Mincut Sensitivity Data Structures for the Insertion of an Edge

Surender Baswana
Department of Computer Science & Engineering, IIT Kanpur, Kanpur – 208016, India
sbaswana@cse.iitk.ac.in

Shiv Gupta
Department of Computer Science & Engineering, IIT Kanpur, Kanpur – 208016, India
shivguptamails@gmail.com

Till Knollmann
Heinz Nixdorf Institute, Paderborn University, Fürstenallee 11, 33102 Paderborn, Germany
tillk@mail.upb.de

Abstract

Let $G = (V, E)$ be an undirected graph on n vertices with non-negative capacities on its edges. The *mincut sensitivity problem* for the insertion of an edge is defined as follows.

Build a compact data structure for G and a given set $S \subseteq V$ of vertices that, on receiving any edge $(x, y) \in S \times S$ of positive capacity as query input, can efficiently report the set of all pairs from $S \times S$ whose mincut value increases upon insertion of the edge (x, y) to G.

The only result that exists for this problem is for a single pair of vertices (Picard and Queyranne, Mathematical Programming Study, 13 (1980), 8-16). We present the following results for the single source and the all-pairs versions of this problem.

1. *Single source*: Given any designated source vertex s, there exists a data structure of size $\mathcal{O}(|S|)$[1] that can output all those vertices from S whose mincut value to s increases upon insertion of any given edge. The time taken by the data structure to answer any query is $\mathcal{O}(|S|)$.
2. *All-pairs*: There exists an $\mathcal{O}(|S|^2)$ size data structure that can output all those pairs of vertices from $S \times S$ whose mincut value gets increased upon insertion of any given edge. The time taken by the data structure to answer any query is $\mathcal{O}(k)$, where k is the number of pairs of vertices whose mincut increases.

For both these versions, we also address the problem of reporting the values of the mincuts upon insertion of any given edge. To derive our results, we use interesting insights into the *nearest* and the *farthest* mincuts for a pair of vertices. In addition, a crucial result, that we establish and use in our data structures, is that there exists a directed acyclic graph of $\mathcal{O}(n)$ size that compactly stores the farthest mincuts from all vertices of V to a designated vertex s in the graph. We believe that this result is of independent interest, especially, because it also complements a previously existing result by Hariharan et al. (STOC 2007) that the nearest mincuts from all vertices of V to s is a laminar family, and hence, can be stored compactly in a tree of $\mathcal{O}(n)$ size.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Mathematics of computing → Network flows; Mathematics of computing → Graph algorithms

Keywords and phrases Mincut, Sensitivity, Data Structure

Digital Object Identifier 10.4230/LIPIcs.ESA.2020.52

Related Version A full version including all omitted proofs can be found online here: `http://www.cse.iitk.ac.in/users/sbaswana/Papers-published/esa-2020-fv.pdf`.

Funding *Surender Baswana*: The research work was carried out while the author was at Heinz Nixdorf Institute, on leave from IIT Kanpur. The research was supported by Alexander von

[1] Data structure sizes are in *words* unless specified otherwise, where a word occupies $\Theta(\log n)$ bits.

Humboldt Foundation.
Till Knollmann: This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre On-The-Fly Computing (GZ: SFB 901/3) under the project number 160364472.

1 Introduction

Let $G = (V, E)$ be a graph on $n = |V|$ vertices and $m = |E|$ edges with a non-negative capacity on each edge. A *mincut* for a pair of vertices u and v is a set of edges with least capacity whose removal disconnects v from u. It is a fundamental concept in graph theory. Moreover, the area of designing algorithms for the mincut and its variants has been extensively researched ever since the seminal result on the maxflow-mincut duality by Ford and Fulkerson [4].

It is often the case that one is more interested in the mincuts between vertices belonging to a relatively small part of the input graph than the mincuts between all vertices. Hence, consider a subset of vertices $S \subseteq V$ and any subset of pairs of vertices $Q \subseteq S \times S$ whose mincut we are interested in. The objective is to have the knowledge about how sensitive the mincuts of pairs of vertices from Q are, with respect to any change in the subgraph induced by S. This change could be a change in the capacity of an existing edge in the subgraph induced by S or insertion of a new edge between any two vertices in S. This knowledge of the impact on various mincuts due to any change in the network can make the network administrators well prepared for such changes when they indeed occur in future.

An important measure of the impact of a change in the capacity of an edge is the number of pairs of vertices whose mincut value changes. The change in the capacity of an edge could be either an increase or a decrease. We focus on the case when the capacity of an edge is allowed to increase only. For this data structure problem, the query input is a new edge with positive capacity or an existing edge whose capacity is increased. It can easily be observed that as far as the mincut between any pair of vertices is concerned, increasing the capacity of an existing edge (x, y) by amount Δ is equivalent to adding one more edge between x and y with capacity Δ. So henceforth, we only consider the insertion of an edge. On receiving any such edge, the objective is to efficiently report the pairs of vertices whose mincut increases.

Based on the discussion above, we now formally define the problem of mincut sensitivity.

▶ **Problem 1.** *Preprocess $G = (V, E)$, a set $S \subseteq V$, and a set $Q \subseteq S \times S$ to build a compact data structure that, on receiving any edge $(x, y) \in S \times S$ of positive capacity as query input, can efficiently report all those pairs from Q whose mincut value increases upon insertion of the edge (x, y) to G.*

We expect the bounds for the data structure of Problem 1 to depend on only the size of S instead of V. For simultaneously achieving efficient query time and compact space, the only previous solution that exists for this problem is for a single pair of vertices only, i.e., when $|Q| = 1$. It consists of a data structure that occupies $\mathcal{O}(|S|)$ space and achieves $\mathcal{O}(1)$ query time. This solution follows from an observation made by Picard and Queyranne in their seminal paper [10].

1.1 Our Contribution

We address the single source and the all-pairs versions of Problem 1, along with the problem of reporting the new value of the mincut between any pair of vertices after the edge insertion.

Single source all destinations: In the single source case, we are interested in the mincuts between a designated vertex $s \in S$ and all other vertices from set S, i.e., the (s,t)-mincut, for all $t \in S \setminus \{s\}$.

▶ **Theorem 1.** *For an undirected graph and any subset S of vertices with a designated vertex $s \in S$, there exists an $\mathcal{O}(|S|)$ size data structure that can report all those vertices from S whose mincut value to s increases upon insertion of any given edge from $S \times S$. The time taken by this data structure to answer any such query is $\mathcal{O}(|S|)$.*

The $\mathcal{O}(|S|)$ space and $\mathcal{O}(|S|)$ query time of our data structure for undirected graphs can be much less than $\mathcal{O}(|V|)$ for smaller S. Also, it is in sharp contrast with the following lower bound result for directed graphs that we also prove.

▶ **Theorem 2.** *Any data structure for a directed graph to answer a mincut sensitivity query from a designated source vertex to any designated subset of q vertices must use $\Omega(q^2)$ bits of space for at least one directed graph.*

The proof of Theorem 2 basically establishes that any such data structure can be used to store any balanced bipartite graph on $2|S|$ vertices implicitly. Interestingly, the same proof also establishes an $\Omega(|S|^2)$ lower bound for the single source version of two other fundamental problems for directed graphs, namely, *reachability sensitivity* as well as *distance sensitivity* for the insertion of an edge. These facts add more significance to the result stated in Theorem 1 for the single source mincut sensitivity.

All-pairs: When considering the mincuts between all pairs of vertices in S, i.e., the (u,v)-mincut, for all $u, v \in S$ such that $u \neq v$, we obtain the following result.

▶ **Theorem 3.** *For an undirected graph and any subset S of vertices, there exists an $\mathcal{O}(|S|^2)$ size data structure that can report all those pairs of vertices from $S \times S$ whose mincut value increases upon insertion of any given edge from $S \times S$. The time taken by the data structure to answer any such query is $\mathcal{O}(k)$, where k is the number of pairs whose mincut increases.*

Note that the query time of the data structure in Theorem 3 is optimal. Moreover, if the objective is to report just the number of all-pairs from the set $S \times S$ whose mincut increases upon insertion of any given edge, our data structure can accomplish this objective in $\mathcal{O}(\min(k, |S| \log |S|))$ time, which is $\mathcal{O}(|S| \log |S|)$ always.

▶ Remark 4. Our results for mincut sensitivity directly extend to maxflow sensitivity as well due to the equivalence between maxflow and mincut [4].

To achieve all our results, we use interesting insights into the *nearest* and the *farthest* mincuts for a pair of vertices – two concepts that exist since the seminal work of Ford and Fulkerson on maximum flow [4]. Additionally, a crucial result about the farthest mincuts that we establish and use in one of our data structures is the following.

▶ **Theorem 5.** *For an undirected graph on n vertices and a designated source vertex s, there exists a directed acyclic graph (DAG) of size $\mathcal{O}(n)$ that compactly stores the farthest mincuts from all vertices $v \in V \setminus \{s\}$ to s. For any v, the set of vertices defining the farthest mincut from v to s can be reported in time that is of the order of the size of the set.*

The graph theoretic result of Theorem 5 is of independent interest in addition to its applications in the mincut sensitivity problem. This is because, not only it adds to our understanding of mincuts, but it also complements an earlier result of Hariharan et al. [8] that showed that the nearest mincuts from all vertices to s form a laminar family, and hence, can be stored in a tree data structure occupying only $\mathcal{O}(n)$ space.

1.1.1 On reporting the value of mincut

In addition to reporting the pairs of vertices whose mincut increases upon insertion of a given edge, it may be important to output the new values of their mincuts. Indeed, if the edge capacities in the graph are integral and the inserted edge has unit capacity, our data structures from Theorems 1 and 3 can also report the new values of the affected mincuts upon insertion of an edge, i.e., the value of the mincut will be increased by one for the reported pairs of vertices. However, if there is no restriction on the capacity of the inserted edge, we show that even for the single source case it is not possible to accomplish this objective with any data structure of subquadratic size.

▶ **Theorem 6.** *There exists a set $\mathcal{G}$ of undirected graphs on n vertices with integer edge capacities in the range $[1, n^{2+\varepsilon}]$ (for any $\varepsilon > 0$) for which the following claim holds true. Any data structure for an undirected graph that can report the value of the mincut between a designated source vertex and any other vertex upon insertion of any edge of integer capacity polynomial in n must require $\Omega(n^2 \varepsilon \log n)$ bits of space for at least one graph from $\mathcal{G}$.*

For the all-pairs case, it turns out that any such data structure also provides a generalization of the flow-tree [5, 7]. That is, the data structure will also be able to report the mincut value between a vertex u and a pair $\{x, y\}$ of vertices for any $u, x, y \in V$. Chitnis, Kamma, and Krauthgamer [3] showed that there will be total $\mathcal{O}(n^2)$ distinct mincut values separating any vertex from any pair of vertices in an undirected graph. However, to the best of our knowledge, designing an $\mathcal{O}(n^2)$ size data structure that returns the value of any such mincut in non-trivial query time is still an open problem.

