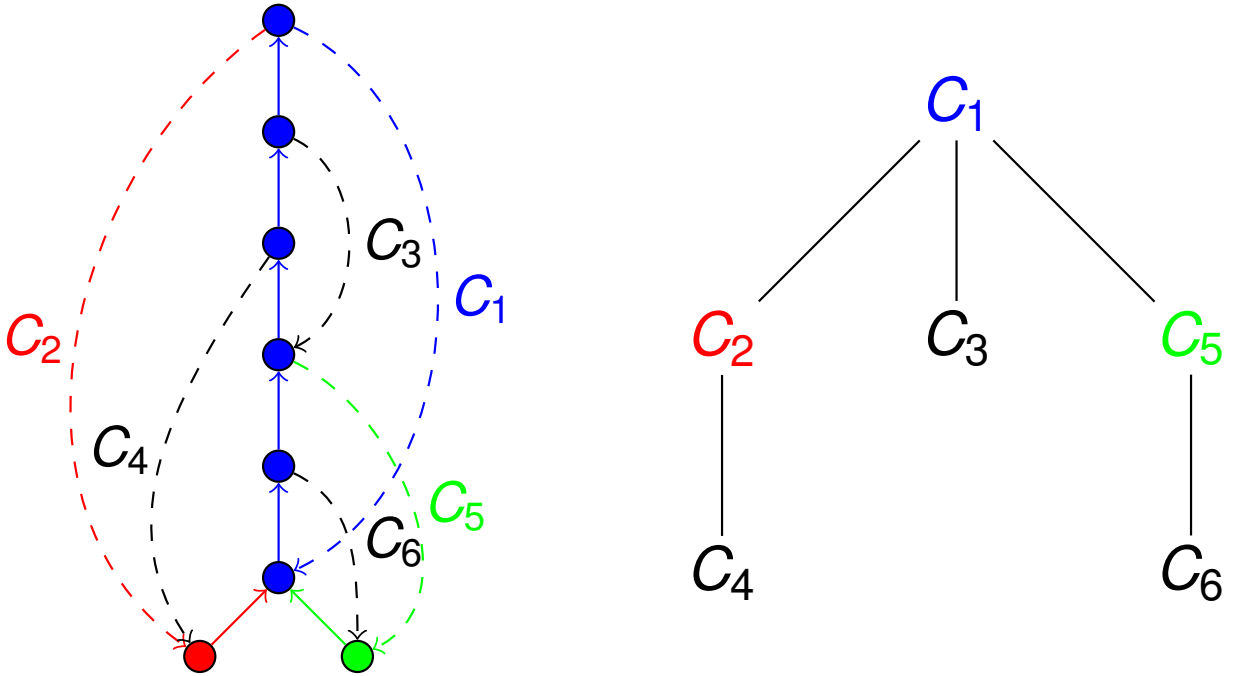




Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Two Edge and Vertex Connectivity

- Two-edge-connectivity:

  No: exhibit a bridge (= a cut consisting of a single edge)
  Yes: exhibit an ear decomposition

- Two-vertex-connectivity:

  No: exhibit a cut-vertex (= a vertex-cut consisting of a single vertex)
  Yes: exhibit an open ear decomposition

All of this is easily done in linear time using the chain-decomposition (Jens Schmidt)

Two Edge and Vertex Connectivity

- Two-edge-connectivity:

  No: exhibit a bridge (= a cut consisting of a single edge)
  Yes: exhibit an ear decomposition

- Two-vertex-connectivity:

  No: exhibit a cut-vertex (= a vertex-cut consisting of a single vertex)
  Yes: exhibit an open ear decomposition

All of this is easily done in linear time using the chain-decomposition (Jens Schmidt)

Three Edge and Vertex Connectivity

3-edge-connectivity and 3-vertex-connectivity are well studied problems. Many linear time solutions known, e.g.:

- 1973: Hopcroft and Tarjan with a correction by Gutwenger and Mutzel
- 1992: Nagamochi and Ibaraki
- 1992: Taoka, Watanabe, and Onaga
- 2007, 2009: Tsin
- Italiano and Galil: reduce edge-connectivity to vertex-connectivity

None of these algorithms is certifying.

They exhibit 2-cuts in the negative case and state 3-connectedness otherwise.

For a user, it is a bit like saying: “I tried hard to find a 2-cut and could not find one. Therefore, I now believe that the graph is 3-connected”.

Three Edge and Vertex Connectivity

3-edge-connectivity and 3-vertex-connectivity are well studied problems. Many linear time solutions known, e.g.:

- 1973: Hopcroft and Tarjan with a correction by Gutwenger and Mutzel
- 1992: Nagamochi and Ibaraki
- 1992: Taoka, Watanabe, and Onaga
- 2007, 2009: Tsin
- Italiano and Galil: reduce edge-connectivity to vertex-connectivity

None of these algorithms is certifying.

They exhibit 2-cuts in the negative case and state 3-connectedness otherwise.

For a user, it is a bit like saying: “I tried hard to find a 2-cut and could not find one. Therefore, I now believe that the graph is 3-connected”.

Three-Edge-Connectedness in Time $O(m^2)$.

For every edge e: certify that $G \setminus e$ is 2-edge-connected.

