Computing min-cuts in hypergraphs

Chandra Chekuri and Chao Xu

July 18, 2017

University of Illinois, Urbana-Champaign

A hypergraph

A graph $G = (V, E)$ consists of vertices V and edges $E \subset \{\{u, v\} \,|\, u, v \in V\}$.

A hypergraph

A graph $G = (V, E)$ consists of vertices V and edges $E \subset \{\{u, v\} \, | u, v \in V\}$.

A hypergraph $H = (V, E)$ consists of vertices V and edges $E \subset \{U | U \subset V, |U| \geq 2\}$.

A hypergraph

A graph $G = (V, E)$ consists of vertices V and edges $E \subset \{\{u, v\} \,|u, v \in V\}$.

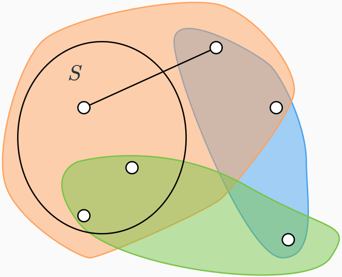
A hypergraph $H = (V, E)$ consists of vertices V and edges $E \subset \{U|U \subset V, |U| \geq 2\}$.

Each edge is assigned with a positive weight $w : E \to \mathbb{R}_+$.

Cut function

- Edge $e \in E$ crosses $S \subset V$, if $e \cap S$ and $e \cap V \setminus S$ are both non-empty.
- $\delta(S)$ is the set of all edges cross S.
- The cut function $c : 2^V \to \mathbb{R}_+$

$$c(S) = \sum_{e \in \delta(S)} w(e)$$



Cuts and min-cuts

- A cut is a bipartition of the vertices $(S, V \setminus S)$. We also call the set S a cut.

Cuts and min-cuts

- A cut is a bipartition of the vertices $(S, V \setminus S)$. We also call the set S a cut.
- A cut S is a (global) min-cut if $c(S)$ is minimized. It's value is denoted as $\lambda(G)$.

Cuts and min-cuts

- A cut is a bipartition of the vertices $(S, V \setminus S)$. We also call the set S a cut.
- A cut S is a (global) min-cut if $c(S)$ is minimized. It's value is denoted as $\lambda(G)$.
- A cut S a st-cut, if $s \in S$ and $t \in V \setminus S$.

Cuts and min-cuts

- A cut is a bipartition of the vertices $(S, V \setminus S)$. We also call the set S a cut.
- A cut S is a (global) min-cut if $c(S)$ is minimized. It's value is denoted as $\lambda(G)$.
- A cut S a *st*-cut, if $s \in S$ and $t \in V \setminus S$.
- The max-flow min-cut theorem is relates max-*st*-flow with min-*st*-cut (also called local min-cuts). It's value is denoted as $\lambda(G, s, t)$.

Why min-cut?

The value of min-cut measures the connectivity of the graph.

Hence min-cuts are useful in clustering and combinatorial optimization.

- disconnect railway networks

Why min-cut?

The value of min-cut measures the connectivity of the graph.

Hence min-cuts are useful in clustering and combinatorial optimization.

- disconnect railway networks
- image segmentation

Why min-cut?

The value of min-cut measures the connectivity of the graph.

Hence min-cuts are useful in clustering and combinatorial optimization.

- disconnect railway networks
- image segmentation
- graph partitioning

Why min-cut?

The value of min-cut measures the connectivity of the graph.

Hence min-cuts are useful in clustering and combinatorial optimization.

- disconnect railway networks
- image segmentation
- graph partitioning
- useful constraints for traveling salesman problem

Why min-cut?

The value of min-cut measures the connectivity of the graph.

Hence min-cuts are useful in clustering and combinatorial optimization.

- disconnect railway networks
- image segmentation
- graph partitioning
- useful constraints for traveling salesman problem
- ...

Why all min-cuts?

- Reinforce railway networks

Why all min-cuts?

- Reinforce railway networks edge connectivity augmentation
  [Gabow 1991; Naor, Gusfield, Martel 1997]

Why all min-cuts?

- Reinforce railway networks edge connectivity augmentation [Gabow 1991; Naor, Gusfield, Martel 1997]
- data security [Kao 1996]

Why all min-cuts?

- Reinforce railway networks edge connectivity augmentation [Gabow 1991; Naor, Gusfield, Martel 1997]
- data security [Kao 1996]
- graph drawing [Kant 1993]

Why all min-cuts?

- Reinforce railway networks edge connectivity augmentation [Gabow 1991; Naor, Gusfield, Martel 1997]
- data security [Kao 1996]
- graph drawing [Kant 1993]
- edge splitting [Nagamochi, Nakamura, Ibaraki 2000]
- ...

Outline

Our contribution: a deterministic algorithm that finds ALL min-cut in a hypergraph in the same running time as finding a *single* min-cut.

Outline

Our contribution: a deterministic algorithm that finds ALL min-cut in a hypergraph in the same running time as finding a *single* min-cut.

- How to find a single min-cut in a graph?

Outline

Our contribution: a deterministic algorithm that finds ALL min-cut in a hypergraph in the same running time as finding a *single* min-cut.

- How to find a single min-cut in a graph?
- What do we mean by ALL min-cuts and how did we find them?