1.2 Overview of our results

We begin with the result of Picard and Queyranne [10] for mincut sensitivity for a source-destination pair (s, t). For any maximum flow f from s to t, let G_f be the corresponding residual graph. Notice that there is no path from s to t in G_f. Let R be the set of vertices which are reachable from s in G_f, and let T be the set of vertices from which t is reachable in G_f. Picard and Queyranne [10] made the following crucial observation.

▶ **Lemma 7** (Picard and Queyranne [10])**.** *The maxflow from s to t increases upon insertion of an edge (x, y) if and only if x belongs to R and y belongs to T.*

Without loss of generality, we can assume that the vertices of set S are labeled from 1 to $|S|$. Based on Lemma 7, the data structure for mincut sensitivity for a (s, t)-pair where $s, t \in S$, stores each of $R \cap S$ and $T \cap S$ in Boolean arrays indexed by the vertices of set S.

The subsets R and $V \backslash T$ in Lemma 7 turn out to be the smallest, and the largest subsets of vertices defining a mincut from s to t, respectively. In the literature on mincuts, R and $V \backslash T$ are respectively called the *nearest* and the *farthest* mincut from s to t; we define these notions more formally in the next section. Therefore, in order to design a compact and efficient data structure for the single source and the all-pairs versions of the mincut sensitivity problem, it is natural to explore if we can have a compact way to store these two types of cuts for multiple pairs of vertices. We now provide an overview of the compact data structures for the nearest and the farthest mincuts, and the way these data structures are used to solve the mincut sensitivity problem.

Compact data structures for the nearest and the farthest mincuts

Interestingly, Hariharan et al. [8] showed that the nearest mincuts from all vertices to a designated vertex s in an undirected graph form a laminar family – If the subsets of vertices

defining the nearest mincuts from u to s, and v to s intersect, then one of them must be a subset of the other. As a result, the nearest mincuts from all vertices to s can be stored compactly in a tree data structure occupying $\mathcal{O}(n)$ space only. However, the farthest mincuts do not constitute a laminar family. Let F_u and F_v be the subsets of vertices that define the farthest mincuts to s from u and v respectively. It is quite possible that F_u and F_v intersect each other but none of them is a subset of the other. In other words, two intersecting farthest mincuts to s may *cross* each other. Moreover, there may be $\Theta(n)$ vertices whose farthest mincuts to s cross the farthest mincut of a single vertex to s. This poses a challenge for designing a compact data structure for storing all the farthest mincuts to s. However, we overcome this challenge using crucial insights into the farthest mincuts.

Using the submodularity of cuts, we first establish the existence of a DAG on $\mathcal{O}(n)$ nodes that stores the farthest mincuts from all vertices to any designated vertex s. However, this DAG could have $\mathcal{O}(n^2)$ edges, and establishing the sparsity of the DAG turns out to be the main hurdle. We overcome this by proving the following interesting property of the farthest mincuts to s:

For any three vertices, either the intersection of their farthest mincuts to s is empty or the farthest mincut from at least one of them is a subset of one of the other two.

Using this property, we are able to prune away all the unnecessary edges from the DAG structure storing farthest mincuts to s. As a result, each node in the DAG turns out to have at most 2 incoming edges, so the size of the DAG is $\mathcal{O}(n)$.

For the objective of solving mincut sensitivity for a subset $S \subseteq V$, we present data structures for the nearest mincuts (likewise the farthest mincuts) that consist of vertices of S only instead of V. Their size is $\mathcal{O}(|S|)$. See Theorem 17 and Theorem 5.

Solving the mincut sensitivity problem

For solving the single source mincut sensitivity problem, we use the tree data structure storing the nearest mincuts and the DAG data structure storing the farthest mincuts, from all vertices of the set S to s. The size of the data structure is $\mathcal{O}(|S|)$. Following Lemma 7, it takes $\mathcal{O}(|S|)$ time using this data structure to determine whether the insertion of a given edge increases the (s, v)-mincut value for any vertex $v \in S$. This leads to $\mathcal{O}(|S|^2)$ time to find all vertices from S whose mincut value from s increases due to the insertion of a given edge. In order to reduce the query time to $\mathcal{O}(|S|)$, we make use of the following insight:

A vertex $x \in S$ belongs to the farthest mincut from v to s if and only if x is reachable from v in the DAG structure storing the farthest mincuts to s.

Solving the all-pairs version of the mincut sensitivity problem with optimal query time turns out to be more challenging. As the underlying graph is undirected, the subset of vertices that defines the farthest mincut from u to v is the complement of the subset of the vertices that defines the nearest mincut from v to u. As a result, keeping the nearest-mincut tree data structure for each vertex of S suffices to solve this problem. The data structure takes $\mathcal{O}(1)$ time to determine for any pair (u, v), whether insertion of an edge, say (x, y), increases (u, v)-mincut value. This observation implies an $\mathcal{O}(|S|^2)$ time algorithm for computing all pairs of vertices from set S whose mincut value increases upon insertion of edge (x, y). But it is quite wasteful if k, the number of pairs whose mincut value increases, is much smaller than $|S|^2$. To accomplish $\mathcal{O}(k)$ query time, we make use of multiple insights into the structure of the nearest mincuts. The most crucial insight is the following:

The vertices whose mincut value to s increases upon insertion of an edge (x, y) lie contiguously on the paths from x and y to their lowest common ancestor in the tree that stores the nearest mincuts to s.

This insight leads to an $\mathcal{O}(|S| + k)$ query time. To get rid of the additive factor of $|S|$, we use another insight that helps finding the *right* pool of vertices whose nearest-mincut tree we need to query.

1.3 Related work

Our research is related to the field of dynamic graph algorithms that emphasizes on efficient data structures to handle changes in a network. For dynamic graph algorithms, the objective is to maintain the solution of a problem for an online sequence of edge insertions or deletions with worst case time complexity better than that of the best static algorithm. There do exist efficient dynamic algorithms for maintaining a global mincut – an incremental algorithm by Goranci, Henzinger, and Thorup [6], and a fully dynamic algorithm by Thorup [12]. However, there does not exist any dynamic algorithm for all-pairs mincuts whose worst case time complexity is better than the best static algorithm. Hartmann and Wagner [9] presented a fully dynamic algorithm for maintaining an all-pairs mincut tree for undirected graphs. Although it achieves a significant speedup over the best static algorithm on many real world graphs, its worst case asymptotic time complexity is not better than the best static algorithm for an all-pairs mincut tree.

1.4 Organization of the paper

Equipped with notations, definitions, and well known lemmas introduced in Section 2, we present the compact data structures for nearest and farthest mincuts in Sections 3 and Section 4, respectively. The data structures for the single source and the all-pairs versions of the mincut sensitivity problem are presented in Section 5 and 6 respectively. We present the lower bound results (Theorems 2 and 6) for the mincut sensitivity problem in Section 7.

2 Preliminaries

Our results consider an undirected graph $G = (V, E)$ on n vertices where each edge is assigned a non-negative capacity through a function $c : E \to R^+$.

▶ **Definition 8** ((s,t)-cut)**.** *A subset of edges whose removal disconnects t from s is called an (s,t)-cut. An (s,t)-mincut is an (s,t)-cut of smallest capacity.*

▶ **Definition 9** (set defining a cut)**.** *A subset $A \subset V$ is said to define an (s,t)-cut if $s \in A$ and $t \notin A$. The corresponding cut is denoted by cut$(A, \bar{A})$ or more compactly cut(A).*

When there is no scope of confusion, we do not distinguish between a mincut and the set defining the mincut. We can extend the capacity function c on edges to any subset $A \subset V$ in a natural way as follows: $c(A)$ denotes the sum of the capacities of all those edges which have exactly one endpoint in A. With this generalization, we now state a well-known property of cuts, namely, the submodularity of cuts.

▶ **Lemma 10** (Submodularity of cuts)**.** *Given an undirected graph $G = (V, E)$ with positive edge capacities, the following inequality holds true for any two subsets $A, B \subset V$.*

$$c(A) + c(B) \geq c(A \cup B) + c(A \cap B)$$

The following lemma states an important property of an (s,t)-mincut.

▶ **Lemma 11.** *Let $A \subset V$ define an (s,t)-mincut with $s \in A$. For any subset $A' \subset A$ with $s \notin A'$, if α is the number of edges incident on A' from $V \backslash A$, and β is the number of edges incident on A' from $A \backslash A'$, then $\alpha \leq \beta$.*

Proof. With α and β as defined in the lemma, let γ represent the number of edges between $A \setminus A'$ and $V \setminus A$. Let λ_{st} denote the value of a (s,t)-mincut. The value of a (s,t)-mincut is equal to the sum of the number of edges from $A \setminus A'$ to $V \setminus A$ and the number of edges from A' to $V \setminus A$, i.e.,

$$\alpha + \gamma = \lambda_{st} \tag{1}$$

Note that since $s \in A \setminus A'$, $A \setminus A'$ defines an (s,t)-cut, and the capacity of this cut is $\beta + \gamma$. Using the fact that any (s,t)-mincut has the least capacity among all (s,t)-cuts, we have:

$$\lambda_{st} \leq \beta + \gamma, \text{ i.e., } \alpha + \gamma \leq \beta + \gamma \text{ (using Equation 1)} \tag{2}$$

Hence, $\alpha \leq \beta$. ◀

2.1 The nearest and the farthest mincuts

▶ **Definition 12** (Nearest and farthest mincuts from s to t)**.** *The subset $A \subset V$ with $s \in A$ is said to define the nearest (likewise the farthest) mincut from s to t if (1) cut$(A, \bar{A})$ defines an (s,t)-mincut, and (2) For every other subset $A' \subset V$ that defines an (s,t)-mincut, $A \subset A'$ (likewise $A' \subset A$). We use s_t^N and s_t^F to denote the nearest and the farthest mincut from s to t, respectively.*

One can easily show using Lemma 10 that the nearest and the farthest mincut from s to t are unique. Additionally, t_s^N and s_t^F partition the set of vertices V as stated in the following lemma.

▶ **Lemma 13.** *For any pair of vertices $s, t \in V$, (i) $s_t^N \cap t_s^N = \emptyset$, and (ii) $s_t^F = V \setminus t_s^N$.*

Proof. We first show that $s_t^F = V \setminus t_s^N$. Let A denote s_t^F and let A' denote $V \setminus t_s^N$. Using the submodularity of cuts (Lemma 10),

$$c(A) + c(A') \geq c(A \cap A') + c(A \cup A') \tag{3}$$

It is evident that both $A \cap A'$ and $A \cup A'$ define (s,t) cuts, since s belongs to both A and A' and t belongs to none. A as well as A' define some (s,t)-mincut. Therefore, it follows from Equation 3 that $c(A \cup A') = c(A \cup A') = c(A) = c(A')$. In other words, $A \cup A'$ define a (s,t)-mincut. This implies that $A' \subseteq A$, since otherwise it would imply that $|A \cup A'| > |A|$ which contradicts the fact that A defines the farthest mincut from s to t.

