Certifying Algs for 3-Connectivity

Kurt Mehlhorn
Adrian Neumann
Jens Schmidt

max planck institut
informatik

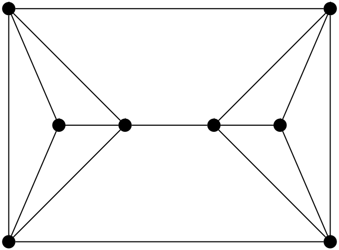
11. November 2014

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



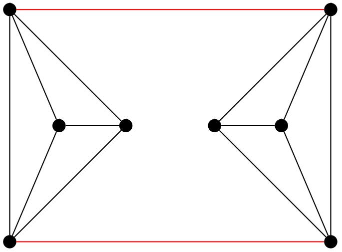
3-edge- and 3-vertex connected

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

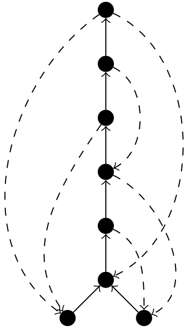
2-vertex-connected, but not 3-vertex-connected

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

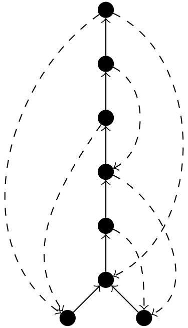
A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



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

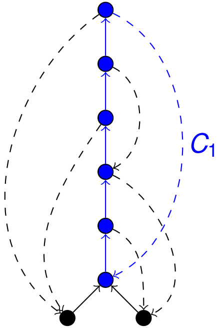
A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



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.

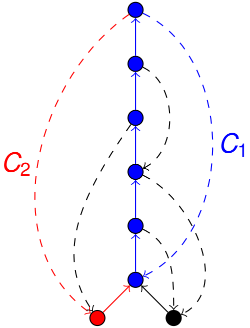


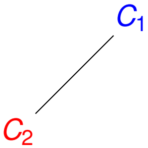
C_1

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.

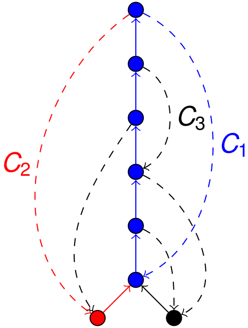


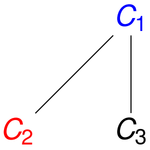


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

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

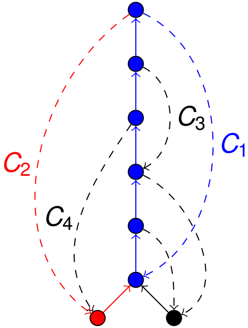


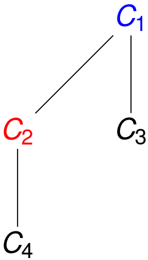


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

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

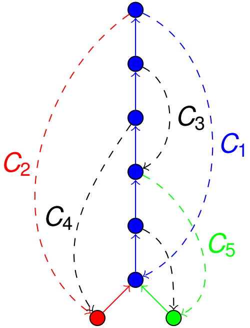


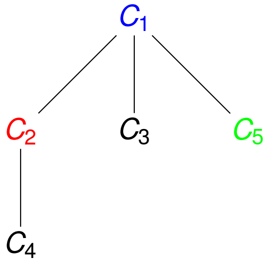


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

Chain Decomposition

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

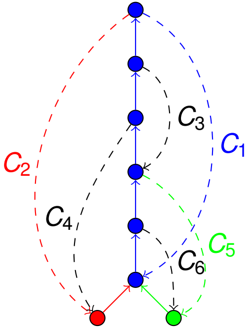


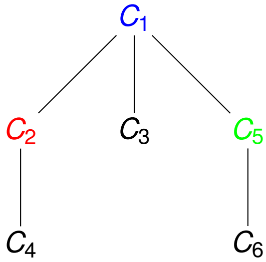


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore back edges top-down.

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.

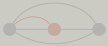
In order to do better, we need structural insight.