Outline

Our contribution: a deterministic algorithm that finds ALL min-cut in a hypergraph in the same running time as finding a *single* min-cut.

- How to find a single min-cut in a graph?
- What do we mean by ALL min-cuts and how did we find them?
- A new algorithm for finding ALL min-cuts in graphs.

Outline

Our contribution: a deterministic algorithm that finds ALL min-cut in a hypergraph in the same running time as finding a *single* min-cut.

- How to find a single min-cut in a graph?
- What do we mean by ALL min-cuts and how did we find them?
- A new algorithm for finding ALL min-cuts in graphs.
- How to modify the algorithm to work for hypergraphs?

n is the # of vertices. m is the # of edges.

The fundamental problem: Finding a min-cut

- Naive:

$$\lambda(G) = \min_{s,t \in V} \lambda(G, s, t)$$

The fundamental problem: Finding a min-cut

- Naive:

$$\lambda(G) = \min_{s,t \in V} \lambda(G, s, t)$$

Find min-*st*-cuts over all $s, t \in V$.

$\binom{n}{2}$ max flows . Max flow takes $O(mn)$ time [Orlin 2013]

$= O(n^3 m)$

The fundamental problem: Finding a min-cut

- Naive:

$$\lambda(G) = \min_{s,t \in V} \lambda(G, s, t)$$

  Find min-*st*-cuts over all $s, t \in V$.
  $\binom{n}{2}$ max flows . Max flow takes $O(mn)$ time [Orlin 2013]
  $= O(n^3 m)$

- Smarter:

$$\lambda(G) = \min_{t \in V} \lambda(G, s, t)$$

The fundamental problem: Finding a min-cut

- Naive:

$$\lambda(G) = \min_{s,t \in V} \lambda(G, s, t)$$

  Find min-*st*-cuts over all $s, t \in V$.

  $\binom{n}{2}$ max flows . Max flow takes $O(mn)$ time [Orlin 2013]

  $= O(n^3 m)$

- Smarter:

$$\lambda(G) = \min_{t \in V} \lambda(G, s, t)$$

  Fix $s \in V$, find min-*st*-cut over all $t \in V$.

  $n - 1$ max flows $= O(n^2 m)$

The fundamental problem: Finding a min-cut

- Naive:

$$\lambda(G) = \min_{s,t \in V} \lambda(G, s, t)$$

  Find min-st-cuts over all $s, t \in V$.
  $\binom{n}{2}$ max flows . Max flow takes $O(mn)$ time [Orlin 2013]
  $= O(n^3 m)$

- Smarter:

$$\lambda(G) = \min_{t \in V} \lambda(G, s, t)$$

  Fix $s \in V$, find min-st-cut over all $t \in V$.
  $n - 1$ max flows $= O(n^2 m)$

- Fastest known:

$$\lambda(G) = \min(\lambda(G, s, t), \lambda(G/st))$$

  $O(nm + n^2 \log n)$ [Nagamochi & Ibaraki 1992]

The min-cut algorithm

Recurrence relation

$$\lambda(G) = \min(\lambda(G, s, t), \lambda(G/st))$$

The min-cut algorithm

Recurrence relation

$$\lambda(G) = \min(\lambda(G, s, t), \lambda(G/st))$$

Algorithm $\textsc{MinCut}$(G):

1. Find a min-*st*-cut for some $s, t \in V$.
2. Find $\textsc{MinCut}(G/st)$.
3. Return the min of the two.

The min-cut algorithm

Recurrence relation

$$\lambda(G) = \min(\lambda(G, s, t), \lambda(G/st))$$

Algorithm $\textsc{MinCut}$(G):

1. Find a min-st-cut for some $s, t \in V$.
2. Find $\textsc{MinCut}(G/st)$.
3. Return the min of the two.

- $n - 1$ calls to $\textsc{MinCut}$.

The min-cut algorithm

Recurrence relation

$$\lambda(G) = \min(\lambda(G, s, t), \lambda(G/st))$$

Algorithm $\textsc{MinCut}$(G):

1. Find a min-st-cut for some $s, t \in V$.
2. Find $\textsc{MinCut}(G/st)$.
3. Return the min of the two.

- $n - 1$ calls to $\textsc{MinCut}$.
- $n - 1$ max flow computation.

The min-cut algorithm

Recurrence relation

$$\lambda(G) = \min(\lambda(G, s, t), \lambda(G/st))$$

Algorithm $\textsc{MinCut}$(G):

1. Find a min-*st*-cut for some $s, t \in V$.
2. Find $\textsc{MinCut}(G/st)$.
3. Return the min of the two.

- $n - 1$ calls to $\textsc{MinCut}$.
- $n - 1$ max flow computation.
- $O(n^2 m)$ again.
- How to improve this?

Find a min-st-cut

Find a min-st-cut for some $s, t \in V$.

Find a min-st-cut

Find a min-st-cut for some $s, t \in V$.

Find a min-st-cut

Find a min-st-cut for some $s, t \in V$.

Find a min-st-cut

Find a min-st-cut for some $s, t \in V$.

- Aim: pick s and t, where the max st-flow can be computed quickly.

Find a min-st-cut

Find a min-st-cut for some $s, t \in V$.

