Minimum Cut in $\mathcal{O}(m \log^2 n)$ Time

Paweł Gawrychowski[1] Shay Mozes[2] Oren Weimann[3]

[1]University of Wrocław, Poland

[2]The Interdisciplinary Center Herzliya, Israel

[3]University of Haifa, Israel

Slides by Paweł Gawrychowski1

(Global) Minimum Cut

Input: undirected edge-weighted graph $G = (V, E)$

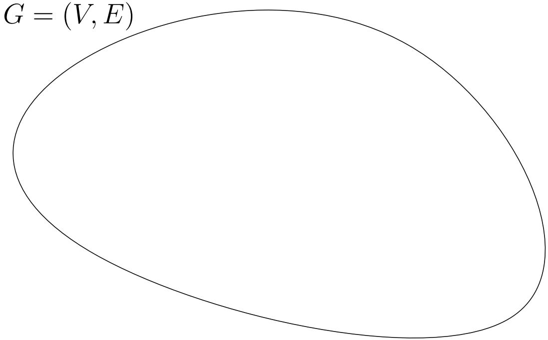
Output: nonempty $S \subset V$ minimizing the total weight of edges between S and $V \setminus S$

Solvable in polynomial time with $n - 1$ maximum flow computations.

(Global) Minimum Cut

Input: undirected edge-weighted graph $G = (V, E)$

Output: nonempty $S \subset V$ minimizing the total weight of edges between S and $V \setminus S$

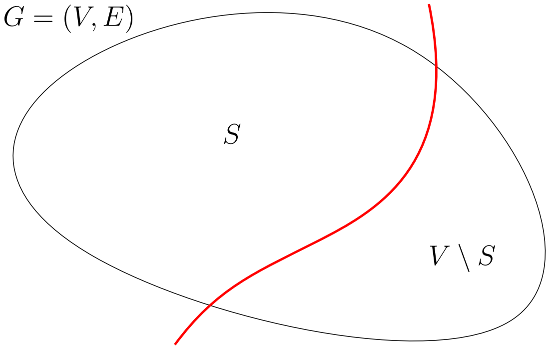


Solvable in polynomial time with $n - 1$ maximum flow computations.

(Global) Minimum Cut

Input: undirected edge-weighted graph $G = (V, E)$

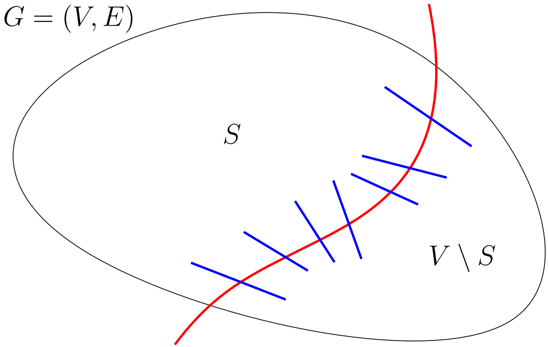
Output: nonempty $S \subset V$ minimizing the total weight of edges between S and $V \setminus S$



Solvable in polynomial time with $n - 1$ maximum flow computations.

(Global) Minimum Cut

Input: undirected edge-weighted graph $G = (V, E)$
Output: nonempty $S \subset V$ minimizing the total weight of edges between S and $V \setminus S$

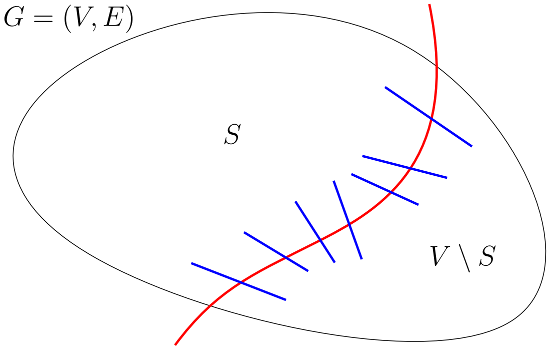


Solvable in polynomial time with $n - 1$ maximum flow computations.

(Global) Minimum Cut

Input: undirected edge-weighted graph $G = (V, E)$

Output: nonempty $S \subset V$ minimizing the total weight of edges between S and $V \setminus S$



Solvable in polynomial time with $n - 1$ maximum flow computations.

The fastest maximum flow algorithm works in $\mathcal{O}(nm)$ time, so applying it $n-1$ times results in $\mathcal{O}(n^2 m)$ complexity. Is there a faster algorithm?

Hao and Orlin 1994

Roughly speaking, a single maximum flow computation suffices, resulting in $\mathcal{O}(mn \log(n^2/m))$ complexity.

Nagamochi and Ibaraki 1992

In $\mathcal{O}(m + n \log n)$ time we can either find the global minimum cut or isolate an edge that doesn't cross it. This edge can then be contracted and the procedure repeated, resulting in $\mathcal{O}(mn + n^2 \log n)$ complexity.

Karger and Stein 1996

A different method based on recursion and contracting a randomly chosen edge finds the global minimum cut in $\mathcal{O}(n^2 \log^3 n)$ time with high probability.

Is there a more efficient algorithm for sparse graphs?

The fastest maximum flow algorithm works in $\mathcal{O}(nm)$ time, so applying it $n-1$ times results in $\mathcal{O}(n^2 m)$ complexity. Is there a faster algorithm?

Hao and Orlin 1994

Roughly speaking, a single maximum flow computation suffices, resulting in $\mathcal{O}(mn\log(n^2/m))$ complexity.

Nagamochi and Ibaraki 1992

In $\mathcal{O}(m + n\log n)$ time we can either find the global minimum cut or isolate an edge that doesn't cross it. This edge can then be contracted and the procedure repeated, resulting in $\mathcal{O}(mn + n^2\log n)$ complexity.

Karger and Stein 1996

A different method based on recursion and contracting a randomly chosen edge finds the global minimum cut in $\mathcal{O}(n^2\log^3 n)$ time with high probability.

Is there a more efficient algorithm for sparse graphs?

The fastest maximum flow algorithm works in $\mathcal{O}(nm)$ time, so applying it $n-1$ times results in $\mathcal{O}(n^2 m)$ complexity. Is there a faster algorithm?

Hao and Orlin 1994

Roughly speaking, a single maximum flow computation suffices, resulting in $\mathcal{O}(mn \log(n^2/m))$ complexity.

Nagamochi and Ibaraki 1992

In $\mathcal{O}(m + n \log n)$ time we can either find the global minimum cut or isolate an edge that doesn't cross it. This edge can then be contracted and the procedure repeated, resulting in $\mathcal{O}(mn + n^2 \log n)$ complexity.

Karger and Stein 1996

A different method based on recursion and contracting a randomly chosen edge finds the global minimum cut in $\mathcal{O}(n^2 \log^3 n)$ time with high probability.

Is there a more efficient algorithm for sparse graphs?

The fastest maximum flow algorithm works in $\mathcal{O}(nm)$ time, so applying it $n-1$ times results in $\mathcal{O}(n^2 m)$ complexity. Is there a faster algorithm?

Hao and Orlin 1994

Roughly speaking, a single maximum flow computation suffices, resulting in $\mathcal{O}(mn\log(n^2/m))$ complexity.

Nagamochi and Ibaraki 1992

In $\mathcal{O}(m+n\log n)$ time we can either find the global minimum cut or isolate an edge that doesn't cross it. This edge can then be contracted and the procedure repeated, resulting in $\mathcal{O}(mn+n^2\log n)$ complexity.

Karger and Stein 1996

A different method based on recursion and contracting a randomly chosen edge finds the global minimum cut in $\mathcal{O}(n^2\log^3 n)$ time with high probability.

Is there a more efficient algorithm for sparse graphs?

The fastest maximum flow algorithm works in $\mathcal{O}(nm)$ time, so applying it $n-1$ times results in $\mathcal{O}(n^2m)$ complexity. Is there a faster algorithm?

Hao and Orlin 1994

Roughly speaking, a single maximum flow computation suffices, resulting in $\mathcal{O}(mn\log(n^2/m))$ complexity.

Nagamochi and Ibaraki 1992

In $\mathcal{O}(m + n\log n)$ time we can either find the global minimum cut or isolate an edge that doesn't cross it. This edge can then be contracted and the procedure repeated, resulting in $\mathcal{O}(mn + n^2\log n)$ complexity.

Karger and Stein 1996

A different method based on recursion and contracting a randomly chosen edge finds the global minimum cut in $\mathcal{O}(n^2\log^3 n)$ time with high probability.

Is there a more efficient algorithm for sparse graphs?

Karger's framework

In 1996, Karger announced a faster $\mathcal{O}(m \log^3 n)$ time algorithm that finds the minimum cut with high probability by solving $\mathcal{O}(\log n)$ independent instances of a more structured problem.

Minimum k-respecting cut

Given a spanning tree T, find the minimum cut crossed by exactly k of its edges.

Karger's framework

In 1996, Karger announced a faster $\mathcal{O}(m \log^3 n)$ time algorithm that finds the minimum cut with high probability by solving $\mathcal{O}(\log n)$ independent instances of a more structured problem.

Minimum k-respecting cut

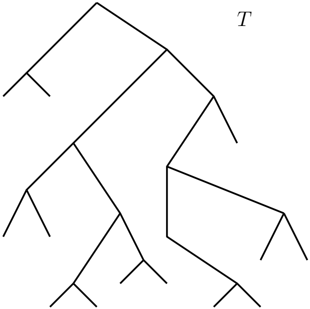
Given a spanning tree T, find the minimum cut crossed by exactly k of its edges.

Karger's framework

In 1996, Karger announced a faster $\mathcal{O}(m \log^3 n)$ time algorithm that finds the minimum cut with high probability by solving $\mathcal{O}(\log n)$ independent instances of a more structured problem.