In order to do better, we need structural insight.

Three-Edge-Connectness

Theorem (Mader, 1978)

A graph is 3-edge-connected iff it can be constructed from a $K_2^3 =$  *by the following three operations*

- *Add an edge between two existing nodes*
- *Split an edge, connect the new node with an old node*

- *Split two edges and connect the two new nodes* 

Theorem (Mehlhorn/Neumann/Schmidt, 2013)

There is a linear time certifying algorithm for 3-edge-connectivity.

It outputs either a 2-edge-cut or a Mader construction sequence.

Three-Edge-Connectness

Theorem (Mader, 1978)

A graph is 3-edge-connected iff it can be constructed from a $K_2^3 =$  *by the following three operations*

- *Add an edge between two existing nodes*
- *Split an edge, connect the new node with an old node*

- *Split two edges and connect the two new nodes* 

Theorem (Mehlhorn/Neumann/Schmidt, 2013)

There is a linear time certifying algorithm for 3-edge-connectivity.

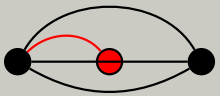
It outputs either a 2-edge-cut or a Mader construction sequence.

Three-Edge-Connectness

Theorem (Mader, 1978)

A graph is 3-edge-connected iff it can be constructed from a $K_2^3 =$  *by the following three operations*

- *Add an edge between two existing nodes*
- *Split an edge, connect the new node with an old node*



- *Split two edges and connect the two new nodes* 

Theorem (Mehlhorn/Neumann/Schmidt, 2013)

There is a linear time certifying algorithm for 3-edge-connectivity.

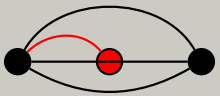
It outputs either a 2-edge-cut or a Mader construction sequence.

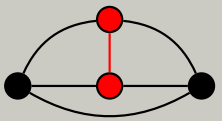
Three-Edge-Connectness

Theorem (Mader, 1978)

A graph is 3-edge-connected iff it can be constructed from a $K_2^3 =$  *by the following three operations*

- *Add an edge between two existing nodes*
- *Split an edge, connect the new node with an old node*



- *Split two edges and connect the two new nodes* 

Theorem (Mehlhorn/Neumann/Schmidt, 2013)

There is a linear time certifying algorithm for 3-edge-connectivity.

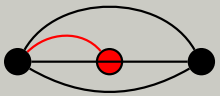
It outputs either a 2-edge-cut or a Mader construction sequence.

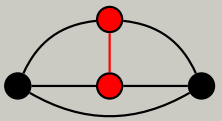
Three-Edge-Connectness

Theorem (Mader, 1978)

A graph is 3-edge-connected iff it can be constructed from a $K_2^3 =$  *by the following three operations*

- *Add an edge between two existing nodes*
- *Split an edge, connect the new node with an old node*



- *Split two edges and connect the two new nodes* 

Theorem (Mehlhorn/Neumann/Schmidt, 2013)

There is a linear time certifying algorithm for 3-edge-connectivity.

It outputs either a 2-edge-cut or a Mader construction sequence.

Content

In This Talk

How to find a construction sequence for a given 3-connected graph in time $O((n+m)\log(n+m))$.

In the paper:

- Correctness proof.
- Linear time algorithm.
- How to verify the certificate.
- A certifying algorithm for 3-edge-connected components.

Content

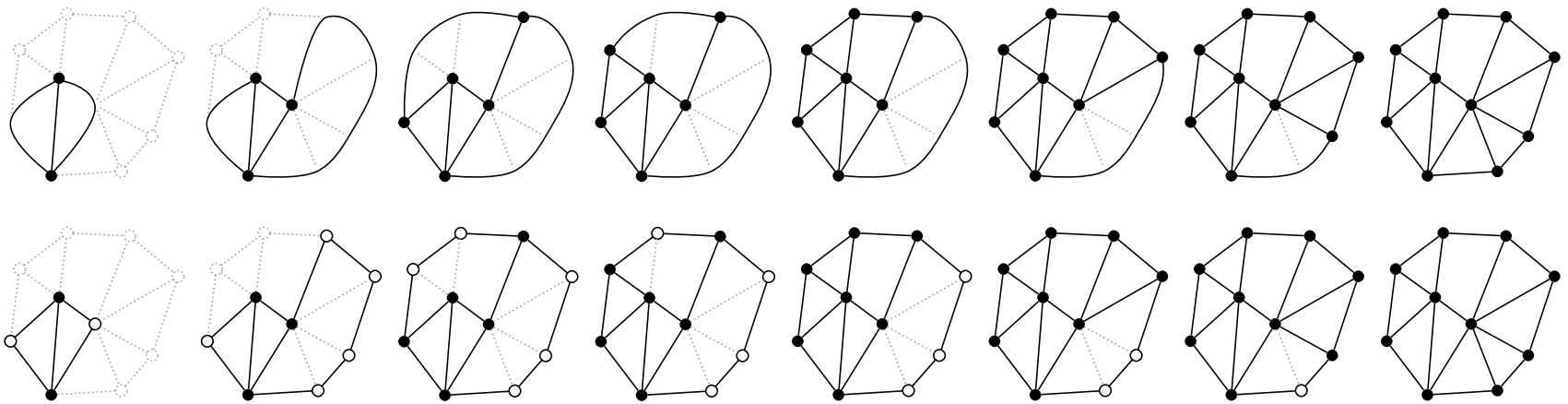
In This Talk

How to find a construction sequence for a given 3-connected graph in time $O((n+m)\log(n+m))$.

