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

The minimum cut of the graph is the minimum weight sum of edge weights that should be erased in order to disconnect the graph. In unweighted case, minimum cut is the quantity that defines the graph's connectivity. Therefore, minimum cut is the basic quantity you may want to calculate when a graph is given, and there are also numerous applications.

There are three ways to calculate the minimum cut of a graph. The methods are listed in order of discovery time. (The summary below was taken from this paper.)

- **Min-cut Max-flow approach**. Global min-cut is the smallest of all global $s-t$ cuts, so we can apply our knowledge for $s-t$ cuts. Simply we can find the s - t cut $\frac{n(n-1)}{2}$. However, Gomory-Hu showed that the problem can be solved by calculating $O(n)$ cuts (*Gomory-Hu Tree*). It is not known how to solve the problem with fewer s - t cut query than this. Finding the minimum $s-t$ cut is nontrivial, but by Min-cut max-flow theorem, you can compute it in polynomial time. However, flow is not a simple operation. Even if the flow is calculated in $O(m)$ time, a minimum of $O(nm)$ time is required, so there is a limit to optimization.
- **Edge contraction approach**. If you can find an edge that does not cross the minimum cut, you can contract that edge. In this way, you get an instance reducing the number of vertices, and solve recursively. Karger-Stein algorithm and Stoer-Wagner algorithm, also introduced in the blog, belong to this kind of algorithm. These algorithms are very clean, and works at times near $O(nm)$, so they are better than the above approach unless Max-flow is solved very effectively. Approaching linear time with them is still unknown.
- **Tree packing approach**. By Nash-Williams Theorem, you can find k Edge-disjoint spanning trees in graphs with a minimum cut of $2k$ or more. This is briefly introduced in the blog Matroid intersection exercise. In a graph with a minimum cut of c, pack the graph with the maximum number of edge-disjoint spanning trees. The number of spanning trees is more than $\frac{c}{2}$, and one spanning tree will have less than 2 intersections with the minimum cut. If so, try all the cuts that can break 2 or fewer edges in the spanning tree. Now the following four problems arise.
    - Is it possible in weighted graphs? Nash-Williams theorem does not hold if graph is weighted.
    - How to pack trees? The simplest polynomial time approach is Matroid Union. This is very slow. Gabow proposed a $O(mc \log m)$ packing algorithm in STOC'91 that combines a matroid approach with a dynamic tree, but it takes time proportional to c.
    - Shouldn't we guess all c spanning trees after packing? This requires time proportional to c.
    - How to try all cuts that have less than 2 intersection with this spanning trees? Naive strategy is to try all intersection and evaluate the value of a cut, which takes $O(n^2m)$ time.

> In STOC'96, Karger solved all four of these problems and proposed a random algorithm that works at $O(m \log^3 n)$ hours. Therefore, unlike the above two approaches, this is close to linear time. In the case of a weighted graph, we can (kind of) consider the weight w as a copy of w duplicate edges. Tree packing is constructed in $O(m + n \log^3 n)$ time by using cut sparsifiers and random sampling. This tree packing only consists of $O(\log n)$ spanning trees, not c, so you can try them all. The last routine can be resolved at $O(m \log^2 n)$ hours. This routine is a key topic in this article.

Given the spanning tree T, the problem of finding the smallest cuts with 2 or less edge intersections with T is known as a **2-respecting min-cut**. Suppose that the time to find this 2-respecting min-cut is $T_{rsp}(n, m)$. The results of Karger's paper, briefly described above are as follows.

Theorem (Karger 2000). Given a weighted graph, let $T_{rsp}(n, m)$ be a complexity of the algorithm for finding a 2-respecting minimum cut. The minimum cut can be found through a random algorithm $O(T_{rsp}(n, m) \log n + m + n \log^3 n)$, or $O(T_{rsp}(n, m) \frac{\log n}{\log \log n} + n \log^6 n)$.

Since then, Karger uses Link-cut tree and *Bough decomposition* to derive the following results. Regarding the Bough decomposition, It may be helpful to refer to the following problem

Theorem. 2-respecting min-cut can be found at $O(m \log^2 n)$ hours. (Karger 2000)

The following is a summary of the state-of-the-art result about the minimum cut algorithm.

