A computational study of Gomory-Hu tree algorithms

Vladimir Kolmogorov
Institute of Science and Technology Austria (ISTA)
vnk@ist.ac.at

Abstract

We present an experimental study of algorithms for computing the Gomory-Hu tree (aka cut tree) in undirected weighted graphs. We develop a new implementation based on two popular maxflow algorithms, IBFS and BK. We compare it with the algorithms from the previous experimental study by Goldberg and Tsioutsiouliklis (2001) and with the more recent algorithm by Akibo et al. (2016) (designed for unweighted simple graphs). Results indicate that on some classes of problems new implementation significantly outperforms previous methods.

1 Introduction

We study the problem of computing the Gomory-Hu tree [15] (aka cut tree) in an undirected weighted graph \( G = (V, E, w) \) with non-negative weights. This is a data structure that allows to efficiently compute the cost of a minimum \( s-t \) cut (denoted as \( f_G(s,t) \)) for any pair of nodes \( s,t \). Formally, the Gomory-Hu tree can be defined as a weighted spanning tree \( T \) with the property that \( f_T(s,t) = f_G(s,t) \) for all pairs \( s,t \). It has numerous applications in various domains (see e.g. Section 1.4 in [3]).

Gomory and Hu [15] showed how to solve the problem using \( n-1 \) maximum flow computations and graph contractions. (As usual, we denote \( n = |V| \) and \( m = |E| \)). An alternative simple algorithm that does not involve contractions has been proposed by Gusfield [16]; it performs all \( n-1 \) maxflow computations on the original graph. \( O(n \cdot MF(n,m)) \) remains the best known complexity for computing the Gomory-Hu tree in general graphs (where \( MF(n,m) \) is the time for computing a maximum flow in a graph with \( n \) nodes and \( m \) edges), though better bounds are known for unweighted graphs [7, 3, 2, 22, 4].

This paper focuses on a computational study of algorithms for computing a Gomory-Hu tree. The first such study by Goldberg and Tsioutsiouliklis appeared in [12]. It relied on the preflow maximum flow algorithm [11] for which efficient implementations had been developed [9]. A more recent study by Akibo et al. [5] focused on real sparse graphs such as social networks and web graphs. It described an implementation that could handle graphs with billions of nodes. This implementation, however, is restricted to unweighted simple graphs. Its features include a tailored bidirectional Dinitz algorithm and graph reduction techniques such as tree packing (which works only for unweighted graphs), contracting degree-2 nodes and elimination of bridges.

The main motivation behind this work was to incorporate more recent maxflow algorithms, in particular the Boykov-Kolmogorov (BK) [8] and the Incremental Breadth-First Search (IBFS) [14, 13] algorithms. They were shown to be state-of-the-art for some classes of problems, so it is natural to ask whether they can speed up the computation of Gomory-Hu trees. We answer this question affirmatively: our results show that suitably modified BK and IBFS algorithms can offer a dramatic improvement for some classes of real-world graphs (in particular, graphs that come from the TSP problem) over the preflow-based implementation of [12].

Partial Gomory-Hu tree Some applications require only a partial knowledge of the tree. In particular, we can consider two scenarios:
• Compute \( f(s, t) \) only for pairs \( s, t \) in a specified subset of nodes \( \hat{V} \subseteq V \). Such problem first appeared in [19]. One application is the odd minimum cut-set problem [23] in which minimum cuts need to be computed only between odd-labeled nodes.

• Compute \( f(s, t) \) only for pairs \( s, t \) with \( f(s, t) \leq k \) for a given constant \( k \). This problem was considered in [18] who introduced the notion of a \( k \)-partial tree and showed how to compute it in expected time \( \tilde{O}(m + nk^3) \) for unweighted graphs. As a motivation, [18] gives the problem of deciding the robustness of a network and finding the edge connectivities of the unreliable portions in it; in such situations, one is interested only in finding the edge connectivities of those pairs of vertices that are poorly connected in the network. Another potential use of \( k \)-partial trees is for solving a separation problem for some classes of inequalities; e.g. the problem of separating \( b \)-matching inequalities can be reduced to an odd minimum cut-set problem [23] with the threshold value \( k = 1 \).

Motivated by these applications, we describe a version of the Gomory-Hu algorithm that combines the two features above. These features are included in our code, but we report the results only for the complete Gomory-Hu algorithm.

2 Algorithm description

In this section we assume that graph \( G \) implicitly comes with a subset \( \hat{V} \subseteq V \), and write \( G = (V, E, w; \hat{V}) \). Such \( G \) will be called a marked graph. For brevity, for a subset \( X \subseteq V \) we will write \( X = X \cap \hat{V} \), when \( G \) is clear from the context. A cut in \( G \) is a subset \( S \) with \( \emptyset \subseteq S \subseteq V \). Its cost will be denoted as \( \text{cost}_G(S) \). It is an \( s \)-\( t \) cut for for distinct \( s, t \in V \) if \( |S \cap \{s, t\}| = 1 \). The cost of a minimum \( s \)-\( t \) cut in \( G \) will be denoted as \( f_G(s, t) \).

A partition tree for graph \( G \) is a spanning tree \( \mathcal{T} = (\Pi, \mathcal{E}) \) where \( \Pi \) is a partition of \( V \) such that \( |\hat{X}| \geq 1 \) for every \( \hat{X} \in \Pi \). An element \( \hat{X} \in \Pi \) will also be called a supernode of \( \mathcal{T} \). The trivial partition tree with a single node will be denoted as \( \mathcal{T}^{\text{empty}} = (\{V\}, \emptyset) \). Each edge \( XY \in \mathcal{E} \) defines a cut in \( G \) in a natural way; it will be denoted as \( C_{\mathcal{T}}(XY) \). We view \( \mathcal{T} \) as a weighted tree where the weight of \( XY \) (equivalently, \( f_{\mathcal{T}}(X, Y) \)) equals \( \text{cost}_G(C_{\mathcal{T}}(XY)) \). The following is a natural generalization of the definition in [18] to marked graphs.

**Definition 1.** Partition tree \( \mathcal{T} = (\Pi, \mathcal{E}) \) is a \( k \)-partial tree for a marked graph \( G = (V, E, w; \hat{V}) \) if for every \( x, y \in \hat{V} \) with \( f_G(x, y) \leq k \) nodes \( x, y \) belong to distinct supernodes of \( \mathcal{T} \) (\( x \in X, y \in Y \)), and \( f_G(x, y) = f_{\mathcal{T}}(X, Y) \). It is a coarsest \( k \)-partial tree if in addition \( f_G(s, t) > k \) for every \( s, t \in \hat{X} \in \Pi \).

Next, we describe an algorithm for computing a \( k \)-partial tree. It maintains a marked graph \( G \) and a partition tree \( \mathcal{T} \) for \( G \). In addition, it maintains weight \( \lambda(X) \in \mathbb{R}_+ \cup \{+\infty\} \) for every supernode \( X \in \Pi \). When computing \( k \)-partial tree, we would set \( \lambda(X) = k \) for all \( X \). We will consider, however, a more general algorithm that is allowed to decrease \( \lambda(X) \) for some supernodes: this flexibility comes “for free”, and could potentially be used for adaptively choosing threshold \( k \). We assume that weight function \( \lambda \) is a part of \( \mathcal{T} \), and write \( \mathcal{T} = (\Pi, \mathcal{E}, \lambda) \).

To initialize, we set \( G \) to be the input graph, \( \mathcal{T} \) to be the empty tree \( \mathcal{T}^{\text{empty}} \) with an arbitrary value \( \lambda(V) \). The pair \((G, \mathcal{T})\) is then modified by repeatedly applying the procedure MinCutStep\((G, \mathcal{T})\) (Algorithm 1). In this algorithm we use the following notation for a graph \( G \), partition tree \( \mathcal{T} = (\Pi, \mathcal{E}, \lambda) \) on \( V \) and supernode \( X \in \Pi \):

- \( H = G[\mathcal{T}, X] \) is the auxiliary graph obtained from \( G \) as follows: (i) take forest \( \mathcal{F} \) on \( \Pi - \{X\} \) obtained from tree \( (\Pi, \mathcal{E}) \) by removing node \( X \); (ii) let \( C_1, \ldots, C_k \) be the connected components of \( \mathcal{F} \), and let \( Y_i = \bigcup_{Y \in C_i} Y \) for \( i = 1, \ldots, k \); (iii) let \( H \) be the graph obtained from \( G \) by contracting each \( Y_i \) to a single vertex \( v_{Y_i} \).
For a set $X'$ let $\mathcal{T}[X \mapsto X']$ be the partition tree obtained from $\mathcal{T}$ by replacing all occurrences of $X$ with $X'$ (e.g., each edge $XY \in \mathcal{E}$ is replaced with the edge $X'Y$), the weight of $X'$ is set to be $\lambda(X') = \lambda(X)$, etc.

Algorithm 1: MinCutStep : $(G, \mathcal{T}) \mapsto (G', \mathcal{T'})$ where $G = (V, E, w; \hat{V})$ and $\mathcal{T} = (\Pi, \mathcal{E}, \lambda)$.