In the paper:

- Correctness proof.
- Linear time algorithm.
- How to verify the certificate.
- A certifying algorithm for 3-edge-connected components.

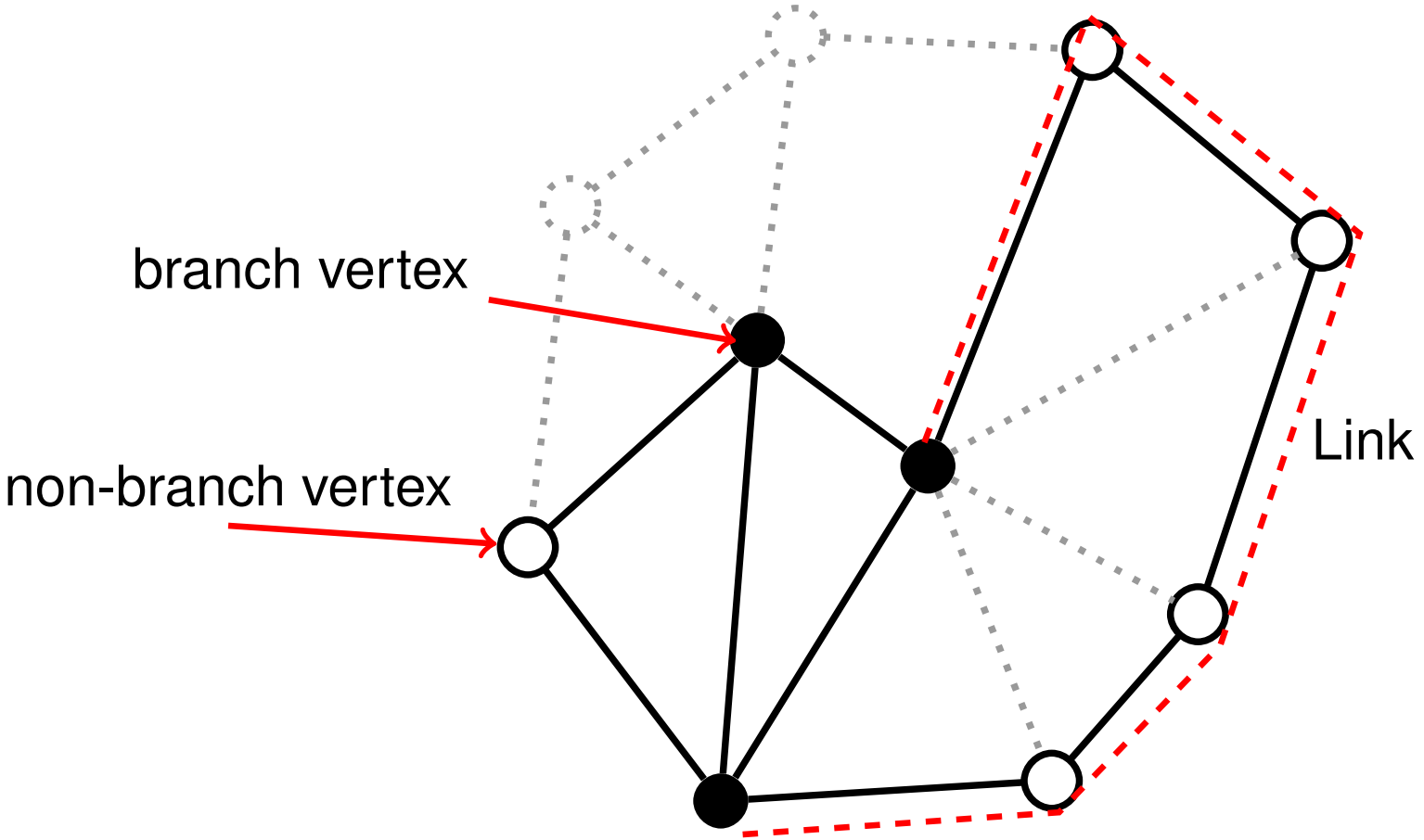
Mader Constructions and Subdivisions



A construction sequence for the graph on the right, once in terms of graphs and once in terms of subdivisions.

It is more convenient to work with subdivisions (a graph whose edges are subdivided by additional vertices), i.e., when we add an edge, we also introduce all vertices that will ever be placed on the edge.