Minimum k-respecting cut

Given a spanning tree T, find the minimum cut crossed by exactly k of its edges.

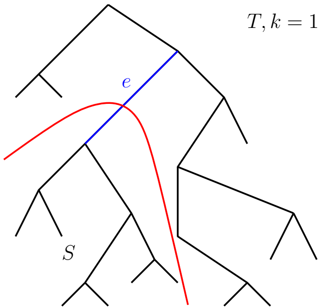


Karger's framework

In 1996, Karger announced a faster $\mathcal{O}(m \log^3 n)$ time algorithm that finds the minimum cut with high probability by solving $\mathcal{O}(\log n)$ independent instances of a more structured problem.

Minimum k-respecting cut

Given a spanning tree T, find the minimum cut crossed by exactly k of its edges.

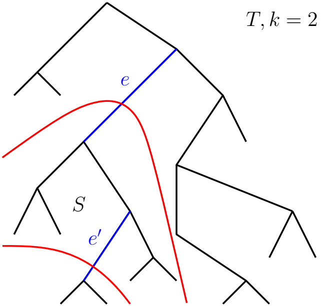


Karger's framework

In 1996, Karger announced a faster $\mathcal{O}(m \log^3 n)$ time algorithm that finds the minimum cut with high probability by solving $\mathcal{O}(\log n)$ independent instances of a more structured problem.

Minimum k-respecting cut

Given a spanning tree T, find the minimum cut crossed by exactly k of its edges.



Karger's framework

Nash-Williams 1961

An unweighted graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees (and clearly not more than c such trees).

If we had such $c/2$ edge-disjoint trees then the average number of edges from the minimum cut per tree is 2, hence the minimum cut 1- or 2-respects one of these trees.

However, this observation was not straightforward to use:

1. One needs to work with weighted graphs.
2. c might be large, and there might be not enough time to find the $c/2$ trees.
3. How to efficiently find the minimum 1- and 2-respecting cut?

Karger's framework

Nash-Williams 1961

An unweighted graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees (and clearly not more than c such trees).

If we had such $c/2$ edge-disjoint trees then the average number of edges from the minimum cut per tree is 2, hence the minimum cut 1- or 2-respects one of these trees.

However, this observation was not straightforward to use:

1. One needs to work with weighted graphs.
2. c might be large, and there might be not enough time to find the $c/2$ trees.
3. How to efficiently find the minimum 1- and 2-respecting cut?

Karger's framework

Nash-Williams 1961

An unweighted graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees (and clearly not more than c such trees).

If we had such $c/2$ edge-disjoint trees then the average number of edges from the minimum cut per tree is 2, hence the minimum cut 1- or 2-respects one of these trees.

However, this observation was not straightforward to use:

1. One needs to work with weighted graphs.
2. c might be large, and there might be not enough time to find the $c/2$ trees.
3. How to efficiently find the minimum 1- and 2-respecting cut?

Karger's framework

Nash-Williams 1961

An unweighted graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees (and clearly not more than c such trees).

If we had such $c/2$ edge-disjoint trees then the average number of edges from the minimum cut per tree is 2, hence the minimum cut 1- or 2-respects one of these trees.

However, this observation was not straightforward to use:

1. One needs to work with weighted graphs.
2. c might be large, and there might be not enough time to find the $c/2$ trees.
3. How to efficiently find the minimum 1- and 2-respecting cut?

Karger's framework

Nash-Williams 1961

An unweighted graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees (and clearly not more than c such trees).

If we had such $c/2$ edge-disjoint trees then the average number of edges from the minimum cut per tree is 2, hence the minimum cut 1- or 2-respects one of these trees.

However, this observation was not straightforward to use:

1. One needs to work with weighted graphs.
2. c might be large, and there might be not enough time to find the $c/2$ trees.
3. How to efficiently find the minimum 1- and 2-respecting cut?

Karger's framework

Nash-Williams 1961

An unweighted graph with minimum cut c contains a set of $c/2$ edge-disjoint spanning trees (and clearly not more than c such trees).

If we had such $c/2$ edge-disjoint trees then the average number of edges from the minimum cut per tree is 2, hence the minimum cut 1- or 2-respects one of these trees.

However, this observation was not straightforward to use:

1. One needs to work with weighted graphs.
2. c might be large, and there might be not enough time to find the $c/2$ trees.
3. How to efficiently find the minimum 1- and 2-respecting cut?

Karger's framework

Weighted tree packing

A set of spanning trees, each with its assigned weight, such that the total weight of all trees containing an edge is at most its weight.

1. Find in $\mathcal{O}(m + n\log n)$ time an unweighted graph H with $\mathcal{O}(n\log n)$ edges and minimum cut $c' = \mathcal{O}(\log n)$, such that the minimum cut in G corresponds to a $7/6$-minimum cut in H.
2. Apply the algorithm of Plotkin-Shmoys-Tardos to find a tree packing of total weight $5/12c'$ in $\mathcal{O}(n\log^3 n)$ time.
3. Fraction of the total weight of all trees in the packing 1- or 2-respected by the minimum cut of G must be at least $1/10$.

Choosing $\mathcal{O}(\log n)$ trees from the packing (with probability equal to the weight) guarantees that w.h.p. we obtain a tree 1- or 2-respected by the minimum cut.

Karger's framework

Weighted tree packing

A set of spanning trees, each with its assigned weight, such that the total weight of all trees containing an edge is at most its weight.

1. Find in $\mathcal{O}(m + n \log n)$ time an unweighted graph H with $\mathcal{O}(n \log n)$ edges and minimum cut $c' = \mathcal{O}(\log n)$, such that the minimum cut in G corresponds to a $7/6$-minimum cut in H.
2. Apply the algorithm of Plotkin-Shmoys-Tardos to find a tree packing of total weight $5/12c'$ in $\mathcal{O}(n \log^3 n)$ time.
3. Fraction of the total weight of all trees in the packing 1- or 2-respected by the minimum cut of G must be at least $1/10$.

Choosing $\mathcal{O}(\log n)$ trees from the packing (with probability equal to the weight) guarantees that w.h.p. we obtain a tree 1- or 2-respected by the minimum cut.

Karger's framework

Weighted tree packing

A set of spanning trees, each with its assigned weight, such that the total weight of all trees containing an edge is at most its weight.

1. Find in $\mathcal{O}(m + n \log n)$ time an unweighted graph H with $\mathcal{O}(n \log n)$ edges and minimum cut $c' = \mathcal{O}(\log n)$, such that the minimum cut in G corresponds to a $7/6$-minimum cut in H.
2. Apply the algorithm of Plotkin-Shmoys-Tardos to find a tree packing of total weight $5/12c'$ in $\mathcal{O}(n \log^3 n)$ time.
3. Fraction of the total weight of all trees in the packing 1- or 2-respected by the minimum cut of G must be at least $1/10$.

Choosing $\mathcal{O}(\log n)$ trees from the packing (with probability equal to the weight) guarantees that w.h.p. we obtain a tree 1- or 2-respected by the minimum cut.

Karger's framework

Weighted tree packing

A set of spanning trees, each with its assigned weight, such that the total weight of all trees containing an edge is at most its weight.

1. Find in $\mathcal{O}(m + n \log n)$ time an unweighted graph H with $\mathcal{O}(n \log n)$ edges and minimum cut $c' = \mathcal{O}(\log n)$, such that the minimum cut in G corresponds to a $7/6$-minimum cut in H.
2. Apply the algorithm of Plotkin-Shmoys-Tardos to find a tree packing of total weight $5/12c'$ in $\mathcal{O}(n \log^3 n)$ time.
3. Fraction of the total weight of all trees in the packing 1- or 2-respected by the minimum cut of G must be at least $1/10$.

Choosing $\mathcal{O}(\log n)$ trees from the packing (with probability equal to the weight) guarantees that w.h.p. we obtain a tree 1- or 2-respected by the minimum cut.

Karger's framework

Weighted tree packing

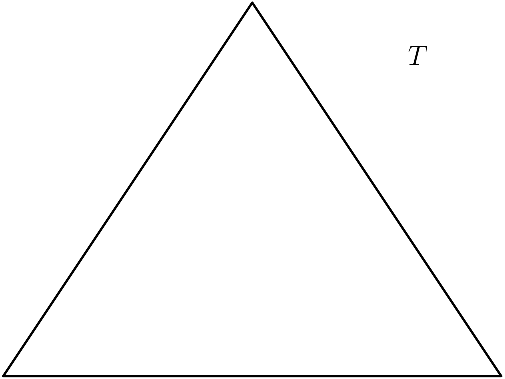
A set of spanning trees, each with its assigned weight, such that the total weight of all trees containing an edge is at most its weight.

1. Find in $\mathcal{O}(m + n \log n)$ time an unweighted graph H with $\mathcal{O}(n \log n)$ edges and minimum cut $c' = \mathcal{O}(\log n)$, such that the minimum cut in G corresponds to a $7/6$-minimum cut in H.
2. Apply the algorithm of Plotkin-Shmoys-Tardos to find a tree packing of total weight $5/12c'$ in $\mathcal{O}(n \log^3 n)$ time.
3. Fraction of the total weight of all trees in the packing 1- or 2-respected by the minimum cut of G must be at least $1/10$.

Choosing $\mathcal{O}(\log n)$ trees from the packing (with probability equal to the weight) guarantees that w.h.p. we obtain a tree 1- or 2-respected by the minimum cut.