- Randomized, simple: $O(m \log n)$, $O(m + n \log^2 n)$ (GNT20, GMW19)
- Randomized, weighted: $O(m \log^2 n)$, $O(m \frac{\log^2 n}{\log \log n} + n \log^6 n)$ (GMW19)
- Deterministic, simple: $O(m \log^2 n \log \log^2 n)$ (HRW17)
- Deterministic, weighted: $O(mn)$ (Nagamochi, Ibaraki 1992)

Goal of this article

In STOC 2020, An algorithm to find the 2-respecting min cut at $O(m \log n + n \log^4 n)$ time was introduced. Karger's first algorithm works on $O(m \log^2 n)$, so it works more efficiently for dense graphs. This article introduces this algorithm.

This article is the first algorithm to run faster than Karger's algorithm in 20 years, but on the day after this paper was submitted to arxiv, an algorithm in $O(m \log n)$ (GMW19) was submitted to arxiv. Therefore, the paper to be introduced is no longer the most efficient algorithm for finding a 2-respecing min cut.

Still, the paper is significant because the approach is different from Karger and GMW19. Let's define the value of the cut when the spanning tree cuts the two edges e, f on T to $cut(e, f)$. The most natural algorithm to compute a 2-respecting min cut is to compute this $cut(e, f)$ for every $e, f \in E(T)$ pair. All algorithms other than this paper computes n^2 entries of this, at least implicitly. After calculating all the values of $cut(e, f)$, return the minimum value. However, in this paper, $\tilde{O}(n)$ important 2-respecting min cut candidates are listed, and then only that values are computed.

To mathematically define this situation, consider the following computational model, called *cut-query*.

- Weighted graph G and a spanning tree T of G is given. You know T, but you don't know G at all. On the other hand, you can query $S \subseteq V(T)$ to return the sum of the weights of the edges between S and $V(T) \setminus S$. Let's call this $\Delta(S)$. (So naturally you can also query $cut(e, f)$). Can we find a 2-respecting min-cut of G?

This paper presents the first $\tilde{O}(n)$ algorithm for this computational model. Therefore, if the value of such a cut can be calculated very quickly in the instance (typically, when the structure of the graph is special or parallel calculation is possible), the algorithm is more appropriate.

It is well known that even in a general sequential model, the value for a given 2-respecting cut can be obtained in $O(\log n)$ using a two-dimensional data structure. Although the algorithm presented in this paper was based on this limited computational model, it was possible to update the record of calculating the 2-respecting cut in a dense graph.

Lastly, although not covered on paper due to time and space, interested readers are encouraged to learn about cut sparsifier and spanning tree random sampling introduced by Karger. The content of this article was summarized as much as possible without needing the concept of cut sparsifier, but in order to fully understand the content of the article, you need to know the concept.

1. An schematic algorithm for 2-respecting min-cut

For the rooted tree T, $v^{\downarrow}$ refers to the set of subtree vertices rooted at v. If $e = (par(v), v)$, $e^{\downarrow}$ is $v^{\downarrow}$. In addition, all given spanning trees are regarded as a rooted tree with an arbitrary vertex as a root. Graph G is not given, but only spanning tree T is given. You only have an access to $\Delta(S)$ query. We also don't care which algorithm is used for computing $\Delta(S)$ (we will get back to that later). We analyzes the complexity of the algorithm based on the number of times it is called.

Also, let's assume that when the spanning tree T is given, all *1-respecting min-cut*, whose intersection with the edge of T is 1 or less, are already found with $O(n)$ calls. Note that the intersection of the cut and T is nonempty.