Now we shall prove that $A \subseteq A'$. The proof is by contradiction. Suppose A is not subset of A', then the fact $A' \subseteq A$, that we established above, would imply that $|A| > |A'|$. In other words, $|V \setminus A| < |t_s^N|$ since $A' = V \setminus t_s^N$. A defines a (s,t)-mincut, so $V \setminus A$ defines a valid (t,s)-mincut since the graph is undirected. As a result, if $|V \setminus A| < |t_s^N|$, it would imply that there is a smaller set than t_s^N that defines a (t,s)-mincut – a contradiction.

So we can conclude that $s_t^F = V \setminus t_s^N$. As a result, $s_t^F \cap t_s^N = \emptyset$. This also implies that $s_t^N \cap t_s^N = \emptyset$ since s_t^N is a subset of s_t^F. ◀

In the light of Lemma 13, we can restate Lemma 7 for undirected graphs as follows.

▶ **Lemma 14** (Picard and Queyranne [10])**.** *The insertion of an edge (x,y) can increase the mincut between s and t if and only if $x \in s_t^N$ and $y \in t_s^N$ or vice versa.*

▶ Remark 15. In order to explore the relationship among the farthest mincuts from a set of vertices to a vertex s, we focus only on the connected component of s. This is because for each vertex outside this component, its farthest mincut to s is obvious. Therefore, without loss of generality we assume G to be connected in the rest of the paper.

3 A compact data structure for all nearest mincuts to vertex s

The following theorem plays the key role in compactly storing all nearest mincuts to s.

▶ **Theorem 16** (Hariharan et al. [8]). *For any two distinct vertices $u, v \in S$, either u_s^N and v_s^N are mutually disjoint or one of them is a subset of the other.*

For a given subset $S \subseteq V$, let $\mathcal{N} = \{x_s^N \cap S | x \in S \backslash \{s\}\}$. Using Theorem 16 we can arrange the sets of $\mathcal{N}$ in a forest of disjoint trees as follows. We refer to a vertex in this forest as *node*. For each set present in $\mathcal{N}$, we create a unique node in the forest. We assign each vertex $v \in S$ to the node ν corresponding to $v_s^N \cap S$. The parent of a node ν is defined as the unique node μ such that the set corresponding to μ is the smallest superset of the set corresponding to ν, if such a superset exists. If no such superset exists, ν will be the root of a tree. We create a dummy node and assign it as the parent of the root of every tree in this forest. Let us denote the resulting rooted tree by $\mathcal{T}(s)$. Figure 1 shows an example graph and the corresponding $\mathcal{T}(s)$ for the case $S = V$.

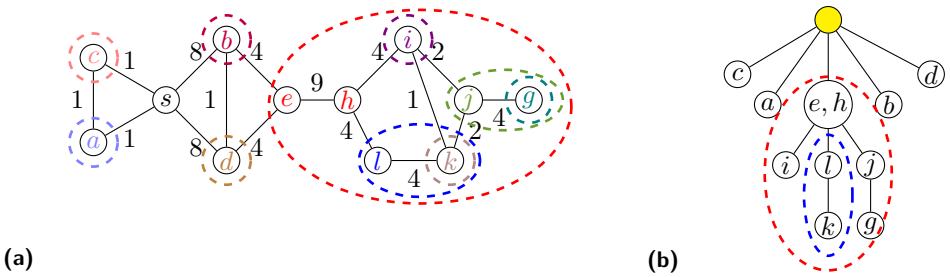


Figure 1 **(a)** The nearest mincut from a vertex to s is encircled with same color. **(b)** Tree $\mathcal{T}(s)$.

It can be observed that if a vertex $v \in S$ is assigned to node ν, then the subtree rooted at ν stores the set $v_s^N \cap S$. So it follows that a vertex, say x, belongs to $v_s^N \cap S$ if either x and v are assigned to the same node in $\mathcal{T}(s)$ or the node containing v is an ancestor of the node containing x. This check can be easily done in $\mathcal{O}(1)$ time if we augment $\mathcal{T}(s)$ to answer lowest common ancestor (LCA) query for any pair of nodes (see [2]). We can thus state the following theorem.

▶ **Theorem 17.** *For an undirected graph $G = (V, E)$, a subset $S \subseteq V$, and any vertex $s \in S$, there exists an $\mathcal{O}(|S|)$ size data structure $\mathcal{T}(s)$ that can report in $\mathcal{O}(1)$ time whether $x \in v_s^N$ for any $x, v \in S$.*

4 A compact data structure for all farthest mincuts to vertex s

In this section, we present a novel data structure that compactly stores the farthest mincuts to vertex s from a subset of vertices. Our main result can be summarized as follows.

▶ **Theorem 18.** *For an undirected graph $G = (V, E)$, any subset $S \subseteq V$, and a designated vertex $s \in S$, there exists a directed acyclic graph $\mathcal{D}(s)$ having $\mathcal{O}(|S|)$ nodes and $\mathcal{O}(|S|)$ edges that can report $v_s^F \cap S$ in time of the order of the size of $v_s^F \cap S$ for any $v \in S \backslash \{s\}$.*

Lemma 13(ii) implies that $s_v^N = V \backslash v_s^F$. So we can compute $s_v^N \cap S$ in $\mathcal{O}(|S|)$ time once we have $v_s^F \cap S$. Therefore, we can state the following corollary of Theorem 18.

▶ **Corollary 19.** *For an undirected graph $G = (V, E)$, any subset $S \subseteq V$, and a designated vertex $s \in S$, there exists a data structure of $\mathcal{O}(|S|)$ size that takes just $\mathcal{O}(|S|)$ time to compute $s_v^N \cap S$ for any $v \in S \backslash \{s\}$.*

Next, we show how to compute a DAG storing all farthest mincuts to s in space $\mathcal{O}(|S|^2)$. Thereafter, we reduce its space complexity to $\mathcal{O}(|S|)$ only.

4.1 A DAG of size $\mathcal{O}(|S|^2)$

Our data structure to store all farthest mincuts to a designated vertex s uses the observation captured in Lemma 20. In its core, it tells us that certain farthest mincuts are related by a subset relation which can be exploited to store them compactly.

▶ **Lemma 20.** *Let x and v be any two vertices in G. If $x \in v_s^F$, then $x_s^F \subseteq v_s^F$.*

Proof. We use Lemma 10 on the submodularity of cuts and provide a proof by contradiction. Let A and B refer to the sets v_s^F and x_s^F, respectively. It is given that $x \in v_s^F$. Assume that $x_s^F \not\subseteq v_s^F$. This would imply that $B \backslash A \neq \emptyset$, hence A must be a proper subset of $A \cup B$. Refer to Figure 2 for the illustration of this configuration.

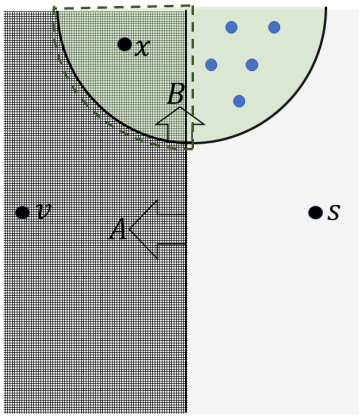


Figure 2 $A \cap B$ is the region surrounded by the dotted boundary; and the triangular region below it has non zero number of vertices signifying that $A \cap B$ is a proper subset of A.

Observe that $A \cap B$ defines a valid (x, s)-cut since x is present in both A and B, whereas $s \notin x_s^F$. This observation implies that $c(A \cap B) \geq c(B)$ since B defines an (x, s)-mincut. This inequality and Lemma 10 imply the following inequality.

$$c(A \cup B) \leq c(A) \tag{4}$$

Now observe that $A \cup B$ defines a valid (v, s)-cut since v belongs to A, whereas s belongs neither to A nor to B. Since A defines a (v, s)-mincut, so Inequality 4 implies that $c(A \cup B)$ must be equal to the capacity of an (v, s)-mincut. But A is a proper subset of $A \cup B$. This would imply that the cut defined by A is not the farthest mincut from v to s – a contradiction. ◀

Let $\mathcal{F} = \{v_s^F \cap S \mid v \in S \backslash \{s\}\}$. We now use Lemma 20 to build a directed acyclic graph $D = (\mathcal{V}, \mathcal{E})$ that stores $\mathcal{F}$ as follows. We use *node* to refer to a vertex of this DAG.

For each set present in $\mathcal{F}$, we create a unique node in D. The set of nodes thus created constitutes $\mathcal{V}$. We denote by $\mathcal{F}(\nu)$ the set in $\mathcal{F}$ corresponding to node ν. The edge set $\mathcal{E}$ of D is defined as follows.

$$\mathcal{E} = \{(\nu, \mu) \mid \mathcal{F}(\mu) \subset \mathcal{F}(\nu)\}$$

It can be observed that if $X \subset Y$ for any two sets X and Y in $\mathcal{F}$, then $|X| < |Y|$. Hence D is acyclic. To efficiently retrieve $x_s^F \cap S$ for any given $x \in S$, we can augment D as follows.

- We create an array J_s indexed by vertices of set S such that for any $v \in S \setminus \{s\}$, $J_s[v]$ stores the pointer to node μ that corresponds to $v_s^F \cap S$, that is, $\mathcal{F}(\mu) = v_s^F \cap S$.
- Each node μ of D stores a list $L(\mu)$ of all those vertices $v \in S$ such that $J_s[v] = \mu$.
- We introduce a dummy node and add an edge from it to every other node which has no incoming edge.

Lemma 21 follows immediately from Lemma 20 and the construction of D described above.

▶ **Lemma 21.** *Let x and u be any two vertices of set S. x is present in $u_s^F \cap S$ if and only if either $J_s[u] = J_s[x]$ or there is an edge from $J_s[u]$ to $J_s[x]$ in D.*

Lemma 21 implies that for each vertex $v \in S \setminus \{s\}$, $v_s^F \cap S$ is the set of vertices stored in the list $L(J_s[v])$ and the lists of all the nodes with an incoming edge from $J_s[v]$ in D.

The subset relation $\subset$ is transitive. So, if there is a path from a node ν to another node μ in D, then (ν, μ) is also an edge in D. In other words, the transitive closure of D is D itself. This observation in conjunction with Lemma 21 leads us to the following lemma which will be crucial for our data structure for the single source mincut sensitivity problem.

▶ **Lemma 22.** *Let x and u be any two vertices in set S. x is present in $u_s^F \cap S$ if and only if $J_s[x]$ is reachable from $J_s[u]$ in D.*

Notice that D has $\mathcal{O}(|S|)$ nodes, but it could have $\Theta(|S|^2)$ edges. So the total space occupied by D could be $\Theta(|S|^2)$. A natural idea to overcome this hurdle is to remove as many edges as possible from D without affecting the reachability between any pair of its vertices so that Lemma 22 continues to hold. In other words, we compute another DAG D^τ which is the transitive reduction of D. Aho, Garey, and Ullman [1] showed that computing the transitive reduction of a DAG is as easy as computing its transitive closure. While in general a transitive reduction does not always lead to a reduced number of edges, it does so in the case of D. In the following subsection we present crucial insights into crossing farthest mincuts that ensure that each node of D^τ will have at most two incoming edges. The data structure $\mathcal{D}(s)$ in Theorem 18 for storing all farthest mincuts to s is D^τ only.