Precondition: there exists $X \in \Pi$ with $|X| \geq 2$.

1. pick supernode $X \in \Pi$ with $|X| \geq 2$ and distinct $s, t \in \hat{X}$
2. optionally, decrease $\lambda(X)$ to a non-negative value
3. form auxiliary graph $H = G[\mathcal{T}, X]$
4. compute minimum $s$-$t$ cut $S$ in $H$ and its cost $\text{cost}_H(S)$
5. if $\text{cost}_H(S) \leq \lambda(X)$ then
   6. update $\Pi := (\Pi - \{X\}) \cup \{A, B\}$ where $(A, B) = (X \cap S, X - S)$, set $\lambda(A) = \lambda(B) = \lambda(X)$
   7. update $\mathcal{E} := \mathcal{E} \cup \{AB\}$
   8. for each edge $XY \in \mathcal{E}$ do the following: (i) find index $i$ with $Y \in C_i$, $Y \subseteq Y_i$;
      (ii) define $C = \begin{cases} A & \text{if } v_{Y_i} \in S \\ B & \text{if } v_{Y_i} \notin S \end{cases}$; (iii) update $\mathcal{E} := (\mathcal{E} - \{XY\}) \cup \{CY\}$
9. else
   10. modify $G$ by contracting $\{s, t\}$ in $G$ to a single node called $s$ (“merging step”)
   11. update $V := V - \{t\}$, $\hat{V} := \hat{V} - \{t\}$, $\mathcal{T} := \mathcal{T}[X \mapsto (X - \{t\})]
12. return $(G, \mathcal{T})$

Clearly, each edge $YZ$ in $\mathcal{T}$ has the same weight as the corresponding edge $Y'Z'$ in $\mathcal{T}'$ (since these edges define the same cut in the graph, possibly after merging $s, t$).

The desired partition tree is obtained by embedding Alg. 1 in a recursive procedure CutTree$(G, \mathcal{T})$ as shown in Algorithm 2.

Algorithm 2: CutTree$(G, \mathcal{T})$.

Input: marked graph $G = (V, E, w; \hat{V})$, partition tree $\mathcal{T} = (\Pi, \mathcal{E}, \lambda)$ on $V$.
Output: partition tree $\mathcal{T}^* = (\Pi^*, \mathcal{E}^*, \lambda^*)$.

1. if $|X| = 1$ for all $X \in \Pi$ then return $\mathcal{T}$
2. call $(G', \mathcal{T'}) \leftarrow \text{MinCutStep}(G, \mathcal{T})$
3. call $\mathcal{T}^* \leftarrow \text{CutTree}(G', \mathcal{T'})$
4. if $G' \neq G$ (i.e. Alg. 1 used lines 10-11) then
   5. take supernode $X^* \in \Pi^*$ containing $s$, update $\mathcal{T}^* := \mathcal{T}^*[X^* \mapsto (X^* \cup \{t\})]
6. return $\mathcal{T}^*$

We say that $\mathcal{T}$ is consistent with $G$ if for any edge $XY$ of $\mathcal{T}$ there exists $x \in \hat{X}$, $y \in \hat{Y}$ such that $f_G(x, y) = f_\mathcal{T}(XY) = \text{cost}_G(C_\mathcal{T}(XY))$. Clearly, $\mathcal{T}^\text{empty}$ is consistent with $G$.

Theorem 2. Suppose $\mathcal{T}$ is consistent with $G$, $(G', \mathcal{T'}) = \text{MinCutStep}(G, \mathcal{T})$ and $\mathcal{T}^* = \text{CutTree}(G, \mathcal{T})$.
(a) $\mathcal{T}'$ is consistent with $G'$.
(b) $\mathcal{T}^*$ is consistent with $G$.
(c) For every supernode $X$ of $\mathcal{T}^*$ and every $s, t \in X$ there holds $f_G(s, t) > \lambda^*(X)$ where $\lambda^*$ is the weight function of $\mathcal{T}^*$.

Theorem 3. Suppose that step 2 of Alg. 1 (decreasing $\lambda(X)$) is never invoked. Then the output $\mathcal{T}^* = \text{CutTree}(G, \mathcal{T}^\text{empty})$ is a coarsest $k$-partial tree for $G$, where $k$ equals $\lambda(V)$ in the initial tree $\mathcal{T}^\text{empty}$.

The proofs of these theorems use fairly standard techniques, and we defer them to Appendix A.

Next, we discuss efficiency issues. Recall that in the introduction we considered two scenarios: (a)
compute $f(s,t)$ only for pairs in the specified subset $\hat{V}$; (b) compute $f(s,t)$ only for pairs with $f(s,t) \leq k$. Algorithm 2 seems appropriate for the first scenario: there is no clear reason to run maxflow computations for vertices not in $\hat{V}$. In contrast, using Algorithm 2 for the second scenario might help since we may be able to terminate maxflow computation at line 4 earlier once the current flow reaches $\lambda(X)$, but it may also be detrimental since it may prevent supernode $X$ from being split. In our informal tests we did not find a scenario where the merging step would have a clear benefit; most often the two versions had a similar performance.

Note that it may be possible to use the Hao-Orlin algorithm [17] for some of the merging steps. Recall that the latter performs the following sequence of operations for a graph on nodes $V$: (1) pick arbitrary source $s$, set $S = \{s\}$; (2) pick $t \in V - S$ and compute minimum $S$-$t$ cut; (3) add $t$ to $S$ and go to step (2). The choice of $t$ in step (2) is not arbitrary, and is determined by the Hao-Orlin algorithm. If this $t$ happens to be in the current supernode then it is natural to use it as the next sink, assuming that the previous computation resulted in a merging step. We have not implemented this approach, however.

3 Implementational details

Maxflow algorithms As stated in the introduction, the main motivation behind this work was to incorporate BK [8] and IBFS [13] algorithms. Importantly, these algorithms support dynamic graph updates (for the BK algorithm such updates were described in [10]). Since we need to call maxflow algorithm for different $s$-$t$ pairs, the functionality that we need is to change the capacity of arcs from the source $c_{si}$ and to the sink $c_{ti}$. (In fact, both implementations do not store these arcs implicitly, but instead maintain the difference $c_{si} - c_{ti}$, which is the “flow excess” at node $i$).

We added the following operation to BK and IBFS codes: contract a given list of nodes (given in an array) to a single node. Importantly, this is done “locally” by traversing only nodes in the given list and their incident arcs (and without traversing the entire graph).

As observed in [13], the data structure used for storing the graph may have a significant impact on the performance. The original BK implementation used the “adjacency list” (AL) representation, while the IBFS code used the “forward star” (FS) representation. [13] changed the BK code to use FS representation and showed a significant speed-up on some instances, which it attributed to a better cache utilization (the FS representation stores incident outgoing arcs of nodes in a consecutive array).

We modified BK and IBFS codes to use a common base graph class, which can be either AL or FS. Note that with the FS representation we may need to allocate new memory for arcs during graph contraction, while in AL we can simply replace previously allocated arcs. Consistent with [13], we found that FS performed better, sometimes significantly. We report results for both AL and FS implementations to facilitate comparison with [12], which used the AL representation. When the used representation is not clear from the context, we will refer to the algorithms as IBFS$^{FS}$, BK$^{FS}$, IBFS$^{AL}$, BK$^{AL}$.

On some classes of problems most of the $s$-$t$ cuts in the Gomory-Hu algorithms are very unbalanced (e.g. containing a single node). It is important that in such cases we do not traverse the entire graph during dynamic graph updates. In particular, a minimum $s$-$t$ cut should ideally be computed without traversing all nodes. We implemented the following approach: we maintain the lists of nodes $V^+$, $V^-$ with the positive excess and the negative excesses, respectively. (Each flow pushing operation updates these lists if necessary). 2 After the maxflow computation is finished,

1 A natural way to implement early termination is to add a new node $s'$, new edge $(s,s')$ of weight $\lambda(X)$, and compute maximum flow from $s'$ to $t$.
2 We believe that IBFS code already maintains buckets with lists $V^+$ and $V^-$, but we have not exploited them in the current version.
we start two breadth-first searches (one from $V^+$ and one from $V^-$), processing the component that currently has the smaller size.

Note that the BK code maintains a single list of “active” nodes in the source and the sink search trees. Guided by the same motivation as above, we modified this to maintain separate lists. When growing search trees, we then pick an active node from the list that has the smaller size. In addition, we terminate the maxflow algorithm if either list becomes empty (while the original BK implementation stops when no active nodes are left).

Order of operations The overall structure of our implementation is the same as in [12]: we pick a pair of nodes $s - t$ in the current component $X$ (initially $X = V$), compute a minimum $s-t$ cut, pick the side $S$ so that $|A| \leq |B|$ where $A = X \cap S$ and $B = X - S$, recursively solve the smaller problem (on $A$), contract nodes in $S$ according to the computed partition tree on $S$, and then update the current component $X := B$.