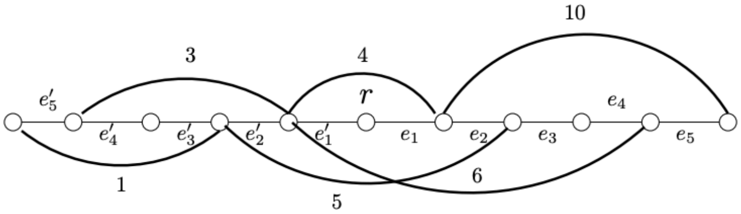
1.1 When T is a path

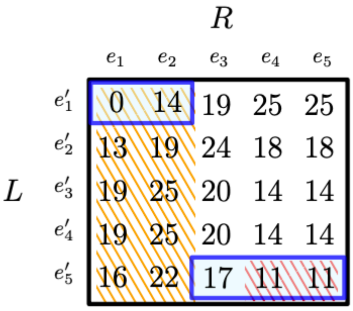
First, let's solve the problem for the special case where T is a path. In this case, we try divide and conquer on T. Think of a recursive function that computes the minimum value of the cut obtained by cutting two edges in the section $[l, r]$ of T. If you create this recursive function, you can conquer it by calling the whole T.

In divide and conquer, take the midpoint r, and solve by the position of two edges to cut.

- Recursively solve when both edges are to the left of r
- Recursively solve when both edges are to the right of r
- Solve the case when one is left of r and the other is right of r **efficiently**

We will solve the third case with just $O(n \log n)$ $cut(e, f)$ queries. Solving the rest of the case recursively, the master theorem calls the oracle up to $O(n \log^2 n)$ times in this case.





$$
L \quad \begin{array}{c|ccccc}
 & e_1 & e_2 & e_3 & e_4 & e_5 \\
\hline
e'_1 & 0 & 14 & 19 & 25 & 25 \\
e'_2 & 13 & 19 & 24 & 18 & 18 \\
e'_3 & 19 & 25 & 20 & 14 & 14 \\
e'_4 & 19 & 25 & 20 & 14 & 14 \\
e'_5 & 16 & 22 & 17 & 11 & 11
\end{array}
$$

Let's index the edges in the interval as $e_1, e_2, \ldots, e_n, e'_1, e'_2, \ldots, e'_m$. From the vertex r, e is numbered from left to right and e' is numbered from left to right. Let's say $F(i,j) = cut(e_i, e'_j)$. Now we prove the following:

Theorem 1.1. For all $1 \leq i \leq n-1, 1 \leq j \leq m-1$, Let A be a *Monge array* if $A(i,j) + A(i+1, j+1) \leq A(i, j+1) + A(i+1, j)$ is satisfied. Then F is a Monge array.

To prove this, the following Lemma is used. Proof is omitted.

Lemma 1.2. Edge $g = (u,v)$ belongs to $cut(e,f)$ if and only if exactly one of e, f is on the unique path connecting (u,v) on T.

Proof of Theorem 1.1. Let's define $F_e(i,j)$ as the value that the edge $e \notin T$ contributes to $F(i,j)$. That is, $F(i,j) = \sum_{e \in E(G)-E(T)} F_e(i,j)$. Since the sum of the Monge array is a Monge array, you can simply prove that each of them are MongeThere are three cases.

- Case 1. The section formed by the edge does not include r and is on the left side of it.
- Case 2. The section formed by the edge includes r
- Case 3. The section formed by the edge does not include r and is on his right

Below is a picture of the contribution of the Monge array for each case (Case 1, 2, 3 in order from left to right). Using the definition of Monge array, we can see that the matrices created in these three cases are all Monge arrays.