4.2 Bounding the in-degree of D^τ by 2

A set of vertices $\mathcal{I} \subset V$ is said to be a set of *incomparable* vertices with respect to the mincuts to s if for each $u, v \in \mathcal{I}$ with $u \neq v$ it holds that $u \notin v_s^F$ and $v \notin u_s^F$. The following lemma highlights an important property for a pair of incomparable vertices.

▶ **Lemma 23.** *For any two incomparable vertices u and v, there does not exist any edge between the set $u_s^F \cap v_s^F$ and the set $V \setminus (u_s^F \cup v_s^F)$.*

If $u_s^F \cap v_s^F = \emptyset$, the lemma holds true vacuously. So let us consider the case when $u_s^F \cap v_s^F \neq \emptyset$. Let B denote the set u_s^F and R denote the set v_s^F. Figure 3 depicts these sets with their intersection $B \cap R$ shown shaded.

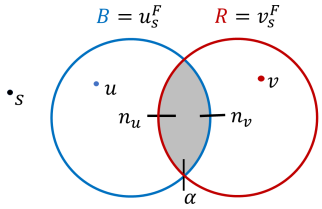


Figure 3 Intersection of the farthest mincuts from u and v to s.

Let n_u, n_v, α be the number of edges incident on $B \cap R$ from the sets $B \backslash R$, $R \backslash B$, and $V \backslash (B \cup R)$ respectively. To prove the lemma, we need to show that $\alpha = 0$. Since u and v are incomparable vertices, so $u \in B \backslash R$ and $v \in R \backslash B$. Since B is an (s, u)-mincut, we get the following inequality by applying Lemma 11 with $A = B$ and $A' = B \cap R$.

$$n_v + \alpha \leq n_u \tag{5}$$

Similarly, since R is an (s, v)-mincut, we get the following inequality by applying Lemma 11 with $A = R$ and $A' = B \cap R$.

$$n_u + \alpha \leq n_v \tag{6}$$

Adding Inequalities 5 and 6, we get $\alpha \leq 0$. But α is non-negative, so we get $\alpha = 0$.
We shall now use Lemma 11 and Lemma 23 to derive the following lemma which will play a crucial role in establishing that D^τ has indegree 2 only.

▶ **Lemma 24.** *For any three incomparable vertices $u, v, w \in V$, $u_s^F \cap v_s^F \cap w_s^F = \emptyset$.*

Proof. We give a proof by contradiction. Let B, L, and R denote the sets u_s^F, v_s^F, and w_s^F, respectively. Figure 4(i) illustrates these sets. For a clear distinction, we have used different colors for these sets in this figure, and correspondingly assigned the labels B (for blue), L (for light green), and R (for red) to these sets.

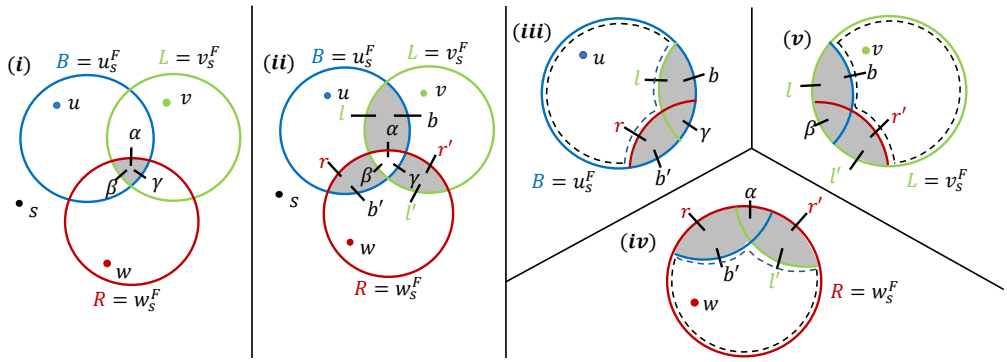


Figure 4 Intersection of the farthest mincuts to s.

Suppose the common intersection $B \cap L \cap R$ of these sets (shown shaded in Figure 4(i)) is not an empty set. By applying Lemma 23 for $B \cap L$, $L \cap R$, $R \cap B$, we can infer that each vertex in the common intersection will have edges incident only from the sets $B \cap L$, $L \cap R$, $R \cap B$. Considering the set $B \cap L \cap R$ as a single entity, let α, β, γ be the number of

edges incident on it from $(B \cap L)\backslash R$, $(R \cap B)\backslash L$, $(L \cap R)\backslash B$, respectively. As the graph is connected (see Remark 15), we have:

$$\alpha + \beta + \gamma > 0. \tag{7}$$

Let us consider the set $(B \cap L)\backslash R$, that is, the set $B \cap L$ after removing the common intersection $B \cap L \cap R$. It follows from Lemma 23 that the edges incident on this set will be from $B\backslash L$ and $L\backslash B$ only, apart from the edges incident from $B \cap L \cap R$. Similar claims hold for the sets $(B \cap R)\backslash L$ and $(R \cap L)\backslash B$ as well. Figure 4(*ii*) shows these sets as shaded regions along with the edges incident on them. For example, l, b, α are the number of edges incident on $(B \cap L)\backslash R$ from $B\backslash L$, $L\backslash B$, and $B \cap R \cap L$, respectively.

The rest of the proof is as follows. Exploiting the fact that u, v, w are incomparable, we suitably apply Lemma 11 to arrive at inequalities that eventually leads to contradict Inequality 7.

B defines an (s, u)-mincut. Since u is incomparable with both v and w, it is not present in the set $B \cap (R \cup L)$ shown shaded in Figure 4(*iii*). Notice that the number of edges incident on this set from $V\backslash B$ is $b + b' + \gamma$ whereas the number of edges incident on this set from the rest of B, that is, the set $B\backslash(R \cup L)$ (enclosed by dotted boundary in Figure 4(*iii*)) is at most $l + r$. So we get the following inequality by substituting B and $B \cap (R \cup L)$ in place of A and A' respectively in Lemma 11:

$$b + b' + \gamma \leq l + r \tag{8}$$

In Figure 4, R defines an (s, w)-mincut and L defines an (s, v)-mincut. Hence, with similar arguments as above, analyzing the (s, w)-mincut in Figure 4(*iv*), and analyzing the (s, v)-mincut in Figure 4(*v*), we get the following inequalities, respectively:

$$r + r' + \alpha \leq b' + l', \qquad\qquad l + l' + \beta \leq b + r'$$

Adding the above inequalities with Inequality 8 and canceling identical terms on either sides we get $\alpha + \beta + \gamma \leq 0$. This contradicts Inequality 7 and completes the proof. ◀

The following is a simple corollary of Lemma 24.

▶ **Corollary 25.** *Let $A \in \mathcal{F}$. If U, V, W are any three distinct sets from $\mathcal{F}$ such that $A \subset U$, $A \subset V$, and $A \subset W$. Then at least one of the sets from $\{U, V, W\}$ must be a proper subset of one of the remaining two.*

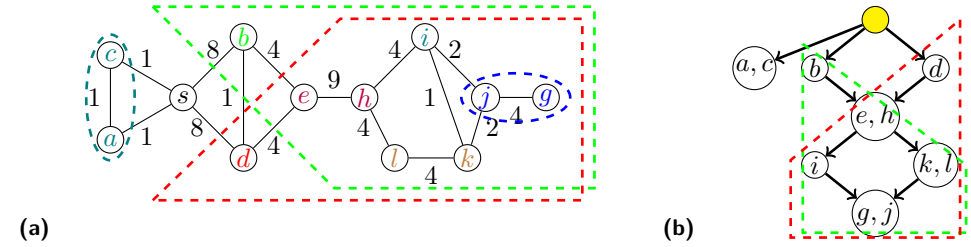
We can use Corollary 25 to establish the following lemma.

▶ **Lemma 26.** *The indegree of any node in D^τ will be at most 2.*

Proof. By definition D^τ preserves the reachability of D and has minimum possible number of edges. So there does not exist any edge in D^τ that can be removed without affecting the reachability between any pair of vertices in D. The presence of a node with indegree at least three in D^τ contradicts this property of D^τ as follows.

Suppose there is a node μ in D^τ with incoming edges from three nodes, say ν_1, ν_2, ν_3. The construction of D implies that $\mathcal{F}(\mu) \subset \mathcal{F}(\nu_1)$, $\mathcal{F}(\mu) \subset \mathcal{F}(\nu_2)$, and $\mathcal{F}(\mu) \subset \mathcal{F}(\nu_3)$. So it follows from Corollary 25 that at least one of the sets from $\{\mathcal{F}(\nu_1), \mathcal{F}(\nu_2), \mathcal{F}(\nu_3)\}$ is a proper subset of one of the remaining two. Without loss of generality, assume that $\mathcal{F}(\nu_1)$ is a proper subset of $\mathcal{F}(\nu_2)$. So there exists an edge in D from ν_2 to ν_1, and thus ν_1 is reachable from ν_2 in D. Since transitive reduction preserves reachability, so there must exist a path from ν_2 to ν_1 in D^τ as well. This path concatenated with (ν_1, μ) is a path from ν_2 to μ in D^τ. The presence of this path from ν_2 to μ makes the edge (ν_2, μ) redundant in D^τ. Hence the edge (ν_2, μ) can be removed from D^τ without affecting reachability – a contradiction. ◀

Figure 5 shows farthest mincuts from a sample of vertices to s in our example graph. Notice that the farthest mincut b_s^F *crosses* the farthest mincut d_s^F. Also, the farthest mincuts from j and g to s are identical, so j and g are mapped to a single node in D^τ.



(a) **(b)**

Figure 5 **(a)** A dotted boundary defines a farthest mincut to s from a vertex of the same color. **(b)** The DAG D^τ.

5 Single source mincut sensitivity for insertion of an edge

We now present an $\mathcal{O}(|S|)$ space data structure that can report all those vertices from S whose mincut value to s increases upon insertion of any given edge $(x, y) \in S \times S$. The data structure will consist of the tree structure $\mathcal{T}(s)$ from Theorem 17 and DAG structure $\mathcal{D}(s)$ from Theorem 18.