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)

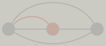
There is a linear time certifying algorithm for 3-edge-connectivity.

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

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.

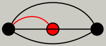
It outputs either a 2-edge-cut or a Mader construction sequence.

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.

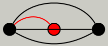
It outputs either a 2-edge-cut or a Mader construction sequence.

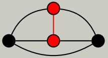
Three-Edge-Connectness

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.

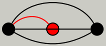
It outputs either a 2-edge-cut or a Mader construction sequence.

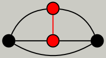
Three-Edge-Connectness

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

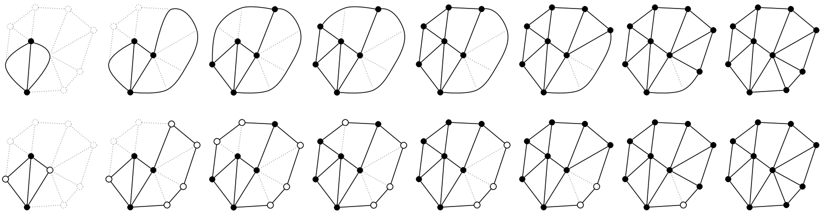
In This Talk

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.

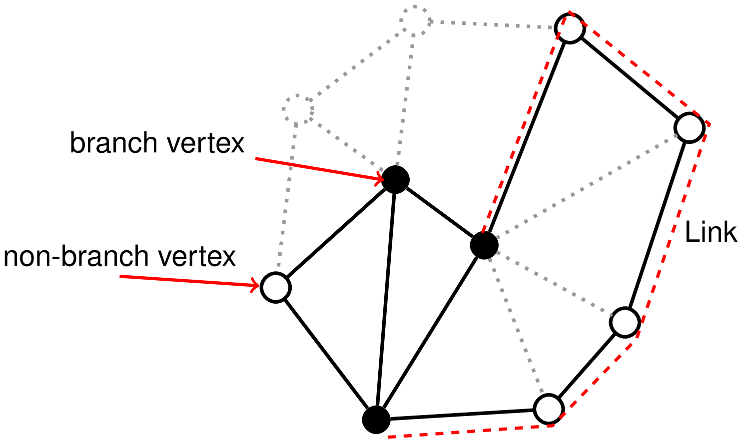
Mader Constructions and Subdivisions



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

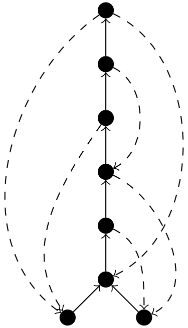


A First Algorithm

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

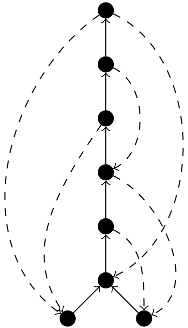
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



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

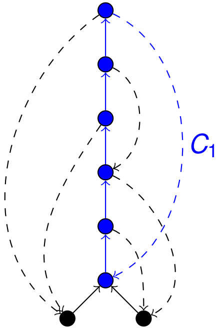
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.



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.

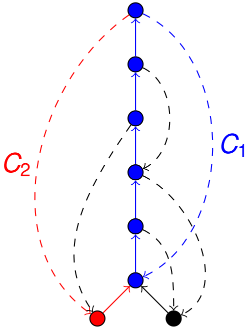


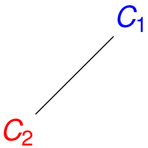
C_1

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.

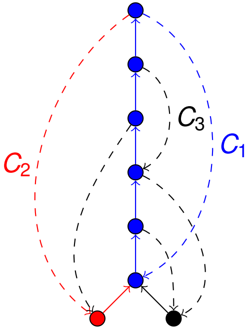


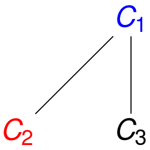


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

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.