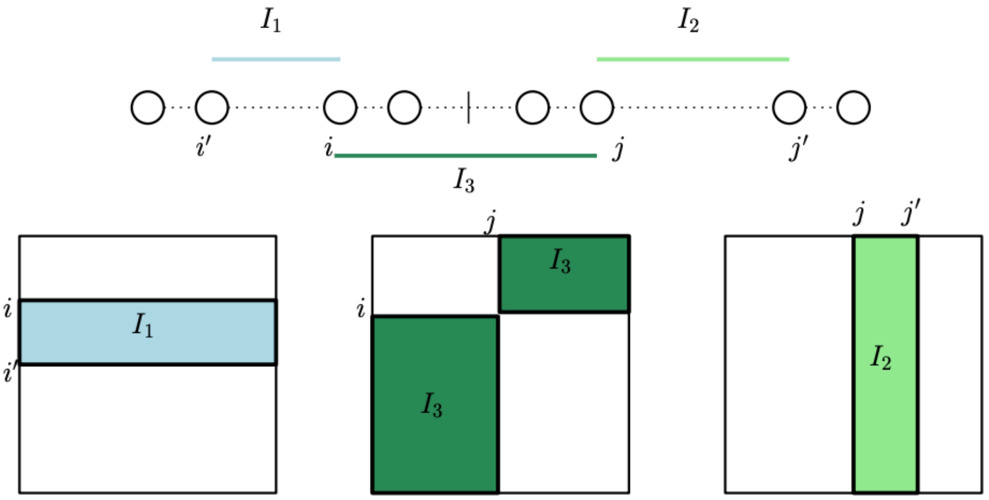


Figure 5: Contribution of each type of Interval (L, R, Crossing) in the cost matrix M.

Theorem 1.3. The minimum value of $n \times m$ Monge array can be found through $O((n+m)\log n)$ F calls.

Proof of Theorem 1.3. In the programming contest, It is well known under the name Divide and Conquer optimization. To briefly reintroduce, let's say $GlobalMin([i_1, i_2], [j_1, j_2]) = min_{i \in [i_1, i_2], j \in [j_1, j_2]} F(i, j)$. When you call this function, you can calculate the total minimum. Assuming $m = (i_1 + i_2)$, let's calculate $min_{j \in [j_1, j_2]} F(m, j)$ directly, and calculate the argument j_{opt} with the lowest value. . Now, according to the definition of the monge array, the minimum value in the $[i_1, m-1]$ section comes from j_{opt} or later, and the minimum in the $[m+1, i_2]$ section is j_{opt} or earlier. Based on this, if the possible section is reduced by j in the division and conquer, only the maximum value of $O(n+m)$ is called in each division and conquer level.

Remark. $O(n+m)$ calls can be accomplished using the SMAWK algorithm, but the author stated that SMAWK is not used since it is unsuitable for non-sequential models.

1.2. When T is a star graph

Now let's solve when T is a star graph. A star graph is a tree with all vertices rooted or adjacent to the root. That is, all nodes except the root are leaves. Define $deg(i)$ as the sum of the weights of all edges adjacent to i, and define $C(i, j)$ as the weight of the edges connecting i and j (0 if none). Define e_i as the edge between i and root (e_{root} is undefined). Let $\Delta(S)$ be defined as the sum of the weights of the edges with both ends in S and $V - S$ respectively.

Observation 1.4. Suppose that every minimum 2-respecting cut contains exactly two tree edges. If the two pair of edges that give a 2-respecting min cut are (e_i, e_j), $deg(i) < 2C(i, j), deg(j) < 2C(i, j)$ holds.

Proof. $cut(e_i, e_j) = deg(i) + deg(j) - 2C(i, j)$. Because $deg(i)$ is the value of the 1-respecting cut with intersection of e_i, $cut(e_i, e_j) < deg(i)$ holds. For the same reason, $cut(e_i, e_j) < deg(j)$. Combine this with above equation.

If all the minimum 2-respecting cuts do not contain two tree edges, there is nothing more to worry about since everything is done in the step of just finding the 1-respecting cut. So let's say the assumption is true. If you want to find a minimum 2-respecting cut that includes any edge e_i, check $cut(e_i, e_j)$ only for j that satisfies $deg(i) < 2C(i, j)$. There exists at most one j with $\frac{deg(i)}{2} < C(i, j)$. This is because the value of $deg(i)$ is exceeded when there are two or more. Such j is the most weighted edge adjacent to i. If you found j, there is only one thing to call with $cut(e_i, e_j)$, so you can solve the problem with $O(n)$ calls.

Now we describe how to find this j. In the cut-query model, for two disjoint subset of vertices $A, B \subseteq V(G), A \cap B = \emptyset,\ between(A, B) = \sum_{i \in A, j \in B} C(i, j) = \frac{\Delta(A) + \Delta(B) - \Delta(A+B)}{2}$. Now for i, finding such j is possible with a binary search, and the problem is solved with $O(n \log n)$ calls.