Mader Constructions and Subdivisions

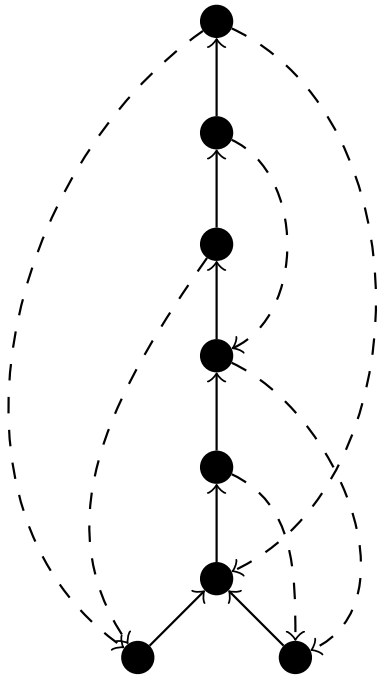


A First Algorithm

1. Find a K_2^3 subdivision. Initialize $G_c = K_2^3$
2. Find a path P in $G - G_c$ from a node u to a node v, such that
   a) at least one of $\{u, v\}$ has degree at least three, or
   b) u and v lie on different links
3. Add P to the current subgraph
4. If the current subgraph is not G, goto 2.

Chain Decomposition

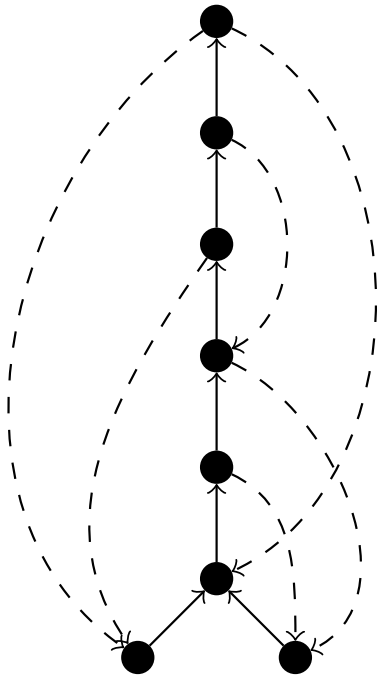
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

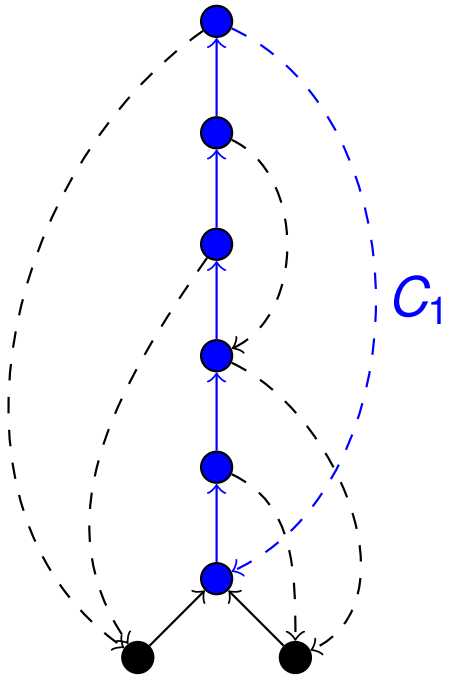
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