1-Respecting Minimum Cut

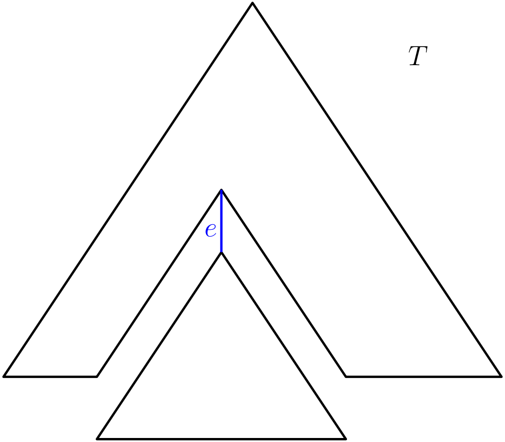
Karger showed that this is actually fairly easy:



This information can be propagated in $\mathcal{O}(m)$ overall time.

1-Respecting Minimum Cut

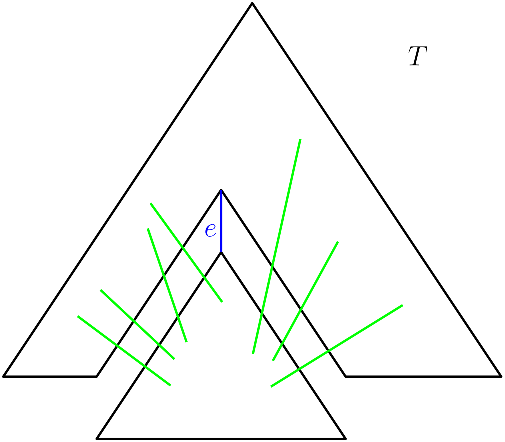
Karger showed that this is actually fairly easy:



This information can be propagated in $\mathcal{O}(m)$ overall time.

1-Respecting Minimum Cut

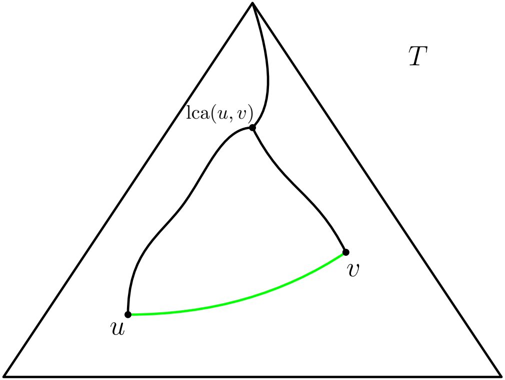
Karger showed that this is actually fairly easy:



This information can be propagated in $\mathcal{O}(m)$ overall time.

1-Respecting Minimum Cut

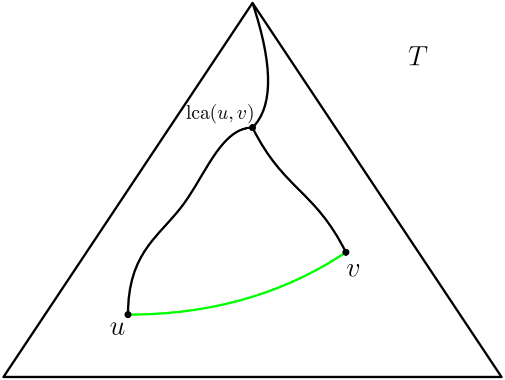
Karger showed that this is actually fairly easy:



This information can be propagated in $\mathcal{O}(m)$ overall time.

1-Respecting Minimum Cut

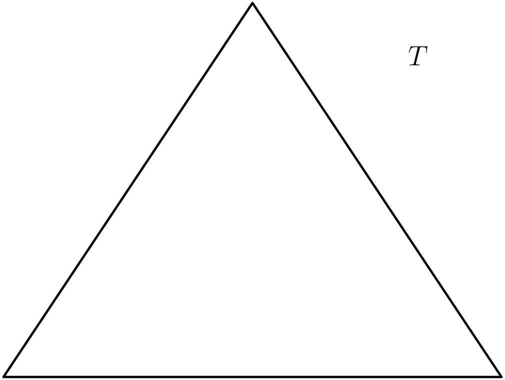
Karger showed that this is actually fairly easy:



This information can be propagated in $\mathcal{O}(m)$ overall time.

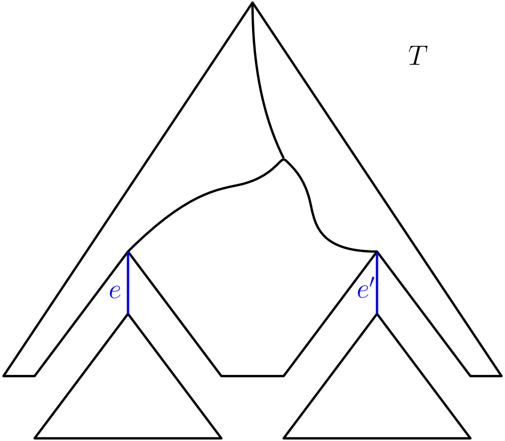
2-Respecting Minimum Cut

Sought e and e' can either be independent or descendant.



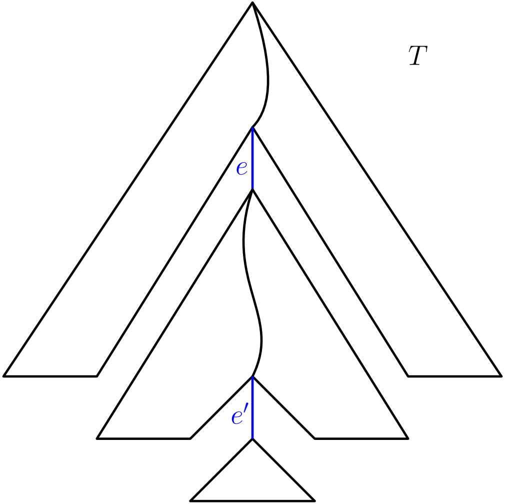
2-Respecting Minimum Cut

Sought e and e' can either be independent or descendant.



2-Respecting Minimum Cut

Sought e and e' can either be independent or descendant.



2-Respecting Minimum Cut

For both cases, Karger's algorithm operates in $\mathcal{O}(\log n)$ phases that iteratively contracting paths (called *boughs*) consisting of nodes with exactly one child terminating at a leaf. Each phase is implemented with link-cut trees in $\mathcal{O}(m \log n)$ time.

Karger 1996

Minimum cut that 2-respects a spanning tree can be found in $\mathcal{O}(m \log^2 n)$ time.

But... no one was able to succeed in doing so!

2-Respecting Minimum Cut

For both cases, Karger's algorithm operates in $\mathcal{O}(\log n)$ phases that iteratively contracting paths (called *boughs*) consisting of nodes with exactly one child terminating at a leaf. Each phase is implemented with link-cut trees in $\mathcal{O}(m \log n)$ time.

Karger 1996

Minimum cut that 2-respects a spanning tree can be found in $\mathcal{O}(m \log^2 n)$ time.

But... no one was able to succeed in doing so!

2-Respecting Minimum Cut

For both cases, Karger's algorithm operates in $\mathcal{O}(\log n)$ phases that iteratively contracting paths (called *boughs*) consisting of nodes with exactly one child terminating at a leaf. Each phase is implemented with link-cut trees in $\mathcal{O}(m \log n)$ time.

Karger 1996

Minimum cut that 2-respects a spanning tree can be found in $\mathcal{O}(m \log^2 n)$ time.

- Give a dynamic path-minimization data structure taking constant amortized time per operation. This would reduce the running time by a $\log n$ factor. We are not using the full power of dynamic trees (in particular, the tree on which we operate is static, and the sequence of operations is known in advance), so this might be possible.
- Extend the approach for boughs directly to trees in order to avoid the $O(\log n)$ different phases needed for pruning boughs. This too would reduce the running time by a $\log n$ factor.

But... no one was able to succeed in doing so!

2-Respecting Minimum Cut

For both cases, Karger's algorithm operates in $\mathcal{O}(\log n)$ phases that iteratively contracting paths (called *boughs*) consisting of nodes with exactly one child terminating at a leaf. Each phase is implemented with link-cut trees in $\mathcal{O}(m \log n)$ time.

Karger 1996

Minimum cut that 2-respects a spanning tree can be found in $\mathcal{O}(m \log^2 n)$ time.

- Give a dynamic path-minimization data structure taking constant amortized time per operation. This would reduce the running time by a $\log n$ factor. We are not using the full power of dynamic trees (in particular, the tree on which we operate is static, and the sequence of operations is known in advance), so this might be possible.
- Extend the approach for boughs directly to trees in order to avoid the $O(\log n)$ different phases needed for pruning boughs. This too would reduce the running time by a $\log n$ factor.

But... no one was able to succeed in doing so!

Recent Developments

Henzinger, Rao, and Wang 2017

A faster $\mathcal{O}(m \log^2 n (\log \log n)^2)$ time deterministic algorithm for simple unweighted graphs.

Ghaffari, Nowicki and Thorup 2020

An even faster $\mathcal{O}(\min\{m + n \log^3 n, m \log n\})$ time algorithm for simple unweighted graphs.

Lovett and Sandlund 2020

A different take on the 2-respecting problem based on top trees instead of link-cut trees, and iterating over the edges of T guided by heavy-light decomposition. Also $\mathcal{O}(m \log^2 n)$ time, but simpler.

Recent Developments

Henzinger, Rao, and Wang 2017

A faster $\mathcal{O}(m \log^2 n (\log \log n)^2)$ time deterministic algorithm for simple unweighted graphs.

Ghaffari, Nowicki and Thorup 2020

An even faster $\mathcal{O}(\min\{m + n \log^3 n, m \log n\})$ time algorithm for simple unweighted graphs.

Lovett and Sandlund 2020

A different take on the 2-respecting problem based on top trees instead of link-cut trees, and iterating over the edges of T guided by heavy-light decomposition. Also $\mathcal{O}(m \log^2 n)$ time, but simpler.

