Deterministic and Exact Fully-dynamic Minimum Cut of Superpolylogarithmic Size in Subpolynomial Time

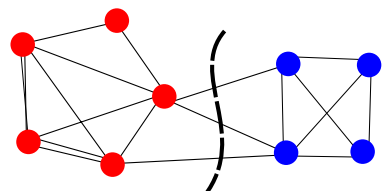
Antoine El-Hayek
ISTA

Monika Henzinger
ISTA

Jason Li
CMU

The Minimum Cut Problem

- Given a Graph $G = (V, E)$, partition V into V_1 and V_2 minimizing the number of crossing edges.
- Parallel edges are allowed
- Dynamic updates: Insertion and deletion of edges
- We denote by λ the value of the minimum cut.



Minimum Cut results

In the **static** case:

Reference	**Running Time**	**Deterministic**	**Weighted**
Karger (2000)	Near-linear time	No	Yes
Kawarabayashi, Thorup (2015)	Near-linear time	Yes	No
Henzinger, Rao, Wang (2020)	Near-linear time	Yes	No
Henzinger, Li, Rao, Wang (2024)	Near-linear time	Yes	Yes

Dynamic Minimum Cut Results

For **Dynamic Exact** minimum cut:

Reference	**Upd. Time**	**Deterministic**	**Conditions**
Henzinger (1997)	$O(\lambda \log n)$	Yes	Incremental
Thorup (2007)	$\tilde{O}(\lambda^{14.5}\sqrt{n})$	Yes	None
Goranci, Henzinger, Thorup (2018)	$\tilde{O}(1)$	Yes	Incremental
Goranci, Henzinger, Nanongkai, Saranurak, Thorup, Wulff-Nilsen (2023)	$\tilde{O}(m^{30/31})$	Yes	None
Jin, Sun, Thorup (2024)	$n^{o(1)}$	Yes	$\lambda \leq (\log n)^{o(1)}$
de Vos, Christiansen (2025)	$\tilde{O}(\lambda^{5.5}\sqrt{n})$, $\tilde{O}(m^{11/12})$	Yes	None
El-Hayek, Henzinger, Li (2026)	$n^{o(1)}$	Yes	$\lambda \leq 2^{\Theta(\log^{3/4-c} n)}$

Dynamic Minimum Cut Results

For **Dynamic Approximate** minimum cut:

Reference	**Upd. Time**	**Approx. Ratio**	**Weighted**	**Det.**
Thorup, Karger (2000)	$\tilde{O}(\lambda^2)$	$\sqrt{2+o(1)}$	No	Yes
Thorup (2007)	$\tilde{O}(\sqrt{n})$	$1+\frac{1}{\log^{0.05} n}$	No	Yes
Goranci, Räcke, Saranurak, Tan (2021)	$n^{o(1)}$	$n^{o(1)}$	No	Yes
van den Brand, Chen, Kyng, Liu, Meierhans, Gutenberg, Sachdeva (2024)	$n^{o(1)}$	$n^{o(1)}$	Yes	Yes
El-Hayek, Henzinger, Li (2025)	$n^{o(1)}$	$1+\frac{1}{\log^{0.25} n}$	No	No
El-Hayek, Henzinger, Li (2026)	$n^{o(1)}$	$1+2^{-\Theta(\log^{3/4-c} n)}$	Yes	No

Derandomising LocalKCut

LocalKCut is a data structure:

- **Preprocessing**:
  - Input: An unweighted graph G, a volume ν, a maximum cut value $\lambda_{\max}$
  - Running time: $m^{1+o(1)}$
- **Query**:
  - Input: a vertex $v \in G$
  - Output: a set $\mathcal{S}$ of **all** cuts S that satisfy:
    - $v \in S$
    - $\mathrm{Vol}(S) \leq \nu$
    - $\mathrm{Cut}(S) \leq \lambda_{\max}$
  - Query time: $n^{o(1)}$

Where $\mathrm{Vol}(S) = \sum_{v \in S} \deg(v)$

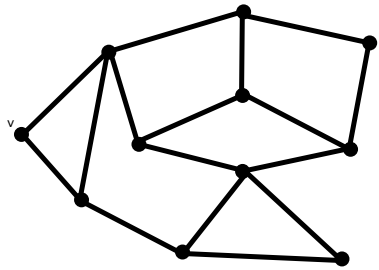


Figure: An example with $\nu = 9$ and $\lambda_{\max} = 3$

Derandomising LocalKCut

LocalKCut is a data structure:

- **Preprocessing**:
  - Input: An unweighted graph G, a volume ν, a maximum cut value $\lambda_{\max}$
  - Running time: $m^{1+o(1)}$
- **Query**:
  - Input: a vertex $v \in G$
  - Output: a set $\mathcal{S}$ of **all** cuts S that satisfy:
    - $v \in S$
    - $\mathrm{Vol}(S) \leq \nu$
    - $\mathrm{Cut}(S) \leq \lambda_{\max}$
  - Query time: $n^{o(1)}$