- Aim: pick s and t, where the max st-flow can be computed quickly.
- How?

Find a min-st-cut

Find a min-st-cut for some $s, t \in V$.

- Aim: pick s and t, where the max st-flow can be computed quickly.
- How? maximum adjacency ordering.

Adjacency

$$d(A, B) = \sum_{e \in \delta(A) \cap \delta(B)} w(e)$$

Adjacency

$$d(A, B) = \sum_{e \in \delta(A) \cap \delta(B)} w(e)$$

$v_1, \ldots, v_n$ is a maximum adjacency ordering(MA-ordering) if for all $1 \leq i \leq j \leq n$,

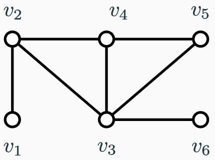
$$d(\{v_1, \ldots, v_{i-1}\}, v_i) \geq d(\{v_1, \ldots, v_{i-1}\}, v_j).$$

Adjacency

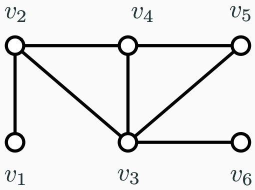
$$d(A,B) = \sum_{e \in \delta(A) \cap \delta(B)} w(e)$$

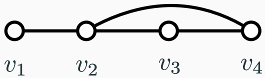
$v_1, \ldots, v_n$ is a maximum adjacency ordering(MA-ordering) if for all $1 \le i \le j \le n$,

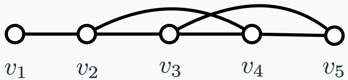
$$d(\{v_1, \ldots, v_{i-1}\}, v_i) \ge d(\{v_1, \ldots, v_{i-1}\}, v_j).$$

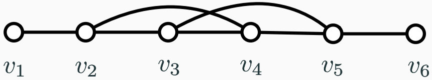


$$d(\{v_1, \ldots, v_{i-1}\}, v_i) \geq d(\{v_1, \ldots, v_{i-1}\}, v_j)$$

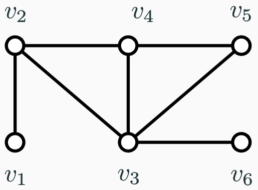


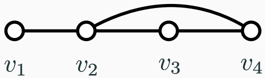


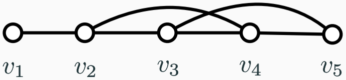


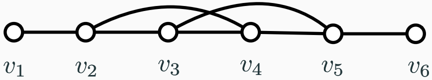


$$d(\{v_1, \ldots, v_{i-1}\}, v_i) \geq d(\{v_1, \ldots, v_{i-1}\}, v_j)$$









MA-ordering can be found in $O(m + n \log n)$ time.

$\{v_n\}$ forms a min $v_{n-1}v_n$-cut.

$\{v_n\}$ forms a min $v_{n-1}v_n$-cut.

Lemma ([Arikati & Mehlhorn 1999])

Given graph $G = (V, E)$ and MA ordering $v_1, \ldots, v_n$. A max $v_{n-1}v_n$-flow can be found in $O(m)$ time.

A min-cut of a graph can be found in $n - 1 \times$ MA-ordering, so $O(nm + n^2 \log n)$ time.

What about hypergraphs?

- The exact same algorithm works for hypergraphs.
- Different ordering:
  - MA ordering [Klimmek & Wagner 1996]
  - Tight ordering [Mak & Wong 2000]
  - Queyranne ordering [Queyranne 1998]

Finding ALL min-cuts

Finding ALL min-cuts

What is the desired output?

- List all min-cuts?

Finding ALL min-cuts

What is the desired output?

- List all min-cuts? Can there be too many min-cuts?

Finding ALL min-cuts

What is the desired output?

- List all min-cuts? Can there be too many min-cuts? No. There are at most $\binom{n}{2}$ min-cuts. [Karger 1993]

Finding ALL min-cuts

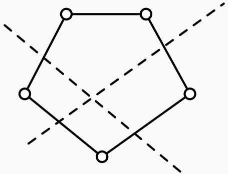
What is the desired output?

- List all min-cuts? Can there be too many min-cuts? No. There are at most $\binom{n}{2}$ min-cuts. [Karger 1993]
- $O(n^3)$ space sufficient: $(S, V \setminus S)$ requires $\min(|S|, |V \setminus S|)$ space.

Finding ALL min-cuts

What is the desired output?

- List all min-cuts? Can there be too many min-cuts? No. There are at most $\binom{n}{2}$ min-cuts. [Karger 1993]
- $O(n^3)$ space sufficient: $(S, V \setminus S)$ requires $\min(|S|, |V \setminus S|)$ space.
- $\Omega(n^3)$ space required: a cycle, space usage $\sum_{i=1}^{n/2} ni = \Omega(n^3)$.



Finding ALL min-cuts: desirable properties

Find a data structure:

- Small: size is smaller than listing all min-cuts.
- Simple: the structure is simple, so one can query the data structure quickly.

Finding ALL min-cuts: desirable properties

Find a data structure:

- Small: size is smaller than listing all min-cuts.
- Simple: the structure is simple, so one can query the data structure quickly. enumeration, counting, filtering...