Recent Developments

Henzinger, Rao, and Wang 2017

A faster $\mathcal{O}(m \log^2 n (\log\log n)^2)$ time deterministic algorithm for simple unweighted graphs.

Ghaffari, Nowicki and Thorup 2020

An even faster $\mathcal{O}(\min\{m + n\log^3 n, m\log n\})$ time algorithm for simple unweighted graphs.

Lovett and Sandlund 2020

A different take on the 2-respecting problem based on top trees instead of link-cut trees, and iterating over the edges of T guided by heavy-light decomposition. Also $\mathcal{O}(m\log^2 n)$ time, but simpler.

Our Contribution

1. The main technical contribution is a deterministic $\mathcal{O}(m \log n)$ time solution for the 2-respecting problem.
2. To obtain an improvement on the overall $\mathcal{O}(m \log^3 n)$ complexity of Karger's algorithm, we also design an alternative sampling procedure that produces the $\mathcal{O}(\log n)$ spanning trees in $\mathcal{O}(m \log^2 n)$ instead of $\mathcal{O}(m + n \log^3 n)$ time.
3. This gives us the minimum cut in $\mathcal{O}(m \log^2 n)$ time.

Mukhopadhyay and Nanongkai 2020

Independently of our result, solves the 2-respecting problem in $\mathcal{O}(m \log n + n \log^3 n)$ randomised time.

Our Contribution

1. The main technical contribution is a deterministic $\mathcal{O}(m \log n)$ time solution for the 2-respecting problem.
2. To obtain an improvement on the overall $\mathcal{O}(m \log^3 n)$ complexity of Karger's algorithm, we also design an alternative sampling procedure that produces the $\mathcal{O}(\log n)$ spanning trees in $\mathcal{O}(m \log^2 n)$ instead of $\mathcal{O}(m + n \log^3 n)$ time.
3. This gives us the minimum cut in $\mathcal{O}(m \log^2 n)$ time.

Mukhopadhyay and Nanongkai 2020

Independently of our result, solves the 2-respecting problem in $\mathcal{O}(m \log n + n \log^3 n)$ randomised time.

Our Contribution

1. The main technical contribution is a deterministic $\mathcal{O}(m \log n)$ time solution for the 2-respecting problem.
2. To obtain an improvement on the overall $\mathcal{O}(m \log^3 n)$ complexity of Karger's algorithm, we also design an alternative sampling procedure that produces the $\mathcal{O}(\log n)$ spanning trees in $\mathcal{O}(m \log^2 n)$ instead of $\mathcal{O}(m + n \log^3 n)$ time.
3. This gives us the minimum cut in $\mathcal{O}(m \log^2 n)$ time.

Mukhopadhyay and Nanongkai 2020

Independently of our result, solves the 2-respecting problem in $\mathcal{O}(m \log n + n \log^3 n)$ randomised time.

Our Contribution

1. The main technical contribution is a deterministic $\mathcal{O}(m \log n)$ time solution for the 2-respecting problem.
2. To obtain an improvement on the overall $\mathcal{O}(m \log^3 n)$ complexity of Karger's algorithm, we also design an alternative sampling procedure that produces the $\mathcal{O}(\log n)$ spanning trees in $\mathcal{O}(m \log^2 n)$ instead of $\mathcal{O}(m + n \log^3 n)$ time.
3. This gives us the minimum cut in $\mathcal{O}(m \log^2 n)$ time.

Mukhopadhyay and Nanongkai 2020

Independently of our result, solves the 2-respecting problem in $\mathcal{O}(m \log n + n \log^3 n)$ randomised time.

Faster Sampling

1. Find constant approximation of c using Matula's algorithm. We implement it in $\mathcal{O}(m \log^2 n)$ time.
2. Edges with weight larger than c can be now contracted, and we think of an edge with weight w as w parallel unweighted edges.
3. Sample $\lceil pm \rceil$ (unweighted) edges, where $p = \Theta(\log n)/c$, to obtain graph H with minimum cut $c' = \mathcal{O}(\log n)$. This can be implemented in $\mathcal{O}(mc \cdot \log m) = \mathcal{O}(m \log^2 n)$ time w.h.p.
4. Apply the following specialised instantiation of Young's variant of the Lagrangian packing technique:

```
1: ℓ(e) := 0 for all e ∈ E(H)
2: while there is no e with ℓ(e) ≥ 1 do
3:     find a minimum spanning tree T w.r.t. ℓ(·)
4:     w(T) = w(T) + 1/(96 ln m')
5:     ℓ(e) = ℓ(e) + 1/(96 ln m') for all e ∈ T
6: end while
```

$\mathcal{O}(c' \cdot \log n) = \mathcal{O}(\log^2 n)$ iterations, each in $\mathcal{O}(m)$ time.

Faster Sampling

1. Find constant approximation of c using Matula's algorithm. We implement it in $\mathcal{O}(m \log^2 n)$ time.
2. Edges with weight larger than c can be now contracted, and we think of an edge with weight w as w parallel unweighted edges.
3. Sample $\lceil pm \rceil$ (unweighted) edges, where $p = \Theta(\log n)/c$, to obtain graph H with minimum cut $c' = \mathcal{O}(\log n)$. This can be implemented in $\mathcal{O}(mc \cdot \log m) = \mathcal{O}(m \log^2 n)$ time w.h.p.
4. Apply the following specialised instantiation of Young's variant of the Lagrangian packing technique:

   $\ell(e) := 0$ for all $e \in E(H)$
   while there is no e with $\ell(e) \geq 1$ **do**
       find a minimum spanning tree T w.r.t. $\ell(\cdot)$
       $w(T) = w(T) + 1/(96 \ln m')$
       $\ell(e) = \ell(e) + 1/(96 \ln m')$ for all $e \in T$
   end while

   $\mathcal{O}(c' \cdot \log n) = \mathcal{O}(\log^2 n)$ iterations, each in $\mathcal{O}(m)$ time.

Faster Sampling

1. Find constant approximation of c using Matula's algorithm. We implement it in $\mathcal{O}(m \log^2 n)$ time.
2. Edges with weight larger than c can be now contracted, and we think of an edge with weight w as w parallel unweighted edges.
3. Sample $\lceil pm \rceil$ (unweighted) edges, where $p = \Theta(\log n)/c$, to obtain graph H with minimum cut $c' = \mathcal{O}(\log n)$. This can be implemented in $\mathcal{O}(mc \cdot \log m) = \mathcal{O}(m \log^2 n)$ time w.h.p.
4. Apply the following specialised instantiation of Young's variant of the Lagrangian packing technique:

   1: $\ell(e) := 0$ for all $e \in E(H)$
   2: **while** there is no e with $\ell(e) \geq 1$ **do**
   3:     find a minimum spanning tree T w.r.t. $\ell(\cdot)$
   4:     $w(T) = w(T) + 1/(96 \ln m')$
   5:     $\ell(e) = \ell(e) + 1/(96 \ln m')$ for all $e \in T$
   6: **end while**

   $\mathcal{O}(c' \cdot \log n) = \mathcal{O}(\log^2 n)$ iterations, each in $\mathcal{O}(m)$ time.

Faster Sampling

1. Find constant approximation of c using Matula's algorithm. We implement it in $\mathcal{O}(m \log^2 n)$ time.
2. Edges with weight larger than c can be now contracted, and we think of an edge with weight w as w parallel unweighted edges.
3. Sample $\lceil pm \rceil$ (unweighted) edges, where $p = \Theta(\log n)/c$, to obtain graph H with minimum cut $c' = \mathcal{O}(\log n)$. This can be implemented in $\mathcal{O}(mc \cdot \log m) = \mathcal{O}(m \log^2 n)$ time w.h.p.
4. Apply the following specialised instantiation of Young's variant of the Lagrangian packing technique:

   $\ell(e) := 0$ for all $e \in E(H)$
   while there is no e with $\ell(e) \geq 1$ **do**
       find a minimum spanning tree T w.r.t. $\ell(\cdot)$
       $w(T) = w(T) + 1/(96 \ln m')$
       $\ell(e) = \ell(e) + 1/(96 \ln m')$ for all $e \in T$
   end while

   $\mathcal{O}(c' \cdot \log n) = \mathcal{O}(\log^2 n)$ iterations, each in $\mathcal{O}(m)$ time.

Faster Sampling

1. Find constant approximation of c using Matula's algorithm. We implement it in $\mathcal{O}(m \log^2 n)$ time.
2. Edges with weight larger than c can be now contracted, and we think of an edge with weight w as w parallel unweighted edges.
3. Sample $\lceil pm \rceil$ (unweighted) edges, where $p = \Theta(\log n)/c$, to obtain graph H with minimum cut $c' = \mathcal{O}(\log n)$. This can be implemented in $\mathcal{O}(mc \cdot \log m) = \mathcal{O}(m \log^2 n)$ time w.h.p.
4. Apply the following specialised instantiation of Young's variant of the Lagrangian packing technique:

```
1: ℓ(e) := 0 for all e ∈ E(H)
2: while there is no e with ℓ(e) ≥ 1 do
3:     find a minimum spanning tree T w.r.t. ℓ(·)
4:     w(T) = w(T) + 1/(96 ln m')
5:     ℓ(e) = ℓ(e) + 1/(96 ln m') for all e ∈ T
6: end while
```

$\mathcal{O}(c' \cdot \log n) = \mathcal{O}(\log^2 n)$ iterations, each in $\mathcal{O}(m)$ time.

Overview

We obtain in $\mathcal{O}(m \log^2 n)$ time a collection of $\mathcal{O}(\log n)$ spanning trees $T_1, T_2, \ldots$ such that the minimum cut 1- or 2-respects some T_i.

