Minimum k-cut algorithm using Gomory–Hu Trees

Students Shreyas Swanand Atre, Panagiotis Syskakis

Problem Description & Applications

Basic terminology

Before introducing the problem, we will go over some basic terms used throughout this report.

A **cut** is a partitioning of the vertices of a graph into two disjoint subsets. In an undirected weighted graph, we associate each cut with a **value**, which is the sum of the weights of edges across the two partitions.

Similarly, a **k-cut** is a partitioning of the graph vertices into k disjoint subsets, and the sum of the weights of cross-partition edges gives the value of the k-cut.

For our scope, out of all possible partitions, there is one (or more) cut with minimal value (minimum algebraic sum of the edge weights). This is called the **minimum cut**. Respectively, a **minimum k-cut** is a k-cut with minimal cut value.

Finally, when given a specific source vertex $s \in V$ and a sink vertex $t \in V$, an **s-t cut** is a cut that separates s from t. Similarly, a minimum s-t cut is an s-t cut that minimizes our metric (sum of weights of cross-cut edges).

The Minimum k-cut problem

The minimum k-cut is an NP-complete combinatorial optimization problem. A brute-force approach has a time complexity that is exponential to the number of nodes: It would require considering all possible ways to partition the n nodes into k partitions. The number of such unique combinations is known as "Stirling number of the second kind", and has a lower bound that grows exponentially with n [5].

Thus, the computation of the Minimum k-cut is usually done using some approximate algorithm. Saran & Vazirani [6] present two such algorithms, the Gomory-Hu-based approach being one of them.

As graphs are used to represent all kinds of networks, the problem of minimum cut or minimum k-cut appears in many different contexts, where there is a need to find optimal locations for network segmentation or clustering. Some notable ones we found mentioned are:

- VLSI circuit partitioning while minimizing interconnections
- Minimizing communication in distributed systems and parallel processing
- Community detection in social networks

The Gomory-Hu Tree

A **Gomory-Hu Tree** [3] is a weighted tree containing all graph vertices. Each edge of the Gomory-Hu Tree is associated with a value, which **represents the value of the minimum cut** that would separate the two connected components on either side of that edge.

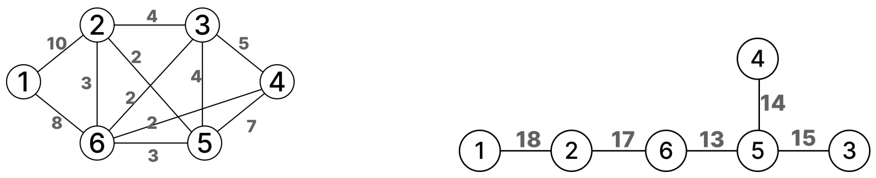


Figure 1: A weighted undirected graph (left) and its Gomory-Hu Tree (right).

It can be proven that for any two vertices, their minimum s-t cut is represented in the minimum-weighted edge in their connecting path in the tree (its weight gives the cut value, and the two connected components created by removing it represent the cut).

Using the Gomory-Hu Tree for minimum k-cut

The Gomory-Hu Tree can also be used to find an approximate solution for the minimum k-cut of a graph. This is done by using the Gomory-Hu tree to find the minimum $k-1$ edges, and using those for the k-cut [6].

This "greedy" approach is an approximate solution for the k-cut problem. However, it is proven to have an approximation factor of $2-\frac{2}{k}$. This factor bounds the ratio of the approximate and the optimal solution, regarding the "cost" metric (in this case, the sum of the weights of the cut edges in the k-cut).

Algorithm

Overview and intuition:

The Gomory-Hu Tree T begins as a single supernode (Fig. 3) containing all graph vertices of an undirected weighted graph G (Fig. 2).

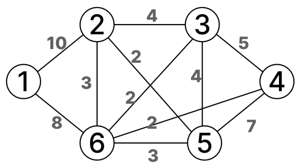


Figure 2: Original graph G

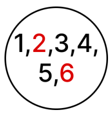


Figure 3: Intermediate Gomory-Hu (GH) tree, iteration 0

Each iteration performs the following steps:

1. A supernode S and two random nodes in it, s and t, are selected
2. S is split into two supernodes S_s and S_t (Fig. 5), based on a minimum cut between s and t (Fig. 4). The minimum cut is performed on a modified graph, $S(G)$, and not on the original graph G, as will be explained later.
3. S is replaced in the tree by S_s and S_t, which get connected with an edge of value equal to that of the minimum s-t cut. Supernodes that were previously connected to S now get connected to either S_s or S_t, depending on which side of the minimum cut they are on.
4. The above steps are repeated for $n-1$ iterations, when n supernodes have been generated with each containing exactly one node.