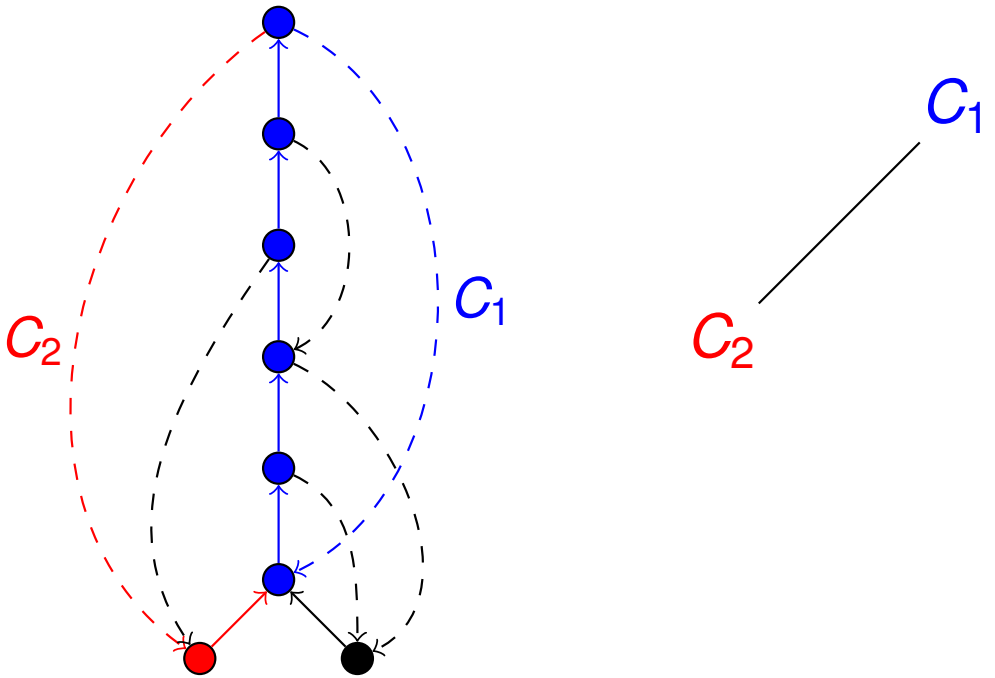


C_1

Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

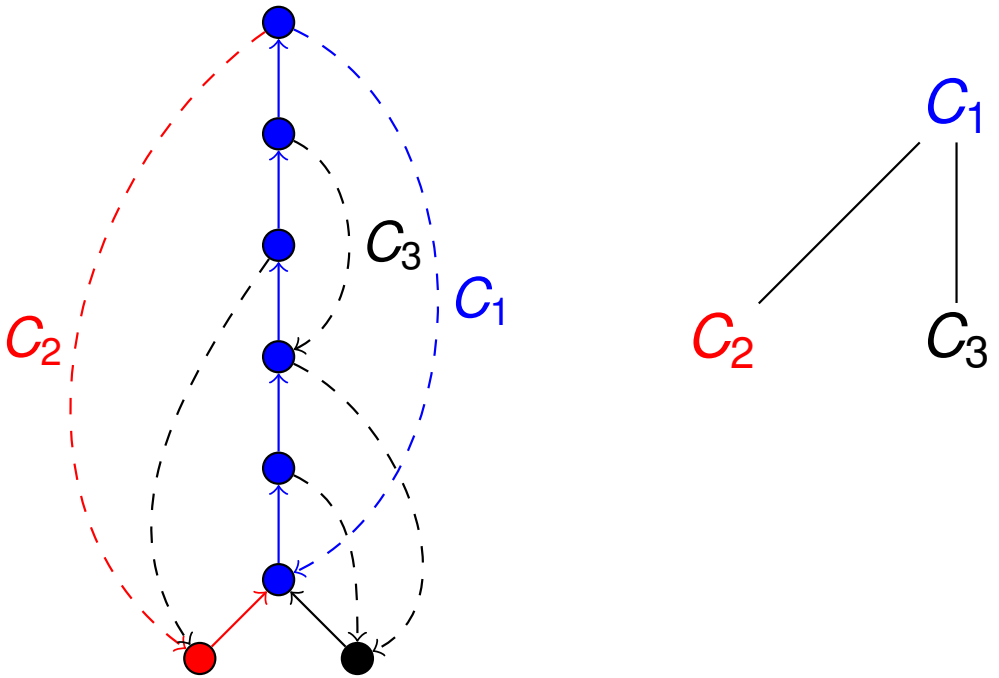
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

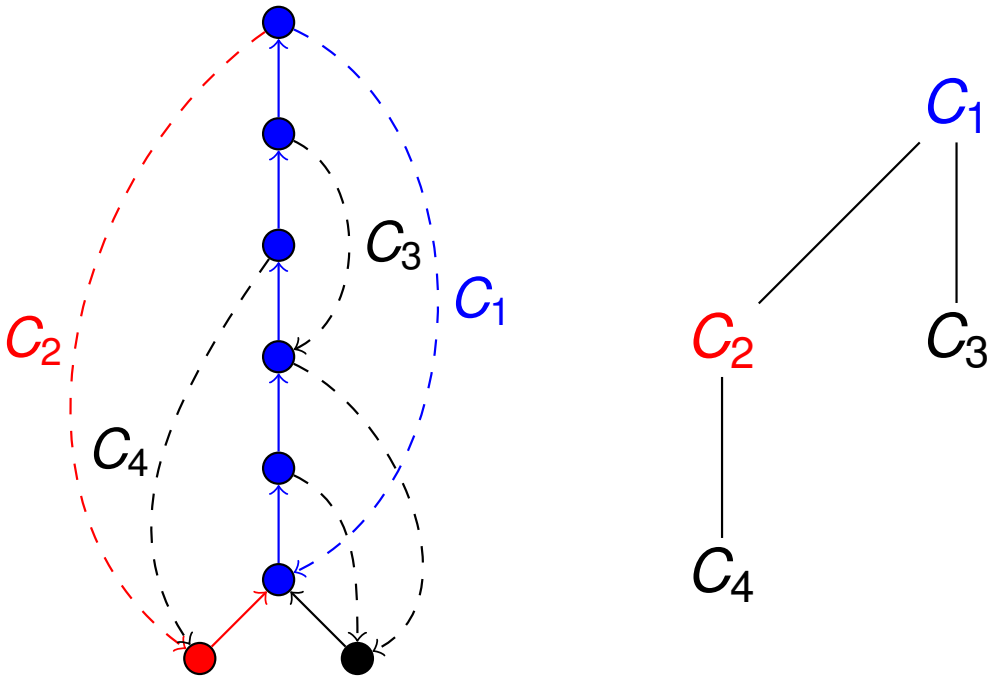
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

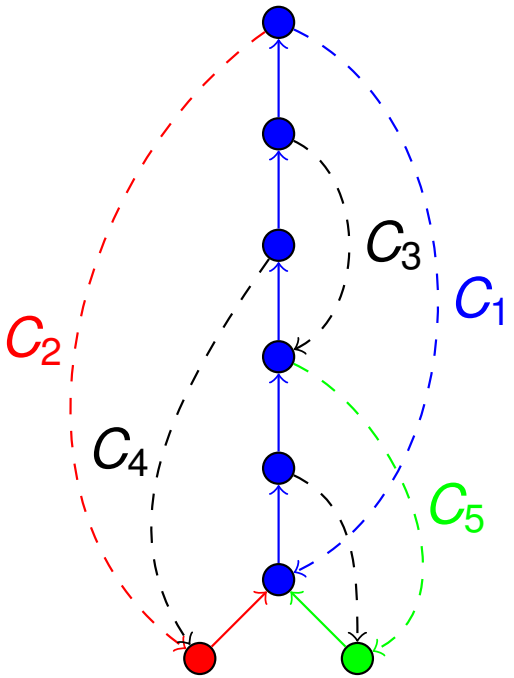
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