An important question is how to select nodes $s, t$. There are several guiding principles proposed in the literature. [12] proposes two heuristics that try to find a balanced cut: (i) choose two heaviest nodes (i.e. nodes with the highest total capacity of incident edges); (ii) after choosing $t$, choose $s$ that is furthest away from $t$ (with respect to unit edge lengths). [5] uses a somewhat opposite strategy: pick $s, t$ which are adjacent in $G$, if exist (otherwise pick an arbitrary pair). They argue that such choice leads to a smaller time for computing maxflow.

We implemented the following heuristic. In the beginning we compute $s, t$ as two heaviest nodes, and for each node $i$ compute distances $D_s(i), D_t(i)$ from these nodes (with respect to the unit edge lengths). After each MinCutStep one the terminals is preserved and the other one gets contracted. Suppose that $s$ is preserved (the other case is symmetric). We then do breadth-first search from (contracted) $t$, and for each layer check whether it has nodes eligible to become new sink. If yes, then we choose a node with the the smallest $D_s(i)$. Thus, this strategy attempts to find new sink $t$ which is close both to $s$ and to the old sink $t$.

4 Experimental results

We used a single 8-core machine with the processor Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz and 16Gb RAM. All codes were written in C / C++. For each instance we reordered nodes and edges by performing a breadth-first search (in order to improve locality). Next, we describe baseline methods, test instances, and discuss the results.

4.1 Baseline methods

The study [12] reported 4 algorithms named $gus, gh, ghs$ and $ghg$. The first one is an implementation of the Gusfield algorithm [16] that works on the original graph, while the other three perform graph contractions. $gh$ and $ghs$ differ in the source-sink selection strategy: the former picks an $s-t$ pair at random while the latter chooses $s$ which is furthest away from the current $t$. $ghg$ uses the Hao-Orlin algorithm instead of the standard preflow algorithm. [12] shows that a single Hao-Orlin run can identify several valid cuts (1 or 2 in their implementation).

We also evaluated the algorithm from [5] that we term MG (which stands for “Massive Graphs” from the title of the paper). Their code supports only unweighted simple graphs. It implements several preprocessing heuristics for graph reduction, namely tree packing for identifying singleton valid cuts, removing bridge edges, and contracting degree-2 nodes. In our experiments these heuristics did not significantly affect the running time (usually by a small factor less than two), so we report results for the default version that includes all heuristics. It uses a bidirectional Dinitz algorithm with a “goal-oriented search”: it precomputes a shortest path tree from a fixed sink $t$ and then uses this tree for multiple sources $s$. It also uses heuristics such as augmenting non-shortest paths with “detour edges” (see [5] for more details).
Finally, we experimented with the Gomory-Hu algorithm from the Lemon graph library [10].

4.2 Test instances

**Weighted graphs** First, we tested the algorithms on synthetic classes of graphs used in [12] and in earlier studies (only of bigger sizes). We used the generators that come with the code in [12]. In tables 1-18 we specify the exact commands that were used to generate the instances. The generators are randomized (except for CYC1 and DBLCYC1); as in [12], we report the numbers averaged over 5 runs with different random seeds.

Second, we tested TSP instances from the TSPLIB dataset [1]. Each instance is given by a set of points and a function that allows to compute a distance between two points. To obtain a sparse instance, we took the $kn$ edges with the smallest weight for $k=2,4,8$.

Results for weighted graphs are presented in tables 1-19. The first column has the format $n(d)m$ where $d \leq n - 1$ is the diameter of the computed Gomory-Hu tree (or a range of this diameter, if different algorithms produce different trees). Each entry has two rows: the upper and bottom rows correspond to implementations that use the adjacency list (AL) and the forward star (FS) representations, respectively. The caption in each table also gives the time of the Lemon’s implementation for the first instance in the corresponding table. All running times are given in seconds. The number in parentheses is the time spent in maxflow computations, while the main number is the total runtime.

We do not report results of ghs on TSP instances since for most of them ghs crashed with a segmentation fault.

**Unweighted simple graphs** We also compared IBFS, BK and MG on unweighted simple graphs. MG stores arcs incident to each node in a consecutive array; accordingly, we use the forward star versions of IBFS and BK.

First, we tested the algorithms on synthetic instances by taking the last instance in each of the tables 1-19 and changing the weight of all edges to one. Results are given in Table 20. Table 21 shows the results on TSP instances from Table 19 converted to unit weights.

Finally, we took large social and web graphs from [21] used in [5] (Table 22). Some of them are directed; we converted them to undirected. We also removed bridge edges and took the largest remaining connected component (motivated by the preprocessing technique in [5]).

4.3 Discussion of results

The runtime for IBFS and BK implementations is dominated by maxflow computations; for most of the instances the overhead is negligible. In contrast, for gus, gh, ghs and ghg the overhead can be many times larger than maxflow times. This is especially pronounced for instances with a large tree diameter - CYC1 and TSP. (This is consistent with the results in [12]). In our implementations we tried to avoid overheads by making sure that all update operations are done locally, without traversing the entire graph (when contracting smaller component, identifying minimum s-t cut, etc). Similar considerations were discussed in [5 Section II] for the MG implementation.

**Results on weighted graphs** IBFS FS is consistently the fastest algorithm followed by BK FS. This, however, can mostly be attributed to the used graph representation that has better locality and thus better cache utilization. The codes from [12] could potentially be modified to use the forward star representation. It is also not inconceivable that an alternative implementation could eliminate the overhead in these codes. We thus focus our subsequent discussion on the maxflow computation times for the adjacency list versions of the algorithms.

On this metric, algorithms from [12] are clear winners on many classes of problems (NOI1, NOI3, NOI4, NOI5, NOI6, PR7, PR8, REG1) and on par with IBFS and/or BK on many other
Table 1: BIKEWHE: bikewheelgen 1024, ..., 4196. (Lemon: 296).

|        | IBFS        | BK       | gus       | gh   | ghs       | ghg       |
|--------|-------------|----------|-----------|------|-----------|-----------|
| 1024   | 0.029 (0.027) | 0.065 (0.064) | 3.25 (3.17) | 3.07 (2.95) | 1.88 (1.78) | 0.75 (0.66) |
| 2045   | 0.019 (0.018) | 0.053 (0.052) |          |      |           |           |
| 2048   | 0.12 (0.11)  | 0.75 (0.75)  | 24.7 (24.3) | 24.0 (23.4) | 13.8 (13.2) | 5.06 (4.64) |
| 4093   | 0.072 (0.070) | 0.60 (0.60)  |          |      |           |           |
| 4196   | 0.49 (0.48)  | 6.28 (6.27)  | 220 (216)  | 181 (177) | 113 (109)  | 30.7 (28.0) |
| 8398   | 0.32 (0.31)  | 5.50 (5.50)  |          |      |           |           |

Table 2: CYC1: cyclegen 4196, ..., 16784. (Lemon: 22.5)

|        | IBFS        | BK       | gus       | gh   | ghs       | ghg       |
|--------|-------------|----------|-----------|------|-----------|-----------|
| 4196   | 0.014 (0.0071) | 0.010 (0.0049) | 2.97 (1.02) | 1.36 (0.0070) | 1.36 (0.0070) | 1.08 (0.0084) |
| 4195   | 0.022 (0.011)  | 0.011 (0.0051) |          |      |           |           |
| 4192   | 0.031 (0.016)  | 0.023 (0.013)  | 14.9 (4.39) | 8.45 (0.024) | 8.46 (0.023) | 6.51 (0.027) |
| 8392   | 0.030 (0.015)  | 0.024 (0.013)  |          |      |           |           |
| 8389   | 0.063 (0.032)  | 0.064 (0.042)  | 73.9 (23.1) | 39.5 (0.053) | 41.0 (0.051) | 30.9 (0.064) |
| 16784  | 0.063 (0.031)  | 0.065 (0.042)  |          |      |           |           |

classes (CYC1, NOI2, PR1, PR5, PR6). IBFS and ghs are the best on PATH. ghs is clear winner on TREE (followed by IBFS), IBFS is a clear winner on WHE, and BK is a clear winner on DBLCYC1 and TSP.

The implementation of Lemon does not appear to be competitive.

Results on unweighted simple graphs Here the comparison seems to be very clear: MG outperforms BK / IBFS on synthetic graphs in Table 20 and large social and web graphs in Table 22 while BK outperforms MG / IBFS on TSP instances in Table 21 (by large margins, especially when the graph sizes get bigger).

Observe that TSP instances have a large tree diameter (e.g. a hundred), while in the other two cases the diameter is relatively small (usually less than ten). This might be one possible reason for different behaviors.

4.4 Summary

Based on current results, we conclude that all three families of algorithms that we tested (the ones from [12], the ones from [5], and ours) have their merits: each one of them is preferable for certain classes of problems. The results also suggest natural directions for further improvements:

- Modify the data structures used in the codes of [12], and try to eliminate the overhead over maxflow computations.

- Extend the method in [5] to weighted graphs. Based on Table 20, it may be well be the case that such extension would outperform other algorithms on synthetic instances.