In the example presented here, starting with the one initial supernode (Fig. 3) s=2 and t=6 are arbitrarily selected. Their minimum cut is computed (Fig. 4) and has a value of 17, and thus the initial node is replaced (Fig. 5). For the next iteration, s=2 and t=1 are selected from the left supernode.

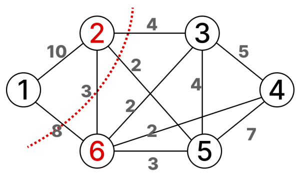


Figure 4: $S_1(G)$

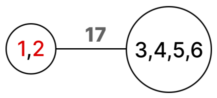


Figure 5: Intermediate GH Tree, iteration 1

In step 2 of the algorithm, there is an important caveat: The minimum s-t cut is NOT computed on the original graph G, but on $S(G)$, which is a contracted version of G (depicted in the left-hand side figures). $S(G)$ is constructed as follows: if we imagine the working supernode absent from the intermediate Gomory-Hu Tree, the tree is separated into connected components. $S(G)$ is created by contracting nodes in G that belong to the same such connected component. The minimum s-t cut is found on this contracted graph, which is uniquely constructed in every iteration. This way, we avoid finding an s-t cut that separates vertices of supernodes other than the one we are working with, and each neighboring supernode in T falls cleanly on either side of the cut.

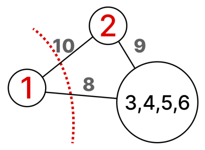


Figure 6: $S_2(G)$

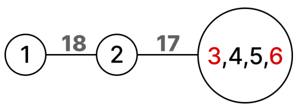


Figure 7: Intermediate GH Tree, iteration 2

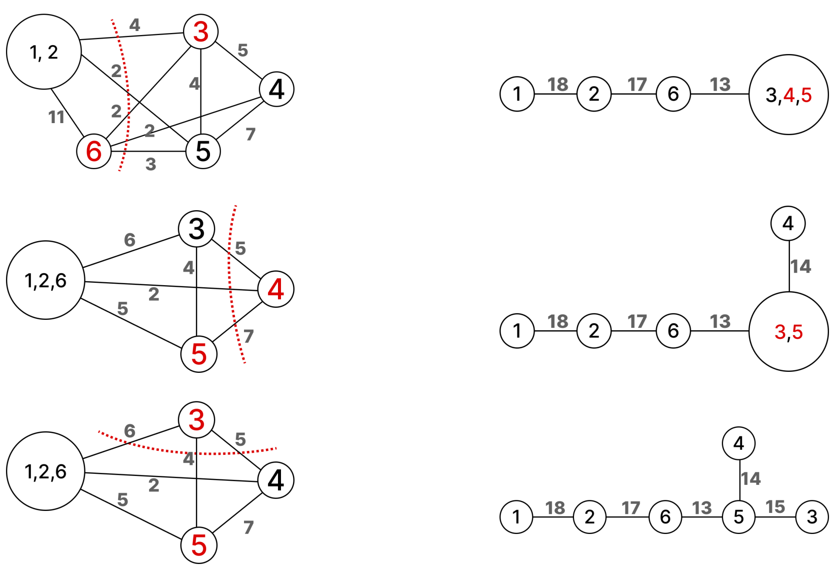


Figure 8: The rest of the algorithm iterations, showing the contracted graphs $S_i(G)$ where the min-cuts are computed (left) and the intermediate Gomory-Hu trees (right).