Finding ALL min-cuts: desirable properties

Find a data structure:

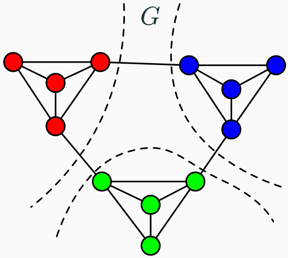
- Small: size is smaller than listing all min-cuts.
- Simple: the structure is simple, so one can query the data structure quickly. enumeration, counting, filtering...

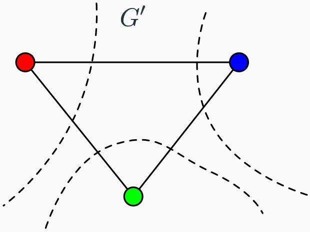
The best kind of data structure: a smaller, simple graph with the same min-cut structure.

Representation

$G' = (V', E')$ is a representation of $G = (V, E)$ if there exist a function $\phi : V \to V'$, such that

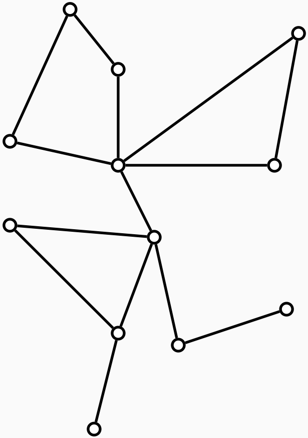
- S is a min-cut in G, then $\phi(S)$ is a min-cut in G'.
- S' is a min-cut in G', then $\phi^{-1}(S')$ is a min-cut in G.





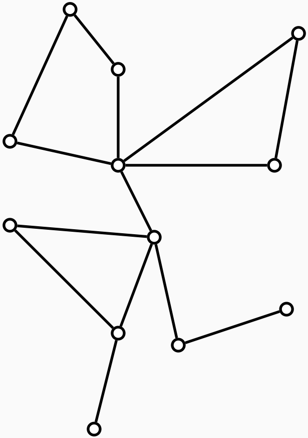
Cactus

A graph is called a cactus if no two cycles share an edge.



Cactus

A graph is called a cactus if no two cycles share an edge.



Every graph has a cactus representation. [Karzanov & Timofeev 1986]

Cactus representation: Algorithms, a long history

Large number of work

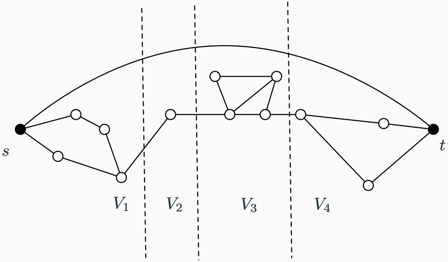
- $O(mn\log(n^2/m))$ [Gabow 1993]
- $O(mn + n^2\log n + n^* m\log n)$ [Nagamochi & Kameda 1996]
  (n^* is # of vertices in the representation)
- $O(mn + n^2\log n + \gamma m\log n)$ [Nagamochi, Nakao, Ibaraki 2000]
  (γ is # of cycles in the representation)
- $O(mn + n^2\log n)$ [Nagamochi, Nakamura, Ishii 2003]

Cactus representation: The state of the art

An edge e is critical, if e crosses a min-cut.

Theorem ([Nagamochi & Kameda 1996])

If st is an critical edge, then there exists a partition $\{V_1, \ldots, V_k\}$ of V, such that $\left\{\bigcup_{i=1}^{j} V_i | 1 \leq j < k\right\}$ are the set of min-st-cuts.



Can be found in $O(m + n \log n)$ time: MA-ordering.

Cactus representation: Previous method

The partition give us a partial cactus.

Algorithm $\textsc{All-MinCut}(\mathsf{G})$:

1. $A \leftarrow$ cactus of all min-st-cut for some critical edge st.
2. $B \leftarrow \textsc{All-MinCut}(G/st)$.
3. Cleverly combine cactus A and B into a cactus for G.

Running time$=n-1\times$ MA ordering, same as finding a single min-cut.

Why is the algorithm unsatisfactory

- Complicated

Why is the algorithm unsatisfactory

- Complicated
- Does not generalize to hypergraphs.

Finding cactus representation for graphs

Our approach: a new algorithm

The decomposition framework [Fujishige 1983, Cunningham 1983].

Our approach: a new algorithm

The decomposition framework [Fujishige 1983, Cunningham 1983].

- decompose the graph to structurally simple graphs.

Our approach: a new algorithm

The decomposition framework [Fujishige 1983, Cunningham 1983].

- decompose the graph to structurally simple graphs.
- preserves information on min-cuts, and might lose information on other cuts.

Our approach: a new algorithm

The decomposition framework [Fujishige 1983, Cunningham 1983].

- decompose the graph to structurally simple graphs.
- preserves information on min-cuts, and might lose information on other cuts. Exactly what we need.

Our approach: a new algorithm

The decomposition framework [Fujishige 1983, Cunningham 1983].

- decompose the graph to structurally simple graphs.
- preserves information on min-cuts, and might lose information on other cuts. Exactly what we need.

Remark: Decomposition framework actually handles all (symmetric) submodular functions ([Fujishige 1983]) [Cunningham 1983].