Where $\mathrm{Vol}(S) = \sum_{v \in S} \deg(v)$

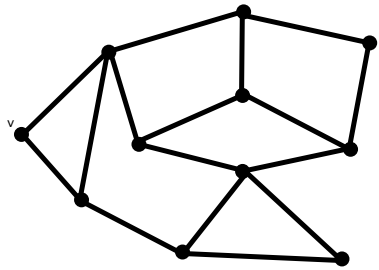


Figure: An example with $\nu = 9$ and $\lambda_{\max} = 3$

Derandomising LocalKCut

LocalKCut is a data structure:

- **Preprocessing**:
  - Input: An unweighted graph G, a volume ν, a maximum cut value $\lambda_{\max}$
  - Running time: $m^{1+o(1)}$
- **Query**:
  - Input: a vertex $v \in G$
  - Output: a set $\mathcal{S}$ of **all** cuts S that satisfy:
    - $v \in S$
    - $\mathrm{Vol}(S) \leq \nu$
    - $\mathrm{Cut}(S) \leq \lambda_{\max}$
  - Query time: $n^{o(1)}$

Where $\mathrm{Vol}(S) = \sum_{v \in S} \deg(v)$

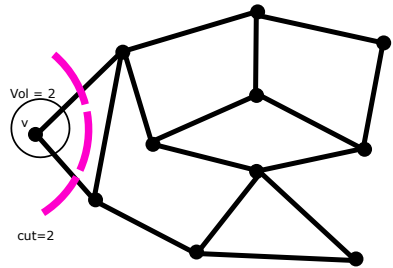


Figure: An example with $\nu = 9$ and $\lambda_{\max} = 3$

Derandomising LocalKCut

LocalKCut is a data structure:

- **Preprocessing**:
  - Input: An unweighted graph G, a volume ν, a maximum cut value $\lambda_{\max}$
  - Running time: $m^{1+o(1)}$
- **Query**:
  - Input: a vertex $v \in G$
  - Output: a set $\mathcal{S}$ of **all** cuts S that satisfy:
    - $v \in S$
    - $\mathrm{Vol}(S) \leq \nu$
    - $\mathrm{Cut}(S) \leq \lambda_{\max}$
  - Query time: $n^{o(1)}$

Where $\mathrm{Vol}(S) = \sum_{v \in S} \deg(v)$

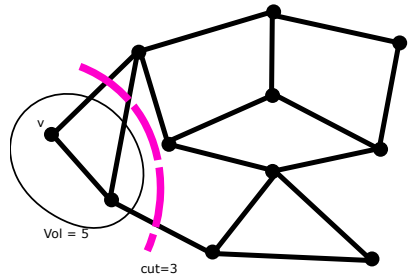


Figure: An example with $\nu = 9$ and $\lambda_{\max} = 3$

Derandomising LocalKCut

LocalKCut is a data structure:

- **Preprocessing**:
  - Input: An unweighted graph G, a volume ν, a maximum cut value $\lambda_{\max}$
  - Running time: $m^{1+o(1)}$
- **Query**:
  - Input: a vertex $v \in G$
  - Output: a set S of **all** cuts S that satisfy:
    - $v \in S$
    - $\mathrm{Vol}(S) \leq \nu$
    - $\mathrm{Cut}(S) \leq \lambda_{\max}$
  - Query time: $n^{o(1)}$

Where $\mathrm{Vol}(S) = \sum_{v \in S} \deg(v)$

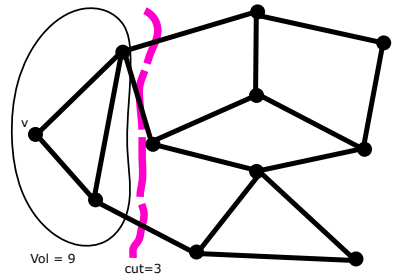
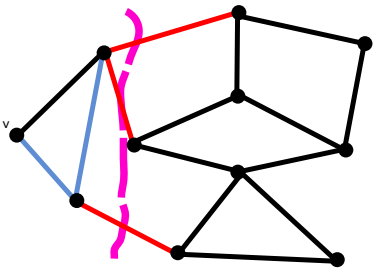


Figure: An example with $\nu = 9$ and $\lambda_{\max} = 3$

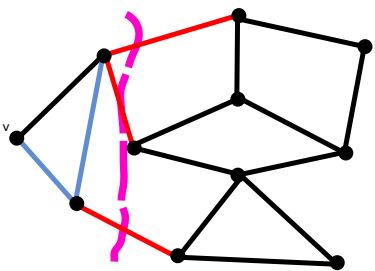
Derandomising LocalKCut: Main Idea



- Assume we have a coloring of the edges into blue and red edges such that:
  - There exists a tree in S whose edges are all blue
  - The edges crossing S are all red

  then a simple BFS starting at v easily finds S in time $O(\nu)$.