1.3 General tree T: Orthogonal pair of edges

Again, like Observation 1.4, suppose that every minimum 2-respecting cut contains exactly two tree edges. When calculating $cut(e, f)$, we divide the case into two things.

- e, f is *orthogonal*: $e^{\downarrow} \cap f^{\downarrow} = \emptyset$.
- Not: $e^{\downarrow} \subseteq f^{\downarrow}$ or vice versa. (It is always easy to see this unless you do *orthogonal*.)

This paragraph only covers the case where e and f are *orthogonal*.

If e, f is *orthogonal*, the proposition of Observation 1.4 can be taken almost as it is.

Observation 1.5. Suppose that every minimum 2-respecting cut contains exactly two tree edges. If the two pairs of edges giving a 2-respecting min cut are (e_i, e_j) and the two are orthogonal, $\Delta(e_i^{\downarrow}) < 2 \times between(e_i^{\downarrow}, e_j^{downarrow}), \Delta(e_j^{\downarrow}) < 2 \times between(e_i^{\downarrow}, e_j^{\downarrow})$ holds.

Proof of Observation 1.5. $cut(e_i, e_j) = \Delta(e_i^{\downarrow}) + \Delta(e_j^{\downarrow}) - 2 \times between(e_i^{\downarrow}, e_j^{\downarrow})$. $\Delta(e_i^{\downarrow})$ is the value of the 1-respecting cut with e_i intersection, so $cut(e_i, e_j) < \Delta(e_i^{\downarrow})$ is established . For the same reason, $cut(e_i, e_j) < \Delta(e_j^{\downarrow})$. You can combine this with the above equation.

Naturally, it would be nice to enumerate all e_j that satisfies these conditions for every e_i.

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 with e_i, then e_j is *cross-interesting* to e_i.

Lemma 1.6. For all e_i, cross-interesting e_j forms a path from any vertex v to the root.

Proof of Lemma 1.6. If any e_j is cross-interesting, $e_{par(j)}$ is still cross-interesting under the condition that it is orthogonal with e_i. This is because the RHS of the $between$ function increases. If any two orthogonal e_j, e_k are cross-interesting to e_i, $\Delta(e_i^{\downarrow}) < between(e_i^{\downarrow}, e_j^{\downarrow}) + between(e_i^{\downarrow}, e_k^{\downarrow}) = between(e_i^{\downarrow}, e_j^{\downarrow} \cup e_k^{\downarrow})$ holds. However, the value of the between function increases as the RHS increases, and it is contradictory because $\Delta(e_i^{\downarrow}) = between(e_i^{\downarrow}, V(T) - e_i^{\downarrow})$.

Finding the path of e_j for e_i is not easy. In a typical *cut-query* environment, these paths can be found in *cut sparsifier*. In the sequential algorithm, a set of cross-interesting edges can be found using random sampling. This part is covered in Chapter 2, and here I'm just stating that I can do this. Let's say you have obtained the path of cross-interesting e_j.

Let's do a heavy-light decomposition (HLD) for T. In the case of HLD, a path from a node to the root direction can be decomposed into a maximum of $O(\log n)$ paths. Therefore, the cross-interesting edge for all e_i is represented by the union of $O(\log n)$ paths. This translates the problem into the following query problem:

- Query (e_i, P): Given e_i edge and P tree path on HLD, $e_j \in P$, e_i and e_j for orthogonal P Calculate the minimum value of $cut(e_i, e_j)$.

For convenience, P is considered as a whole path in HLD, not the subsegment of path decomposed in HLD. This is okay because redundancy doesn't affect the answer. We need to efficiently handle these queries when there are $O(n \log n)$ number of such. Here, let's say that in the minimum cut, not only e_j is cross-interesting for e_i as well as **vice versa**, that is, e_i should be interesting for e_j. This condition corresponds exactly to the proposition originally defined in Observation 1.5, which we just overlooked for convenience in introducing Lemma 1.6.

If the HLD path to which e_i belongs is Q, let's process the query with same $\{Q, P\}$ at once ((P, Q) and (Q, P) are considered in same occasion). There are (Q, P) pairs of $O(n \log n)$ equal to the number of queries.