Decomposition: Refinement

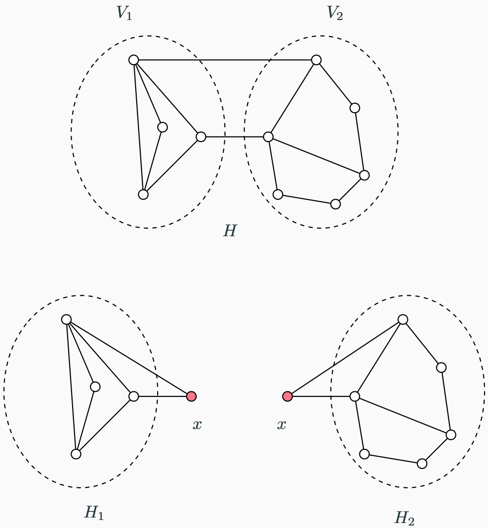
A min-cut with at least two vertices on each side is called a split.

Definition

Given a graph G, $\{G_1, G_2\}$ is a refinement of G if G_1 and G_2 are graphs obtained through a split (V_1, V_2) and a new marker vertex x as follows.

1. G_1 is G/V_2, such that V_2 gets contracted to x.
2. G_2 is G/V_1, such that V_1 gets contracted to x.

Example of a refinement



Decomposition

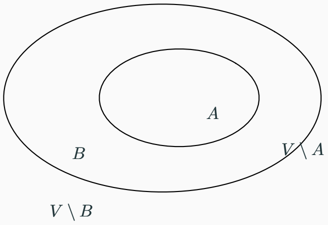
Definition

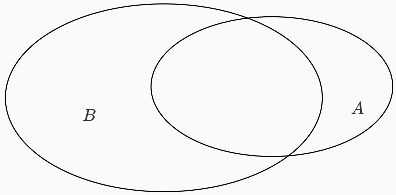
A set of graphs $\mathcal{D} = \{G_1, \ldots, G_k\}$ is a decomposition of G if it is obtained from $\{G\}$ by replacing an element by its refinements.

Two decomposition are equivalent if they are the same up to relabeling marker vertices.

Crossing

Two cuts $(A, V \setminus A)$ and $(B, V \setminus B)$ are crossing if $A \cap B$, $A \setminus B$ and $B \setminus A$ are all non-empty.

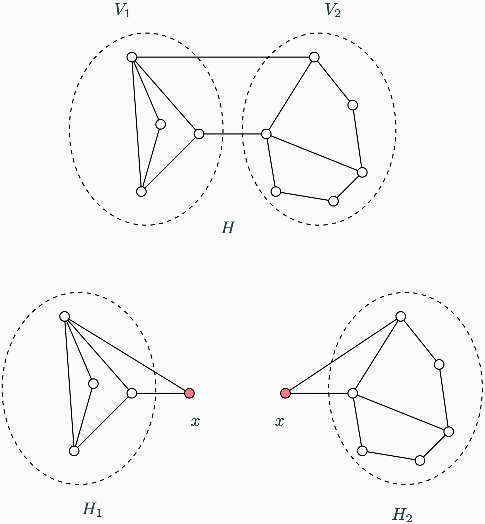




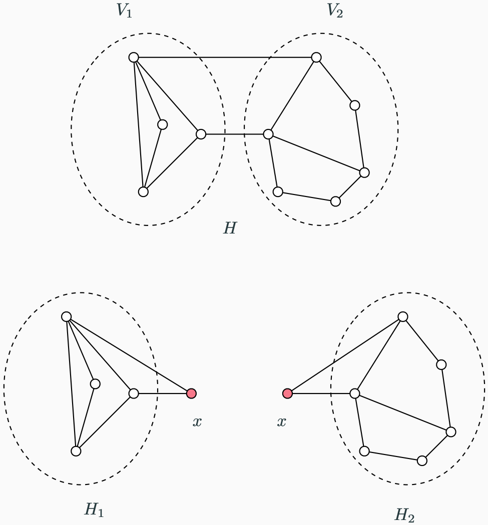
If $(A, V \setminus A)$ and $(B, V \setminus B)$ are non-crossing, then contract A or B preserves the other cut.

A split is good if no min-cut crosses it.

Refinement preserves cuts



Refinement preserves cuts

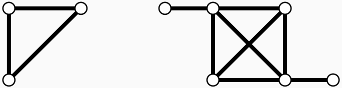


The min-cut in H_1 and H_2 are the min-cuts in H.

Refine through good splits preserves all min-cuts.

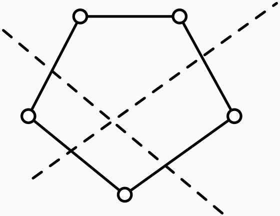
Definition

A graph G is prime if it does not contain any split.



Good splits and cycles

There exist graphs that are not prime but have no good splits.



Theorem

A graph without any good split is either a prime or a cycle.

Standard decomposition

A decomposition consists of primes and cycles is called a standard decomposition.

Standard decomposition

A decomposition consists of primes and cycles is called a standard decomposition.

Observation: Any refinement of a standard decomposition is a standard decomposition.

Standard decomposition

A decomposition consists of primes and cycles is called a standard decomposition.