Acknowledgements

I thank Sagi Hed for kindly providing the IBFS code from [13].
Table 3: DBLCyc1: dblcyclegen 2048, ..., 8392. (Lemon: 4223).

| BR   | IBFS   | BK     | gus    | gh     | ghs    | ghp    |
|------|--------|--------|--------|--------|--------|--------|
| 2048 (5) | 0.069 (0.064) | 0.0047 (0.0033) | 28.7 (28.1) | 1.73 (1.39) | 0.65 (0.35) | 0.28 (0.075) |
| 4096 | 0.056 (0.051) | 0.0045 (0.0030) |        |        |        |        |
| 4196 (5) | 0.27 (0.26) | 0.010 (0.0074) | 206 (203) | 13.6 (11.0) | 4.33 (1.95) | 3.00 (0.83) |
| 8392 | 1.05 (1.00) | 0.041 (0.035) | 1775 (1761) | 74.9 (61.9) | 20.2 (8.68) | 11.5 (1.86) |

Table 4: NOI1: noigen 400 50 1 400, ..., 1000 50 1 1000. (Lemon: 3.53).

| BR   | IBFS   | BK     | gus    | gh     | ghs    | ghp    |
|------|--------|--------|--------|--------|--------|--------|
| 400 (2) | 0.71 (0.71) | 0.77 (0.76) | 0.55 (0.13) | 0.54 (0.16) | 0.59 (0.21) | 0.59 (0.32) |
| 39900 | 0.21 (0.21) | 0.43 (0.43) |        |        |        |        |
| 600 (2) | 7.41 (7.37) | 4.21 (4.18) | 1.82 (0.73) | 2.84 (0.86) | 3.13 (1.15) | 3.12 (1.68) |
| 89850 | 0.63 (0.62) | 1.43 (1.43) |        |        |        |        |
| 800 (2) | 27.7 (27.6) | 15.2 (15.1) | 4.72 (1.85) | 7.28 (2.15) | 8.01 (2.89) | 8.11 (4.39) |
| 159800 | 1.52 (1.49) | 3.45 (3.43) |        |        |        |        |
| 1000 (2) | 61.6 (61.4) | 36.1 (35.9) | 9.43 (3.61) | 14.5 (4.16) | 15.9 (5.65) | 16.5 (9.04) |
| 249750 | 3.07 (3.02) | 6.95 (6.92) |        |        |        |        |

Table 5: NOI2: noigen 400 50 2 400, ..., 1000 50 2 1000. (Lemon: 3.66).

| BR   | IBFS   | BK     | gus    | gh     | ghs    | ghp    |
|------|--------|--------|--------|--------|--------|--------|
| 400 (3) | 0.23 (0.22) | 0.16 (0.16) | 0.36 (0.15) | 0.18 (0.053) | 0.20 (0.071) | 0.21 (0.12) |
| 39900 | 0.074 (0.068) | 0.11 (0.10) |        |        |        |        |
| 600 (3) | 0.93 (0.89) | 0.66 (0.63) | 1.88 (0.79) | 0.82 (0.23) | 0.96 (0.33) | 1.00 (0.54) |
| 89850 | 0.22 (0.21) | 0.34 (0.33) |        |        |        |        |
| 800 (3) | 3.15 (3.05) | 1.73 (1.65) | 4.83 (1.90) | 2.79 (0.74) | 3.24 (1.11) | 3.24 (1.74) |
| 159800 | 0.52 (0.50) | 0.79 (0.78) |        |        |        |        |
| 1000 (3) | 8.40 (8.22) | 4.06 (3.90) | 9.64 (3.81) | 7.30 (1.92) | 8.34 (2.83) | 8.21 (4.40) |
| 249750 | 0.99 (0.95) | 1.63 (1.60) |        |        |        |        |

Table 6: NOI3: noigen 1000 d 1 1000, d=5,10,25,50,75,100. (Lemon: 3.88).

| BR   | IBFS   | BK     | gus    | gh     | ghs    | ghp    |
|------|--------|--------|--------|--------|--------|--------|
| 1000 (2) | 1.47 (1.46) | 0.63 (0.62) | 0.65 (0.27) | 1.09 (0.40) | 1.07 (0.39) | 1.18 (0.68) |
| 24975 | 0.51 (0.50) | 0.43 (0.43) |        |        |        |        |
| 1000 (2) | 3.54 (3.53) | 1.53 (1.52) | 1.48 (0.59) | 2.32 (0.76) | 2.45 (0.90) | 2.78 (1.59) |
| 49950 | 0.91 (0.90) | 0.86 (0.85) |        |        |        |        |
| 1000 (2) | 27.2 (27.1) | 9.46 (9.39) | 4.57 (1.87) | 7.23 (2.23) | 7.93 (2.92) | 8.48 (4.82) |
| 124875 | 1.88 (1.85) | 2.49 (2.47) |        |        |        |        |
| 1000 (2) | 61.3 (61.1) | 36.1 (35.9) | 9.42 (3.61) | 14.4 (4.13) | 16.0 (5.69) | 16.5 (8.99) |
| 249750 | 2.86 (2.82) | 6.91 (6.87) |        |        |        |        |
| 1000 (2) | 90.9 (90.7) | 69.5 (69.2) | 13.9 (5.19) | 21.5 (5.97) | 24.1 (8.52) | 24.8 (13.5) |
| 374625 | 3.72 (3.65) | 12.4 (12.3) |        |        |        |        |
| 1000 (2) | 122 (121) | 102 (102) | 18.6 (6.76) | 28.7 (7.71) | 32.1 (11.2) | 30.9 (15.6) |
| 499500 | 4.90 (4.81) | 17.8 (17.8) |        |        |        |        |
|            | IBFS       | BK         | gus        | gh         | ghs        | ghg        |
|------------|------------|------------|------------|------------|------------|------------|
| 1000 (3)   | 0.41 (0.40)| 0.21 (0.20)| 0.71 (0.32)| 0.47 (0.18)| 0.45 (0.17)| 0.55 (0.35)|
| 24975      | 0.17 (0.16)| 0.17 (0.16)| 0.76 (0.25)| 0.80 (0.29)| 0.90 (0.53)|
| 1000 (3)   | 0.97 (0.95)| 0.37 (0.35)| 1.53 (0.65)| 2.53 (0.70)| 2.81 (0.97)| 2.81 (1.52)|
| 49950      | 0.31 (0.30)| 0.24 (0.24)| 0.17 (0.16)| 0.17 (0.16)|
| 1000 (3)   | 2.70 (2.63)| 1.27 (1.21)| 4.71 (1.99)| 2.81 (0.97)|
| 124875     | 0.55 (0.52)| 0.66 (0.64)| 0.71 (0.32)|
| 1000 (3)   | 9.80 (9.61)| 4.19 (4.03)| 9.64 (3.82)| 8.32 (2.82)| 8.26 (4.43)|
| 249750     | 0.99 (0.95)| 1.62 (1.59)| 7.29 (1.92)|
| 1000 (3)   | 19.0 (18.7)| 8.85 (8.59)| 14.1 (5.44)| 13.8 (7.04)|
| 374625     | 1.43 (1.37)| 2.49 (2.44)| 12.8 (3.13)|
| 1000 (3)   | 25.8 (25.3)| 15.8 (15.4)| 18.4 (4.27)| 21.3 (6.90)| 19.4 (9.44)|
| 499500     | 1.79 (1.70)| 3.72 (3.65)|

Table 7: NOI4: noigen 1000 d 2 1000, d=5,10,25,50,75,100. (Lemon: 4.34).

|            | IBFS       | BK         | gus        | gh         | ghs        | ghg        |
|------------|------------|------------|------------|------------|------------|------------|
| 1000 (2)   | 62.5 (62.4)| 36.3 (36.1)| 9.48 (3.63)| 14.5 (4.15)|
| 249750     | 3.47 (3.42)| 6.94 (6.91)| 15.9 (5.67)| 16.5 (9.03)|
| 1000 (3)   | 8.26 (8.08)| 4.18 (4.02)| 6.95 (3.81)| 7.28 (1.91)| 8.41 (2.85)| 8.22 (4.41)|
| 249750     | 0.92 (0.88)| 1.63 (1.59)| 9.98 (4.13)| 0.91 (0.26)| 1.61 (0.60)| 1.48 (0.80)|
| 1000 (4)   | 1.23 (1.02)| 2.89 (2.70)| 1.31 (1.28)| 10.4 (4.56)| 0.51 (0.22)| 0.78 (0.32)| 0.94 (0.54)|
| 249750     | 0.25 (0.22)| 2.05 (2.02)|
| 1000 (5.6) | 1.52 (1.21)| 4.93 (4.67)| 11.9 (5.99)| 1.14 (0.64)| 0.69 (0.31)| 1.14 (0.78)|
| 249750     | 0.19 (0.15)| 2.10 (2.02)|
| 1000 (6.4) | 4.40 (3.17)| 8.63 (7.96)| 13.2 (7.25)| 2.42 (1.43)| 0.84 (0.41)| 1.77 (1.30)|
| 249750     | 0.32 (0.24)| 2.51 (2.45)|
| 1000 (8)   | 5.87 (3.21)| 11.9 (10.6)| 14.2 (8.27)| 7.08 (4.08)| 1.56 (0.71)| 3.76 (2.84)|
| 249750     | 0.36 (0.21)| 3.07 (2.97)|
| 1000 (8)   | 13.3 (6.24)| 20.8 (17.8)| 12.8 (6.84)| 9.57 (3.63)| 15.3 (10.4)|
| 249750     | 0.72 (0.39)| 4.41 (4.22)|
| 1000 (6.4) | 19.4 (9.16)| 28.7 (25.2)| 11.3 (5.40)| 17.7 (7.80)| 21.8 (7.99)| 30.6 (20.7)|
| 249750     | 1.19 (0.60)| 5.44 (5.20)|
| 1000 (6.2) | 20.3 (10.2)| 30.1 (26.6)| 11.1 (5.25)| 17.7 (7.56)| 23.2 (8.49)| 34.7 (24.2)|
| 249750     | 1.34 (0.70)| 5.68 (5.45)|