Let $A_x = \{v \in S | x \in s_v^N\}$ and $A_y = \{v \in S | y \in s_v^N\}$. It follows from Lemma 14 that if (s, v)-mincut increases, then v must belong to A_x or A_y. Furthermore, for any $v \in A_x$, (s, v)-mincut increases if $y \in v_s^N$. Using Theorem 17, it takes just $\mathcal{O}(1)$ time to do this check for any given $v \in A_x$ (likewise A_y). So, in order to report all vertices from S whose mincut from s increases upon insertion of edge (x, y) in $\mathcal{O}(|S|)$ time, all we need is an $\mathcal{O}(|S|)$ time algorithm to compute A_x and A_y. We now provide $\mathcal{O}(|S|)$ time algorithm to compute A_x; we can compute A_y in $\mathcal{O}(|S|)$ time in a similar manner.

It follows from Lemma 13(ii) that computing A_x is equivalent to computing the set $\bar{A_x} = \{v \in S | x \in v_s^F\}$. Recall that $J_s[x]$ is the node containing x in $\mathcal{D}(s)$. It follows from Lemma 22 that $x \in v_s^F$ if and only if $J_s[x]$ is reachable from $J_s[v]$ in $\mathcal{D}(s)$. Therefore, we can compute $\bar{A_x}$ by first reversing the edges of $\mathcal{D}(s)$ and then traversing all the nodes reachable from $J_s[x]$. For each node λ reachable from $J_s[x]$ in the reversed $\mathcal{D}(s)$, x is present in v_s^F for each vertex $v \in L(\lambda)$. Since $\mathcal{D}(s)$ has $\mathcal{O}(|S|)$ edges, it takes $\mathcal{O}(|S|)$ time to reverse it and traverse it to compute $\bar{A_x}$. This establishes the proof of Theorem 1 for the single source mincut sensitivity problem.

6 All-pairs mincut sensitivity data structure for insertion of an edge

Our data structure consists of the nearest-mincut tree $\mathcal{T}(z)$ from Theorem 17 for each $z \in S$. Each of these trees occupies $\mathcal{O}(|S|)$ space, so the space occupied by the data structure is $\mathcal{O}(|S|^2)$. For the rest of this section, x, y, z are any arbitrary vertices from S. Upon insertion of edge (x, y), let k be the number of pairs of vertices from $S \times S$ whose mincut value increases. We present an $\mathcal{O}(k)$ time algorithm to output all these pairs using four insights into the nearest-mincut trees. Our first insight is stated in Lemma 27. It implies that for all vertices belonging to a node μ in $\mathcal{T}(z)$, it suffices to determine for any single vertex, say $u \in \mu$, whether the (z, u)-mincut value increases upon insertion of any given edge.

▶ **Lemma 27.** *Let $u, v \in S$ be any two vertices belonging to the same node in $\mathcal{T}(z)$. Upon insertion of any given edge, (u, z)-mincut value increases iff (v, z)-mincut value increases.*

Proof. Let (x, y) be the edge that is inserted. Let μ and ν be respectively the nodes in $\mathcal{T}(z)$ to which x and y belong. It follows from Lemma 28 that the node containing u and v in $\mathcal{T}(z)$ must be an ancestor of either μ or ν (it could also be the node μ or ν). Without loss of generality, assume that this node is either ν or an ancestor of ν. This implies that $y \in u_z^N$ and $y \in v_z^N$. This fact in conjunction with Lemma 14 imply that the insertion of edge (x, y) will increase the mincut from u (likewise from v) to z if $x \in z_u^N$ (likewise $x \in z_v^N$). So in order to the prove this lemma, it suffices if we can establish that $z_u^N = z_v^N$. We accomplish this task as follows.

Let $A = z_u^N$ and $A' = z_v^N$. Since both u and v belong to the same node in $\mathcal{T}(z)$, so capacity of a (z, u)-mincut is the same as the capacity of a (z, v)-mincut. Hence $c(A) = c(A')$. The fact that, u and v belong to the same node in $\mathcal{T}(z)$, also implies that $u_z^N = v_z^N$. So it follows from Lemma 13(ii) that $z_u^F = z_v^F$. Hence, by definition, neither u nor v belongs to z_u^F. Since $z_u^N \subseteq z_u^F$ and $z_v^N \subseteq z_v^F$, so it follows that neither u nor v belongs to $A \cup A'$. Hence $A \cup A'$ defines a valid (z, u)-cut and a valid (z, v)-cut. $A \cap A'$ also defines a valid (z, u)-cut and a valid (z, v)-cut since z belongs to both A as well as A' and $A \cap A' \subseteq A \cup A'$. So using the submodularity of cuts (Lemma 10), it follows that each of $A \cup A'$ and $A \cap A'$ is also a valid (z, u)-mincut and a valid (z, v)-mincut. If $A \neq A'$, then it would imply that $A \cap A'$ is a proper subset of at least one of A or A'. This would imply that either A is not the nearest (z, u)-mincut or A' is not the nearest (z, v)-mincut – a contradiction. Hence $A = A'$. That is, $z_u^N = z_v^N$. ◀

Let μ and ν be the nodes in $\mathcal{T}(z)$ containing x and y respectively. Our second insight, stated in the following lemma, specifies the location of vertices in $\mathcal{T}(z)$ whose mincut value to s increases upon insertion of edge (x, y).

▶ **Lemma 28.** *Let $\omega = LCA(\mu, \nu)$ in $\mathcal{T}(z)$. Upon insertion of edge (x, y), the mincut value from z to only those vertices may increase that belong to the nodes of (1) the path from μ to ω but excluding ω, and (2) the path from ν to ω but excluding ω.*

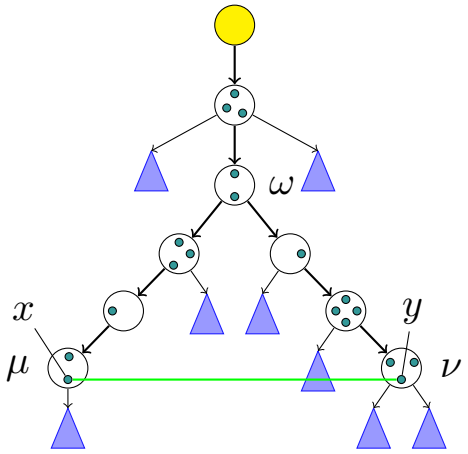


Figure 6 The tree $\mathcal{T}(z)$ from the perspective of μ and ν. If v is a vertex whose node in $\mathcal{T}(z)$ does not belong to either of ω-ν and ω-μ paths, then v must be present in one of the subtrees hanging from these paths (shown in blue). Now consider any node, say γ, lying on the path from ω to the root of $\mathcal{T}(z)$. Both x and y belong to the subtree rooted at γ.

Proof. Let us view $\mathcal{T}(z)$ from the perspective of the paths from μ and ν to the root of $\mathcal{T}(z)$. The reader is advised to refer to Figure 6 for a better understanding. If v is a vertex whose node in $\mathcal{T}(z)$ does not belong to these paths, then v must be present in one of the subtrees (shown in blue in Figure 6) hanging from these paths. Notice that neither x nor y belongs to the subtree containing v. So it follows from Lemma 14 that the mincut from z to v is not affected by the insertion of edge (x, y). Now consider any node, say γ, lying on the path from ω to the root of $\mathcal{T}(z)$. Both x and y belong to the subtree rooted at γ. So using Lemma 14 again, the mincut from z to any vertex of γ remains unaffected by the insertion of edge (x, y). ◀

Our third and most crucial insight is that the vertices whose mincut value to z increases upon insertion of edge (x, y) belong to a *contiguous* sequence of nodes on the paths from the node containing y and the node containing x to their LCA in $\mathcal{T}(z)$. The following lemma states this insight for the node containing y.

▶ **Lemma 29.** *Let ω be the LCA of the nodes containing x and y in $\mathcal{T}(z)$. Let u and v be any two vertices lying on the path from the node containing y to ω in $\mathcal{T}(z)$ such that the node containing u is an ancestor of the node containing v. If (z, u)-mincut value increases upon insertion of edge (x, y), then (z, v)-mincut value also increases upon insertion of (x, y).*

Proof. It follows from the construction of $\mathcal{T}(z)$ that $y \in v_z^N$ and $v_z^N \subseteq u_z^N$. It is given that the insertion of edge (x, y) increases (z, u)-mincut value and u is an ancestor of y in $\mathcal{T}(z)$. So Lemma 14 implies:

$$x \in z_u^N \tag{9}$$

It follows from Lemma 13(i) that $z_u^N \cap u_z^N = \emptyset$. So $v \notin z_u^N$ since $v \in u_z^N$. Applying Lemma 13(ii), we get $v \in u_z^F$. So it follows from Lemma 20 that $v_z^F \subseteq u_z^F$. Applying Lemma 13(ii) again, it follows that $z_u^N \subseteq z_v^N$. Using this fact and Equation 9, we can infer that $x \in z_v^N$. Since we have already established that $y \in v_z^N$, so using Lemma 14 we can conclude that (z, v)-mincut value will also increase upon insertion of edge (x, y). ◀

It is a simple corollary of Lemma 29 that if (y, z)-mincut value does not increase upon insertion of edge (x, y), then for any vertex v present in any ancestor of the node containing y in $\mathcal{T}(z)$, (v, z)-mincut value will also not increase. So, to compute all-pairs of vertices whose mincut increases, we need to explore the nearest-mincut tree of only those vertices z whose mincut value to y (and likewise to x) increases.

We now describe how to process $\mathcal{T}(z)$ for a vertex z given that (y, z)-mincut value increases upon insertion of (x, y). For each such z, first we enumerate all vertices present in the node, say ν, to which y belongs. We then begin an upward traversal of $\mathcal{T}(z)$ starting from the parent of ν. For any node, say λ, that we traverse, we pick any arbitrary vertex from it, say v, and determine whether $x \in z_v^N$ by querying $\mathcal{T}(v)$. It takes $\mathcal{O}(1)$ time to answer this query (see Theorem 17). If $x \in z_v^N$, it follows from Lemma 27 that each vertex present in λ has its mincut value to z increased. So we enumerate all vertices from λ, and continue processing the parent of λ in a similar manner. If $x \notin z_v^N$, we stop the traversal. It follows from Lemma 29 that the vertices enumerated in this way are precisely the vertices whose mincut value to z increases. To efficiently identify each vertex z, such that the (y, z)-mincut value increases upon insertion of edge (x, y), we exploit the fourth insight into the nearest-mincut trees which is stated in the following lemma. This lemma can be seen as a corollary of Lemma 14.

▶ **Lemma 30.** *(y, z)-mincut value increases upon insertion of edge (x, y) iff $x \in z_y^N$.*

It follows from Lemma 30 that the vertices present in the node containing x and its ancestors in $\mathcal{T}(y)$ are precisely the vertices whose mincut value to y increases. We can identify all these vertices in optimal time by traversing $\mathcal{T}(y)$ upward from the node containing x.

We have described above the processing of each z such that the (y, z)-mincut value increases due to the insertion of edge (x, y). A similar processing must be carried out for all vertices z, such that the (x, z)-mincut value increases due to the insertion of edge (x, y).