Derandomising LocalKCut: Main Idea



- Assume we have a coloring of the edges into blue and red edges such that:
  - There exists a tree in S whose edges are all blue
  - The edges crossing S are all red

  then a simple BFS starting at v easily finds S in time $O(\nu)$.
- Assume we have $n^{o(1)}$ many colorings with at least one of them satisfying the above property, running a BFS for each coloring also finds S

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

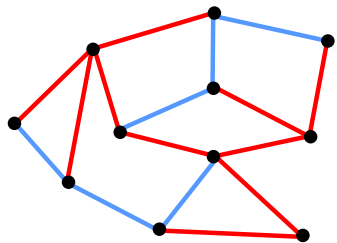


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

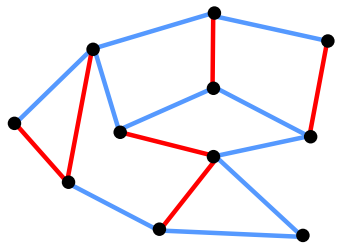


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

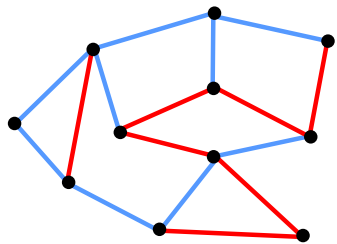


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

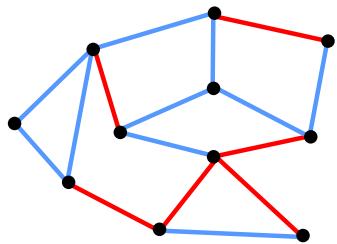


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

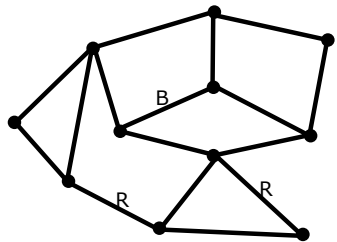


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

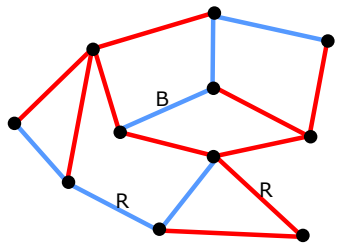


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

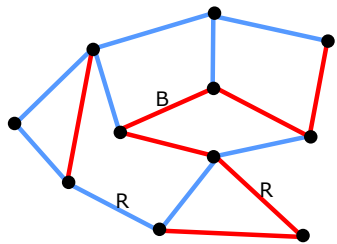


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

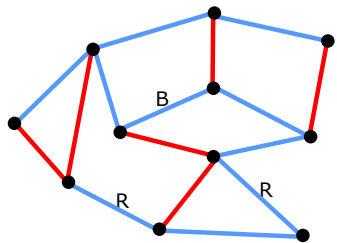


Figure: example for $r = 2$ and $b = 1$

Creating Colorings

Theorem [Chitnis, Cygan, Hajiaghayi, Pilipczuk, Pilipczuk 2016]

Let $G = (V, E)$ be a graph and $r, b \in \mathbb{N}$. There exists a family of $N = (r + b)^{O(\min\{r,b\})} \log m$ red-blue colorings of the edges such that:
for any $R, B \subseteq E$, $R \bigcap B = 0, |R| \leq r, |B| \leq b$, there exists a coloring where all edges in R are red and all edges in B are blue.
This family is computable in $N \cdot m$ deterministic time.

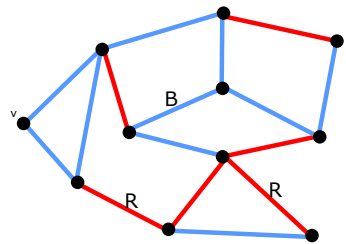
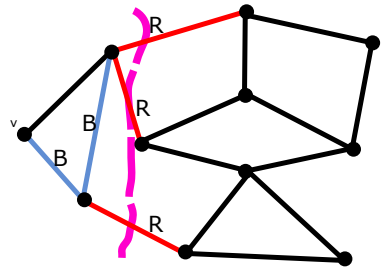


Figure: example for $r = 2$ and $b = 1$

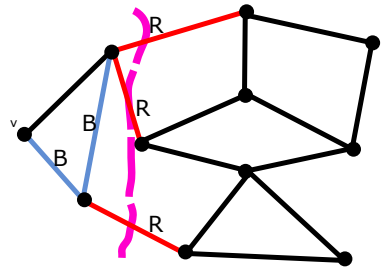
Creating Colorings

- **Attempt 1:** $r = \lambda_{\max}, b = \nu$, deterministic LocalKCut would be trivial:
  - Simply make all edges of a tree in S blue and all edges crossing S red