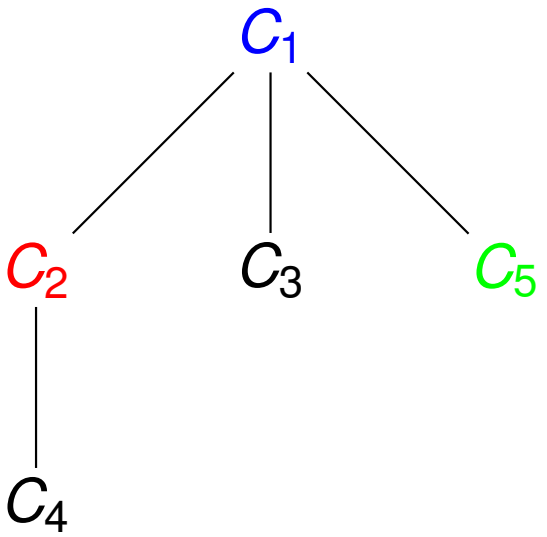


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

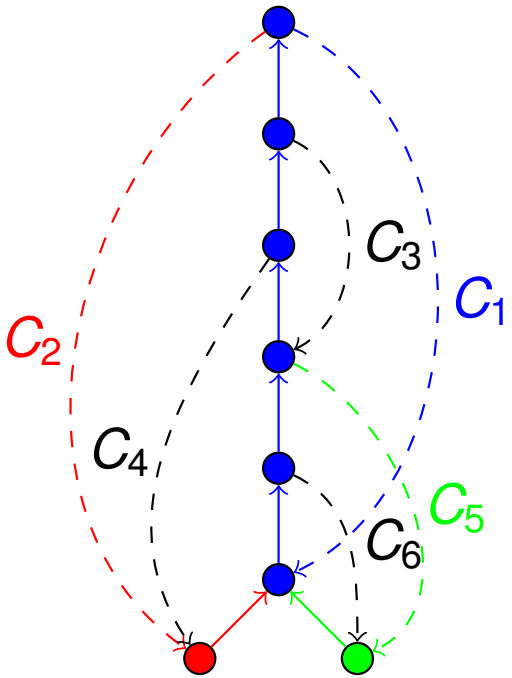


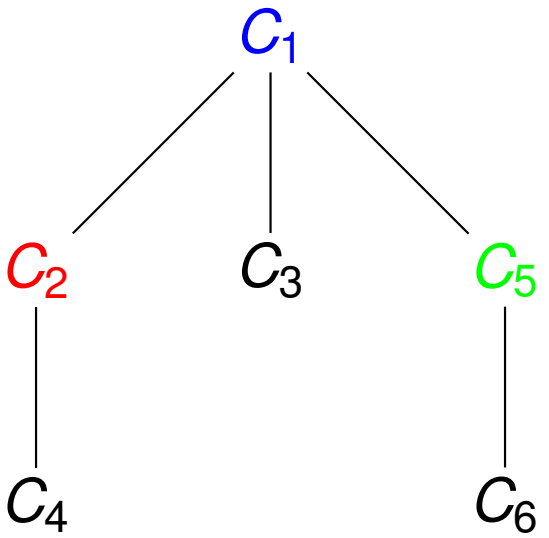


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

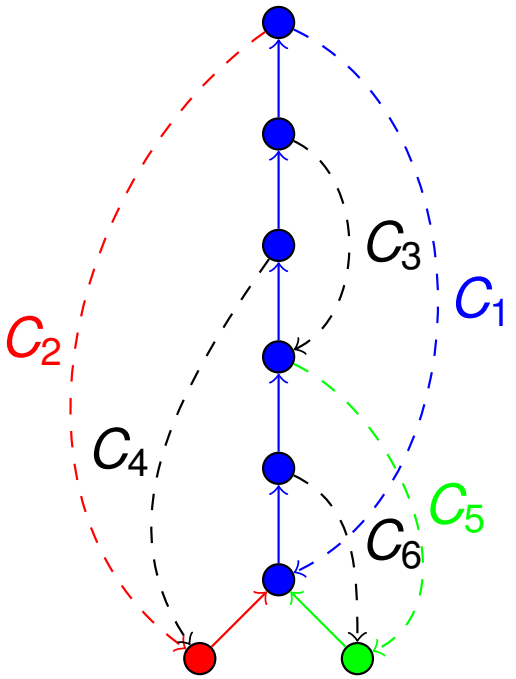


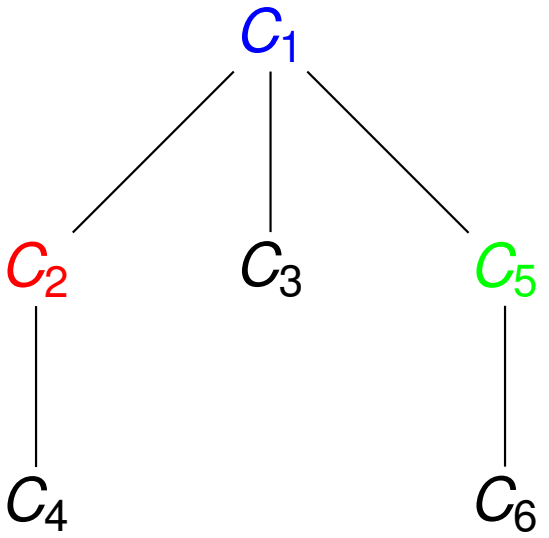


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.