It follows from the description given above that we can compute all those pairs of vertices from $S \times S$ whose mincut value increases upon the insertion of any given edge in $\mathcal{O}(k)$ time, where k is the number of these pairs. If our goal is to just report the value of k, we can accomplish it in $\mathcal{O}(\min(k, |S| \log |S|))$ time by suitably augmenting the nearest-mincut trees (see Appendix). We can thus conclude with Theorem 31 which extends Theorem 3.

▶ **Theorem 31.** *For an undirected graph $G = (V, E)$, and a subset S of vertices, there exists an $\mathcal{O}(|S|^2)$ size data structure that can report all pairs of vertices whose mincut increases upon insertion of a query edge. The guaranteed query time is $\mathcal{O}(k)$, where k is the number of pairs whose mincut increases. We can also report k in $\mathcal{O}(\min(k, |S| \log |S|))$ time.*

7 Lower bounds

In the following subsection, we show that for a directed graph, we can not have any data structure of $o(n^2)$ bits that can report the set of vertices whose mincut from s increases upon insertion of any query edge. Thus, for directed graphs, there can not exist any non-trivial data structure for mincut sensitivity for the insertion of an edge. For undirected graphs, we provide lower bounds on the size of any data structure that can report the value of a mincut between a pair of vertices upon insertion of any query edge of arbitrary capacity.

7.1 Lower bound for directed graphs

Let $\mathcal{A}$ be an algorithm that takes a directed graph H and a designated source s as input. It outputs a data structure $\mathcal{A}(H)$ which reports, for any query edge (x, y), the set of all the vertices whose mincut from s increases upon insertion of (x, y) to H.

Let $\mathcal{G}$ be a family of the directed graphs on $2n + 1$ vertices defined as follows. The vertex set of each graph in $\mathcal{G}$ consists of 3 disjoint subsets – $\{s\}, V_1, V_2$, where s is a designated source vertex, and V_1 and V_2 consist of n vertices each. The edge set of any graph in $\mathcal{G}$ will be a subset of $(V_1 \times V_2)$. For each pair $(u, v) \in (V_1 \times V_2)$, the edge (u, v) is either present or absent in a graph in $\mathcal{G}$. So the total number of graphs in the family $\mathcal{G}$ is 2^{n^2}.

▶ **Lemma 32.** *If H_1 and H_2 are any two different graphs from the family $\mathcal{G}$, then $\mathcal{A}(H_1)$ and $\mathcal{A}(H_2)$ must be different.*

Proof. Since H_1 and H_2 differ, we can assume without loss of generality that there is an edge, say (u, v), which is present in H_1 but absent in H_2. It can be observed that upon insertion of edge (s, u), the maxflow from s to v increases from 0 to 1 in H_1 whereas it continues to remain 0 in H_2. So for the query edge (u, v), the output of data structures $\mathcal{A}(H_1)$ and $\mathcal{A}(H_2)$ will be different – v will belong to the output of $\mathcal{A}(H_1)$ but not to the output of $\mathcal{A}(H_2)$. Hence, $\mathcal{A}(H_1)$ cannot be identical to $\mathcal{A}(H_2)$. ◀

Since there are 2^{n^2} graphs in the family $\mathcal{G}$, it follows from Lemma 32 that for each $H \in \mathcal{G}$, $\mathcal{A}(H)$ will have a unique binary representation. Hence, there must be at least one graph H from the family $\mathcal{G}$ such that $\mathcal{A}(H)$ will have at least n^2 bits.

*

7.2 Lower bound for reporting the values of mincuts upon edge insertion

We first define the notion of mincut between a vertex and a pair of vertices as follows.

▶ **Definition 33** (Mincut between a vertex and a pair of vertices)**.** *Let u, x and y be any three vertices. A set E' of edges is said to be a cut between u and the pair $\{x, y\}$ if there is no path between u and x or between u and y in the graph $(V, E \backslash E')$. The set E' is said to be a mincut between u and the pair $\{x, y\}$ if the capacity of E' is least among all the cuts that separate u and the pair $\{x, y\}$.*

▶ **Definition 34** (Set defining a mincut between a vertex and and a pair of vertices)**.** *Let u, x and y be any three vertices. A set A is said to define a mincut between u and the pair $\{x, y\}$ if $u \in A$, $x \notin A, y \notin A$, and the set of edges between A and $V \backslash A$ forms a mincut between u and the pair $\{x, y\}$.*

We shall now define two problems and establish a relationship between them. The first problem is the mincut sensitivity problem capable of reporting the mincut value as well, formally defined in Problem 2. The second problem deals with a mincut between a vertex and a pair of vertices, formally defined in Problem 3.

▶ **Problem 2.** *Preprocess a given undirected graph to build a data structure that can report, for any query edge (x, y) of capacity Δ for any $x, y \in V$, and $\Delta > 0$, the value of the mincut between any two vertices in the graph upon insertion of the edge (x, y) with capacity Δ.*

We now define the problem of the mincut between a vertex and a pair of vertices as follows.

▶ **Problem 3.** *Preprocess a given undirected graph to build a data structure that can report the value of the mincut between u and a pair $\{x, y\}$ for any $u, x, y \in V$.*

▶ **Lemma 35.** *Let G' be the graph after inserting an edge (x, y) of capacity $c_0(u)$ in G, where $c_0(u)$ is any number greater than $\sum_{(u,v)\in E} c(u, v)$. A mincut between u and the pair $\{x, y\}$ in G is also a mincut between u and x in G'.*

Proof. Since $\{u\}$ defines a cut between u and x, and its capacity is less than $c_0(u)$, so the capacity of the (u, x)-mincut in G' must be less than c_0. This implies that the edge (x, y) can not be present in any (u, x)-mincut in G'. Therefore, y must lie on the side of x in every (u, x)-mincut in G'. Hence an (u, x)-mincut in G' is also a cut that separates u and the pair $\{x, y\}$ in G. Observe that the capacity of this cut in G' is identical to its capacity in G. On the other hand, a mincut that separates u from $\{x, y\}$ in G is also a cut of the same capacity that separates u from x in G' since it follows from Definition 33 that the edge (x, y) never appears in any cut between u and the pair $\{x, y\}$. Therefore, we can conclude that a mincut between u and the pair $\{x, y\}$ in G is also a mincut with the same capacity between u and x in G'. ◀

We now exploit Lemma 35 to provide the connection between Problem 2 and Problem 3.

Let Q be a data structure for solving Problem 2. That is, Q can report the value of a (u, v)-mincut upon inserting an edge (x, y) of capacity Δ for any $u, v, x, y, \in V$ and $\Delta > 0$. We can use Q to solve Problem 3 as follows.

Suppose we wish to retrieve the capacity of the mincut between u and a pair $\{x, y\}$. We can retrieve it using Q as follows. Let c_0 be any integer greater than $\sum_{(u,v)\in E} c(u, v)$. We ask Q for the value of the (u, x)-mincut after we insert the edge (x, y) of capacity c_0. It follows from Lemma 35 that the value returned by Q will be the same as the capacity of the mincut that separates s from the pair $\{u, v\}$ in G. So we can state the following lemma.

▶ **Lemma 36.** *A lower bound on the space complexity of Problem 3 will hold as a lower bound on the space complexity of Problem 2 as well.*

In the light of Lemma 36, it suffices if we address the lower bound on the space complexity of Problem 3. In the next subsection, we provide a lower bound on the space complexity of the single source version of Problem 3. In the subsequent subsection, we address the space complexity of the all-pairs version of Problem 3.

7.2.1 Lower bound on the single source version of Problem 3

We shall now establish a lower bound on the space complexity of the single source version of Problem 3. In particular, we shall establish the following theorem.

▶ **Theorem 37.** *There exists an undirected graph on n vertices with a designated source vertex s and $\Theta(n^2)$ edges with integer edge capacities varying in the range $[1, n^{2+\varepsilon}]$ (for any $\varepsilon > 0$) for which there can not exist any data structure of size $o(n^2\varepsilon \log n)$ bits, that can report the value of a mincut between s and any pair of vertices in the graph.*

Lemma 36 and Theorem 37 would imply Theorem 6. To establish Theorem 37, first we give an overview of the proof.

Suppose we have a $n \times n$ matrix M of integers, where each entry takes any value from the range $[1, W]$, where $W = n^{\varepsilon}$ for any given $\varepsilon > 0$. We use this matrix to build a graph G_M with a designated source vertex s. The weight of each edge in G_M will be in the range $[1, n^2W]$. Let $D(G_M)$ be any data structure for the the single source version of Problem 3. We show that we can retrieve all elements of M by making appropriate queries to $D(G_M)$. Since any data structure that stores M must use $n^2\varepsilon \log n$ bits of space, the data structure $D(G_M)$ is bound to take $n^2\varepsilon \log n$ bits of space.

Construction of $G(M)$:

Given the matrix M, we construct the graph G_M as follows. Let $U = \{u_1, u_2, \ldots, u_n\}$ and $U' = \{u'_1, u'_2, \ldots, u'_n\}$ be two disjoint sets of vertices.

- The vertex set of G_M is $U \cup U' \cup \{s\}$, where s is the designated source vertex.
- The edge set of G_M consists of 3 types of edges as follows.

E_1: This set consists of edges between vertices of set U and vertices of set U': For each $1 \leq i, j \leq n$, there is an edge between u_i and u'_j and the weight of this edge is $M[i, j]$.

E_2: This set of edges joins the vertices of U along a path that terminates at s as follows. For each $1 \leq i < n$, we add an edge between vertex u_i and u_{i+1} with a weight of inW. We also add an edge between vertex u_n and s with a weight of n^2W.

E_3: This set of edges joins the vertices of U' along a path that terminates at s as follows. For each $1 \leq i < n$, we add an edge between vertex u'_i and u'_{i+1} with a weight of inW. We also add an edge between vertex u'_n and s with a weight of n^2W.

Notice that each edge of G_M has a weight in the range $[1, n^2W]$. The graph G_M is shown in Figure 7. We now state the following lemma on the values of mincuts between s and other vertices in G_M.

▶ **Lemma 38.** *For any $1 \leq i \leq n$, the set $\{u_1, u_2, \ldots, u_i\}$ defines a (s, u_i)-mincut.*

Proof. Using the maxflow-mincut theorem, it suffices if we can establish that there is a valid flow from s to i that saturates all edges of the cut defined by $\{u_1, u_2, \ldots, u_i\}$.

The capacities of the edges increase along the path $u_1 - u_2 - \ldots - u_n - s$. The capacity of edge (u_1, u_{i+1}) is inW. So we can send flow of value inW from s to u_i along the path $s - u_n - u_{n-1} - \ldots - u_i$ that saturates edge (u_i, u_{i+1}). The remaining edges of the cut defined by $\{u_1, u_2, \ldots, u_i\}$ are only between vertices from set $\{u_1, u_2, \ldots, u_i\}$ and U'. For each $1 \leq j \leq i, 1 \leq \ell \leq n$, we assign flow along the edge (u'_ℓ, u_j) equal to its capacity. So all that we have to establish is that we can assign flow along the remaining edges of G_M ensuring the capacity constraints as well as the conservation constraint. We accomplish this objective as follows.