Table 8: NOI5: noigen 1000 50 k 1000, k=1,2,5,10,30,50,100,200,400,500. (Lemon: 67.0).
| IBFS | BK  | gus  | gh   | ghs  | ghg  |
|------|-----|------|------|------|------|
| 1000 (2) 249750 | 61.3 (61.1) | 36.3 (36.2) | 9.51 (3.66) | 14.4 (4.15) | 16.0 (5.75) | 16.5 (9.01) |
| 1000 (2) 249750 | 57.5 (57.3) | 37.8 (37.7) | 9.62 (3.78) | 14.7 (4.38) | 16.2 (6.24) | 19.1 (11.5) |
| 1000 (2) 249750 | 61.5 (61.3) | 49.7 (49.5) | 9.87 (4.03) | 15.2 (4.90) | 16.6 (6.08) | 19.2 (11.7) |
| 1000 (2) 249750 | 61.5 (61.3) | 57.5 (57.3) | 9.62 (3.66) | 14.7 (4.38) | 16.2 (6.24) | 20.2 (12.6) |
| 1000 (2) 249750 | 57.5 (57.3) | 49.7 (49.5) | 9.87 (4.03) | 15.2 (4.90) | 16.6 (6.08) | 20.2 (12.6) |
| 1000 (2) 249750 | 61.5 (61.3) | 57.5 (57.3) | 9.62 (3.66) | 14.7 (4.38) | 16.2 (6.08) | 20.2 (12.6) |
| 1000 (2) 249750 | 61.5 (61.3) | 49.7 (49.5) | 9.87 (4.03) | 15.2 (4.90) | 16.6 (6.08) | 19.2 (11.7) |

Table 9: NOI6: noigen 1000 50 2 P, P=1,10,50,100,150,200,500,1000,5000. (Lemon: 66.6).
|          | IBFS          | BK            | gus            | gh             | ghs            | ghg            |
|----------|---------------|---------------|----------------|----------------|----------------|----------------|
| 400 (5) | 0.014 (0.013) | 0.012 (0.011)| 0.041 (0.022)  | 0.030 (0.014)  | 0.023 (0.0087) | 0.034 (0.023)  |
| 400 (3) | 0.099 (0.095) | 0.070 (0.067) | 0.22 (0.11)    | 0.17 (0.074)   | 0.15 (0.057)   | 0.20 (0.13)    |
| 1200 (3)| 0.31 (0.30)   | 0.20 (0.20)   | 0.64 (0.31)    | 0.47 (0.20)    | 0.41 (0.16)    | 0.54 (0.35)    |
| 2000 (3)| 1.42 (1.40)   | 0.85 (0.83)   | 3.75 (1.73)    | 2.17 (0.79)    | 1.93 (0.68)    | 2.10 (1.23)    |
| Table 12: PR5: prgen 400 2 2, ..., 2000 2 2. (Lemon: 0.21). |
|          | IBFS          | BK            | gus            | gh             | ghs            | ghg            |
| 400 (3) | 0.044 (0.042) | 0.032 (0.030) | 0.094 (0.043)  | 0.061 (0.023)  | 0.059 (0.022)  | 0.070 (0.043)  |
| 800 (3) | 0.45 (0.44)   | 0.29 (0.28)   | 0.69 (0.29)    | 0.41 (0.14)    | 0.42 (0.16)    | 0.48 (0.28)    |
| 1200 (3)| 1.45 (1.42)   | 0.81 (0.79)   | 3.23 (1.40)    | 1.56 (0.48)    | 1.69 (0.58)    | 1.73 (0.97)    |
| 2000 (3)| 8.93 (8.80)   | 4.26 (4.14)   | 16.8 (7.12)    | 11.7 (3.50)    | 12.9 (4.58)    | 14.6 (8.38)    |
| Table 13: PR6: prgen 400 10 2, ..., 2000 10 2. (Lemon: 0.67). |
|          | IBFS          | BK            | gus            | gh             | ghs            | ghg            |
| 400 (3) | 0.24 (0.23)   | 0.26 (0.25)   | 0.43 (0.18)    | 0.23 (0.066)   | 0.24 (0.084)   | 0.24 (0.13)    |
| 800 (3) | 3.04 (2.97)   | 3.11 (3.04)   | 2.61 (2.10)    | 3.16 (0.89)    | 3.51 (1.24)    | 3.53 (1.89)    |
| 1200 (3)| 14.2 (14.0)   | 10.3 (10.1)   | 17.6 (7.01)    | 12.8 (3.43)    | 14.3 (4.94)    | 14.1 (7.34)    |
| 2000 (3)| 51.8 (51.3)   | 39.6 (39.2)   | 42.2 (16.9)    | 30.0 (7.85)    | 34.3 (11.7)    | 33.9 (17.5)    |
| Table 14: PR7: prgen 400 50 2, ..., 2000 50 2. (Lemon: 4.29). |
|          | IBFS          | BK            | gus            | gh             | ghs            | ghg            |
| 400 (3) | 1.02 (1.00)   | 1.18 (1.16)   | 1.07 (0.43)    | 0.48 (0.14)    | 0.52 (0.19)    | 0.50 (0.26)    |
| 800 (3) | 12.1 (12.0)   | 15.1 (15.0)   | 9.92 (3.90)    | 6.89 (1.83)    | 7.76 (2.66)    | 7.33 (3.71)    |
| 1200 (3)| 51.0 (50.7)   | 65.4 (65.0)   | 34.2 (13.5)    | 24.9 (6.43)    | 28.2 (9.71)    | 26.8 (13.5)    |
| 2000 (3)| 214 (212)     | 343 (341)     | 173 (68.8)     | 121 (30.9)     | 138 (47.2)     | 133 (67.4)     |
| Table 15: PR8: prgen 400 100 2, ..., 2000 100 2. (Lemon: 12.9). |
| k     | IBFS         | BK          | gus        | gh         | ghs        | ghg        |
|-------|--------------|-------------|------------|------------|------------|------------|
| 1000  | 0.0009 (0.0003) | 0.0008 (0.0003) | 0.089 (0.031) | 0.14 (0.054) | 0.12 (0.039) | 0.15 (0.059) |
| 1000  | 0.0010 (0.0003) | 0.0008 (0.0003) | 0.19 (0.11) | 0.29 (0.16) | 0.29 (0.17) | 0.25 (0.16) |
| 1000  | 0.59 (0.59) | 0.36 (0.36) | 0.46 (0.21) | 0.71 (0.28) | 0.71 (0.32) | 0.59 (0.34) |
| 1000  | 2.69 (2.67) | 2.69 (2.67) | 2.13 (0.89) | 3.33 (1.06) | 3.67 (1.45) | 4.04 (2.57) |
| 1000  | 14.0 (13.9) | 11.9 (11.8) | 4.77 (1.99) | 7.13 (2.18) | 8.03 (3.09) | 13.6 (9.16) |

Table 16: REG1: regulargen 1000 k -w1000, k=1,4,16,64. (Lemon: 0.82).