Observation: Any refinement of a standard decomposition is a standard decomposition.

A standard decomposition is minimal, if it is not a refinement of any other standard decomposition.

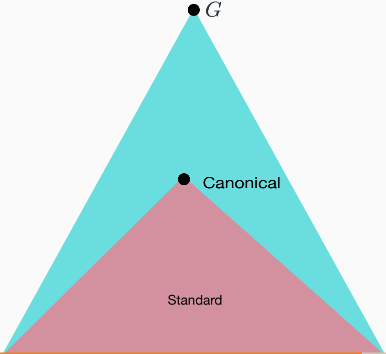
Canonical decomposition

Theorem ([Fujishige 1983,Cunningham 1983])

There exist a unique minimal standard decomposition.

Such decomposition is called the canonical decomposition.

Canonical decomposition is what we want: obtained through only applying good splits.



Properties of the canonical decomposition

Properties of the canonical decomposition

- Obtained through refinement applied to good splits.

Properties of the canonical decomposition

- Obtained through refinement applied to good splits.
- Consists of only primes and cycles.

Properties of the canonical decomposition

- Obtained through refinement applied to good splits.
- Consists of only primes and cycles.
- Recover a cactus representation from a canonical decomposition is easy: $O(n)$ time. [Cheng 1999]

Properties of the canonical decomposition

- Obtained through refinement applied to good splits.
- Consists of only primes and cycles.
- Recover a cactus representation from a canonical decomposition is easy: $O(n)$ time. [Cheng 1999]

New Goal: Find the canonical decomposition.

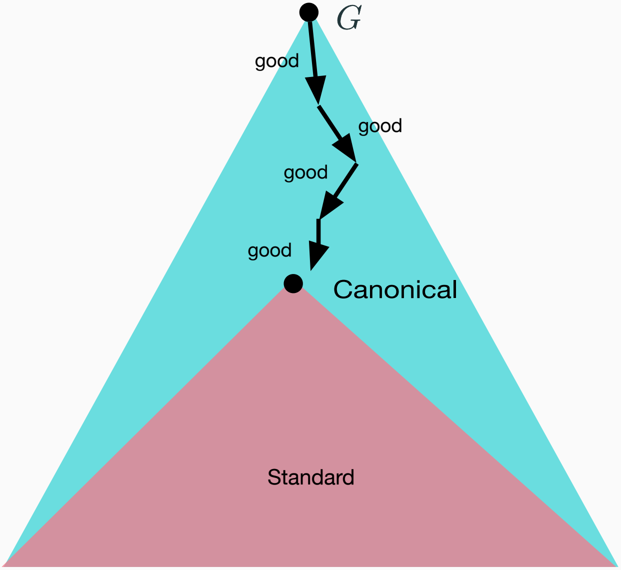
Algorithm, first attempt

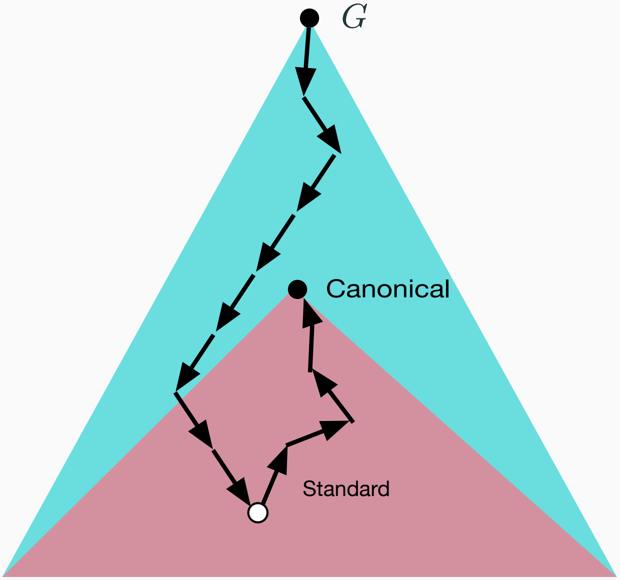
1. Find a good split.
2. Produce a refinement $\{G_1, G_2\}$ using the good split.
3. Recurse on G_1 and G_2.

Algorithm, first attempt

1. Find a good split.
2. Produce a refinement $\{G_1, G_2\}$ using the good split.
3. Recurse on G_1 and G_2.

Algorithm running time are dominated by finding $O(n)$ good splits.
Finding a good split is no easier than finding a min-cut.


G
good
good
good
good
Canonical
Standard


G
Canonical
Standard

Algorithm, second attempt

Merge: $\{G_1, \ldots, G_k\}$ a decomposition of G, G_1 and G_2 shares a marker vertex. Find a decomposition $\{G', G_3, \ldots, G_k\}$ of G, where $\{G_1, G_2\}$ is a refinement of G'.

Algorithm, second attempt

Merge: $\{G_1, \ldots, G_k\}$ a decomposition of G, G_1 and G_2 shares a marker vertex. Find a decomposition $\{G', G_3, \ldots, G_k\}$ of G, where $\{G_1, G_2\}$ is a refinement of G'.

