Weighted Min-Cut: Sequential, Cut-Query and Streaming Algorithms

Jaehyun Koo (koosaga)

July 23, 2020

Prior researches and contribution

Global Minimum Cut with Flows

Mincut Maxflow Theorem: The value of $s-t$ mincut equals to $s-t$ maxflow.

Polynomial Time Maximum Flow: Edmonds-Karp showed that the maximum flow can be computed in $O(nm^2)$ time.

This two result is still an important core of theoretical CS, so it's reasonable to reduce the $s-t$ cut to n^2 time of flows..

Indeed, the **Gomory-Hu tree** shows that we only need $O(n)$ iteration of maximum flow.

Global Minimum Cut with Flows

Mincut Maxflow Theorem: The value of $s-t$ mincut equals to $s-t$ maxflow.

Polynomial Time Maximum Flow: Edmonds-Karp showed that the maximum flow can be computed in $O(nm^2)$ time.

This two result is still an important core of theoretical CS, so it's reasonable to reduce the $s-t$ cut to n^2 time of flows..

Indeed, the **Gomory-Hu tree** shows that we only need $O(n)$ iteration of maximum flow.

However, flow is not really a cheap primitive. And we use it too much.

Even if we have $O(m)$ flow, this is at least $O(nm)$!

Global Minimum Cut with Contractions

Global Minimum Cut divides the graph into two connected components.

Since it's minimum, not many edges will go across the optimal min-cut.

Many edges will connect the vertices of same partition.

Global Minimum Cut with Contractions

Global Minimum Cut divides the graph into two connected components.

Since it's minimum, not many edges will go across the optimal min-cut.

Many edges will connect the vertices of same partition.

Karger: Then we can just randomly contract and win! (*Karger-Stein algorithm*)

Nagamochi-Ibaraki: Contract something that is most thickly attached to current component. (*Stoer-Wagner algorithm*)

Global Minimum Cut with Contractions

Global Minimum Cut divides the graph into two connected components.

Since it's minimum, not many edges will go across the optimal min-cut.

Many edges will connect the vertices of same partition.

Karger: Then we can just randomly contract and win! (*Karger-Stein algorithm*)

Nagamochi-Ibaraki: Contract something that is most thickly attached to current component. (*Stoer-Wagner algorithm*)

Both algorithms are **much simpler** than the flow approach.

Both algorithm has a magnitude of about $O(nm)$. It is surely fast, but...

Global Minimum Cut with Spanning Trees

Nash-Williams Theorem: In a $2k$-connected graph, there is k disjoint spanning trees.

Global Minimum Cut with Spanning Trees

Nash-Williams Theorem: In a $2k$-connected graph, there is k disjoint spanning trees.

Takeaway?

Global Minimum Cut with Spanning Trees

Nash-Williams Theorem: In a $2k$-connected graph, there is k disjoint spanning trees.

Takeaway?

Obviously, each disjoint spanning trees have nonempty intersection with edge sets in global min-cut.

By pigeonhole principle, there exists some disjoint spanning trees that have at most 2 intersection with global min-cut.

Given c spanning trees, There exists $O(cn^2)$ possible cuts for each spanning tree, which is polynomial!

Question: How to compute those spanning trees?

Question: How to try all cuts faster?

Global Minimum Cut with Spanning Trees

Question: How to compute those spanning trees?

Naively, you can use matroid intersection to obtain slow polynomial time solution. Gabow discovered an matroid-based algorithm that computes $\frac{c}{2}$ disjoint spanning trees given a c-connected graph (STOC'91) in $O(mc \log m)$.

Still, there are a lot of problems: This is slow, there are too many spanning trees, and it doesn't work for weighted graphs.

Karger discovered a simple algorithm that can sample a spanning tree that have at most 2 intersection with weighted global min-cut with $\frac{1}{3}$ probability.

Sampling $O(\log n)$ trees, we can find a global min-cut w.h.p.

This concept is based on the *cut sparsifiers*, which we won't cover in this lecture. For now, let's just assume that we only have to try $O(\log n)$ trees somehow.