Algorithm Steps:

Let us now describe the complete algorithm:

Input: A weighted undirected graph $G = (V_G, E_G)$

Output: A Gomory-Hu Tree $T = (V_T, E_T)$ for G

Steps:

1. Set $V_T = \{V_G\}$ and $E_T = \varnothing$. This is the initial supernode containing all vertices of G.
2. While there is some supernode $X \in V_T$ where $|X| > 1$, repeat steps 3 to 5:
3. Construct the contracted graph $S = (V_S, E_S)$ as follows: For each connected component $C = (V_C, E_C)$ in $T - X$, let $S_C = \cup_{v \in V_C} v$. That is, S_C is a supernode consisting of all vertices in each connected component. Then, $V_S = X \cup \{S_C | \text{C is a connected component in } T - X\}$. E_S is found by retargeting E_G to their corresponding supernodes in V_S.
4. Choose any pair $s, t \in X$, and compute a minimum s-t cut (A, B) in S. This splits X into $X_A = X \cap A$ and $X_B = X \cap B$.

5. For each supernode $Y \in V_T$ previously connected to X, connect to X_A if $Y \cap A \neq \varnothing$ (meaning that Y's nodes fall in the A-side of the cut), otherwise connect to X_B.

Big-O Complexity:

Since the algorithm is expressed in terms of high-level graph operations, reasoning about time complexity requires us to make a few assumptions about the graph data structures. We assume O(1) access, insertion, and deletion (assuming we know the position of the element), while traversing the edges of a k-degree node is O(k). Access to the attributes of a given node or edge (such as edge weight, supernode elements, etc.) is also O(1).

The algorithm will finish after $n = |V_G|$ iterations, as each iteration produces a new supernode in T. On each iteration, we perform the following computations:

1. Finding connected components in $T - X$. The algorithm doesn't specify exactly how this should be done, but a simple procedure is to use a graph traversal algorithm (for example DFS). For each tree node, if unvisited, start a DFS to visit the whole connected component. Since each node is considered only once, this has complexity $O(|V| + |E|)$.

2. Performing vertex contractions to form S. Firstly, this requires making a copy of G ($O(|V| + |E|)$). To contract edge e_{ab} between nodes a and b, we delete b and re-target its edges to a. This needs some number of operations proportional to the degree of b, which in the worst case is —V—-1. Each edge contraction removes an edge, so in the worst case, we have $|E|$ contraction operations. So, this step is bounded by $O(|V||E|)$ (for graphs with a bounded or constant average vertex degree, this can be claimed to be $O(|V|)$)

3. Computing the minimum s-t cut in S. There are various algorithms to do this. We will consider the Push-Relabel algorithm, which is considered one of the most efficient minimum-cut algorithms, and has a time complexity of $O(|V|^2\sqrt{|E|})$ [1].

4. Replacing X with X_A and X_B in the tree. Assuming that we can probe which side of the cut the neighbors of X are in $O(1)$, X has $|V| - 1$ neighbors in the worst case, so this step is $O(|V|)$.

Combining all the above, for the $|V|$ iterations, we get:

$$|V| \cdot O(|V| + |E| + |V||E| + |V|^2\sqrt{|E|} + |V|)$$

which is equivalent to:

$$O(|V|^2|E| + |V|^3\sqrt{|E|})$$

Proof of Correctness

Proof of the Gomory-Hu Algorithm

The proof is partly based on the following invariants:

At any stage of the algorithm, for the intermediate Gomory-Hu Tree T:

1. A new edge is added in T, which represents a minimum cut in the original graph

2. All edges in T represent a minimum cut of some nodes contained in the supernodes they connect.

These two invariants guarantee that on termination (where all $|V| - 1$ supernodes contain exactly one node) all nodes in T are connected by an edge that represents the minimum cut between them.

In addition, the following two properties are essential:

1. For any nodes not connected in T, their minimum cut can also be found in T, and is represented by the minimum-value edge in the path that connects them.

2. The minimum cut in the contracted graph (which is a "local" operation relative to the working supernode in each iteration) is a minimum cut in the original graph ("globally").

Let us now prove these statements individually.

Let $G(V_G, E_G)$ be an undirected graph, with $w_{i,j}$ and $f_{i,j}$ representing the weight and the minimum cut value between vertices $i, j \in V_G$ respectively.

Let us first prove that Gomory-Hu Trees can exist, meaning, there can be a tree that encodes the minimum cut for any two nodes of G.