Lemma: If G is 3-edge-connected then there is a Mader construction that adds the chains parent-first.

An Improved Algorithm

Observations

- If G is 2-vertex-connected: $C_1 \cup C_2 = K_2^3$
- We start with $G_c = C_1 \cup C_2$; current graph
- Chains become visible as soon as both endpoints belong to G_c
- A visible chain can be added (is addable) to G_c, if its endpoints lie on different links or one is a branch vertex.
- Conversely: a visible chain is not addable if its endpoints are on the same link.
- Adding a chain makes its endpoints branch vertices (if not already branching); this may make other chains addable. It also makes the children of the chain visible.

An Improved Algorithm

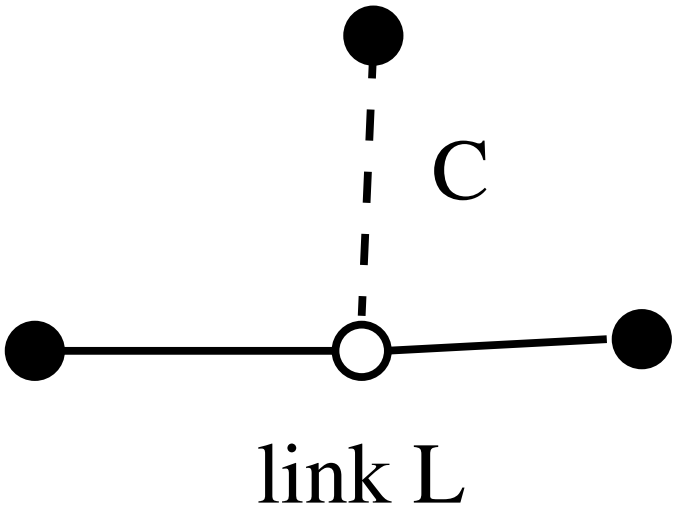
1. Initialize graph to $C_1 \cup C_2 \sim K_2^3$ and iterate over children C of C_1 and C_2. Add addable C's to the list of addable chains, associate others with a link.
2. Take a chain C from the list of addable chains.
   a) Add C. This turns endpoints that are non-branching to branching vertices and splits the links containing these vertices. So we split zero or one or two links.
   b) Check whether splitting a link L makes chains addable; such chains have both endpoints on L, but not both endpoints on L_1 or L_2.
   c) Process the children of C: some are addable and some have both endpoints on inner vertices of C. Associate the latter with the link C.
3. If there are addable chains left, goto 2.

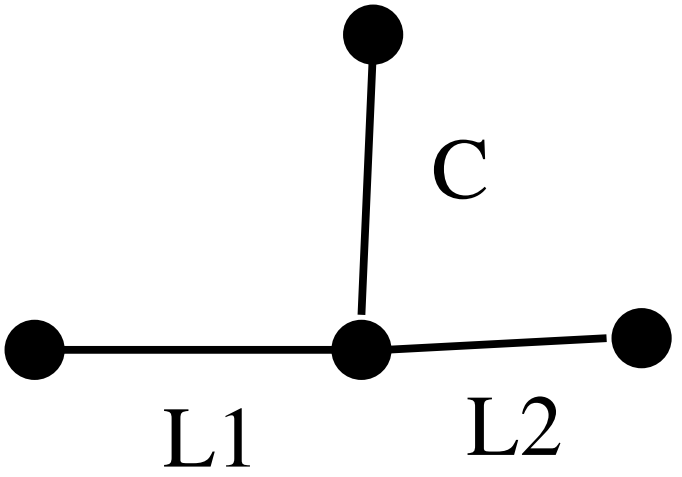
Can be implemented such that the runtime is in

$$O((n+m)\log(n+m)).$$

An Improved Algorithm

1. Initialize graph to $C_1 \cup C_2 \sim K_2^3$ and iterate over children C of C_1 and C_2. Add addable C's to the list of addable chains, associate others with a link.
2. Take a chain C from the list of addable chains.
   a) Add C. This turns endpoints that are non-branching to branching vertices and splits the links containing these vertices. So we split zero or one or two links.