Finding 2-respecting minimum cut

Question: How to try all cuts faster?

Given a spanning tree T, a cut **2-respects** a tree if there is at most 2 edge in T that connects the different part of the cut.

Let $T_{rsp}(n, m)$ be a time to find 2-respecting min-cut. Then the global min-cut can be computed in $O(T_{rsp}(n, m) \log n)$.

Theorem (Karger 2000). $T_{rsp}(n, m) = O(m \log^2 n)$

Finding 2-respecting minimum cut

Question: How to try all cuts faster?

Given a spanning tree T, a cut **2-respects** a tree if there is at most 2 edge in T that connects the different part of the cut.

Let $T_{rsp}(n, m)$ be a time to find 2-respecting min-cut. Then the global min-cut can be computed in $O(T_{rsp}(n, m) \log n)$.

Theorem (Karger 2000). $T_{rsp}(n, m) = O(m \log^2 n)$

Theorem (This paper). $T_{rsp}(n, m) = O(m \log n + n \log^4 n)$

Finding 2-respecting minimum cut

Question: How to try all cuts faster?

Given a spanning tree T, a cut **2-respects** a tree if there is at most 2 edge in T that connects the different part of the cut.

Let $T_{rsp}(n, m)$ be a time to find 2-respecting min-cut. Then the global min-cut can be computed in $O(T_{rsp}(n, m) \log n)$.

Theorem (Karger 2000). $T_{rsp}(n, m) = O(m \log^2 n)$

Theorem (This paper). $T_{rsp}(n, m) = O(m \log n + n \log^4 n)$

Theorem (GMW19). $T_{rsp}(n, m) = O(m \log n)$ (??)

Finding 2-respecting minimum cut

The algorithm in this paper was the fastest 2-respecting min-cut algorithm in dense graph *for one day* (per arXiv).

But this is unique in a sense that it does not require all 2-respecting cut. Approaches taken by Karger and GMW19 requires all 2-respecting min-cut at least implicitly.

For example, consider the *cut-query* computation model: You are only given a tree T, but not a graph, and you can find a size of cut between S and $V \setminus S$ in each query.

This algorithm can solve the min-cut problem with $\tilde{O}(n)$ cut queries, which is good for specific computation environment (especially on parallelization).

Our explanation will also assume the *cut-query* computation model: We don't know the graph. Then we will talk about the implementation in sequential model.

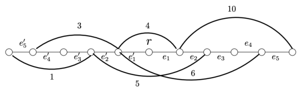
How to solve it

When T is a path

We use divide and conquer on path.

Take the middle position of path. If both position of the cut lies before, or after this middle position, you can recursively find it.

Interesting case is when you have the cut for each side.



R

L	e_1	e_2	e_3	e_4	e_5
e'_1	0	14	19	25	25
e'_2	13	19	24	18	18
e'_3	19	25	20	14	14
e'_4	19	25	20	14	14
e'_5	16	22	17	11	11

Let $e_1, e_2, \dots, e_n, e'_1, e'_2, \dots, e'_m$ be the indices of the edges to cut (in the interval we are considering). e is numbered from left, e' is from right.

Let $F(i,j) = cut(e_i, e'_j)$. The matrix in right denotes $F(i,j)$. *Deja vu?*

When T is a path

Are we gonna do it? Yes, we are gonna do it!!

Theorem. For all $1 \leq i \leq n-1, 1 \leq j \leq m-1$. Let A be a **Monge array** if $A(i,j) + A(i+1,j+1) \leq A(i,j+1) + A(i+1,j)$. F is a Monge array.

Lemma. If A, B is a Monge array, $A + B$ also is.

Proof of Theorem. An edge $g = (u,v)$ contributes to $cut(e,f)$ iff there exists only one of e, f in a unique path between u, v. Take three case of possible edges, and observe all of them are Monge - so do their sum.