1. Find any standard decomposition.
2. Merge elements in the decomposition while maintaining a standard decomposition.

Algorithm, second attempt

Merge: $\{G_1, \ldots, G_k\}$ a decomposition of G, G_1 and G_2 shares a marker vertex. Find a decomposition $\{G', G_3, \ldots, G_k\}$ of G, where $\{G_1, G_2\}$ is a refinement of G'.

1. Find any standard decomposition.
2. Merge elements in the decomposition while maintaining a standard decomposition.

Second operation is $O(nm)$ time:

1. Inspect each marker vertex

Algorithm, second attempt

Merge: $\{G_1, \ldots, G_k\}$ a decomposition of G, G_1 and G_2 shares a marker vertex. Find a decomposition $\{G', G_3, \ldots, G_k\}$ of G, where $\{G_1, G_2\}$ is a refinement of G'.

1. Find any standard decomposition.
2. Merge elements in the decomposition while maintaining a standard decomposition.

Second operation is $O(nm)$ time:

1. Inspect each marker vertex ($O(n)$ of them)
2. Check if merge maintains a standard decomposition.

Algorithm, second attempt

Merge: $\{G_1, \ldots, G_k\}$ a decomposition of G, G_1 and G_2 shares a marker vertex. Find a decomposition $\{G', G_3, \ldots, G_k\}$ of G, where $\{G_1, G_2\}$ is a refinement of G'.

1. Find any standard decomposition.
2. Merge elements in the decomposition while maintaining a standard decomposition.

Second operation is $O(nm)$ time:

1. Inspect each marker vertex ($O(n)$ of them)
2. Check if merge maintains a standard decomposition. Easy: Is the new graph a cycle?

Algorithm, second attempt

Merge: $\{G_1, \ldots, G_k\}$ a decomposition of G, G_1 and G_2 shares a marker vertex. Find a decomposition $\{G', G_3, \ldots, G_k\}$ of G, where $\{G_1, G_2\}$ is a refinement of G'.

1. Find any standard decomposition.
2. Merge elements in the decomposition while maintaining a standard decomposition.

Second operation is $O(nm)$ time:

1. Inspect each marker vertex ($O(n)$ of them)
2. Check if merge maintains a standard decomposition. Easy: Is the new graph a cycle?

New Goal: Find a standard decomposition.

How to find a standard decomposition

Idea: We don't have to always apply refinement all the time. Any progress is fine.

How to find a standard decomposition

Idea: We don't have to always apply refinement all the time. Any progress is fine.

For fixed $s, t \in V$, either there is a split that separates s and t(called a *st*-split). Or we can contract s and t.

Split oracle

Problem

Given G and the min-cut value λ, outputs either a split in G or a pair of vertices $\{s, t\}$ such that there is no st-split in G.

An algorithm solve the above problem is called a split oracle.

Split oracle

Problem

Given G and the min-cut value λ, outputs either a split in G or a pair of vertices $\{s, t\}$ such that there is no st-split in G.

An algorithm solve the above problem is called a split oracle.

If we have a fast split oracle, then we can solve the problem.

1. Apply the split oracle on G.
2. If the oracle returns a split, find a refinement of H and recurse on both sides.
3. Otherwise, the oracle output a pair s, t. Contract s and t and recurse.

Theorem

A split oracle can be implemented in $O(m + n \log n)$ time.

Sketch

- Use MA-ordering to find a max *st*-flow for some s and t.
- Enumerate at most 3 min *st*-cuts using the maximum flow. It either finds a split or decide there is none. [Provan & Shier 1996]

The reductions

1. cactus representation:

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition:

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition: $O(nm)+$finding a standard decomposition

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition: $O(nm)+$finding a standard decomposition
3. standard decomposition:

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition: $O(nm)+$finding a standard decomposition
3. standard decomposition: $O(nm) + O(n) \times$ split oracle

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition: $O(nm)+$finding a standard decomposition
3. standard decomposition: $O(nm) + O(n) \times$ split oracle
4. split oracle:

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition: $O(nm)+$finding a standard decomposition
3. standard decomposition: $O(nm) + O(n) \times$ split oracle
4. split oracle: MA-ordering $O(m + n \log n)$.

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition: $O(nm)+$finding a standard decomposition
3. standard decomposition: $O(nm) + O(n) \times$ split oracle
4. split oracle: MA-ordering $O(m + n \log n)$.

Total running time: $O(nm + n^2 \log n)$.

The reductions

1. cactus representation: $O(m)+$ finding the canonical decomposition.
2. canonical decomposition: $O(nm)+$finding a standard decomposition
3. standard decomposition: $O(nm) + O(n) \times$ split oracle
4. split oracle: MA-ordering $O(m + n \log n)$.

Total running time: $O(nm + n^2 \log n)$. Same as finding a single min-cut.

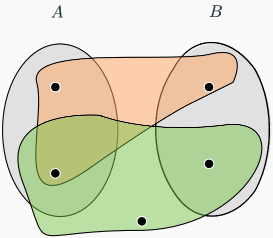
What about hypergraphs?

Our algorithm almost work for hypergraphs. Just a few small changes.

1. We need to use tight ordering instead of MA-ordering.
2. There are non-cycles that also doesn't have any good split.
3. Cactus is not the right representation.