| k     | IBFS         | BK          | gus        | gh         | ghs        | ghg        |
|-------|--------------|-------------|------------|------------|------------|------------|
| 800   | 2.02 (1.93)  | 6.06 (5.97)  | 3.17 (2.22) | 7.71 (2.44) | 6.48 (1.22) | 7.98 (4.16) |
| 800   | 0.17 (0.14)  | 1.49 (1.47)  |            |            |            |            |
| 800   | 0.84 (0.73)  | 2.18 (2.10)  | 3.63 (0.74) | 8.16 (3.06) | 3.26 (0.65) | 3.02 (1.19) |
| 800   | 0.998 (0.073) | 0.61 (0.59)  |            |            |            |            |
| 800   | 0.57 (0.47)  | 2.35 (2.27)  | 3.89 (1.00) | 8.82 (3.60) | 1.46 (0.31) | 1.33 (0.53) |
| 800   | 0.081 (0.059) | 0.79 (0.77)  |            |            |            |            |
| 800   | 0.37 (0.24)  | 3.11 (3.05)  | 4.22 (1.31) | 9.19 (4.08) | 0.64 (0.14) | 0.61 (0.26) |
| 800   | 0.053 (0.031) | 0.82 (0.80)  |            |            |            |            |
| 800   | 0.40 (0.20)  | 3.32 (3.25)  | 4.48 (1.60) | 9.36 (4.30) | 0.43 (0.11) | 0.45 (0.21) |
| 800   | 0.051 (0.029) | 0.95 (0.93)  |            |            |            |            |
| 800   | 0.80 (0.36)  | 4.99 (4.91)  | 4.81 (1.87) | 9.85 (4.84) | 0.56 (0.16) | 0.72 (0.38) |
| 800   | 0.073 (0.039) | 1.36 (1.34)  |            |            |            |            |
| 800   | 1.53 (0.69)  | 7.38 (7.25)  | 5.24 (2.14) | 11.8 (5.93) | 1.14 (0.36) | 1.56 (0.90) |
| 800   | 0.12 (0.073) | 2.11 (2.08)  |            |            |            |            |
| 800   | 3.18 (1.54)  | 9.41 (9.14)  | 5.56 (2.43) | 11.2 (5.76) | 1.61 (0.58) | 2.98 (2.02) |
| 800   | 0.25 (0.16)  | 2.78 (2.74)  |            |            |            |            |
| 800   | 5.92 (2.89)  | 12.7 (12.1)  | 5.65 (2.66) | 11.0 (5.80) | 3.00 (1.08) | 5.64 (4.11) |
| 800   | 0.37 (0.23)  | 3.64 (3.59)  |            |            |            |            |

Table 17: TREE: treegen 800 50 k 1000, k=1,3,5,10,20,50,100,200,400. (Lemon: 28.9).

| k     | IBFS         | BK          | gus        | gh         | ghs        | ghg        |
|-------|--------------|-------------|------------|------------|------------|------------|
| 1024  | 0.038 (0.037) | 0.099 (0.099) | 3.29 (3.19) | 3.47 (3.33) | 2.05 (1.92) | 0.16 (0.084) |
| 2046  | 0.029 (0.027) | 0.096 (0.095) |            |            |            |            |
| 2046  | 0.16 (0.15)  | 0.83 (0.83)  | 26.6 (26.0) | 27.0 (26.2) | 13.6 (12.8) | 0.91 (0.43) |
| 4094  | 0.11 (0.11)  | 0.82 (0.82)  |            |            |            |            |
| 4196  | 0.68 (0.67)  | 6.82 (6.81)  | 207 (204)  | 176 (172)  | 101 (96.6) | 4.54 (1.93) |
| 8390  | 0.49 (0.49)  | 6.71 (6.71)  |            |            |            |            |

Table 18: WHE: wheelgen 1024, ..., 4196. (Lemon: 274).
|          | IBFS          | BK            | gus      | gh       | ghg      |
|----------|---------------|---------------|----------|----------|----------|
| rl5934   |               |               |          |          |          |
|          | 5933 (117-127)| 0.037 (0.013) | 0.028 (0.0071) | 6.19 (0.69) | 3.98 (0.054) | 2.67 (0.042) |
|          | 11868         | 0.036 (0.012) | 0.027 (0.0068) |          |          |          |
|          | 5934 (72)     | 0.28 (0.20)   | 0.061 (0.046) | 12.4 (5.11) | 4.47 (0.84) | 2.63 (0.49) |
|          | 23736         | 0.21 (0.19)   | 0.053 (0.040) |          |          |          |
|          | 5934 (18)     | 4.21 (4.16)   | 0.25 (0.23)   | 24.4 (13.7) | 28.6 (15.3) | 33.1 (24.0) |
|          | 47472         | 2.22 (2.18)   | 0.20 (0.18)   |          |          |          |
| rl11849  |               |               |          |          |          |
|          | 11846 (203-204)| 0.088 (0.040) | 0.060 (0.017) | 32.6 (4.07) | 18.9 (0.16) | 13.7 (0.13) |
|          | 23698         | 0.093 (0.043) | 0.056 (0.016) |          |          |          |
|          | 11845 (69-92) | 0.87 (0.73)   | 0.17 (0.13)   | 56.2 (22.9) | 26.9 (6.74) | 16.8 (5.05) |
|          | 47396         | 38.6 (38.3)   | 0.57 (0.50)   |          |          |          |
|          | 94792         | 11.9 (11.8)   | 0.41 (0.38)   |          |          |          |
| usa13509 |               |               |          |          |          |
|          | 13504 (73-80) | 0.68 (0.15)   | 0.56 (0.035) | 43.7 (4.80) | 30.5 (2.30) | 38.4 (2.14) |
|          | 27018         | 0.67 (0.13)   | 0.58 (0.037) |          |          |          |
|          | 13499 (88-93) | 0.80 (0.63)   | 0.28 (0.13)   | 54.3 (9.04) | 27.4 (1.36) | 21.3 (1.55) |
|          | 54036         | 0.48 (0.32)   | 0.27 (0.11)   |          |          |          |
|          | 13492 (62-66) | 4.55 (4.42)   | 0.56 (0.47)   |          |          |          |
|          | 108072        | 1.46 (1.38)   | 0.40 (0.35)   |          |          |          |
| brd14051 |               |               |          |          |          |
|          | 14049 (173-183)| 0.49 (0.35)   | 0.18 (0.052) | 50.9 (10.8) | 30.5 (1.83) | 23.0 (1.44) |
|          | 28102         | 0.43 (0.28)   | 0.18 (0.047) |          |          |          |
|          | 14051 (79)    | 3.33 (3.21)   | 0.32 (0.26)   | 105 (57.5) | 82.6 (41.1) | 33.3 (14.5) |
|          | 56204         | 1.97 (1.8)    | 0.27 (0.23)   |          |          |          |
|          | 14051 (18-20) | 29.6 (29.3)   | 0.86 (0.78)   |          |          |          |
|          | 112408        | 10.6 (10.4)   | 0.66 (0.62)   |          |          |          |
| d18512   |               |               |          |          |          |
|          | 18512 (174-198)| 0.74 (0.55)   | 0.25 (0.072) | 95.9 (22.4) | 53.7 (2.53) | 40.9 (1.95) |
|          | 37024         | 0.58 (0.39)   | 0.24 (0.066) |          |          |          |
|          | 18510 (50-72) | 7.00 (6.7)    | 0.54 (0.42)   | 213 (126)  | 190 (103)  | 114 (70.7) |
|          | 74048         | 3.78 (3.59)   | 0.43 (0.36)   |          |          |          |
|          | 18510 (15)    | 56.4 (55.9)   | 1.23 (1.13)   | 360 (240)  | 676 (383)  | 843 (671)  |
|          | 148096        | 20.9 (20.6)   | 0.94 (0.89)   |          |          |          |
| pla33810 |               |               |          |          |          |
|          | 33678 (553-558)| 0.98 (0.88)   | 0.21 (0.13)   | 349 (91.6) | 196 (17.3) | 134 (4.03) |
|          | 67620         | 0.80 (0.70)   | 0.19 (0.12)   |          |          |          |
|          | 33678 (205-214)| 5.87 (5.73)   | 0.44 (0.36)   | 424 (110)  | 255 (77.3) | 143 (29.9) |
|          | 135240        | 4.12 (3.99)   | 0.39 (0.33)   |          |          |          |
|          | 270480        | 14.5 (14.3)   | 0.97 (0.84)   |          |          |          |
| pla85900 |               |               |          |          |          |
|          | 85895 (730-743)| 7.16 (6.66)   | 0.79 (0.58)   | 2491 (719) | 1364 (211) | 883 (57.6) |
|          | 171800        | 5.96 (5.50)   | 0.76 (0.55)   |          |          |          |
|          | 85900 (596-687)| 29.0 (28.5)   | 1.56 (1.31)   | 3005 (859) | 1843 (651) | 965 (240)  |
|          | 343600        | 20.3 (19.8)   | 1.32 (1.14)   |          |          |          |
|          | 85900 (629-633)| 98.6 (97.6)   | 3.14 (2.77)   | 3816 (985) | 2460 (1142)| 1184 (418) |
|          | 687200        | 49.8 (49.0)   | 2.78 (2.54)   |          |          |          |

Table 19: TSP instances (for average degree \( k = 2, 4, 8 \)). (Lemon: 6.37).
## Table 20: Unweighted simple graphs: synthetic instances. Each line corresponds to the last instance in tables [1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1][1]
A Correctness of Algorithm 2: proof of Theorems 2 and 3

First, we state two well-known facts about undirected weighted graphs (see e.g. Lemmas 8.33 and 8.35 in [20]).

Lemma 4. For any sequence of distinct $k \geq 2$ vertices $v_1, \ldots, v_k$ there holds

$$f_G(v_1, v_k) \geq \min_{1 \leq i < k} f_G(v_i, v_{i+1})$$

Lemma 5. Let $s, t$ be distinct nodes in $G$ and $U$ be a minimum $u-v$ cut in $G$ for some $u, v$. Then there exists a minimum $s$-$t$ cut $W$ with $W \subseteq U$.