The Gomory-Hu Tree T must satisfy the following two requirements for any nodes $i, p \in G$:

1. If edge e_{ip} in T, it has value f_{ip}
2. If edge e_{ip} NOT in T, and supposing $e_{ij}, e_{jk}, ...e_{op}$ are in the tree and form a connecting path from i to p:
$$f_{ip} = min(f_{ij}, f_{jk}, ...f_{op})$$

Lemma 1

Let i, j, k be three vertices in the graph, and let f_{ik}, f_{ij}, f_{jk} represent the minimum cut value between corresponding vertices. It must hold that:

$$f_{ik} \geq min(f_{ij}, f_{jk})$$

and by induction, for $i, j, ...p \in V_G$:

$$f_{ip} \geq min(f_{ij}, f_{jk}, ...f_{op})$$

This can be proven by contradiction: Let's suppose that this didn't hold, so $f_{ik} < min(f_{ij}, f_{jk})$. Let's also suppose (WLOG) that, on the minimum cut represented by f_{ik}, j falls on the side of i. This means that the cut separates j from k. $f_{ik} < f_{jk}$ is now a contradiction, since f_{jk} is defined as a minimum cut, thus a cut with a smaller value cannot exist. Similarly, in the case where f_{ik} separates i from j, $f_{ik} < f_{ij}$ is a contradiction. Since one of the two cases must happen, it must hold that either $f_{ik} \geq f_{jk}$ or $f_{ik} \geq f_{ij}$, proving that $f_{ik} \geq min(f_{ij}, f_{jk})$.

Lemma 2

Similarly to the familiar maximum-spanning tree over the edge weights of a graph, let us imagine a maximum-spanning tree where the edge values are given by the minimum s-t cut values. This is a tree containing all vertices of G, where the value of edge e_ij is f_ij, and, out of all possible such trees, the tree is selected such that the sum of the edge values is maximal.

For any edge e_{ip} NOT in the minimum-spanning tree, and supposing $e_{ij}, e_{jk}, ...e_{op}$ are in the tree:

$$f_{ip} \leq min(f_{ij}, f_{jk}, ...f_{op})$$

This must hold because if the opposite were true, there would be a contradiction: a spanning tree of higher value could be produced by removing the minimum of those edges and adding e_{ip}.

Combining with Lemma 1, for any edge e_{ip} NOT in the maximum-spanning tree:

$$f_{ip} = min(f_{ij}, f_{jk}, ...f_{op})$$

And thus this maximum-spanning tree is a Gomory-Hu Tree.

The following Lemmas will now demonstrate that the algorithm steps actually lead to the construction of a Gomory-Hu Tree.

Lemma 3

The Gomory-Hu algorithm, after selecting two nodes in some supernode, contracts nodes in all other connected components created by omitting that supernode from the Tree. This Lemma proves that the minimum cut in the contracted graph is equivalent to a minimum cut in the original.

Let $(A, \hat{A})$ a minimum s-t cut in an undirected weighted graph G. Let $S(G)$ be a modification of G, such that all nodes in $\hat{A}$ are contracted. A minimum cut between $i, j \in A$ in $S(G)$ is numerically equal to an equivalent minimum cut in G.

Proof:

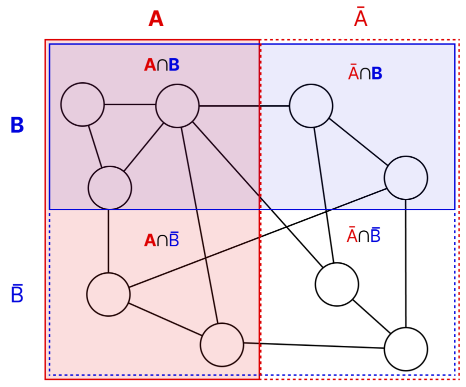


Figure 9

Let's consider $(A, \hat{A})$ a minimum cut between nodes k,l in the original graph G. W.l.o.g, consider $k \in A$ and $l \in \hat{A}$. Now, consider nodes $i, j \in A$, and $(B, \hat{B})$ a minimum cut that separates them. Let b_{XY} denote the sum of weights of edges between sets X, Y. Also, let's denote the four disjoint "quadrants":