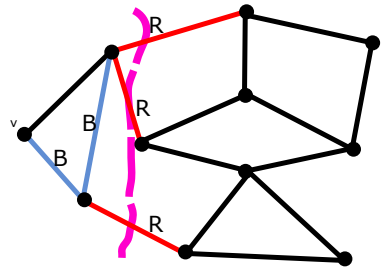
Creating Colorings

- **Attempt 1:** $r = \lambda_{\max}, b = \nu$, deterministic LocalKCut would be trivial:
  - Simply make all edges of a tree in S blue and all edges crossing S red
  - But then $N = (\lambda_{\max} + \nu)^{O(\min\{\lambda_{\max}, \nu\})} \cdot \log m >> n^{o(1)}$



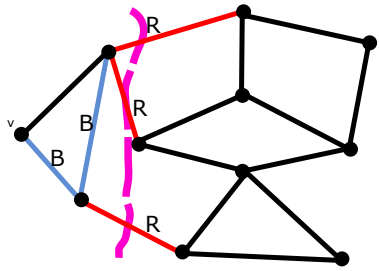
Creating Colorings

- **Attempt 1:** $r = \lambda_{\max}, b = \nu$, deterministic LocalKCut would be trivial:
  - Simply make all edges of a tree in S blue and all edges crossing S red
  - But then $N = (\lambda_{\max} + \nu)^{O(\min\{\lambda_{\max}, \nu\})} \cdot \log m >> n^{o(1)}$
- **Attempt 2:** we create a red-blue coloring with $r = 2\beta, b = \nu$, and a yellow-green coloring with $r = y = \lambda_{\max}$, $b = g = 2\beta$



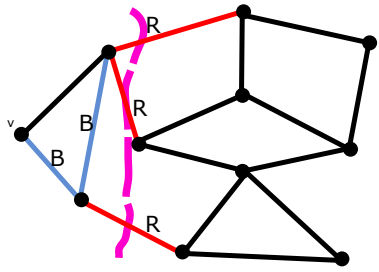
Creating Colorings

- **Attempt 1:** $r = \lambda_{\max}, b = \nu$, deterministic LocalKCut would be trivial:
  - Simply make all edges of a tree in S blue and all edges crossing S red
  - But then $N = (\lambda_{\max} + \nu)^{O(\min\{\lambda_{\max}, \nu\})} \cdot \log m >> n^{o(1)}$
- **Attempt 2:** we create a red-blue coloring with $r = 2\beta, b = \nu$, and a yellow-green coloring with $r = y = \lambda_{\max}$, $b = g = 2\beta$
  - $\beta = O(1)$ is a constant to determine later.
  - $N = \nu^{O(1)} = n^{o(1)}$, $N = {\lambda_{\max}}^{O(1)} = n^{o(1)}$ respectively

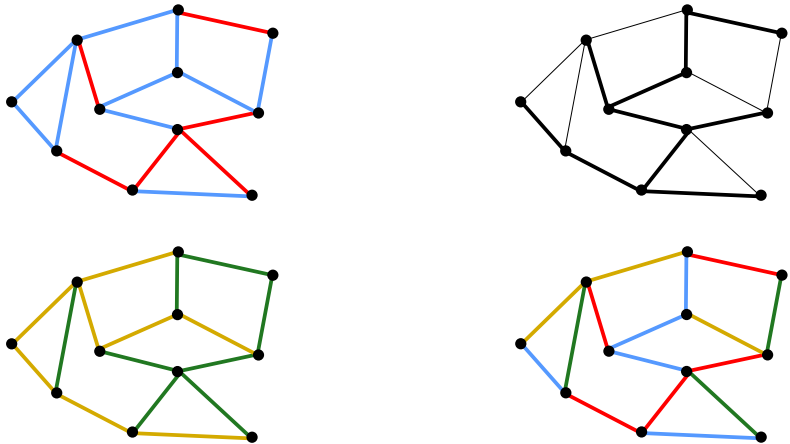


Creating Colorings

- **Attempt 1:** $r = \lambda_{\max}, b = \nu$, deterministic LocalKCut would be trivial:
  - Simply make all edges of a tree in S blue and all edges crossing S red
  - But then $N = (\lambda_{\max} + \nu)^{O(\min\{\lambda_{\max}, \nu\})} \cdot \log m >> n^{o(1)}$
- **Attempt 2:** we create a red-blue coloring with $r = 2\beta, b = \nu$, and a yellow-green coloring with $r = y = \lambda_{\max}$, $b = g = 2\beta$
  - $\beta = O(1)$ is a constant to determine later.
  - $N = \nu^{O(1)} = n^{o(1)}$, $N = {\lambda_{\max}}^{O(1)} = n^{o(1)}$ respectively
- We need to stitch the colorings together, using **Greedy Tree Packings**