Modification 1: Tight adjacency and tight ordering

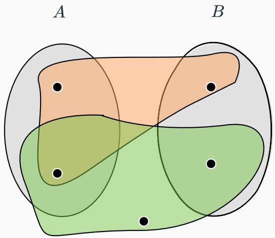
$$d'(A, B) = \sum_{\substack{e \in \delta(A) \cap \delta(B) \\ e \subseteq A \cup B}} w(e)$$



$d'(A, B) = 1$

Modification 1: Tight adjacency and tight ordering

$$d'(A, B) = \sum_{\substack{e \in \delta(A) \cap \delta(B) \\ e \subseteq A \cup B}} w(e)$$

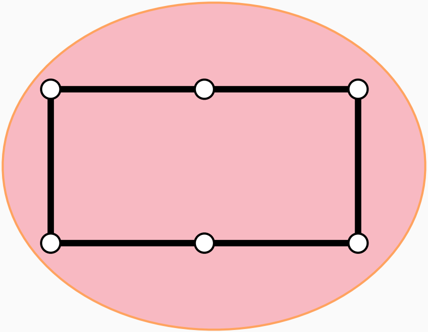


$d'(A, B) = 1$

$v_1, \ldots, v_n$ is a tight ordering(MA ordering) if for all $1 \leq i \leq j \leq n$,

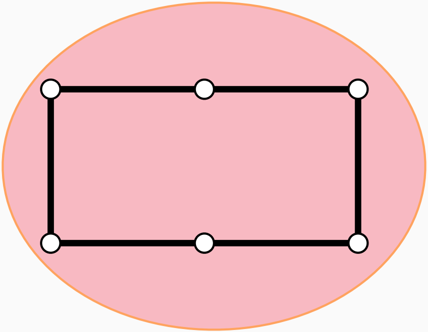
$$d'(\{v_1, \ldots, v_{i-1}\}, v_i) \geq d'(\{v_1, \ldots, v_{i-1}\}, v_j).$$

Modification 2: Solid polygons



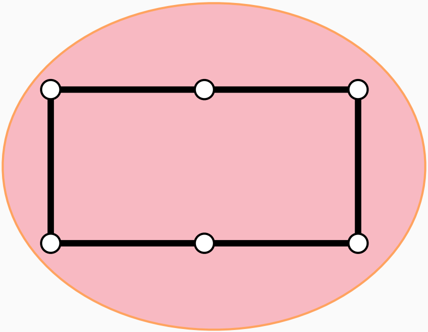
A hypergraph is a solid polygon if it consist of a (possibly 0 weight) cycle and a (possibly 0 weight) hyperedge covering all vertices.

Modification 2: Solid polygons



A hypergraph is a solid polygon if it consist of a (possibly 0 weight) cycle and a (possibly 0 weight) hyperedge covering all vertices. Hypergraphs with no good splits are either primes or solid polygons.

Modification 2: Solid polygons

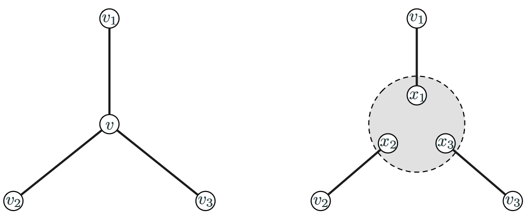


A hypergraph is a solid polygon if it consist of a (possibly 0 weight) cycle and a (possibly 0 weight) hyperedge covering all vertices. Hypergraphs with no good splits are either primes or solid polygons. A standard decomposition consist of primes and solid polygons.

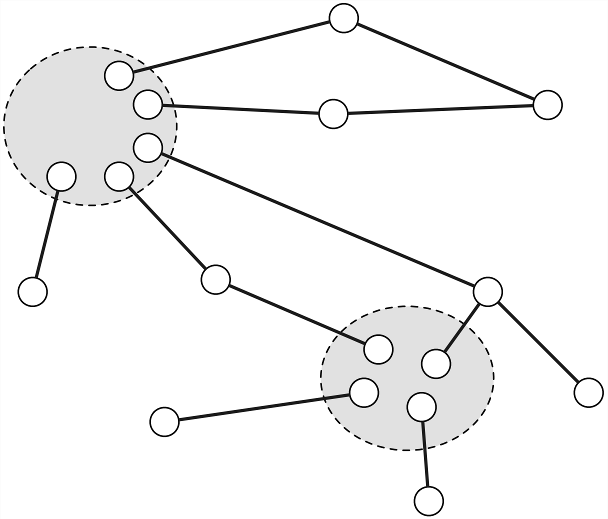
Modification 3: Hypercactus

Definition

A hypergraph is called a hypercactus if it can be obtained through the following operations from a cactus: Split the edges incident to a star, and "blow up" the vertex into a hyperedge.



Modification 3: Hypercactus Example



Modification 3: Hypercactus representation

Theorem ([Cheng 1999, Fleiner & Jordán 1999])
Every hypergraph have a hypercactus representation.

Modification 3: Hypercactus representation

Theorem ([Cheng 1999, Fleiner & Jordán 1999])
Every hypergraph have a hypercactus representation.

Remark: Fleiner & Jordán actually showed there is a hypercactus representation for all symmetric submodular functions.

Conclusion

All hypergraph min-cuts can be computed in the same running time as finding a single min-cut.

Thank you!