Now we iterate over every tree T_i and:

1. Find the minimum 1-respecting cut in $\mathcal{O}(m)$ time.
2. Find the minimum 2-respecting cut defined by dependent edges in $\mathcal{O}(m \log n)$ time.
3. To find the minimum 2-respecting cut defined by independent edges, obtain in $\mathcal{O}(m \log n)$ time a number of instances of a bipartite problem of total size $\mathcal{O}(m)$.
4. Solve each size-s instance of a bipartite problem in $\mathcal{O}(s \log s)$ time.

This sums up to $\mathcal{O}(m \log^2 n)$ as promised.

Overview

We obtain in $\mathcal{O}(m \log^2 n)$ time a collection of $\mathcal{O}(\log n)$ spanning trees $T_1, T_2, \ldots$ such that the minimum cut 1- or 2-respects some T_i.

Now we iterate over every tree T_i and:

1. Find the minimum 1-respecting cut in $\mathcal{O}(m)$ time.
2. Find the minimum 2-respecting cut defined by dependent edges in $\mathcal{O}(m \log n)$ time.
3. To find the minimum 2-respecting cut defined by independent edges, obtain in $\mathcal{O}(m \log n)$ time a number of instances of a bipartite problem of total size $\mathcal{O}(m)$.
4. Solve each size-s instance of a bipartite problem in $\mathcal{O}(s \log s)$ time.

This sums up to $\mathcal{O}(m \log^2 n)$ as promised.

Overview

We obtain in $\mathcal{O}(m \log^2 n)$ time a collection of $\mathcal{O}(\log n)$ spanning trees $T_1, T_2, \ldots$ such that the minimum cut 1- or 2-respects some T_i.

Now we iterate over every tree T_i and:

1. Find the minimum 1-respecting cut in $\mathcal{O}(m)$ time.
2. Find the minimum 2-respecting cut defined by dependent edges in $\mathcal{O}(m \log n)$ time.
3. To find the minimum 2-respecting cut defined by independent edges, obtain in $\mathcal{O}(m \log n)$ time a number of instances of a bipartite problem of total size $\mathcal{O}(m)$.
4. Solve each size-s instance of a bipartite problem in $\mathcal{O}(s \log s)$ time.

This sums up to $\mathcal{O}(m \log^2 n)$ as promised.

Overview

We obtain in $\mathcal{O}(m \log^2 n)$ time a collection of $\mathcal{O}(\log n)$ spanning trees $T_1, T_2, \ldots$ such that the minimum cut 1- or 2-respects some T_i.

Now we iterate over every tree T_i and:

1. Find the minimum 1-respecting cut in $\mathcal{O}(m)$ time.
2. Find the minimum 2-respecting cut defined by dependent edges in $\mathcal{O}(m \log n)$ time.
3. To find the minimum 2-respecting cut defined by independent edges, obtain in $\mathcal{O}(m \log n)$ time a number of instances of a bipartite problem of total size $\mathcal{O}(m)$.
4. Solve each size-s instance of a bipartite problem in $\mathcal{O}(s \log s)$ time.

This sums up to $\mathcal{O}(m \log^2 n)$ as promised.

Overview

We obtain in $\mathcal{O}(m \log^2 n)$ time a collection of $\mathcal{O}(\log n)$ spanning trees $T_1, T_2, \ldots$ such that the minimum cut 1- or 2-respects some T_i.

Now we iterate over every tree T_i and:

1. Find the minimum 1-respecting cut in $\mathcal{O}(m)$ time.
2. Find the minimum 2-respecting cut defined by dependent edges in $\mathcal{O}(m \log n)$ time.
3. To find the minimum 2-respecting cut defined by independent edges, obtain in $\mathcal{O}(m \log n)$ time a number of instances of a bipartite problem of total size $\mathcal{O}(m)$.
4. Solve each size-s instance of a bipartite problem in $\mathcal{O}(s \log s)$ time.

This sums up to $\mathcal{O}(m \log^2 n)$ as promised.

Overview

We obtain in $\mathcal{O}(m \log^2 n)$ time a collection of $\mathcal{O}(\log n)$ spanning trees $T_1, T_2, \ldots$ such that the minimum cut 1- or 2-respects some T_i.

Now we iterate over every tree T_i and:

1. Find the minimum 1-respecting cut in $\mathcal{O}(m)$ time.
2. Find the minimum 2-respecting cut defined by dependent edges in $\mathcal{O}(m \log n)$ time.
3. To find the minimum 2-respecting cut defined by independent edges, obtain in $\mathcal{O}(m \log n)$ time a number of instances of a bipartite problem of total size $\mathcal{O}(m)$.
4. Solve each size-s instance of a bipartite problem in $\mathcal{O}(s \log s)$ time.

This sums up to $\mathcal{O}(m \log^2 n)$ as promised.

Overview

We obtain in $\mathcal{O}(m \log^2 n)$ time a collection of $\mathcal{O}(\log n)$ spanning trees $T_1, T_2, \ldots$ such that the minimum cut 1- or 2-respects some T_i.

Now we iterate over every tree T_i and:

1. Find the minimum 1-respecting cut in $\mathcal{O}(m)$ time.
2. Find the minimum 2-respecting cut defined by dependent edges in $\mathcal{O}(m \log n)$ time.
3. To find the minimum 2-respecting cut defined by independent edges, obtain in $\mathcal{O}(m \log n)$ time a number of instances of a bipartite problem of total size $\mathcal{O}(m)$.
4. Solve each size-s instance of a bipartite problem in $\mathcal{O}(s \log s)$ time.

This sums up to $\mathcal{O}(m \log^2 n)$ as promised.

Dependent Edges

Link-cut trees implement the following operations in $\mathcal{O}(\log n)$ time:

1. add Δ to the score of every edge on a path,
2. return edge with the smallest score in a subtree.

We binarise T and sweep over its edges maintaining the scores.

For every $e' \in T(e)$, the weight of a cut defined by $\{e, e'\}$ is the score of e' + the total weight of all edges with exactly 1 endpoint in $T(e)$.

Dependent Edges

Link-cut trees implement the following operations in $\mathcal{O}(\log n)$ time:

1. add Δ to the score of every edge on a path,
2. return edge with the smallest score in a subtree.

We binarise T and sweep over its edges maintaining the scores.

For every $e' \in T(e)$, the weight of a cut defined by $\{e, e'\}$ is the score of e' + the total weight of all edges with exactly 1 endpoint in $T(e)$.

Dependent Edges

Link-cut trees implement the following operations in $\mathcal{O}(\log n)$ time:

1. add Δ to the score of every edge on a path,
2. return edge with the smallest score in a subtree.

We binarise T and sweep over its edges maintaining the scores.

For every $e' \in T(e)$, the weight of a cut defined by $\{e, e'\}$ is the score of e' + the total weight of all edges with exactly 1 endpoint in $T(e)$.

Dependent Edges

Link-cut trees implement the following operations in $\mathcal{O}(\log n)$ time:

1. add Δ to the score of every edge on a path,
2. return edge with the smallest score in a subtree.

We binarise T and sweep over its edges maintaining the scores.

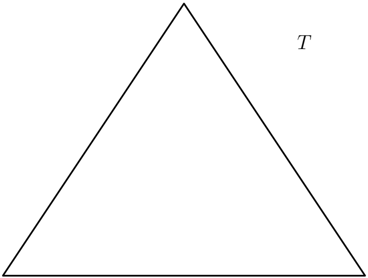
For every $e' \in T(e)$, the weight of a cut defined by $\{e, e'\}$ is the score of e' + the total weight of all edges with exactly 1 endpoint in $T(e)$.

Dependent Edges

Link-cut trees implement the following operations in $\mathcal{O}(\log n)$ time:

1. add Δ to the score of every edge on a path,
2. return edge with the smallest score in a subtree.

We binarise T and sweep over its edges maintaining the scores.



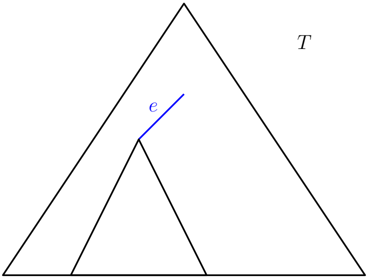
For every $e' \in T(e)$, the weight of a cut defined by $\{e, e'\}$ is the score of e' + the total weight of all edges with exactly 1 endpoint in $T(e)$.

Dependent Edges

Link-cut trees implement the following operations in $\mathcal{O}(\log n)$ time:

1. add Δ to the score of every edge on a path,
2. return edge with the smallest score in a subtree.

We binarise T and sweep over its edges maintaining the scores.



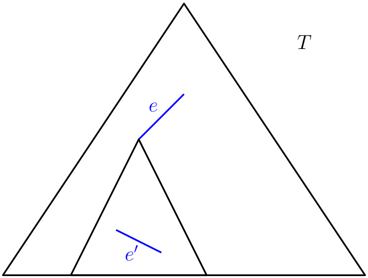
For every $e' \in T(e)$, the weight of a cut defined by $\{e, e'\}$ is the score of e' + the total weight of all edges with exactly 1 endpoint in $T(e)$.

Dependent Edges

Link-cut trees implement the following operations in $\mathcal{O}(\log n)$ time:

1. add Δ to the score of every edge on a path,
2. return edge with the smallest score in a subtree.

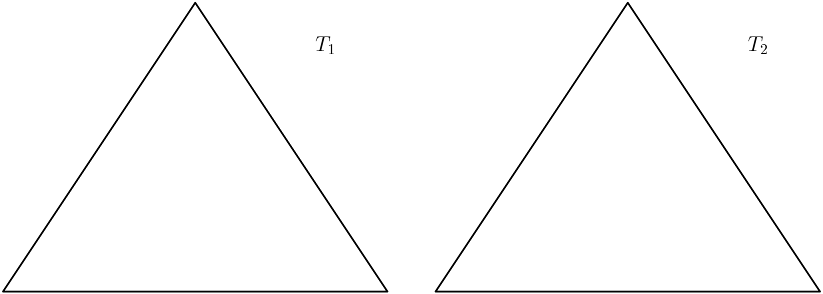
We binarise T and sweep over its edges maintaining the scores.