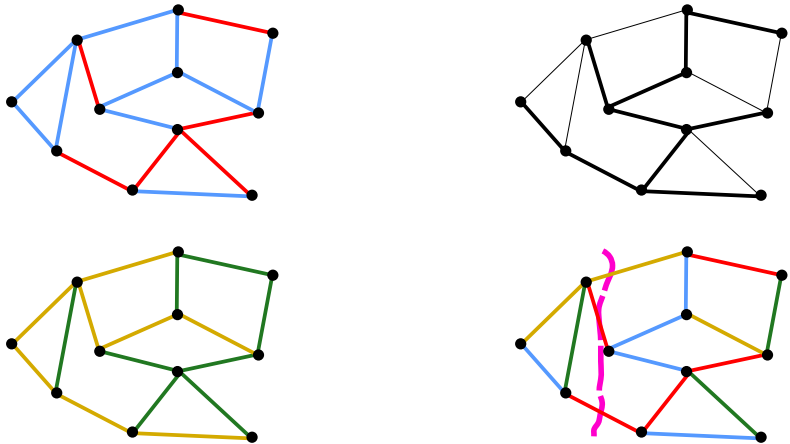


Trees and Coloring



Edges in the tree are colored in Red and Blue, others in Yellow and Green

Trees and Coloring

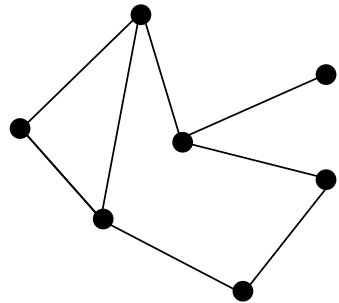


The BFS will run on two colors: Blue and Green, ignoring Red and Yellow

Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

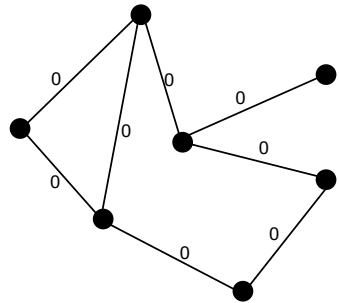
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

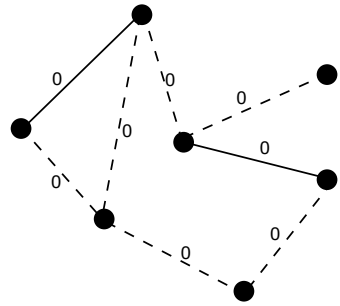
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

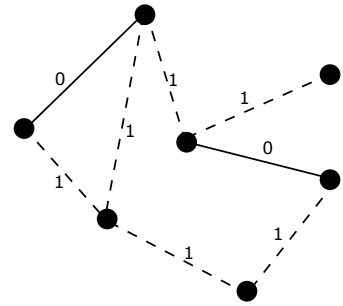
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

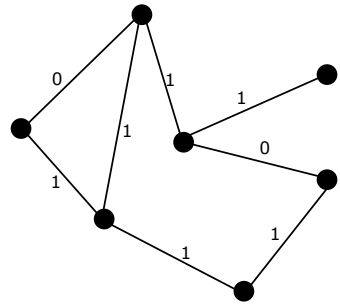
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

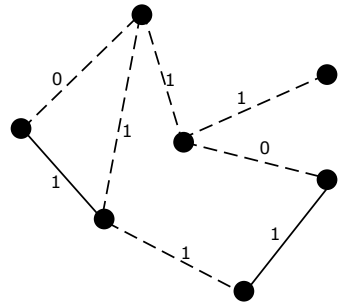
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

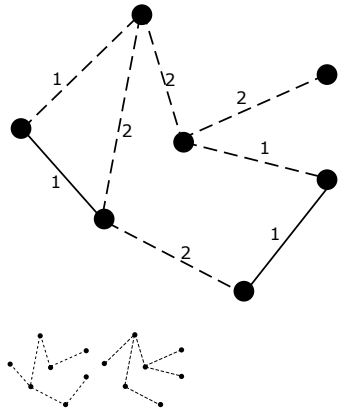
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

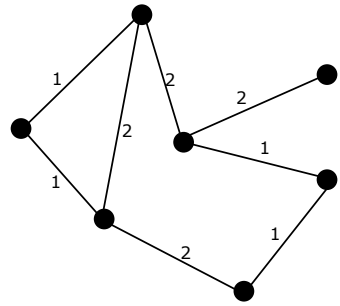
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

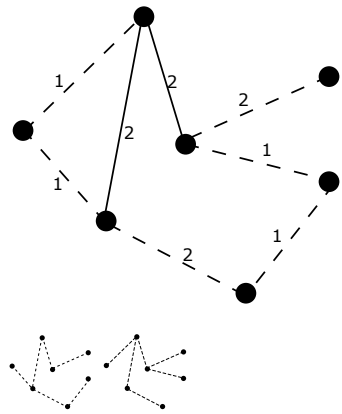
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

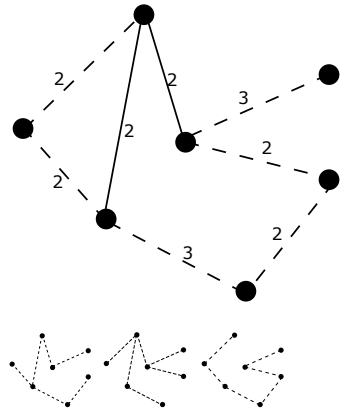
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