$$X = (A \cap B)$$

$$X' = (A \cap \hat{B})$$

$$Y = (\hat{A} \cap B)$$

$$Y' = (\hat{A} \cap \hat{B})$$

We can see that cut $(X \cup Y \cup Y', X')$ separates i,j, however, we know $(B, \hat{B})$ to be a minimal i, j cut, so:

$$b_{B,\hat{B}} \leq b_{(X \cup Y \cup Y'),X'}$$

$$\implies b_{X,X'} + b_{Y,Y'} + b_{X,Y'} + b_{X',Y} \leq b_{X,X'} + b_{X',Y} + b_{X',Y'}$$

$$\implies b_{Y,Y'} + b_{X,Y'} \leq b_{X',Y'}$$

Similarly, we know $(A, \hat{A})$ to be a minimal k, l cut, while cut $(X \cup X' \cup Y', Y)$ also separates them:

$$b_{A,\hat{A}} \leq b_{(X \cup X' \cup Y'),Y}$$

$$\implies b_{X,Y} + b_{X,Y'} + b_{X',Y} + b_{X',Y'} \leq b_{X,Y} + b_{X',Y} + b_{Y,Y'}$$

$$\implies b_{X,Y'} + b_{X',Y'} \leq b_{Y,Y'}$$

Adding these two inequalities yields $b_{X,Y'} \leq 0$ which, since weights are non-negative, implies $b_{X,Y'} = 0$

Substituting that to the inequalities gives $b_{Y,Y'} \leq b_{X',Y'}$ and $b_{X',Y'} \leq b_{Y,Y'} \implies b_{X',Y'} = b_{Y,Y'}$

This suffices to prove $b_{B,\hat{B}} = b_{(X \cup Y \cup Y'),X'}$, thus $((X \cup Y \cup Y'), X')$ is also a minimum j,k cut, and is a cut that will be found even when $\hat{A} = Y \cup Y'$ are contracted into a single node.

Lemma 4

Consider two nodes i and p, and an intermediate Gomory-Hu Tree T. By definition of f_{ip} being the min-cut flow, it holds that $f_{ip} \leq min(e_{ij}, ..e_{op})$, since $e_{ij}, ..e_{op}$ all represent cuts that separate i from p.

Because $e_{ij}, ..e_{op}$ represent minimum flow cuts, we can apply Lemma 1: $f_{ip} \geq min(e_{ij}, ..e_{op})$, which combines with the previous inequality to prove:

$$f_{ip} = min(e_{ij}, ..e_{op})$$

Lemma 5

This lemma proves that an edge in the intermediate Gomory-Hu Tree always represents a minimum cut for some nodes contained in the two supernodes it connects. Formally:

If an edge of value v connects sets A_i and A_j then there is a node N_i in A_i and a node N_j in A_j such that $f_{ij} = v$

Let (A_j, A_i) be a minimum cut on A for nodes N_i, N_j. For this first cut, this is obviously true, as $v = f_{ij}$.

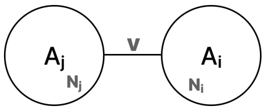


Figure 10: Initial cut

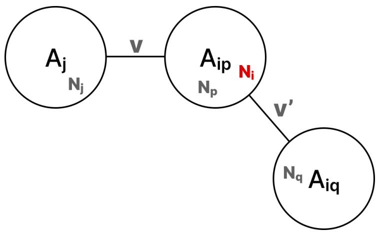


Figure 11: Case 1

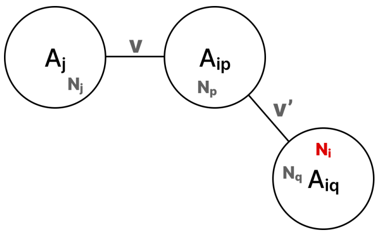


Figure 12: Case 2

Let us consider what happens to edge v for a subsequent cut on A_i, (A_{ip}, A_{iq}) with value $v' = f_{pq}$. There are two cases for N_i:

1. Ni is in A_{ip}, then $v = f_{ij}$ is still applicable, as it is a minimum cut value of nodes N_i and N_j, which are contained on the sets connected by this edge.