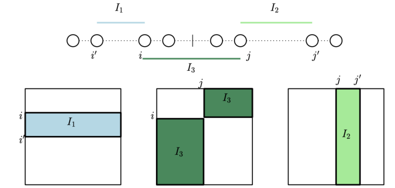


Figure 5: Contribution of each type of Interval (L, R, Crossing) in the cost matrix M.

When T is a path

It is widely known that the (row) minimum of $n \times m$ Monge arrays can be computed in $O((n+m)\log n)$ by so-called *Divide and Conquer optimization*.

Combining this with standard divide and conquer on path, we obtain $O(n \log^2 n)$ algorithm for path.

We will call this a *path algorithm*, and use this as a black box.

SMAWK also works, but the author skipped it for easy parallelization. This algorithm is slow anyway, so nobody cares small improvement...

When T is a star graph

Let $C(i, j)$ be a weight of edge between vertex i and j (0 if not exist), and $deg(i) = \sum_j C(i, j)$. Let $\Delta(S) = \sum_{i \in S, j \notin S} C(i, j)$. Let e_i be an edge $(i, parent(i))$.

Lemma. Suppose the optimal answer have an intersection of exactly two edges. If (e_i, e_j) gives the optimal solution, then $deg(i) < 2C(i, j), deg(j) < 2C(i, j)$.

Proof. $cut(e_i, e_j) = deg(i) + deg(j) - 2C(i, j)$. Since $deg(i), deg(j)$ is a cut with one intersection, $cut(e_i, e_j) < deg(i), cut(e_i, e_j) < deg(j)$. Combine this with first equation.

So, in the optimal solution, there is only one possible candidate of j, which is the maximum-weight edge incident with i.

Finding j can be done with simple binary search, because for disjoint set A, B, $between(A, B) = \sum_{i \in A, j \in B} C(i, j) = \frac{\Delta(A) + \Delta(B) - \Delta(A+B)}{2}$

Orthogonal case

Let $e_i^{\downarrow}$ be the rooted subtree that lies below the edge e_i.

We call the pair of edge e_i, e_j *orthogonal* if their rooted subtrees are disjoint. If it is not (so one is a subset of other), then we call it *parallel*.

For the case of orthogonal edges, we have a following lemma, very similar to the star case.

Lemma. Suppose the optimal answer have an intersection of exactly two edges. If (e_i, e_j) gives the optimal solution, and if they are orthogonal, then $\Delta(e_i^{\downarrow}) < 2 \times between(e_i^{\downarrow}, e_j^{\downarrow}), \Delta(e_j^{\downarrow}) < 2 \times between(e_i^{\downarrow}, e_j^{\downarrow})$.

Proof is similar, so we will skip it.

Orthogonal case

In the star case, there were only one such j that satisfies the condition of Lemma. Also that j was easy to find. This is not the case in general, but hopefully the j forms a nice structure.

Definition (cross-interesting). For some e_i, if $\Delta(e_i^{\downarrow}) < 2 \times between(e_i^{\downarrow}, e_j^{\downarrow})$ and, e_j is orthogonal to e_i, then e_j is *cross-interesting* to e_i.

Lemma. For some e_i, the set of e_j that is cross-interesting forms a path in the direction to the root.

Proof. If some e_j is cross-interesting, $e_{par(j)}$ is also cross-interesting as long as it is orthogonal to e_i. The RHS of $between$ is increasing, and thus the value is increasing. If two orthogonal pair e_j, e_k is both cross-interesting to e_i,
$\Delta(e_i^{\downarrow}) < between(e_i^{\downarrow}, e_j^{\downarrow}) + between(e_i^{\downarrow}, e_k^{\downarrow}) = between(e_i^{\downarrow}, e_j^{\downarrow} \cup e_k^{\downarrow})$.
Since, $\Delta(e_i^{\downarrow}) = between(e_i^{\downarrow}, V(T) - e_i^{\downarrow})$, contradiction.

Orthogonal case

To find the path of e_j, we can use *cut sparsifier*, but let's talk about this later, and suppose we have just found it.

Now we will reduce the problem on general tree T to the collection of path case, and use *path algorithm* as a black box.