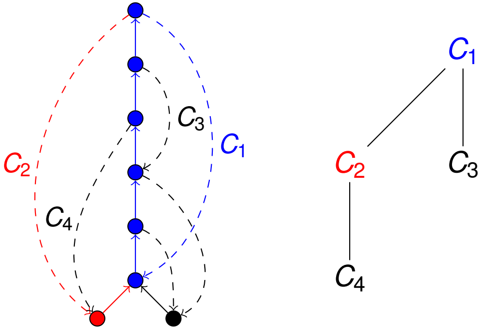




Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

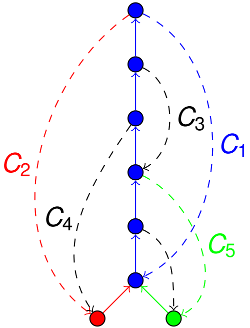
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

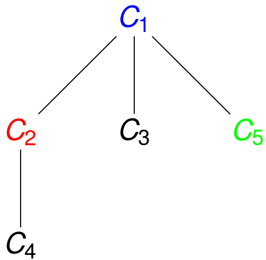


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

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.

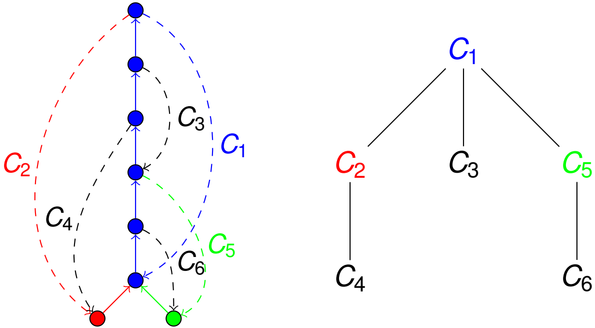




Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

Chain Decomposition

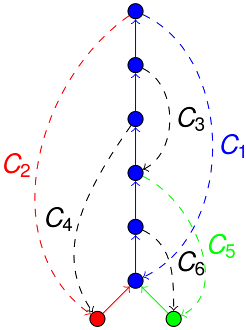
A structure to help find a K_2^3 and subsequent paths. A special ear-decomposition. Perform a DFS and direct tree edges upwards and back edges downwards.

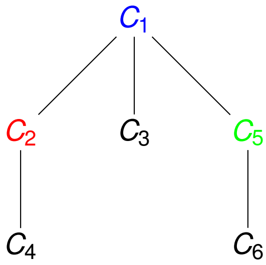


Construction of a chain: Follow a back edge and then tree edges up to a node already seen; the chain containing this node is the parent chain. Explore backedge top-down.

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

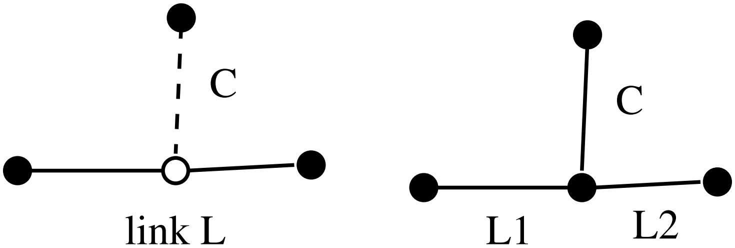
1. Initialize graph to $C_1 \cup C_2 \sim K_2^3$ and iterate over children C of C_1 and C_2. Add addable C's to the list of addable chains, associate others with a link.
2. Take a chain C from the list of addable chains.
   a) Add C. This turns endpoints that are non-branching to branching vertices and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

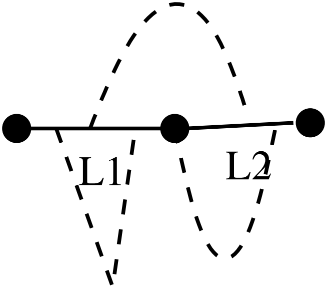
1. Initialize graph to $C_1 \cup C_2 \sim K_2^3$ and iterate over children C of C_1 and C_2. Add addable C's to the list of addable chains, associate others with a link.
2. Take a chain C from the list of addable chains.
   a) Add C. This turns endpoints that are non-branching to branching vertices and 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 $L_{j=\operatorname{argmin}\min_{i=1,2}\ldots}$ 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?