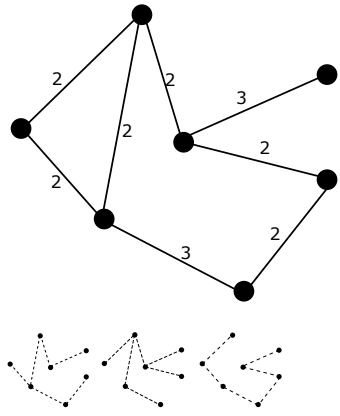
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

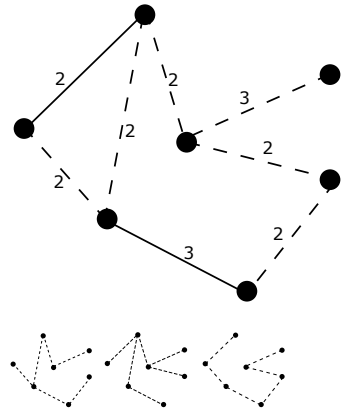
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

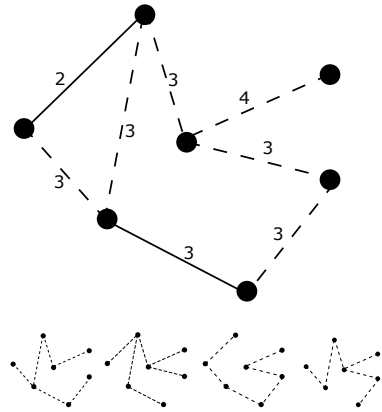
- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Greedy Tree Packing

A greedy tree packing of a graph G with k trees is a set of k trees obtained as follows:

- Set the weights of all the edges of G to 0
- Repeat k times:
  - Find a minimum spanning tree T of G
  - Add T to the packing
  - Increase the weight of every edge of T by one in G
- Return the packing



Trees k-respecting a cut

Definition

In a graph G, a tree is said to k-respect a cut if there are at most k edges crossing the cut.

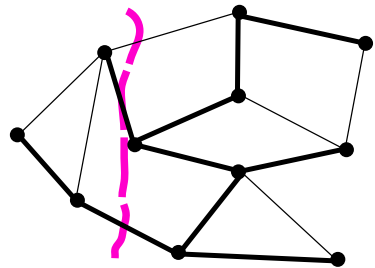


Figure: This tree 2-respects the cut

Trees k-respecting a cut

Definition

In a graph G, a tree is said to k-respect a cut if there are at most k edges crossing the cut.

Theorem [Karger 2000]

Let S be a minimum cut. If we choose $\Theta(\lambda \log m)$ trees in the greedy tree packing, there exists a tree that 2-respects S.

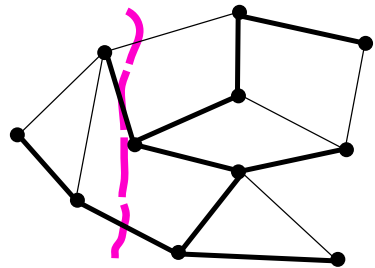


Figure: This tree 2-respects the cut

Trees k-respecting a cut

Definition
In a graph G, a tree is said to k-respect a cut if there are at most k edges crossing the cut.

Theorem [Karger 2000]
Let S be a minimum cut. If we choose $\Theta(\lambda \log m)$ trees in the greedy tree packing, there exists a tree that 2-respects S.

Theorem
Let S be a β-approximate minimum cut. If we choose $\Theta(\lambda\beta^2 \log m) = n^{o(1)}$ trees in the greedy tree packing, there exists a tree that 2β-respects S.

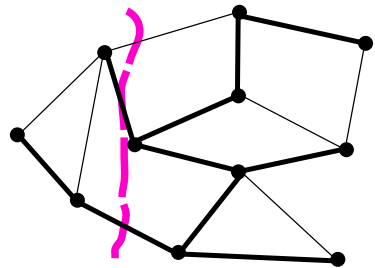
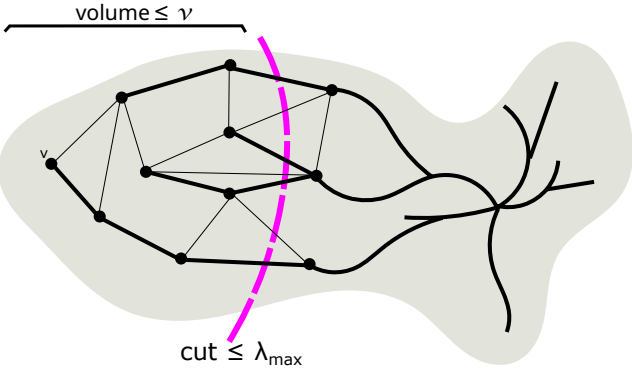