Now first, let's contract all edges that don't belong to Q or P. In this case, the contracted graph may be a straight line or a tri-junction in which Q is combined with P. In the case of a tri-junction, the branch in the direction of root route can be contracted. Therefore, it can be assumed that the contracted graph is straight.

Second, let's contract all edges that did not appear in the form of e_i in the query (e_i, P) or (e_i, Q). As mentioned above, this is an operation that can be performed because the min-cut pair should appear in **both side** of the list. In this case, the different sets of e_i that appear in a query set become the set of edges we will consider. Using the *path algorithm* considered in 1.1 for this set of e_i, we can use $|S| \log^2 |S|$ query to determine the optimal solution. The total number of pairs considered is $O(n \log n)$, so we needed $O(n \log^3 n)$ cut queries to solve this case. All of the contracts mentioned here do not need to be explicit at all, they just need to be understood to the extent that we ignore all non-contracted ones.

1.4 General tree T: Non-orthogonal pair of edges

It can be treated similarly to the orthogonal case. Start by copying all Observations and Lemmas.

Observation 1.7. Suppose that every minimum 2-respecting cut contains exactly two tree edges. If the two pairs of edges giving a 2-respecting min cut are (e_i, e_j) and $e_j^\downarrow \subseteq e_i^\downarrow$, then $\Delta(e_i^\downarrow) < 2 \times between(V(T) - e_i^\downarrow, e_j^\downarrow)$, $\Delta(e_j^\downarrow) < 2 \times between(V(T) - e_i^\downarrow, e_j^\downarrow)$ holds.

Proof of Observation 1.7. $cut(e_i, e_j) = \Delta(e_i^\downarrow) + \Delta(e_j^\downarrow) - 2 \times between(V(T) - e_i^\downarrow, e_j^\downarrow)$. $\Delta(e_i^\downarrow)$ is the value of the 1-respecting cut with e_i intersection, so $cut(e_i, e_j) < \Delta(e_i^\downarrow)$ is established . For the same reason, $cut(e_i, e_j) < \Delta(e_j^\downarrow)$. You can substitute the above equation.

Definition (down-interesting). For some e_i, if $\Delta(e_i^\downarrow) < 2 \times between(V(T) - e_i^\downarrow, e_j^\downarrow)$) is satisfied, then e_j is *down-interesting* for e_i.

Lemma 1.8. For all $e_i = (par(i), i)$, the down-interested e_j forms a path from any vertex v to i.

Proof of Lemma 1.8. If any e_j is down-interesting, $e_{par(j)}$ is still down-interesting under the condition that it is orthogonal with e_i. This is because the RHS of the $between$ function increases. If any two orthogonal e_j, e_k are down-interesting to e_i, $\Delta(e_i^\downarrow) < between(V(T) - e_i^\downarrow, e_j^{downarrow}) + between(V(T) - e_i^\downarrow, e_k^\downarrow) = between(V(T) - e_i^\downarrow, e_j^\downarrow \cup e_k^\downarrow)$ holds. However, the value of the between function increases as the RHS increases, and it is contradictory because $\Delta(e_i^\downarrow) = between(V(T) - e_i^\downarrow, e_i^\downarrow)$.

Through the same sampling method as 1.3, the down-interesting path can be found. Description of this content is omitted. Let's solve the following query problem.

- Query (e_i, P): Given the edge e_i and the path P on the HLD, for all $e_j \in P$ and $e_j^\downarrow \subseteq e_i^\downarrow$, calculate the minimum value of $cut(e_i, e_j)$.

Here, let's assume that $e_i \notin P$ is guaranteed. In that case, after contracting all except P for each HLD path, you can use the path algorithm of 1.1.

When the HLD path to which e_i belongs is called Q, let's process the same query with (Q, P) at once. The order does matter, and Q is below P because of ancestor relations. There are (Q, P) pairs of $O(n \log n)$ equal to the number of queries. Because it cannot be processed in both directions as above, contract the edge that does not appear with e_i only in Q, and leave P as it is. Of course, the rest of the trunks are all contracted, so the result is similar to the above three-way / straight-line form.