Notice that for each $1 \leq \ell \leq n$, the capacity of the edge $(u'_\ell, u'_{\ell+1})$ is equal to ℓnW which is an upper bound on the sum of the flow on each edge $(u'_k, u_j), 1 \leq k \leq \ell, 1 \leq j \leq i$. So assigning flow along the edge from $u'_{\ell+1}$ to u'_ℓ equal to the sum of the flow along each edge $(u'_k, u_j), 1 \leq k \leq \ell, 1 \leq j \leq i$ ensures the capacity constraint of the edge $(u'_{\ell+1}, u'_\ell)$. It can be seen that this assignment of flow also ensures the conservation constraint at each vertex of U'.

Notice that for each $1 \leq j < i$, the capacity of the edge (u_j, u_{j+1}) is jnW which is also an upper bound on the sum of the flow on all edges entering vertices $\{u_1, \ldots, u_j\}$ from vertices of set U'. So assigning flow along the edge (u_j, u_{j+1}) equal to the sum of the flow along each edge $(u'_\ell, u_k), 1 \leq \ell \leq n, 1 \leq k \leq j$ ensures the capacity constraints of the edge (u_j, u_{j+1}). It can be seen that this assignment of flow ensures the conservation constraint at each vertex $u_j, 1 \leq j < i$.

So we are able to establish a valid flow from s to u_i that saturates all edges of the cut defined by $\{u_1, \ldots, u_i\}$. This establishes that the cut defined by $\{u_1, u_2, \ldots, u_i\}$ is a mincut between s and u_i. ◀

Along similar lines as that of the proof of Lemma 38, we can prove the following lemma.

▶ **Lemma 39.** *For any $1 \leq j \leq n$, the set $\{u'_1, u'_2, \ldots, u'_j\}$ defines a (s, u'_j)-mincut.*

Now we state the lemma about the mincut between s and a pair $\{u_i, u'_j\}$ for any $1 \leq i, j \leq n$.

▶ **Lemma 40.** *For any $i, j \leq n$, the cut defined by $\{u_1, u_2, \ldots, u_i\} \cup \{u'_1, u'_2, \ldots, u'_j\}$ is a mincut between s and the pair $\{u_i, u'_j\}$.*

Proof. Using the maxflow-mincut theorem, it suffices if we can establish that there is a valid flow from s to two sink vertices u_i and u'_j that saturates all edges of the cut defined by $\{u_1, u_2, \ldots, u_i\} \cup \{u'_1, u'_2, \ldots, u'_j\}$.

Once again, recall that the capacities of the edges increase along the path $u_1 - u_2 - \ldots - u_n - s$ and the path $u'_1 - u'_2 - \ldots - u'_n - s$. So we can send flow of value inW from s to u_i along the path $s - u_n - u_{n-1} - \ldots - u_i$ that saturates the edge (u_i, u_{i+1}). Likewise, we can send flow of value jnW from s to u'_j along the path $s - u_n - u_{n-1} - \ldots - u'_j$ that saturates the edge (u'_j, u'_{j+1}). We now assign flow along the remaining edges of the cut defined by $\{u_1, \ldots, u_i\} \cup \{u'_1, \ldots, u'_j\}$. These edges are only of two types:

1. The edges between vertices of set $\{u_{i+1}, \ldots, u_n\}$ and $\{u'_1 \ldots, u'_j\}$. For each $i < k \leq n$ and $1 \leq \ell \leq j$, we send flow along the edge (u_k, u'_ℓ) equal to its capacity.
2. The edges between vertices of set $\{u'_{j+1}, u'_{j+2}, \ldots, u'_n\}$ and $\{u_1, \ldots, u_i\}$. For each $1 \leq k \leq i$ and $j < \ell \leq n$, we send flow along the edge (u'_ℓ, u_k) equal to its capacity.

Now all that we have to establish is that we can assign flow along the edges of the paths $s - u_n - \ldots - u_1$ and $s - u'_n - \ldots - u'_1$ ensuring the capacity constraints as well as the conservation constraint. We accomplish this objective as follows.

Consider the edge (u_k, u_{k+1}) for $k > i$. The capacity of this edge is knW and a flow of value inW has already been sent along (u_{k+1}, u_k) towards u_i. The remaining capacity is $(k-i)nW$

which is an upper bound on the sum of the capacity of each edge $(u_q, u'_\ell), i < q \leq k, 1 \leq \ell \leq j$. So sending an extra flow along edge (u_{k+1}, u_k) equal to the sum of the flow along each edge $(u_q, u'_\ell), i < q \leq k, 1 \leq \ell \leq j$ ensures the capacity constraint of the edge (u_{k+1}, u_k). It can be seen that this assignment of flow also ensures the conservation constraint at each vertex $u_k, i < k \leq n$. Along exactly similar lines, we can establish flows along each edge $(u'_{\ell+1}, u'_\ell), j < \ell \leq n$ that will satisfy the capacity constraint of the edge and also ensure the conservation constraint at the vertex u'_ℓ.

The flow that enters a vertex $u_k, 1 \leq k < i$, through edges from $\{u'_{j+1}, \ldots, u'_n\}$ will be sent to sink u_i. We use the edges along the path $u_1 - \ldots - u_i$ to accomplish this task in a similar manner as we did in Lemma 38 so that the capacity constraint of each edge on this path is satisfied and the conservation of flow at each vertex on the path is also ensured. Along similar lines, we ensure that the flow that enters $u'_\ell, 1 \leq \ell < j$ through edges from $\{u_{i+1}, \ldots, u_n\}$ is sent to sink u_j ensuring the capacity constraint and the conservation constraint at each vertex of the path $u'_1 - \ldots - u'_j$. ◀

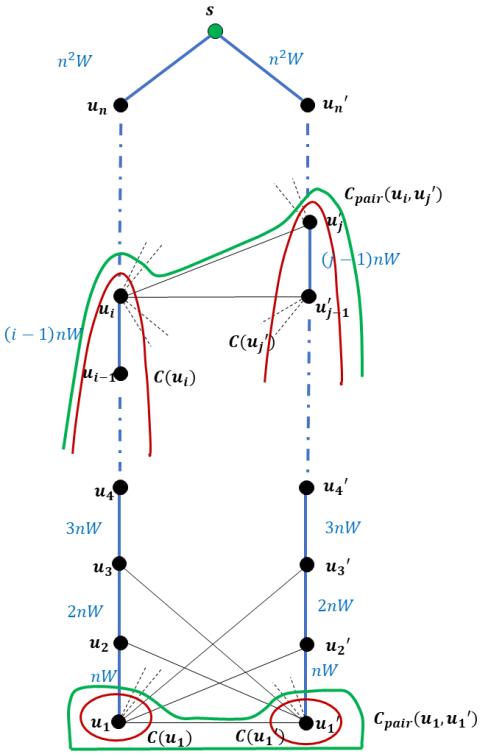


Figure 7 The graph G_M.

Refer to Figure 7 for a better understanding of the graph G_M. We use $C(x)$ to denote the value of a (s, x)-mincut. Likewise, we use $C_{pair}(x, y)$ to denote the value of a mincut between s and the pair $\{x, y\}$. It is easy to observe that $C(x) = C_{pair}(x, x)$.

Retrieving the elements of the matrix M:

Suppose $D(G_M)$ is the data structure for the single source version of Problem 3 for G_M. For a query $Q(x, y)$ made to this data structure, the answer reported will be $C_{pair}(x, y)$.

We now show that we can retrieve all elements of M by making appropriate queries to $D(G_M)$. To begin with, we show the retrieval of $M[1,1]$ as follows. Note that $M[1,1]$ is assigned as the weight of the edge (u_1, u'_1). Lemmas 38, 39, and 40 reveal the sets defining $C_{pair}(u_1,u_1), C_{pair}(u'_1,u'_1)$, and $C_{pair}(u_1,u'_1)$. It can be observed that each edge incident on u_1 from a vertex other than u'_1 contributes to $C_{pair}(u_1,u'_1)$ as well. So it can be observed that

$$M[1,1] = \frac{1}{2}\left(Q(u_1,u_1) + Q(u'_1,u'_1) - Q(u_1,u'_1)\right) \tag{10}$$

So by making three queries to the data structure $D(G_M)$, we can retrieve $M[1,1]$. We now show the retrieval of $M[i,j]$ for any i,j. Note that $M[i,j]$ is assigned as the weight of the edge (u_i,u'_j). Lemmas 38, 39, and 40 reveal the sets defining $C_{pair}(u_i,u_i), C_{pair}(u'_j,u'_j)$, and $C_{pair}(u_i,u'_j)$. Based on this information, the following equation captures the weight of the edge (u_i,u'_j) which is $M[i,j]$.

$$M[i,j] = \frac{1}{2}\left(Q(u_i,u_i) + Q(u'_j,u'_j) - Q(u_i,u'_j)\right) - \sum_{\forall k,\ell: k\le i, \ell<j \text{ or } k<i, \ell\le j} M[k,\ell] \tag{11}$$

We now present a simple dynamic programming algorithm based on Equation 11 that makes queries to $D(G_M)$ and take only $\mathcal{O}(n)$ extra space to retrieve $M[i,j]$ for any $1 \le i,j \le n$. Let us first introduce a term.

$$A(i,j) = \sum_{\forall k,\ell: k\le i, \ell\le j} M[k,\ell]$$

We define $A(i,j) = 0$ if $i \le 0$ or $j \le 0$. Using the term $A(i,j)$ thus defined, we can rephrase Equation 11 as follows.

$$M[i,j] = \frac{1}{2}\left(Q(u_i,u_i) + Q(u'_j,u'_j) - Q(u_i,u'_j)\right) - \left(A(i,j-1) + A(i-1,j) - A(i-1,j-1)\right) \tag{12}$$

We shall now use the above equation to enumerate all values of the matrix M using $\mathcal{O}(n)$ space. In this algorithm, we shall also build and use the matrix A. However, as will soon become clear, we need not keep all of A at any time. In particular, we shall need only two of its *diagonals* at any stage of time.

Let $S(t) = \{M[i,j] \mid i+j = t\}$ and $R(t) = \{A(i,j) \mid i+j = t\}$. We shall enumerate $S(2)$, $S(3)$, and so on. For $S(2)$, we compute $M[1,1]$ as described in Equation 10. Let $t > 2$ be any integer. We now present the algorithm for enumerating the elements of $S(t)$. Let i,j be any two integers with $i+j = t$. Our algorithm makes use of Equation 12. It makes three queries to $D(G_M)$. In addition, it retrieves $A(i,j-1), A(i-1,j), A(i-1,j-1)$. Plugging these values in Equation 12, it computes $M[i,j]$. The only extra space required is for retrieving the three values of A. These values vary depending upon i and j. But the key observation is that for computing any element from $S(t)$, we need only those values of A that belong to the set $\{A(k,\ell) \mid t-2 \le k+\ell \le t-1\}$. This set is precisely $R(t-2) \cup R(t-1)$. So for computing elements of $S(t)$, we need only $R(t-2)$ and $R(t-1)$. So the extra space needed consists of only $2t-3 = \mathcal{O}(n)$ elements. While we compute $S(t)$, we also compute the corresponding entries of A for the set $R(t)$ (to be used for computing $S(t+1)$ in future) using the following equation.

$$A(i,j) = A(i-1,j) + A(i,j-1) - A(i-1,j-1) + M[i,j]$$

We discard the set $R(t-2)$ after we have enumerated $S(t)$. So the extra space used in the enumeration of all elements of M is $\mathcal{O}(n)$ only.