Figure: This tree 2-respects the cut

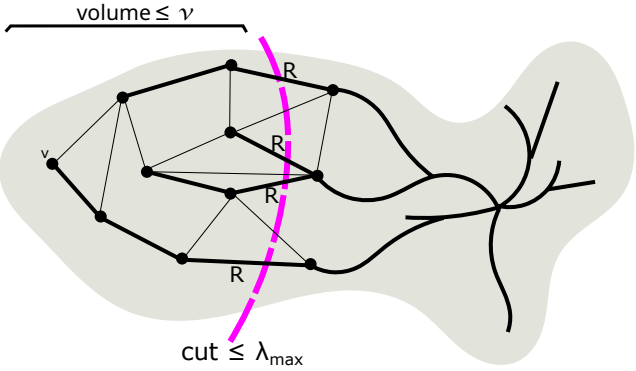
Putting things together

We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



Putting things together

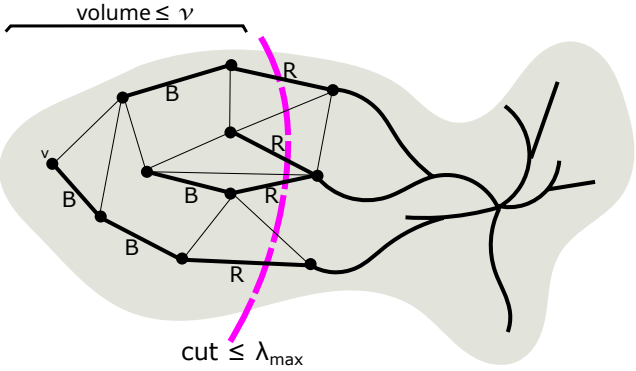
We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



The tree is 2β-respecting the cut
$\Rightarrow$ We can label all tree edges crossing the cut with "R"

Putting things together

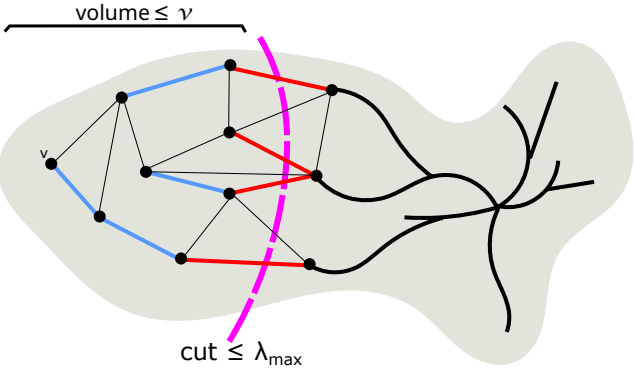
We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



The volume of the tree inside S is at most ν
$\Rightarrow$ We can label all tree edges inside S with "B"

Putting things together

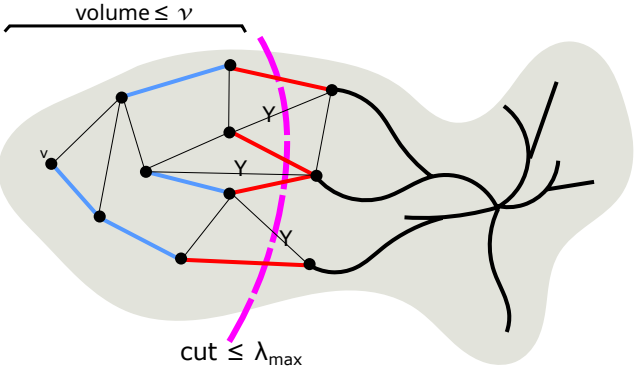
We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



We can find a red-blue coloring that fits the labels

Putting things together

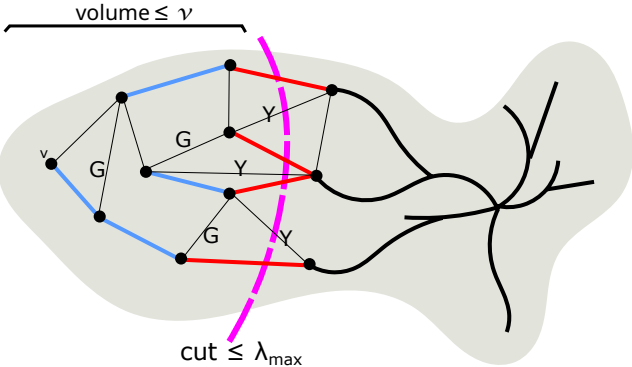
We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



The total number of edges crossing the cut is at most $\lambda_{\max}$
$\Rightarrow$ We can label all non-tree edges crossing S with "Y"