Corollary 6. If $T$ is consistent with $G$ and $X$ is a supernode of $T$ then $f_H(s, t) = f_G(s, t)$ for any distinct $x, y \in X$, where $H = G[X, T]$.

Proof. Consider edge $XY$ in $T$, and let $S$ be the cut $C_T(X, Y)$ with $X \subseteq S$ (then $Y \subseteq V - S$). By Lemma 5 the cost of a minimum $s$-$t$ in $G$ will not change if $V - S$ is contracted to a single node. Applying this argument inductively for all edges $XY$ of $T$ gives the claim.

Lemma 7. Suppose that $T$ is consistent with $G$ and $(G', T') = \text{MinCutStep}(G, T)$. Then $T'$ is consistent with $G'$.

Proof. First, suppose that $\text{MinCutStep}(G, T)$ uses lines 6-8 and so $G = G'$. We repeat the argument given in [20] Section 8.6. For brevity, we write $f(u, v)$ instead of $f_G(u, v)$. We also treat $S$ as a cut in $G$, not in $H$ (with $A \subseteq S$ and $B \subseteq S^\text{def} = V - S$). By Corollary 6 $S$ is a minimum $s$-$t$ cut in $G$.

For supernodes $A, B$ of $T'$ the consistency condition is straightforward: $s \in \hat{A}$ and $t \in \hat{B}$ is a desired pair of nodes with $f(s, t) = f_{T'}(A, B) = \text{cost}_G(S)$. For edges of $T'$ that do not involve $A, B$ the claim is trivial. By symmetry, it remains to consider edge $AY$ of $T'$ with $Y \neq B$. Note that $Y \subseteq S$. Since $T$ is consistent with $G$, there exists $p \in \hat{X}, q \in \hat{Y}$ with $f(p, q) = f_T(X, Y)$. If $p \in A$ then the claim trivially holds. Let us thus assume that $p \in B$. It suffices to prove that $f(p, q) = f(s, q)$, then $s \in \hat{A}$ and $q \in \hat{Y}$ would be a desired pair of nodes (note that $f_T(X, Y) = f_{T'}(A, Y)$). Let $G$ be the graph obtained from $G$ by adding edge $pt$ with a sufficiently large weight. By Lemma 4 there exists a minimum $s$-$q$ cut in $G$ that does not separate $p, t$, and so

$$f(s, q) = f_G(s, q) \geq \min\{f_G(s, t), f_G(t, p), f_G(p, q)\} = \min\{f_G(s, t), f_G(p, q)\} \geq \min\{f(s, t), f(p, q)\}$$

where the first inequality holds by Lemma 4. Since the minimum $s$-$t$ cut $S$ separates $p$ and $q$, we have $f(s, t) \geq f(p, q)$. This implies that $f(s, q) \geq f(p, q)$. Since $C_T(X, Y)$ is an $s$-$q$ cut, we also have $f(s, q) \leq f_T(X, Y) = f(p, q)$. This shows that $f(s, q) = f(p, q)$, as claimed.

Now suppose that $\text{MinCutStep}(G, T)$ uses lines 10-11. Consider edge $Y'Z'$ of $T'$ and the corresponding edge $YZ$ of $T$. Since $T$ is consistent with $G$, there exists $y \in \hat{Y}, z \in \hat{Z}$ with $f_G(y, z) = f_T(Y, Z)$. We can assume w.l.o.g. that $y \neq t$ and $z \neq t$ (by swapping $s$ and $t$, if

| Graph       | $n$ (tree diameter), $m$ | IBFS    | BK       | MG       |
|-------------|--------------------------|---------|----------|----------|
| wiki-Vote   | 4786 (2), 98456          | 0.62 (0.58) | 2.08 (2.05) | 0.13     |
| ca-CondMat  | 19378 (10), 89128        | 3.85 (3.69) | 1.04 (0.96) | 0.19     |
| facebook    | 3964 (10), 88159         | 0.20 (0.18) | 0.47 (0.45) | 0.13     |
| com-dblp    | 264341 (15), 991125      | 700 (664) | 52.9 (36.9) | 7.96     |
| soc-Epinions1 | 36957 (8-9), 366634   | 9.76 (8.73) | 38.9 (38.0) | 0.70     |
| soc-Slashdot0811 | 47603 (8), 439348 | 24.1 (22.7) | 35.0 (34.0) | 0.85     |
| twitter     | 76310 (7), 1337300       | 59.8 (57.2) | 89.2 (88.0) | 4.07     |

Table 22: Unweighted simple graphs: social and web graphs.
necessary. We have $f_{G'}(y, z) \geq f_G(y, z)$ since contraction cannot decrease the cost of minimum cuts, and $f_{G'}(y, z) \leq f_T(Y'Z') = f_T(YZ)$ since $C_T(Y'Z')$ is a $y$-$z$ cut of cost $f_T(YZ)$. This proves the claim.

Let us consider a relaxed version of Algorithm 2 where CutTree($G, T$) may optionally return input tree $T$ before step 1 is reached. Its run for input $(G, T)$ can be described by a sequence $\Lambda = (\Lambda_0, \ldots, \Lambda_\ell)$ with $\Lambda_i = (G_i, T_i, T_i^*)$, $\ell \geq 0$ where $(G_0, T_0) = (G, T^{\text{empty}})$ is the input, $(G_{i+1}, T_{i+1})$ is the result of applying procedure MinCutStep to $(G_i, T_i)$, and $T_i^*$ is the output of CutTree$(G_i, T_i)$. Any such sequence that can be obtained by a run of relaxed Algorithm 2 will be called realizable. For a realizable $\Lambda$ let $\#\Lambda$ be the total number of “merging” steps (lines 10-11 of Alg. 1) or equivalently lines 5 of Alg. 2.

Lemma 8. Consider realizable sequence $\Lambda$ with $\Lambda_0 = (G_0, T_0, T_0^*)$ where $T_0$ is consistent with $G_0$. There exists another realizable sequence $\Lambda'$ with $\#\Lambda' = 0$ and $\Lambda_0' = (G_0, T_0, T_0^*)$.

Proof. It suffices to prove the claim when $\#\Lambda = 1$ and merging is done in the first call MinCutStep($G_0, T_0$); the general case will then easily follow by Lemma 7 and an induction argument (by considering the rightmost suffix containing the merging step). Accordingly, we assume that $\Lambda = (((G, \hat{T}, \hat{T}^*), (G, T_1, T_1^*), \ldots), (G, T_\ell, T_\ell^*))$ where $G$ is a graph on nodes $V$ and $G$ is the graph on nodes $V = V - \{t\}$ obtained from $G$ by merging $s$ and $t$. (We use “hat” for objects in which $s, t$ are not merged, and “no hat” for objects in which $s, t$ are merged). For $i \in [\ell]$ let $X_i$ be the supernode of $T_i$ containing $s$, and denote $X_i = X_i \cup \{t\}$ and $T_i = T_i[\{X_i \mapsto X_i\}]$. Note that we have $T_0 = T, T_\ell = T^*$ and $T_i = T_i^*$. It remains to show that sequence $((G, \hat{T}_1, \hat{T}_1^*), \ldots, (G, \hat{T}_\ell, \hat{T}_\ell^*))$ is realizable, or more specifically that $(G, \hat{T}_{i+1})$ is a valid output of MinCutTree($G, \hat{T}_i$) for $i \in [\ell - 1]$.

Let $\lambda$ and $\lambda_i$ be the weight functions of $T$ and $T_i$ respectively. By assumption, MinCutTree($G, \hat{T}$) uses lines 10-11. From Corollary 6 we conclude that $f_G(s, t) > \lambda(X_1)$. By inspecting the algorithm we conclude that $\lambda(X_0) \geq \lambda_i(X_i)$, and so $f_G(s, t) \geq \lambda_i(X_i)$ for all $i \in [\ell]$.

Let $X, s_i, t_i$ be the supernode of $T_i$ and the nodes chosen in line 1 of MinCutStep($G, T_i$), and let $\hat{X}$ be the supernode of $\hat{T}_i$ corresponding to $X$. Our goal is to show that MinCutTree($G, \hat{T}_i$) can produce $(G, \hat{T}_{i+1})$ if we use choices $\hat{X}$ and $s_i, t_i$ in line 1. Denote $H = G[T_i, X]$ and $\hat{H} = G[\hat{T}_i, \hat{X}]$. If $X \neq X_i$ then $X = \hat{X}, H = \hat{H}$, and the claim holds. Suppose that $X = X_i$ and thus $\hat{X} = X \cup \{t\}$. Let $S$ be a minimum $s_i$-$t_i$ cut in $H$, and $\hat{S}$ be the cut of $\hat{H}$ obtained from $S$ by assigning $t$ to the same component as $s$. We have $\text{cost}_H(S) = \text{cost}_H(S) \leq \lambda_i(X_i) \leq f_G(s, t)$. We also know that $S$ is a minimum $s_i$-$t_i$ cut in $H$ and thus $\hat{S}$ is a minimum $s_i$-$t_i$ cut in $\hat{H}$ among all cuts in which $s, t$ are not separated. Therefore, $\hat{S}$ is a minimum $s_i$-$t_i$ cut in $\hat{H}$, which implies the claim. \hfill $\square$

Lemmas 7 and 8 imply parts (a) and (b) of Theorem 2. To show part (c), let us introduce the following definition. We say that $T$ lower-bounds $G$ if for every supernode $X$ of $T$ and every $s, t \in X$ there holds $f_G > \lambda(X)$.