2. Ni is in A_{iq}, then we can show that $v = f_{jp}$, or in other words, v is a minimum cut value of nodes N_j, N_p. From Lemma 1, we know that:

$$f_{jp} \geq min(f_{ji}, f_{iq}, f_{qp})$$

. In this case, we observe that f_{jp} must be unaffected by the cut represented by f_{iq}, since N_i, N_q fall on the opposite side of the tree. Thus,

$$f_{jp} \geq min(f_{ji}, f_{qp})$$

or equivalently, by how the cuts were created:

$$f_{jp} \geq min(v, v')$$

Since v' is a cut that separates N_i, N_j, whose minimum cut value was v, it must hold that $v' \geq v \implies min(v, v') = v$, so finally

$$f_{jp} \geq v$$

But since v separates N_j, N_p, it must also hold that $f_{jp} \leq v$, thus proving that $f_{jp} = v$

These Lemmas suffice to prove the validity of the algorithm:

At any iteration, a minimum cut between s-t is introduced as a new edge (Lemma 3).

At the final iteration, where each supernode has only one node, Lemma 5 tells us that any edge e_{jk} will have value f_{jk}, while Lemma 4 tells us how to obtain f_{pq} even if e_{pq} is not in the tree.

Proof of Gomory-Hu k-cut approximation factor

Let edges C constitute an optimal k-cut, and $A_1, A_2, ...A_k$ be the k components produced by that cut. Let $C_1, C_2, ...C_k \in C$ be the cuts that separate $A_1, .., A_k$ from the rest of the graph. Each edge in C connects two components, so it can be used maximally twice in all $C_1, ...C_k$. Thus, with $w()$ giving the value of the cut:

$$\sum_{i=0}^{k} w(C_i) = 2w(C)$$

Now, transform a Gomory-Hu Tree T by merging together nodes belonging to A_i. Root that tree in A_k and consider any leaf A_i. For edge e_i connecting A_i with its parent, we know that $w(e_i) \leq w(C_i)$, since e_i was an edge in T. Assuming w.l.o.g. that cuts C_i are in increasing value:

$$\sum_{i=0}^{k-1} w(e_i) \leq \sum_{i=0}^{k-1} w(C_i) \leq 2(1 - \frac{1}{k})w(C)$$

Implementation & Experimental Measurements

The Gomory-Hu algorithm was implemented in C++. The implementation was facilitated by LEMON, a C++ graph library focused on combinatorial optimization tasks.

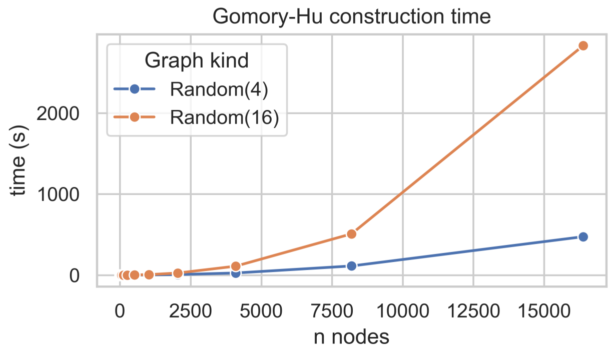
LEMON provides data structures such as a ListGraph class, which represents a graph as an adjacency list. It also allowed us to add node or edge attributes using the template classes NodeMap<T> or EdgeMap<T> respectively, where T can be any data type. These are essentially arrays indexed by the node/edge id, giving O(1) access time. Supernodes, which are extensively used in the construction of the Gomory-Hu Tree, were modeled by associating a vector of nodes with each node of the tree (using LEMON, this is done using a NodeMap<Vector<Node>>)

To construct a contracted graph (which is needed in each iteration of the Gomory-Hu algorithm), we first make a copy of the original graph. Then, a customized implementation of DFS is used to find the connected components in the intermediate Gomory-Hu Trees, contracting nodes as they are visited.

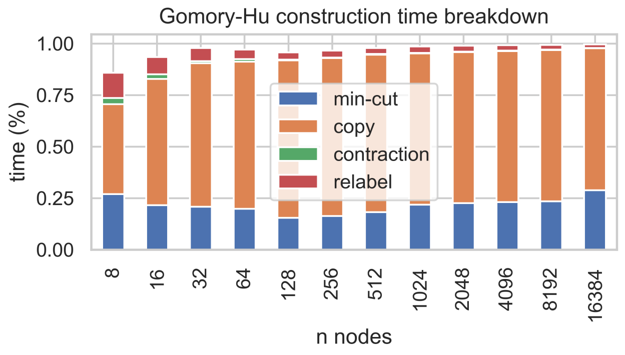
In addition to the original Gomory-Hu Algorithm, we also implemented a variation provided by Dan Gusfield [4]. Gusfield's algorithm, being simpler in its implementation, provided an additional point of reference for the correctness and performance of our implementation.