7.2.2 On the space complexity of the all-pairs version of Problem 3

Gomory and Hu [5] showed that for any undirected graph $G = (V, E)$ on n vertices, there can only be $n - 1$ distinct values of mincuts. Furthermore, these values can be stored in a tree T on the vertices of the set V that stores a mincut between each pair of vertices. The value of a mincut between any two vertices u and v in G is the weight of the least weighted edge on the path in T between u and v. Moreover, the removal of this edge defines a mincut between u and v. The tree T is called a Gomory-Hu tree of G. The flow tree of G is a tree that stores the value of a mincut between any pair of vertices, but not necessarily the set of vertices defining their mincut (see [5],[7]). It is natural to ask whether the notion of a Gomory-Hu tree or a flow tree can be generalized. Problem 3 can be seen as the generalization of a flow tree to report the value of the mincut between a vertex u and a pair $\{x, y\}$ for any $u, x, y \in V$. Two trivial data structures for this problem are as follows.

- We explicitly store the value of a mincut between u and $\{x, y\}$ for each $u, x, y \in V$. The space taken by the data structure will be $\mathcal{O}(n^3 \log n)$ bits if each edge has integral capacity in the range $[1, n^c]$ for a constant c. The query time will be $\mathcal{O}(1)$. This data structure is trivial since it does not optimize the space.
- We keep the graph G itself as the data structure. For a query that asks for the mincut between u and a pair $\{x, y\}$, we just add an edge of value $c_0 > \sum_{(u,v)\in E} c(u, v)$ between x and y, and then compute the maxflow between u and x. This data structure is trivial since it does not optimize the query time.

Chitnis, Kamma, and Krauthgamer [3] addressed the problem of generalization of Gomory-Hu tree and flow tree. They showed that there will be total $\mathcal{O}(n^2)$ distinct values of mincuts between a vertex and any pair of vertices. They also showed that this bound is existentially tight. However, designing an $\mathcal{O}(n^2)$ size data structure that achieves a non-trivial query time to report the value of a mincut between a vertex u and the pair $\{x, y\}$ for any $u, x, y, \in V$ is still an open problem. In other words, to the best of our knowledge, currently there is no non-trivial data structure for Problem 3 even after nearly 50 years of the seminal result by Gomory and Hu [5].

8 Acknowledgements

We would like to convey special thanks to Jannik Castenow from the Heinz Nixdorf Institute and the Paderborn University for many valuable discussions. Additionally, we would like to thank Rajesh Chitnis and Robert Krauthgamer for promptly answering a few of our queries related to their paper [3].

References

1 Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972. doi:10.1137/0201008.

2 Michael A. Bender and Martin Farach-Colton. The level ancestor problem simplified. *Theor. Comput. Sci.*, 321(1):5–12, 2004. doi:10.1016/j.tcs.2003.05.002.

3 Rajesh Chitnis, Lior Kamma, and Robert Krauthgamer. Tight bounds for gomory-hu-like cut counting. In *Graph-Theoretic Concepts in Computer Science - 42nd International Workshop, WG 2016, Istanbul, Turkey, June 22-24, 2016, Revised Selected Papers*, pages 133–144, 2016. doi:10.1007/978-3-662-53536-3_12.

4 L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956. doi:10.4153/CJM-1956-045-5.

5 R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9(4):551–570, 1961. URL: http://www.jstor.org/stable/2098881.

6 Gramoz Goranci, Monika Henzinger, and Mikkel Thorup. Incremental exact min-cut in polylogarithmic amortized update time. *ACM Trans. Algorithms*, 14(2):17:1–17:21, 2018. doi:10.1145/3174803.

7 Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM J. Comput.*, 19(1):143–155, February 1990. URL: http://dx.doi.org/10.1137/0219009, doi:10.1137/0219009.

8 Ramesh Hariharan, Telikepalli Kavitha, Debmalya Panigrahi, and Anand Bhalgat. An $\tilde{O}(mn)$ gomory-hu tree construction algorithm for unweighted graphs. In *Proceedings of the 39th Annual ACM Symposium on Theory of Computing, San Diego, California, USA, June 11-13, 2007*, pages 605–614, 2007. See also the extended version at http://hariharan-ramesh.com/papers/gohu.pdf. doi:10.1145/1250790.1250879.

9 Tanja Hartmann and Dorothea Wagner. Fast and simple fully-dynamic cut tree construction. In *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*, pages 95–105, 2012. doi:10.1007/978-3-642-35261-4_13.

10 Jean-Claude Picard and Maurice Queyranne. On the structure of all minimum cuts in a network and applications. *In Rayward-Smith V.J. (eds) Combinatorial Optimization II. Mathematical Programming Studies*, 13(1):8–16, 1980. doi:10.1007/BFb0120902.

11 Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi:10.1016/0022-0000(83)90006-5.

12 Mikkel Thorup. Fully-dynamic min-cut. *Combinatorica*, 27(1):91–127, 2007. doi:10.1007/s00493-007-0045-2.

9 Appendix

In Section 6, we presented the algorithm for computing all pairs of vertices whose mincut increases upon insertion of any query edge. The time complexity of the algorithm is $\mathcal{O}(k)$, where k is the number of pairs of vertices whose mincut increases upon insertion of a query edge. We now address a variant of this problem – the objective is to report the number k. For this variant, we shall design an algorithm that achieves $\mathcal{O}(\min(k, n \log n))$ time to answer a query. The algorithm is a slight modification of the algorithm presented in Section 6. However, to execute it and achieve the desired query time, we need to suitably augment each nearest-mincut tree $\mathcal{T}(z)$. First we present this augmentation, and then we present our algorithm.

9.1 Augmentation of $\mathcal{T}(z)$

We augment $\mathcal{T}(z)$ so that given any node ν and integer i, we can report the ancestor of ν at depth i in $\mathcal{T}(z)$ efficiently. For this, we employ the Level-Ancestor data structure of Bender and Colton [2]. So we can state the following theorem.

▶ **Theorem 41.** *We can augment $\mathcal{T}(z)$ so that given any node $\mu \in \mathcal{T}(z)$ and integer i, the ancestor of μ at depth i can be reported in $\mathcal{O}(1)$ time.*

We now augment $\mathcal{T}(z)$ so that given any node μ_1 and another node μ_2 which is ancestor of μ_1, we can efficiently compute the number of vertices of G that belong to the nodes lying on the path from μ_1 to μ_2. It is possible to accomplish this objective with $\mathcal{O}(\log n)$ query time without any asymptotic blow up in the size of $\mathcal{T}(z)$ as follows. First we introduce some notations. For a node $\mu \in \mathcal{T}(z)$, let $p(\mu)$ denote the parent of μ and let size(μ) denote the

number of vertices assigned to μ (it is the same as the number of elements in list $L(\mu)$). We augment $\mathcal{T}(z)$ as follows.

We decompose $\mathcal{T}(z)$ into a collection $\mathcal{H}$ of vertex disjoint paths using heavy-light decomposition [11]. This decomposition ensures that for any node μ_1 and another node μ_2 which is an ancestor of μ_1, the path from μ_1 to μ_2 can be expressed as a union of $\mathcal{O}(\log n)$ paths from $\mathcal{H}$. Each path in $\mathcal{H}$ originates from a node of $\mathcal{T}(z)$, called its root, and terminates at a node which is a leaf node in $\mathcal{T}(z)$. Furthermore, we augment each path $p \in \mathcal{H}$ as follows.

- For each node $\nu \in p$, we keep a field $\text{sum}(\nu)$ that stores the sum of $\text{size}(\mu)$ for each μ which is descendant of ν on path p (in our terminology, a node is also a descendant of itself).
- For each node $\nu \in p$, we keep a field labeled $\text{jump}(\nu)$ which stores the pointer to the root of the path p.
- The root node of path p also stores a field labeled $\text{next}(p)$ that stores the pointer to its parent in $\mathcal{T}(z)$.

With the above augmentation, it is an easy exercise to extract the sum of the number of vertices belonging to the nodes of path from μ_1 to ancestor μ_2 in $\mathcal{T}(z)$ – we start from μ_1, and traverse to μ_2 using jump() and next() field. We extract the number of vertices lying on this path using the sum() field stored at the roots of various paths and the corresponding next pointer. The total time complexity will be $\mathcal{O}(\min(t, \log n))$ where t is the number of nodes on the path from u_1 to μ_2. So we can state the following theorem.

▶ **Theorem 42.** *We can augment $\mathcal{T}(z)$ so that given any node μ_1 and another node μ_2 which is ancestor of μ_1, it takes $\mathcal{O}(\min(t, \log n))$ time to compute the total number of vertices present in the nodes of the path from μ_1 to μ_2. Here t is the number of nodes on the path from μ_1 to μ_2 in $\mathcal{T}(z)$.*

9.2 The algorithm

Interestingly, we need to just slightly modify the algorithm presented in Section 6. Suppose (x, y) is the query edge to be inserted. Recall that first we compute all vertices whose mincut to y or x increases due to insertion of (x, y). Let Z_y and Z_x be the set of all these vertices whose mincut to y increased. We process Z_y as follows (Z_x is processed in a similar manner).

Let $z \in Z_y$. Let μ be the node in $\mathcal{T}(z)$ to which y belongs. Recall that, exploiting Lemma 29, we perform a sequential search from μ till we reach the first ancestor of μ that has a vertex whose mincut to z does not increases. This sequential search was necessary to enumerate all vertices whose mincut to z increases upon insertion of the edge (x, y). However, if our objective is to count all vertices in μ and its ancestors whose mincut to z increases upon insertion of (x, y), there is a faster way – we do a binary search over the ancestors of μ to find the highest ancestor of μ, say μ', having a vertex whose mincut to z increases upon insertion of (x, y). For carrying out the binary search efficiently, we use Theorem 41. This ensures that we take $\mathcal{O}(\min(t, \log n))$ time for the search of μ' where t is the number of nodes on the path from μ to μ'. Once we have computed μ', we use Theorem 42 to compute the total number of vertices present in the path from μ to μ'. So we spend only $\mathcal{O}(\min(t, \log n))$ time for tree $\mathcal{T}(z)$. Adding this cost for each z in Z_y and Z_x, the total time complexity for reporting the number of pairs of vertices whose mincut increases upon insertion of edge (x, y) is thus always bounded by $\mathcal{O}(\min(k, n \log n))$.