Subsequently, path algorithm is performed at $O((|P|+|Q|)\log^2(|P|+|Q|))$. For Q, the sum of appearances is $O(n\log n)$. For the P, note that Q is an ancestor path of P, so there exists at most $O(\log N)$ such Q. Therefore, even if there is no compression for P, one path appears $O(\log N)$ times anyway, so we call path algorithm for total $O(n\log n)$ edges.

1.5 Final algorithm

Let's put together all the pieces and organize the final algorithm. first

- 1-respecting cut is obtained by cut-query of $O(n)$ times.
- After calculating the HLD of T, for each of the paths decomposed by HLD, a 2-respecting cut is found using the algorithm of the path case. At this stage, the case where the two edges to be erased belong to one HLD path is processed.
- For the edge e of T, obtain the cross-interesting path and down-interesting path. As of Lemma 1.6/1.8, there are only two paths that have to find the cut corresponding to e. How to get this path is slightly different for each calculation model, and it is explained in Chapter 2 because *cut sparsifier* is needed. However, I guarantee no oracle calls over $O(n\log^3 n)$ are required here.
- In the cross-interesting path, the path algorithm is called through reduction introduced in 1.3.
- In the down-interesting path, the path algorithm is called through reduction introduced in 1.4.

Finally, the minimum cut was obtained with $O(n\log^3 n)$ oracle calls.

2. Implementation detail in each computation model

2.1 $O(\log n)$-time cut query in sequential setting

Except the part of finding an interesting edge set, all queries asked in the above procedure is about 2-respecting cuts. That is, even in the sequential model, the problem can be solved by an efficient computation of 2-respecting cut. by adding the call time of the function to the above algorithm. Here, we introduce a method to calculate 2-respecting cut in $O(\log n)$ per query after preprocessing $O(m\log n)$ time. Therefore, the algorithm described above can be implemented in sequential model at $O(m\log n + n\log^4 n)$ time.

Use a Persistent segment tree (PST) (other 2D query data structures are fine, but PST is the cleanest). On an Euler tour of T, the vertices belonging to any subtree $e_i^\downarrow$ correspond to the section on the Euler tour. Therefore, in all cases, the area of the cut appears as $O(1)$ intervals as follows:

- In case of 1-respecting, $e_i^\downarrow$ is one section.
- If the two edges are orthogonal, $e_i^\downarrow$ and $e_j^\downarrow$ are expressed in two sections.
- If the two edges are not orthogonal, the section called $e_i^\downarrow$ is removed from the section called $e_j^\downarrow$.

We want edges where one endpoint belongs to this section and the other does not. The sets of edges crossing the cuts can be expressed as a union of $O(1)$ rectangles (one end in some interval, other end in another interval) on the Euler tour. Therefore, the entire operation can be done in $O(\log n)$ time per edge after preprocessing in $O(m \log n)$.

2.2 Finding interesting edge set in sequential setting with random sampling

First, to find the entire set of cross-interesting edges, by Lemma 1.6, it is enough to know only the deepest edge of the edges in the set. So for e_i you will find the deepest edge e_j.

Suppose the weights of all edges are 1. In this case, if you sample **any edge** from $\Delta(e_i^{\downarrow})$ with a random probability, you will get the edge in $between(e_i^{\downarrow}, e_j^{\downarrow})$ with at least $\frac{1}{2}$ probability. To satisfy a WHP condition, you can sample $\log n$ edges. The weight of the edge is not 1, but it does not matter if each edge is sampled with a probability that is linearly proportional to the weight. Sampling any edge from $\Delta(e_i^{\downarrow})$ is a very similar method to counting using the Persistent segment tree, and it can be done with complexity $O(\log n)$ with the same data structure used in cut computation.

In this sampled edge, if v is an endpoint that does not belong to $e_i^{\downarrow}$, e_j exists on the path between v and the root. To calculate the $between$ function, $\Delta(e_i^{\downarrow}), \Delta(e_j^{\downarrow})$, and $cut(e_i, e_j)$ are required. They are all 2-respecting cuts and can be calculated. Therefore, you can find these e_j through binary search, and return the lowest depth among all e_j found. If you do this, you can find e_j with $O(\log^2 n)$ cut-query.