We measured our implementation of the Gomory-Hu algorithm on our personal computers. We used synthetic graphs generated with the GTGraph random graph generator. We used two different graph generators:

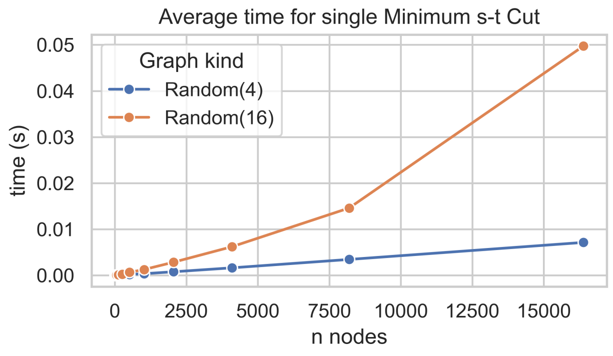
- "Random" generator $G(n, m)$ where n is the number of desired nodes, and m the number of desired edges. Each edge was assigned a random source and target. Using this, we produced graphs where m was a constant multiple of n specifically $m = 4n$ or $m = 16n$ $((|E| \propto |V|)$.
- "Erdős–Rényi" generator $G(n, p)$ where out of n nodes, each possible pair obtains an edge with probability p. Using this, we produced graphs where the number of edges scaled superlinearly with n $(|E| \propto |V|^2)$ [2].



(a) Gomory-Hu Tree construction time



(b) Time breakdown for *Random(16)* graph



(c) Average time of the Push-Relabel Minimum Flow algorithm for random s-t pairs

.

Figure 13: Time measurements of the Gomory-Hu algorithm on "Random" graphs (*Random(x)* indicates the ratio of edges to nodes is x).

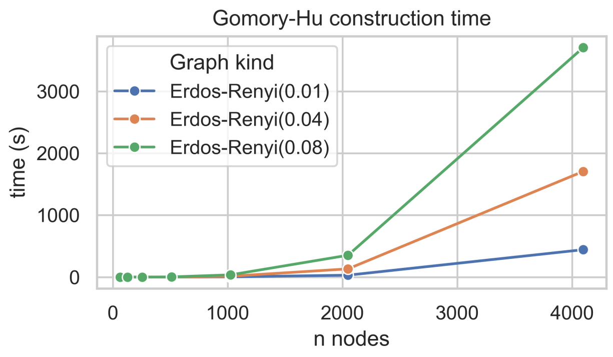
Fig. 13a shows time scaling for the "Random" family of generated graphs for an increasing number of nodes. This is superlinear, as expected.

Fig. 13b shows the proportion taken by each step of the algorithm, measured cumulatively over all iterations ("copy" refers to making a copy of the original graph G used for creating the contracted graph $S(G)$, "contraction" refers to performing the DFS and contracting edges, and "relabel" refers to replacing the original supernode with the two new supernodes created by the "min-cut"). An unexpected amount of time is taken by copying, which is done using a subroutine of the LEMON graph library. We expect that this can be significantly optimized, as memory copying is essentially a $O(n)$ operation, however, time constraints didn't allow us to investigate this further.

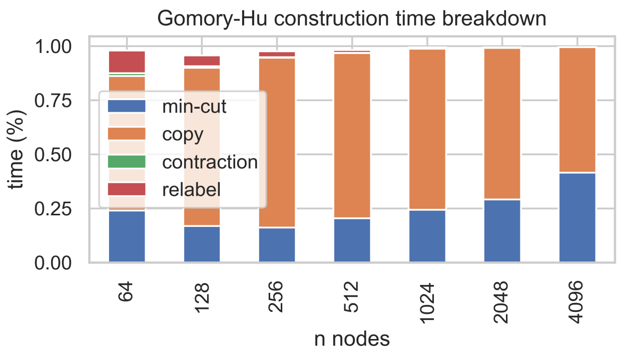
Fig. 13c shows the time scaling only for the minimum cut subroutine, for an increasing number of nodes. Although the scaling is superlinear, we observe it is quite close to linear for the less dense graph.