For every $e' \in T(e)$, the weight of a cut defined by $\{e, e'\}$ is the score of e' + the total weight of all edges with exactly 1 endpoint in $T(e)$.

Bipartite Problem

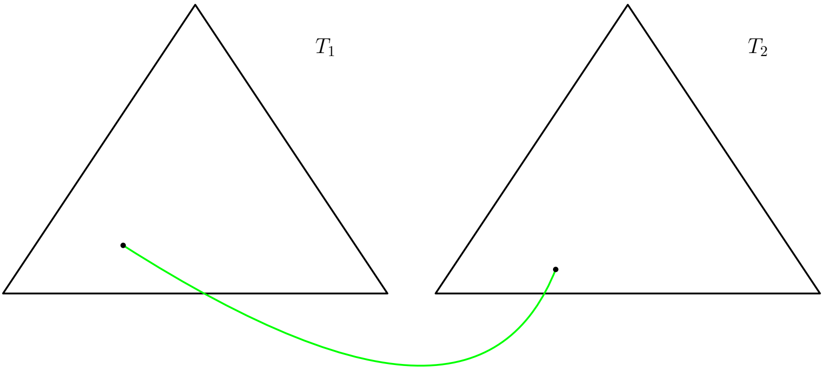
We are given two trees and a number of cross-edges:



We seek $e \in T_1, e' \in T_2$ minimising the sum of their costs - the total weight of all cross-edges between $T_1(e)$ with $T_2(e')$.

Bipartite Problem

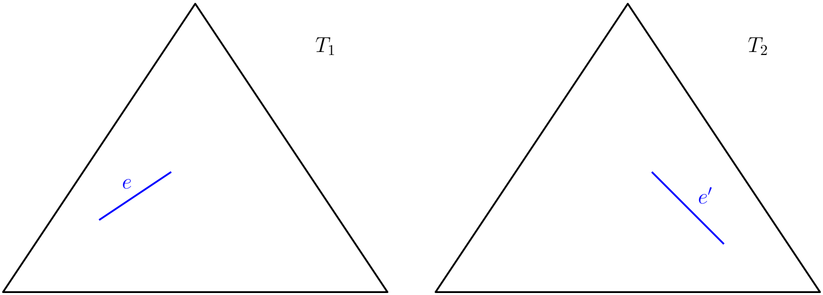
We are given two trees and a number of cross-edges:



We seek $e \in T_1, e' \in T_2$ minimising the sum of their costs - the total weight of all cross-edges between $T_1(e)$ with $T_2(e')$.

Bipartite Problem

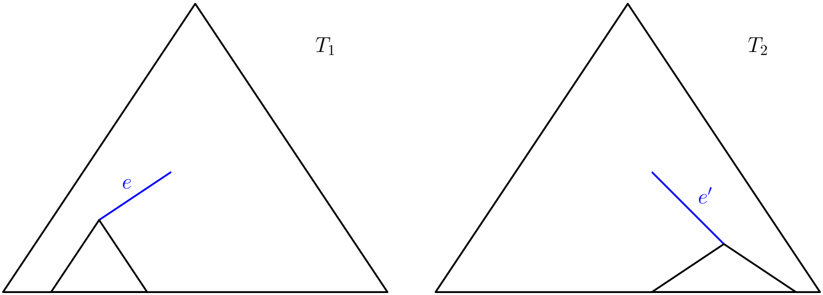
We are given two trees and a number of cross-edges:



We seek $e \in T_1, e' \in T_2$ minimising the sum of their costs - the total weight of all cross-edges between $T_1(e)$ with $T_2(e')$.

Bipartite Problem

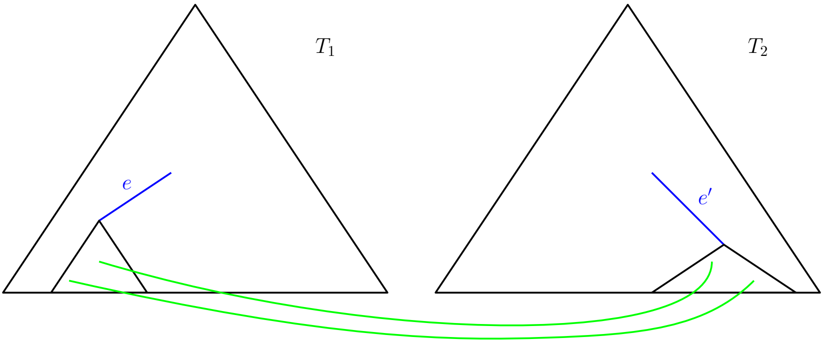
We are given two trees and a number of cross-edges:



We seek $e \in T_1, e' \in T_2$ minimising the sum of their costs - the total weight of all cross-edges between $T_1(e)$ with $T_2(e')$.

Bipartite Problem

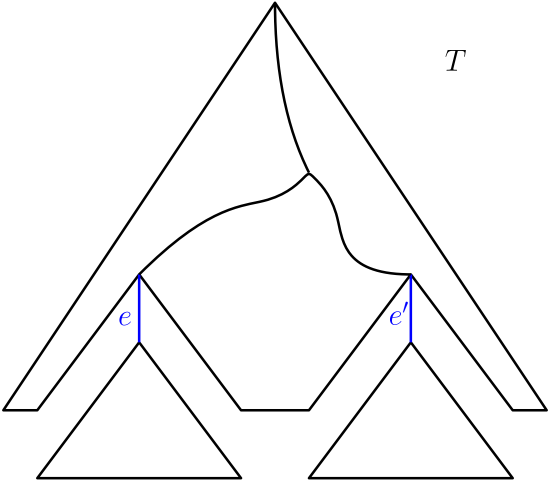
We are given two trees and a number of cross-edges:



We seek $e \in T_1, e' \in T_2$ minimising the sum of their costs - the total weight of all cross-edges between $T_1(e)$ with $T_2(e')$.

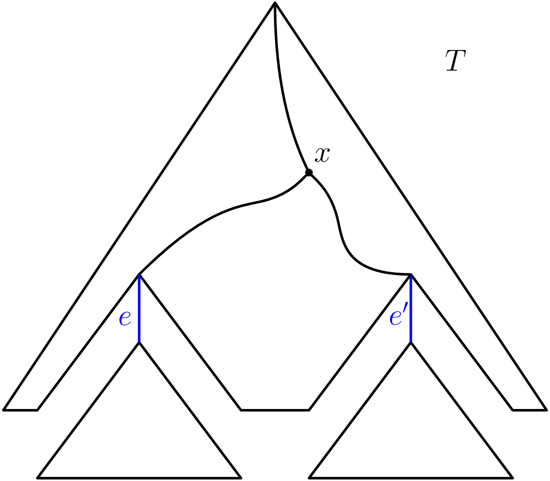
Reduction to Bipartite Problems

We create a separate instance for every node x with two children:



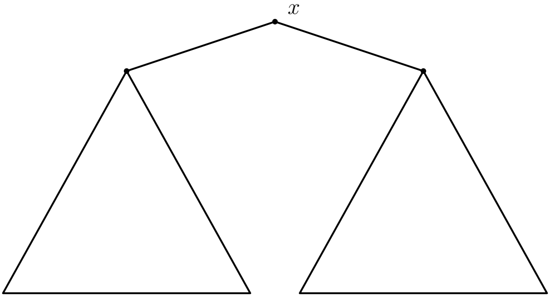
Reduction to Bipartite Problems

We create a separate instance for every node x with two children:



Reduction to Bipartite Problems

We create a separate instance for every node x with two children:

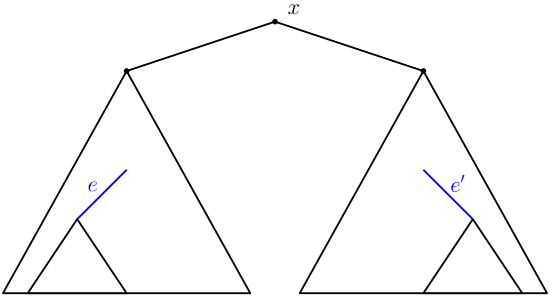


The weight of a cut defined by $\{e, e'\}$ is calculated as follows:

+ the total weight of all edges with one endpoint in $T(e)$,
+ the total weight of all edges with one endpoint in $T(e')$,
− the total weight of all edges between $T(e)$ with $T(e')$.

Reduction to Bipartite Problems

We create a separate instance for every node x with two children:



The weight of a cut defined by $\{e, e'\}$ is calculated as follows:

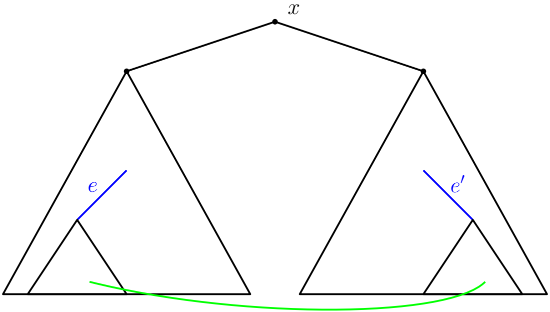
$+$ the total weight of all edges with one endpoint in $T(e)$,

$+$ the total weight of all edges with one endpoint in $T(e')$,

$-$ the total weight of all edges between $T(e)$ with $T(e')$.