Putting things together

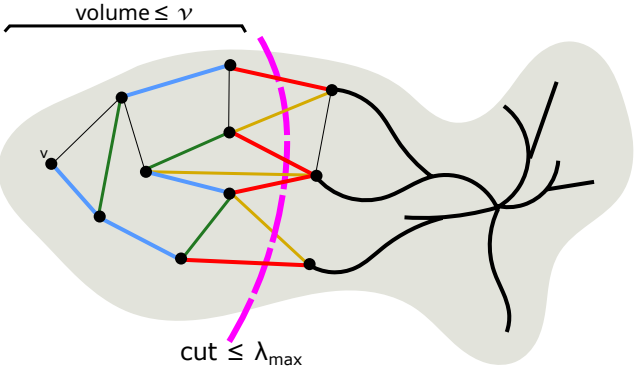
We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



The blue edges inside S form at most 2β components
$\Rightarrow$ We can connect them with at most 2β many edges outside the tree, labeling them "G"

Putting things together

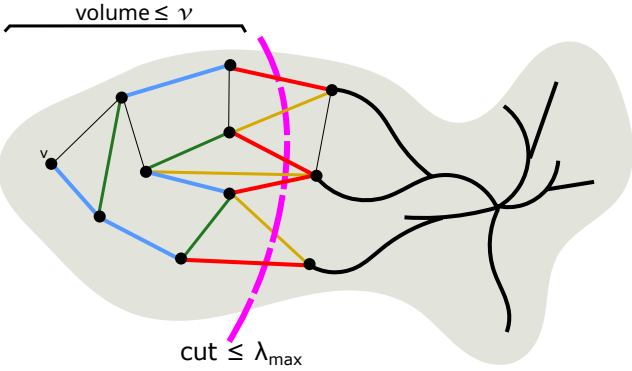
We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



We can find a yellow-green coloring that fits the labels

Putting things together

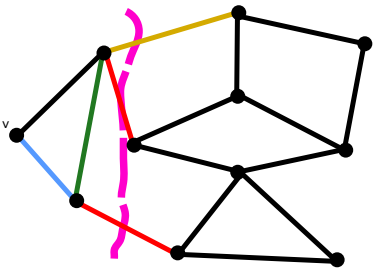
We can choose $r = 2\beta$ red edges, $b = \nu$ blue edges, $y = \lambda_{\max}$ yellow edges, and $g = 2\beta$ green edges.



The Deterministic LocalKCut Data Structure

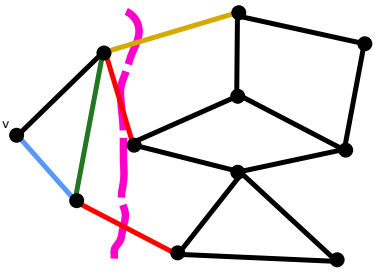
- Preprocessing:
  - Compute Red-Blue and Yellow-Green Colorings
  - Compute a Greedy Tree Packing
- Upon Request:
  - For every (R-B coloring, Y-G coloring, Tree):
    - Color all edges in the tree according to the R-B coloring
    - Color all edges outside the tree according to the Y-G coloring
    - Do a BFS on Blue and Green edges, starting on v, limiting the volume explored to ν
    - If the BFS ends on a set that satisfies the required conditions, add it to the set of cuts to return
- Return the set found

Request time Analysis



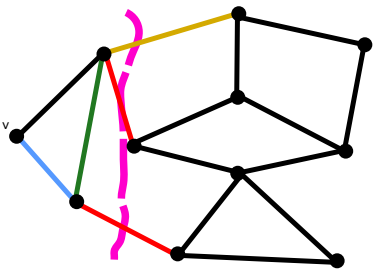
- We have:
  - $n^{o(1)}$ many red-blue colorings
  - $n^{o(1)}$ many yellow-green colorings
  - $n^{o(1)}$ many trees

Request time Analysis



- We have:
  - $n^{o(1)}$ many red-blue colorings
  - $n^{o(1)}$ many yellow-green colorings
  - $n^{o(1)}$ many trees
- For each tuple of two colorings and a tree, we construct a red-blue-yellow-green coloring

Request time Analysis



- We have:
  - $n^{o(1)}$ many red-blue colorings
  - $n^{o(1)}$ many yellow-green colorings
  - $n^{o(1)}$ many trees
- For each tuple of two colorings and a tree, we construct a red-blue-yellow-green coloring
- Overall, we need to run $\left(n^{o(1)}\right)^3$ BFS each of which runs in $n^{o(1)}$ time.

Conclusion

Theorem 1

For **unweighted** graphs, we give a **deterministic** fully-dynamic algorithm that can maintain the **exact** minimum cut in $n^{o(1)}$ update time as long as $\lambda \leq 2^{\Theta(\log^{3/4-c} n)}$.

Theorem 2

For **positively, real-weighted** graphs, we give a **randomised** fully-dynamic algorithm that can maintain a $\left(1 + 2^{-\Theta(\log^{3/4-c} n)}\right)$-**approximate** minimum cut in $n^{o(1)}$ update time.

Open questions:

1. Can we extend the exact algorithm to all values of λ?
2. What about the directed case?
3. What about the exact weighted case? the deterministic weighted case?