Fig. 14 shows the same set of measurements for the "Erdős–Rényi" generated graphs. The results are consistent with the previous ones, taking into account that the number of edges now scales quadratically with the number of nodes.

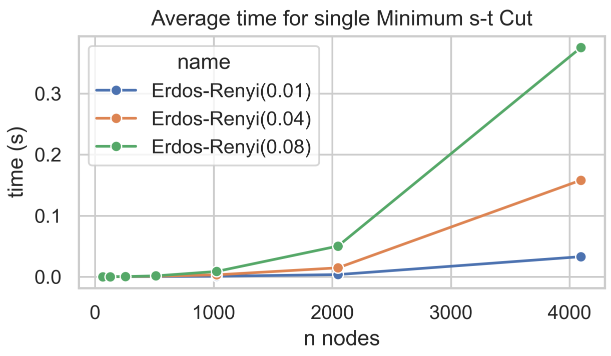
Finally, Fig. 15 presents results from Gusfield's algorithm, which is another algorithm for computing the Gomory-Hu Tree. Its operation is largely based on the Gomory-Hu work and the proofs presented here, and its implementation is much simpler and more efficient. It does not require performing graph contractions, so as Fig. 15b shows, the majority of running time is consumed in the "min-cut" subroutine ($|V|-1$ minimum cut computations in total, just like the original Gomory-Hu algorithm).



(a)



(b) Time breakdown for *Erdos-Renyi(0.08)* graph

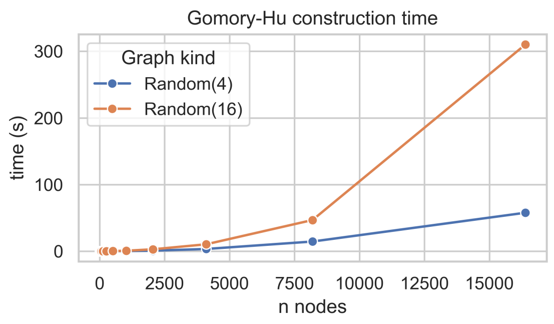


(c) Average time of the Push-Relabel Minimum Flow algorithm for random s-t pairs

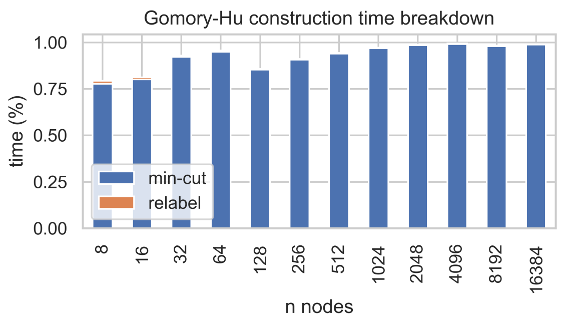
.

Figure 14: Time measurements of the Gomory-Hu algorithm on "Erdos-Renyi" graphs (*Erdos-Renyi(p)* indicates p probability of connection for each possible node pair).

It performs around 10x faster than our implementation of the original Gomory-Hu algorithm, although a large part of that gap could be attributed to the abnormally slow copy operation of the original implementation.



(a)



(b)

Figure 15: Time measurements (a) and time breakdown (b) using **Gusfield's algorithm**, on graphs produced with the "Random" generator.

References

[1] Ravindra K. Ahuja, Murali Kodialam, Ajay K. Mishra, and James B. Orlin. Computational investigations of maximum flow algorithms. *European Journal of Operational Research*, 97(3):509–542, 1997.

[2] Paul Erd6s and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960.

[3] 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.

[4] Dan Gusfield. Very simple methods for all pairs network flow analysis. *SIAM Journal on Computing*, 19(1):143–155, 1990.

[5] B.C. Rennie and A.J. Dobson. On stirling numbers of the second kind. *Journal of Combinatorial Theory*, 7(2):116–121, 1969.

[6] H. Saran and V.V. Vazirani. Finding k-cuts within twice the optimal. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 743–751, 1991.