Reduction to Bipartite Problems

We create a separate instance for every node x with two children:



The weight of a cut defined by $\{e, e'\}$ is calculated as follows:

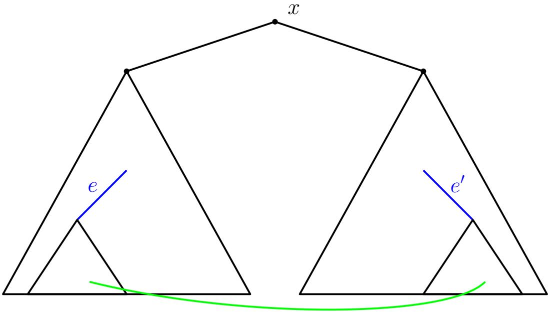
$+$ the total weight of all edges with one endpoint in $T(e)$,

$+$ the total weight of all edges with one endpoint in $T(e')$,

$-$ the total weight of all edges between $T(e)$ with $T(e')$.

Reduction to Bipartite Problems

We create a separate instance for every node x with two children:



The weight of a cut defined by $\{e, e'\}$ is calculated as follows:

+ the total weight of all edges with one endpoint in $T(e)$,
+ the total weight of all edges with one endpoint in $T(e')$,
– the total weight of all edges between $T(e)$ with $T(e')$.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.

With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

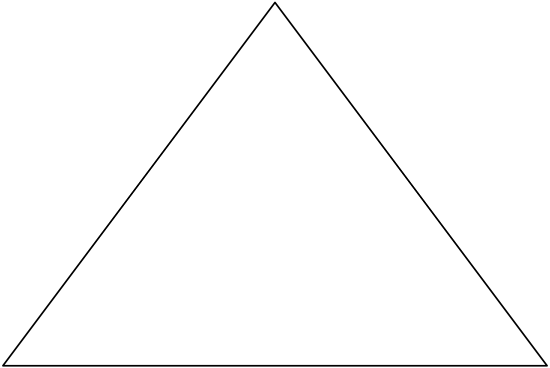
Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.

With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.

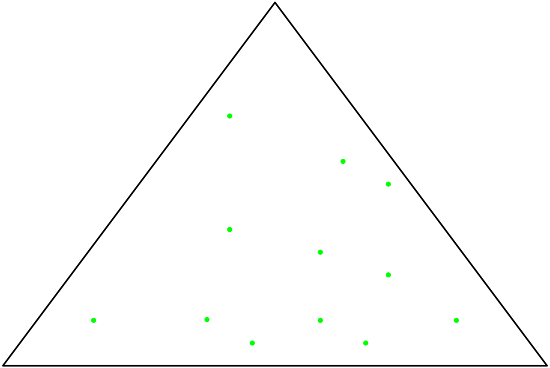


With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.

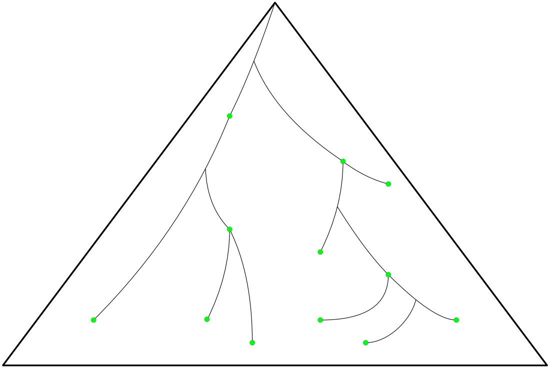


With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.

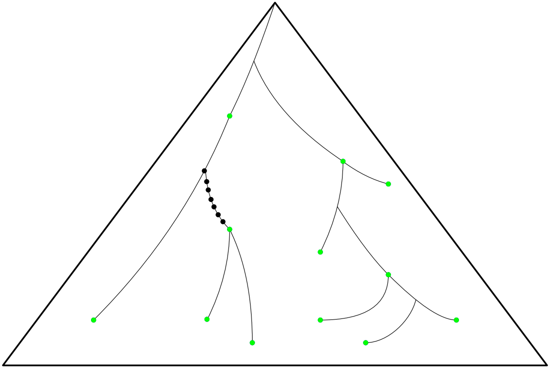


With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.

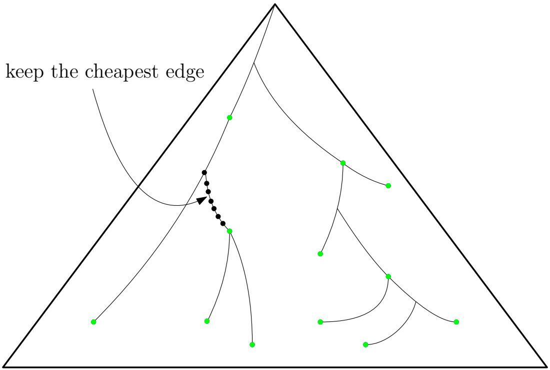


With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.

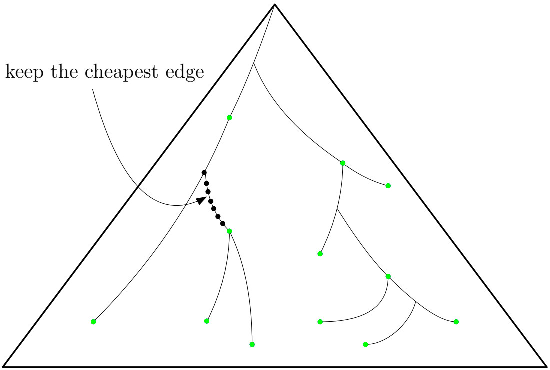


With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Main Trick

$s =$ number of edges (u, v) with $\mathrm{lca}(u, v) = x$.

Even though both trees might be large, only $\mathcal{O}(s)$ edges really matter.



With path-minimum queries we can compress the tree to consist of at most $2s$ edges in $\mathcal{O}(s \log n)$ time.

Solving the Bipartite Problem

Goal: for every $e \in T_1$ find the cheapest $e' \in T_2$.

Divide-and-conquer guided by the heavy-light decomposition of T_1.

In every recursive call we operate on a fragment of T_1: the subtree rooted at u_1 (possibly) without the subtree rooted at u_k.

Solving the Bipartite Problem

Goal: for every $e \in T_1$ find the cheapest $e' \in T_2$.

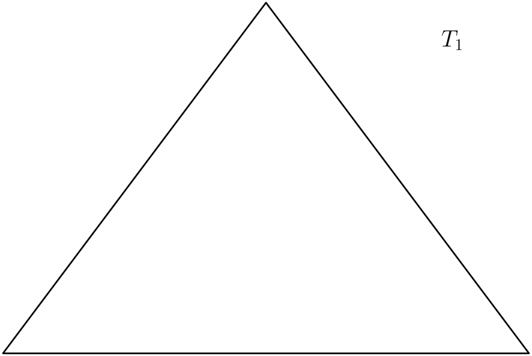
Divide-and-conquer guided by the heavy-light decomposition of T_1.

In every recursive call we operate on a fragment of T_1: the subtree rooted at u_1 (possibly) without the subtree rooted at u_k.

Solving the Bipartite Problem

Goal: for every $e \in T_1$ find the cheapest $e' \in T_2$.

Divide-and-conquer guided by the heavy-light decomposition of T_1.

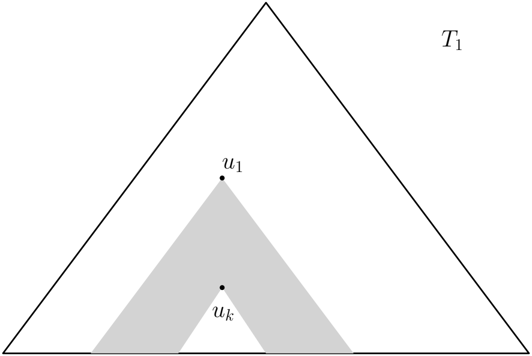


In every recursive call we operate on a fragment of T_1: the subtree rooted at u_1 (possibly) without the subtree rooted at u_k.

Solving the Bipartite Problem

Goal: for every $e \in T_1$ find the cheapest $e' \in T_2$.

Divide-and-conquer guided by the heavy-light decomposition of T_1.

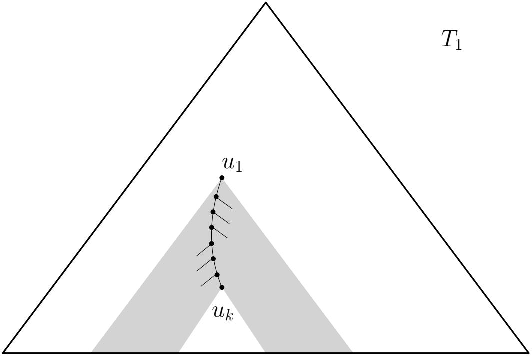


In every recursive call we operate on a fragment of T_1: the subtree rooted at u_1 (possibly) without the subtree rooted at u_k.

Solving the Bipartite Problem

Goal: for every $e \in T_1$ find the cheapest $e' \in T_2$.

Divide-and-conquer guided by the heavy-light decomposition of T_1.



In every recursive call we operate on a fragment of T_1: the subtree rooted at u_1 (possibly) without the subtree rooted at u_k.

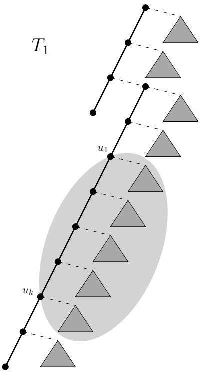
Solving Bipartite Problem

By appropriately splitting the current fragment into smaller fragments, we ensure that the depth of the recursion is $\mathcal{O}(\log n)$, and fragments on every level of the recursion are disjoint. But what about T_2?

We maintain a compressed representation of the relevant part of T_2.