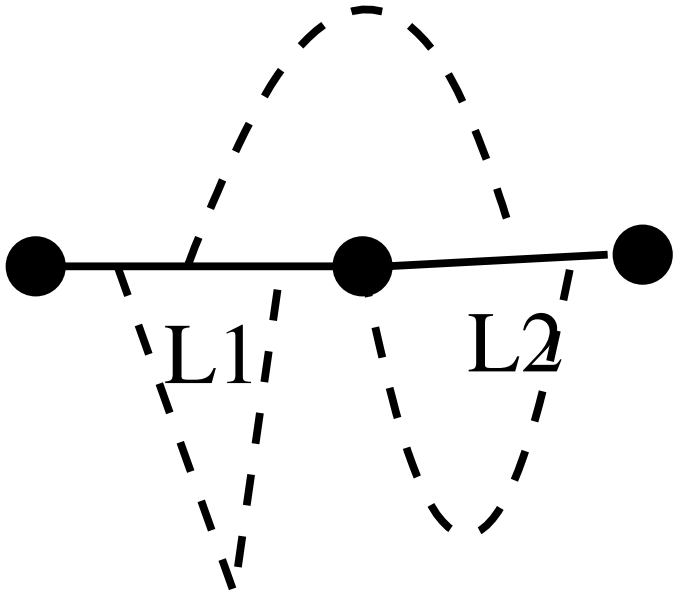




   b) Check whether splitting a link L makes chains addable; such chains have both endpoints on L, but not both endpoints on L_1 or L_2.
   c) Process the children of C: some are addable and some have both endpoints on inner vertices of C. Associate the latter with

An Improved Algorithm

1. Initialize graph to $C_1 \cup C_2 \sim K_2^3$ and iterate over children C of C_1 and C_2. Add addable C's to the list of addable chains, associate others with a link.
2. Take a chain C from the list of addable chains.
   a) Add C. This turns endpoints that are non-branching to branching vertices and splits the links containing these vertices. So we split zero or one or two links.
   b) Check whether splitting a link L makes chains addable; such chains have both endpoints on L, but not both endpoints on L_1 or L_2.



c) Process the children of C: some are addable and some have

An Improved Algorithm

1. Initialize graph to $C_1 \cup C_2 \sim K_2^3$ and iterate over children C of C_1 and C_2. Add addable C's to the list of addable chains, associate others with a link.
2. Take a chain C from the list of addable chains.
   a) Add C. This turns endpoints that are non-branching to branching vertices and splits the links containing these vertices. So we split zero or one or two links.
   b) Check whether splitting a link L makes chains addable; such chains have both endpoints on L, but not both endpoints on L_1 or L_2.
   c) Process the children of C: some are addable and some have both endpoints on inner vertices of C. Associate the latter with the link C.
3. If there are addable chains left, goto 2.

Can be implemented such that the runtime is in

$$O((n+m)\log(n+m)).$$

An Improved Algorithm

1. Initialize graph to $C_1 \cup C_2 \sim K_2^3$ and iterate over children C of C_1 and C_2. Add addable C's to the list of addable chains, associate others with a link.
2. Take a chain C from the list of addable chains.
   a) Add C. This turns endpoints that are non-branching to branching vertices and splits the links containing these vertices. So we split zero or one or two links.
   b) Check whether splitting a link L makes chains addable; such chains have both endpoints on L, but not both endpoints on L_1 or L_2.
   c) Process the children of C: some are addable and some have both endpoints on inner vertices of C. Associate the latter with the link C.
3. If there are addable chains left, goto 2.

Can be implemented such that the runtime is in

$$O((n+m)\log(n+m)).$$

Analysis of Improved Algorithm

- All steps except 2b are certainly linear.
- In 2b we have to look at all chains having both endpoints on L; some become addable and some will have both endpoints on L_1 or L_2. We will look at those again.
- How to process L?
  - process all chains incident to the new branching vertex.
  - work on L from both sides; switch between working on L_1 and L_2: an elementary step is to look at the endpoint of a chain.
  - stop, if either L_1 or L_2 is completely processed, say L_1: for each chain having both endpoints on L and at least one endpoint on L_1, we have seen two or one endpoint. If seen one, the chain is addable. Otherwise, now both endpoints on L_1.
  - cost = # addable ch. + $2 \cdot \min_{i=1,2}$ # chains only incident to L_i
  - charge the latter cost to the non-addable chains moved to $L_{j=\text{argmin}\,\min_{i=1,2}\ldots}$ and observe: whenever a chain is charged, it is moved to a set of half the size.

Analysis of Improved Algorithm

- All steps except 2b are certainly linear.
- In 2b we have to look at all chains having both endpoints on L; some become addable and some will have both endpoints on L_1 or L_2. We will look at those again.
- How to process L?
  - process all chains incident to the new branching vertex.
  - work on L from both sides; switch between working on L_1 and L_2: an elementary step is to look at the endpoint of a chain.
  - stop, if either L_1 or L_2 is completely processed, say L_1: for each chain having both endpoints on L and at least one endpoint on L_1, we have seen two or one endpoint. If seen one, the chain is addable. Otherwise, now both endpoints on L_1.
  - cost = # addable ch. + $2 \cdot \min_{i=1,2}$ # chains only incident to L_i
  - charge the latter cost to the non-addable chains moved to $L_{j=\text{argmin}\,\min_{i=1,2}\ldots}$ and observe: whenever a chain is charged, it is moved to a set of half the size.

A Linear Time Algorithm

- see paper
- also: a linear time certifying alg for computing cactus representation of 2-cuts.

- open problem: our $O((n+m)\log(n+m))$ algorithm is considerably simpler than the $O(n+m)$ algorithm. The linear time algorithm for vertex-connectivity is considerably more complex that the linear time edge-connectivity alg. Can it be simplified by accepting a log-factor?