Take the Heavy-light decomposition of T. For each e_i, interesting path has an intersection with at most $\log N$ paths. Redundancy doesn't affect the answer, so we can decompose the whole problem to the following $O(n \log n)$ query:

Query (e_i, P): For an e_i, and a path P in HLD, find the minimum $cut(e_i, e_j)$ for all $e_j \in P$, such that e_i, e_j are orthogonal.

Orthogonal case

To solve this query, notice that for the optimal cut, not only e_j should be cross-interesting to e_i, but the reverse should also hold.

Let Q be the HLD path that e_i belong. Batch-process the query with the same $\{P, Q\}$ set (order ignored). There exists at most $O(n \log n)$ distinct $\{P, Q\}$ set, obviously.

We can ignore everything not in P, Q: Contract them. Then we will have a line, or a "tri-junction". In the tri-junction case, you can again contract the path headed to root. So now we obtain a line.

For a cut (e_i, e_j) to be interesting, both (e_i, P) and (e_j, Q) should appear in the list of query by above reasons. So, we can contract all edges that does not appear in any query as itself.

Orthogonal case

After the contraction, we have a set of paths that have a size at most $O(n \log n)$, which we can use the *path algorithm*.

We used $O(n \log^3 n)$ queries.

Note that the contraction is just hypothetical: We don't need to explicitly consider anything, because in reality we don't have to contract anything, what we need is to just *ignore* anything not contracted.

Parallel case

Take an edge e_i of the upper side, then we have to consider e_j that lies in the subtree of $e_i^{\downarrow}$.

They also form a path that goes from some vertex v to i: The proof is similar, and how we find that path is also similar, so we will skip it.

We again obtain the following queries of size $O(n \log n)$:

Query (e_i, P)*: For an* e_i*, and a path* P *in HLD, find the minimum* $cut(e_i, e_j)$ *for all* $e_j \in P$*, such that* $e_j^{\downarrow} \subseteq e_i^{\downarrow}$. WLOG $e_i \notin P$ (We run the path algorithm for each HLD path separately.)

Parallel case

We batch process the set with same (P, Q). This time order is important. Contract $E \setminus P \setminus Q$, and possibly one branch in the tri-junction, to make a path.

Now we contract all vertices in Q that does not appear as P. Note that we can't contract P because of the lack of symmetry.

However, since P is in the subtree of Q, there exists at most $O(\log N)$ possible Q that is interested in P. So we can afford not contracting them: We again obtain $O(n \log^3 n)$.

Now the algorithm is complete.. except the stuff about cut sparsifiers.

Implementation by computation model

Querying 2-respecting cut in sequential model

So far, the algorithm was only presented in the *cut-query* model: $\tilde{O}(n)$ queries means $\tilde{O}(nm)$ algorithm for minimum 2-respecting cut, which is not interesting.

On the other hand, we can observe that all the queries asked by the algorithm is 2-respecting, except the case of finding interesting paths.

Take the euler tour of T, then the interesting subtrees can be represented as an union of $O(1)$ disjoint intervals. Edges crossing inside and outside of the set, can be represented as an edge having one end at some interval, and one at another.

This is 2-dimensional range queries: Persistent segment trees can do the job. We have $O(1) \times O(\log n) = O(\log n)$ query time for 2-resp cut.

Finding interesting path in sequential model

Note that we only have to compute the deepest edge in the interesting path, to specify the whole set.

Assume all edges have weight 1. If you sample *any* edge randomly in $\Delta(e_i^\downarrow)$, then the edge belongs to $between(e_i^\downarrow, e_j^\downarrow)$ with probability $\frac{1}{2}$. Sampling $\log n$ edges are sufficient to have high probability.

In fact, the edges are weighted, so we have to assign linear probability, which is not hard anyway.

Sampling an edge can be done with the exact persistent segment tree we built.

Now binary search in the path (or iterate HLD) from the sampled edge, and to the LCA of sampled edge and e_i. The criteria to check the edges can be written as a 2-respecting cuts. Taking the least deep edge, you obtain $O(\log^2 n)$ query for j.