Lemma 9. If $T$ is consistent with $G$ and $T^* = \text{CutTree}(G, T)$ then $T^*$ lower-bounds $G$.

Proof. We use induction on the depth of the recursion. The depth will be zero if CutTree($G, T$) terminates in line 1, i.e. if $|X| = 1$ for all supernodes; then the claim is trivial. Suppose the depth is positive. Let $(G', T')$ be the output of MinCutStep($G, T'$) in line 2, and $T''$ be the output of CutTree($G', T'$). Let $\lambda, \lambda', \lambda''$ be the weight functions of $T, T', T''$ respectively. By the induction hypothesis, $T''$ lower-bounds $G'$. If $G' = G$ then $T^* = T''$ and the claim holds. Suppose therefore that $G'$ was obtained from $G$ by contracting nodes $s, t \in X$ in lines 10-11 of MinCutStep($G, T'$). Note that $T'' = T''[X'\mapsto X']$ where $X'$ is the supernode of $T''$ containing $s$, and $X'' = X'\cup\{t\}$. Consider supernode $U$ of $T^*$ (corresponding to supernode $U'$ of $T''$) and nodes $u, v \in U$. We need to show that $f_G(u, v) > \lambda(U)$. Note, if $u \neq t$ and $v \neq t$ then $f_{G'}(u, v) > \lambda''(U') = \lambda''(U)$ since $T''$ lower-bounds $G'$.\hfill $\square$
First, suppose that \( U \neq X^* \). Denote \( H = G[T^*, U] \) and \( H' = G'[T'^*, U] \). Clearly, we have \( H = H' \), and so Corollary 2 gives \( f_G(u, v) = f_H(u, v) = f_{H'}(u, v) = f_{G'}(u, v) > \lambda^*(U) \). Now suppose that \( U = X^* \), and consider a minimum \( u-v \) cut \( S \) in \( G \) with \( \text{cost}_G(S) = f_G(u, v) \). Two cases are possible.

- \( S \) is an \( s-t \) cut. Since \( \text{MinCutStep}(G, T) \) used lines 10-11, we must have \( f_G(u, v) = \text{cost}_G(S) > \lambda(X) \). By inspecting the algorithm we conclude that \( \lambda(X) \geq \lambda^*(X^*) \), and so \( f_G(u, v) > \lambda^*(X^*) \).

- \( S \) is not an \( s-t \) cut. Let \( S' \) be the cut of \( G' \) obtained from \( S \) by merging \( s \) and \( t \), then \( f_G(u, v) = \text{cost}_G(S) = \text{cost}_{G'}(S') > \lambda^*(U') = \lambda^*(U) \) where the inequality holds since \( T'^* \) lower-bounds \( G' \).

To conclude this section, we prove Theorem 3. Suppose that step 2 of Alg. 1 (decreasing \( \lambda(X) \)) is never invoked. By construction, all partitions \( X \) during the algorithm satisfy \( \lambda(X) = k \), and the weight of edges \( YZ \) of partitions trees is at most \( k \). The claim now follows from Theorem 2 and the result below.

**Lemma 10.** Suppose that \( T \) is consistent with \( G \) and lower bounds \( G \), \( \lambda(X) = k \) for all supernodes of \( T \), and the weight of all edges of \( T \) is at most \( k \). Then \( T \) is a \( k \)-partial tree for \( G \).

**Proof.** For brevity, we write \( f(x, y) \) instead of \( f_G(x, y) \). Consider \( x, y \in V \) with \( f_G(x, y) \leq k \). Nodes \( x, y \) must belong to distinct supernodes of \( T \) (\( x \in X \), \( y \in Y \)) since \( T \) lower-bounds \( G \). Let \( X = X_1, X_2, \ldots, X_\ell = Y \) be the unique path in \( T \) from \( X \) to \( Y \). By assumption, for every \( i \in [\ell - 1] \) there exists pair \( p_i \in X_i, q_{i+1} \in X_{i+1} \) with \( f(p_i, q_{i+1}) = f_T(X_i, X_{i+1}) \). Denote \( q_1 = x \) and \( p_\ell = y \), then by Lemma 4

\[
f(x, y) \geq \min\{A, B\} \quad \text{where} \quad A = \min_{i \in [\ell]} f(p_i, q_i), \quad B = \min_{i \in [\ell-1]} f(p_i, q_{i+1})
\]

By assumptions we have \( A > k \) and \( B \leq k \), therefore \( f(x, y) \geq B = f_T(X, Y) \). Furthermore, there exists an \( x-y \) cut of cost \( f_T(X, Y) \) (it is defined by the minimum-weight edge on the \( X-Y \) path in \( T \)). Therefore, \( f(x, y) = f_T(X, Y) \), as desired.

**References**

[1] TSPLIB dataset. [http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html](http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html)

[2] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. APMF ? APSP? Gomory-Hu tree for unweighted graphs in almost-quadratic time. In *FOCS*, pages 1135–1146, 2021.

[3] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Subcubic algorithms for Gomory-Hu tree in unweighted graphs. In *STOC*, pages 1725–1737, 2021. Full version at arXiv:2012.10281.

[4] Amir Abboud, Robert Krauthgamer, and Ohad Trabelsi. Friendly cut sparsifiers and faster Gomory-Hu trees. In *SODA*, pages 3630–3649, 2022.

[5] Takuya Akiba, Yoichi Iwata, Yosuke Sameshima, Naoto Mizuno, and Yosuke Yano. Cut tree construction from massive graphs. In *Proceedings of the IEEE 16th International Conference on Data Mining (ICDM)*, pages 775–780, 2016.
[6] Karteek Alahari, Pushmeet Kohli, and P. H. S. Torr. Dynamic hybrid algorithms for MAP inference in discrete MRFs. *PAMI*, 32(10):1846–1857, 2010.

[7] A. Bhalgat, R. Hariharan, T. Kavitha, and D. Panigrahi. An o(mn) Gomory-Hu tree construction algorithm for unweighted graphs. In *STOC*, pages 605–614, 2007.

[8] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *PAMI*, 26(9), September 2004.

[9] B. V. Cherkassky and A. V. Goldberg. On implementing push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[10] B. Dezs, A. Jüttner, and P. Kovács. Lemon - an open source C++ graph template library. *Electron. Notes Theor. Comput. Sci.*, 264(5):23–45, 2011.

[11] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum flow problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[12] A. V. Goldberg and K. Tsioutsiouliklis. Cut tree algorithms: an experimental study. *Journal of Algorithms*, 38(1):51–83, 2001. Code: https://web.archive.org/web/20140115171857/http://www.cs.princeton.edu/~kt/cut-tree/experiments/.

[13] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Pushmeet Kohli, Robert E. Tarjan, and Renato F. Werneck. Faster and more dynamic maximum flow by incremental breadth-first search. In *Algorithms - ESA*, pages 619–630, 2015.

[14] Andrew V. Goldberg, Sagi Hed, Haim Kaplan, Robert E. Tarjan, and Renato F. Werneck. Maximum flows by incremental breadth-first search. In *Algorithms - ESA*, pages 457–468, 2011.

[15] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *Journal of the Society for Industrial and Applied Mathematics*, 9:551–570, 1961.

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

[17] J. Hao and J. B. Orlin. A faster algorithm for finding the minimum cut in a directed graph. *Algorithmica*, 17:424–446, 1994.

[18] R. Hariharan, T. Kavitha, and D. Panigrahi. Efficient algorithms for computing all low s-t edge connectivities and related problems. In *SODA*, pages 127–136, 2007.

[19] T. C. Hu. *Integer Programming and Network Flows*. Addison-Wesley, Reading, MA, 1969.

[20] B. Korte and J. Vygen. *Combinatorial Optimization*. Springer Verlag, 2002.

[21] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[22] Jason Li, Debmalya Panigrahi, and Thatchaphol Saranurak. A nearly optimal all-pairs min-cuts algorithm in simple graphs. In *FOCS*, pages 1135–1146, 2021.

[23] Manfred W. Padberg and M. R. Rao. Odd minimum cut-sets and b-matchings. *Mathematics of Operations Research*, 7(1):67–80, 1982.

[24] Jan van den Brand, Yin Tat Lee, Yang P. Liu, Thatchaphol Saranurak, Aaron Sidford, Zhao Song, and Di Wang. Minimum cost flows, MDPs, and ℓ1-regression in nearly linear time for dense instances. In *STOC*, pages 859–869, 2021.