Solving Bipartite Problem

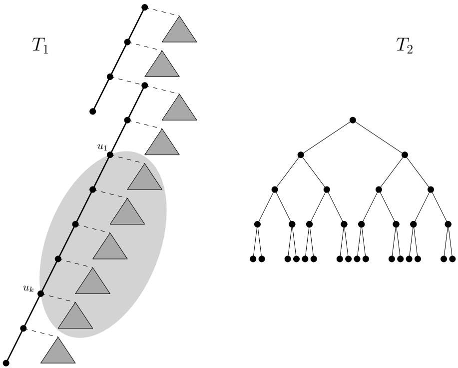
By appropriately splitting the current fragment into smaller fragments, we ensure that the depth of the recursion is $\mathcal{O}(\log n)$, and fragments on every level of the recursion are disjoint. But what about T_2?



We maintain a compressed representation of the relevant part of T_2.

Solving Bipartite Problem

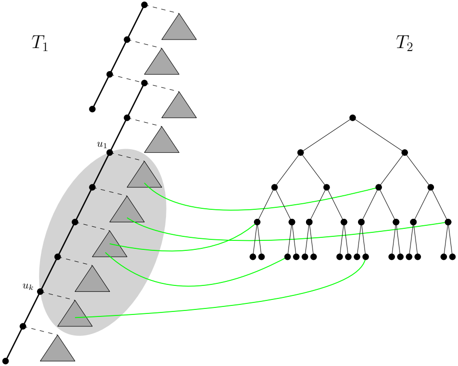
By appropriately splitting the current fragment into smaller fragments, we ensure that the depth of the recursion is $\mathcal{O}(\log n)$, and fragments on every level of the recursion are disjoint. But what about T_2?



We maintain a compressed representation of the relevant part of T_2.

Solving Bipartite Problem

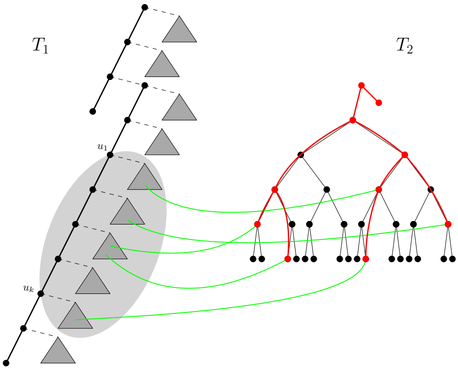
By appropriately splitting the current fragment into smaller fragments, we ensure that the depth of the recursion is $\mathcal{O}(\log n)$, and fragments on every level of the recursion are disjoint. But what about T_2?



We maintain a compressed representation of the relevant part of T_2.

Solving Bipartite Problem

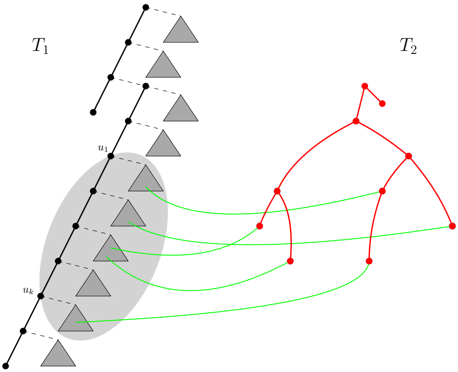
By appropriately splitting the current fragment into smaller fragments, we ensure that the depth of the recursion is $\mathcal{O}(\log n)$, and fragments on every level of the recursion are disjoint. But what about T_2?



We maintain a compressed representation of the relevant part of T_2.

Solving Bipartite Problem

By appropriately splitting the current fragment into smaller fragments, we ensure that the depth of the recursion is $\mathcal{O}(\log n)$, and fragments on every level of the recursion are disjoint. But what about T_2?



We maintain a compressed representation of the relevant part of T_2.

Recursion

In every recursive call we have a fragment f of T_1 and a compressed representation of the relevant part of T_2 of size $\mathcal{O}(s)$, where s is the number of cross-edges with one endpoint in f. Then:

1. We find the cheapest $e' \in T_2$ for $\mathcal{O}(1)$ edges $e \in T_1$ by considering all $\mathcal{O}(s)$ edges in the compressed representation.
2. We partition the remaining into $\mathcal{O}(1)$ fragments $f_1, f_2, \ldots$.
3. For every f_i, we extract a compressed representation of its relevant part of T_2 from the current compressed representation in $\mathcal{O}(s)$ time, and recurse.

We make sure that the depth of the recursion is $\mathcal{O}(\log n)$, and then the whole running time becomes $\mathcal{O}(m \log n)$.

There are some technicalities, in particular we actually maintain two separate compressed representations, please see the paper.

Recursion

In every recursive call we have a fragment f of T_1 and a compressed representation of the relevant part of T_2 of size $\mathcal{O}(s)$, where s is the number of cross-edges with one endpoint in f. Then:

1. We find the cheapest $e' \in T_2$ for $\mathcal{O}(1)$ edges $e \in T_1$ by considering all $\mathcal{O}(s)$ edges in the compressed representation.
2. We partition the remaining into $\mathcal{O}(1)$ fragments $f_1, f_2, \ldots$.
3. For every f_i, we extract a compressed representation of its relevant part of T_2 from the current compressed representation in $\mathcal{O}(s)$ time, and recurse.

We make sure that the depth of the recursion is $\mathcal{O}(\log n)$, and then the whole running time becomes $\mathcal{O}(m \log n)$.

There are some technicalities, in particular we actually maintain two separate compressed representations, please see the paper.

Recursion

In every recursive call we have a fragment f of T_1 and a compressed representation of the relevant part of T_2 of size $\mathcal{O}(s)$, where s is the number of cross-edges with one endpoint in f. Then:

1. We find the cheapest $e' \in T_2$ for $\mathcal{O}(1)$ edges $e \in T_1$ by considering all $\mathcal{O}(s)$ edges in the compressed representation.
2. We partition the remaining into $\mathcal{O}(1)$ fragments $f_1, f_2, \ldots$.
3. For every f_i, we extract a compressed representation of its relevant part of T_2 from the current compressed representation in $\mathcal{O}(s)$ time, and recurse.

We make sure that the depth of the recursion is $\mathcal{O}(\log n)$, and then the whole running time becomes $\mathcal{O}(m \log n)$.

There are some technicalities, in particular we actually maintain two separate compressed representations, please see the paper.

Recursion

In every recursive call we have a fragment f of T_1 and a compressed representation of the relevant part of T_2 of size $\mathcal{O}(s)$, where s is the number of cross-edges with one endpoint in f. Then:

1. We find the cheapest $e' \in T_2$ for $\mathcal{O}(1)$ edges $e \in T_1$ by considering all $\mathcal{O}(s)$ edges in the compressed representation.
2. We partition the remaining into $\mathcal{O}(1)$ fragments $f_1, f_2, \ldots$.
3. For every f_i, we extract a compressed representation of its relevant part of T_2 from the current compressed representation in $\mathcal{O}(s)$ time, and recurse.

We make sure that the depth of the recursion is $\mathcal{O}(\log n)$, and then the whole running time becomes $\mathcal{O}(m \log n)$.

There are some technicalities, in particular we actually maintain two separate compressed representations, please see the paper.

Recursion

In every recursive call we have a fragment f of T_1 and a compressed representation of the relevant part of T_2 of size $\mathcal{O}(s)$, where s is the number of cross-edges with one endpoint in f. Then:

1. We find the cheapest $e' \in T_2$ for $\mathcal{O}(1)$ edges $e \in T_1$ by considering all $\mathcal{O}(s)$ edges in the compressed representation.
2. We partition the remaining into $\mathcal{O}(1)$ fragments $f_1, f_2, \ldots$.
3. For every f_i, we extract a compressed representation of its relevant part of T_2 from the current compressed representation in $\mathcal{O}(s)$ time, and recurse.

We make sure that the depth of the recursion is $\mathcal{O}(\log n)$, and then the whole running time becomes $\mathcal{O}(m \log n)$.

There are some technicalities, in particular we actually maintain two separate compressed representations, please see the paper.

Recursion

In every recursive call we have a fragment f of T_1 and a compressed representation of the relevant part of T_2 of size $\mathcal{O}(s)$, where s is the number of cross-edges with one endpoint in f. Then:

1. We find the cheapest $e' \in T_2$ for $\mathcal{O}(1)$ edges $e \in T_1$ by considering all $\mathcal{O}(s)$ edges in the compressed representation.
2. We partition the remaining into $\mathcal{O}(1)$ fragments $f_1, f_2, \ldots$.
3. For every f_i, we extract a compressed representation of its relevant part of T_2 from the current compressed representation in $\mathcal{O}(s)$ time, and recurse.

We make sure that the depth of the recursion is $\mathcal{O}(\log n)$, and then the whole running time becomes $\mathcal{O}(m \log n)$.

There are some technicalities, in particular we actually maintain two separate compressed representations, please see the paper.

Summary

Minimum cut can be solved in $\mathcal{O}(m \log^2 n)$ time w.h.p.

1. Is there a faster algorithm?
2. Is there a near-linear time deterministic algorithm?

Thank you!

Summary

Minimum cut can be solved in $\mathcal{O}(m \log^2 n)$ time w.h.p.

1. Is there a faster algorithm?
2. Is there a near-linear time deterministic algorithm?

Thank you!

Summary

Minimum cut can be solved in $\mathcal{O}(m \log^2 n)$ time w.h.p.

1. Is there a faster algorithm?
2. Is there a near-linear time deterministic algorithm?

Thank you!

Summary

Minimum cut can be solved in $\mathcal{O}(m \log^2 n)$ time w.h.p.

1. Is there a faster algorithm?
2. Is there a near-linear time deterministic algorithm?

Thank you!

Summary

Minimum cut can be solved in $\mathcal{O}(m \log^2 n)$ time w.h.p.

1. Is there a faster algorithm?
2. Is there a near-linear time deterministic algorithm?

Thank you!