Algorithmica (2020) 82:1601–1615
https://doi.org/10.1007/s00453-019-00658-6

Speeding Up the Gomory-Hu Parallel Cut Tree Algorithm with Efficient Graph Contractions

Charles Maske[1] · **Jaime Cohen**[2] · **Elias P. Duarte Jr.**[1]

Received: 13 March 2019 / Accepted: 1 December 2019 / Published online: 14 December 2019

Abstract
A cut tree is a combinatorial structure that represents the edge-connectivity between all pairs of nodes of an undirected graph. Cut trees have multiple applications in dependability, as they represent how much it takes to disconnect every pair of network nodes. They have been used for solving connectivity problems, routing, and in the analysis of complex networks, among several other applications. This work presents a parallel version of the classical Gomory-Hu cut tree algorithm. The algorithm is heavily based on tasks that compute the minimum cut on contracted graphs. The main contribution is an efficient strategy to compute the contracted graphs, that allows processes to take advantage of previously contracted graph instances, instead of always computing all contractions from the original input graph. The proposed algorithm was implemented using MPI and experimental results are presented for several families of graphs and show significant performance gains.

Keywords Cut trees · Parallel algorithms · Gomory-Hu algorithm · Graph contractions

1 Introduction

Cut trees are combinatorial strutures that represent, in a compact form, the edge-connectivity between all pairs of vertices of a graph [1]. The edge-connectivity between a pair of vertices s and t corresponds to the capacity of the s-t minimum cut. Cut trees can be very useful in network dependability: for instance, if the capacity of every edge of the graph representing a network is equal to 1, cut trees reveal in an efficient way the number of edges that disconnect each and every pair of network nodes. Cut trees have been used to solve multiple relevant combinatorial problems including: routing

✉ Elias P. Duarte Jr.
elias@inf.ufpr.br

1 Department Informatics, Federal University of Parana, Curitiba, Brazil

2 Department Informatics, State University of Ponta Grossa, Ponta Grossa, Brazil

[2], graph partitioning and graph clustering [3,4], among others. Cut trees also can be used in complex network analysis, biological data analysis [5,6], social network analysis [7,8] among others [9–11].

There are two classical algorithms that are used to build a cut tree from a weighted graph: the Gomory and Hu algorithm [12] and the Gusfield algorithm [13]. Both algorithms are similar in the sense that both are heavily dependent on the computation of minimum cuts between pairs of vertices; for a graph with n vertices, $n-1$ minimum cuts are computed. The main difference between these two algorithms is that the Gusfield algorithm computes the $n-1$ minimum cuts on the input graph and the Gomory-Hu algorithm contracts the graph and computes the minumum cuts over the contracted graph instances. The Gomory-Hu algorithm also requires non-trivial data structures to manage the cut tree construction and the contracted graphs.

Parallel versions of both Gusfield and Gomory-Hu algorithms are presented in [14] along with a new hybrid algorithm that combines both strategies. These parallel algorithms employ a master-slave strategy to parallelize the s-t minimum cuts computations. The master process distributes the s and t vertex pairs for the slaves to compute the corresponding cuts. The slaves solve the s-t minimum cuts subproblems and then send the solutions back to the master, which updates the cut tree in construction and sends new tasks to the slaves. The algorithm finishes when the cut tree is complete and there are no more vertices to be separated by any s-t minimum cut. Experiments executed with the parallel version of the Gomory-Hu algorithm showed that the graph contractions are the single most expensive procedure employed by this algorithm, consuming nearly half of the execution time. It has been noted that it is not trivial to improve the efficiency of the graph contraction procedure as the slaves do not share any global data structure.

In this work we propose a new parallel Gomory-Hu algorithm based on an efficient strategy to implement graph contractions. The main goal is to allow slave processes to employ previously contracted graph instances so that they do not have to do all computations on the original input graph for every new task received. The implementation of this strategy requires the master process to maintain information about the tasks sent to each slave; results are stored and possibly used as new tasks are generated. The slave processes in their turn need to take decisions about using or not the contracted graphs from previous steps.

The proposed strategy was implemented with MPI and experiments are reported which were executed on a high performance cluster. Experiments were executed for both synthetic graphs and graphs representing real systems and networks. Results show that the *speedups* are roughly linear for most of the graphs. The comparison results show that the proposed strategy is indeed efficient, in particular for graphs with balanced minimum cuts.

The rest of the paper is organized as follows. In Section 2 we describe both the sequential and the parallel versions of the Gomory-Hu algorithm. In Section 3 the proposed strategy is described and specified. The implementation and experimental results are presented in Section 4. The conclusions follow in Section 5.

2 The Gomory-Hu Algorithm

This section starts with a description of the sequential version of the Gomory-Hu algorithm, which is followed by a description of an existing parallel version of that algorithm.

2.1 The Sequential Gomory-Hu Algorithm

Consider the problem of computing the edge-connectivity between all pairs of vertices of a weighted graph $G = (V, E, c)$, where V is the set of vertices, E is the set of edges and c is the edge capacity function $c : E \rightarrow \mathbb{R}^{+}$. The edge-connectivity between vertices s and t can be computed using a *maximum flow* algorithm which finds out the s-t minimum cut. A $s-t$ minimum cut is a partition $\{S, \overline{S}\}$ of the set of vertices V which necessarily separates vertices s and t, in other words $s \in S$ and $t \in \overline{S}$. The naive solution to this problem is compute all s-t minimum cuts between all the $\binom{|V|}{2}$ combinations of vertex pairs. Gomory and Hu [12] have shown that there are exactly $|V| - 1$ cuts which include at least one minimum cut for each pair of vertices of the graph. A cut tree is a structure that represents these $|V| - 1$ cuts in a compact way.

Figure 1 shows an example graph and the corresponding cut tree. As mentioned above, the cut tree describes the edge-connectivity between all pairs of vertices of the graph. So, for example, between vertices 3 and 7 the s-t minimum cut is $\{S, \overline{S}\} = \{\{0, 1, 2, 3, 4\}, \{5, 6, 7\}\}$ with capacity $c(S, \overline{S}) = 5$. Edge $\{3, 5\}$ of the T tree shows the minimum capacity between vertices 3 and 7 in G, and its removal induces a s-t minimum cut in G.

Next we describe the sequential Gomory-Hu algorithm which receives a graph G as input and builds the corresponding cut tree. In the description of Algorithm 1 the term *node* refers to a cut tree vertex. Line 4 corresponds to the proposed graph contraction strategy that is also described next.

Algorithm 1: Sequential Gomory-Hu Algorithm

```
input  : $G = (V, E_G, c_G)$ an undirected weighted graph
output: $T = (V, E_T, c_T)$ the cut tree of $G$
1 $T \leftarrow (V_T = \{V\}, E_T = \emptyset)$;
2 while $\exists X \in V_T$ such that $|X| > 1$ do
3     select node $X \in T$;
4     contract $G$ to form $G'$;
5     select a pair of vertices $s, t \in X$ ;
6     compute the $s$-$t$ minimum cut in $G'$;
7     update $T$ ;
8 return $T$
```

Let $G = (V_G, E_G, c_G)$ be the input graph. The algorithm starts the construction of the tree $T = (V_T, E_T, c_T)$ with a single node representing all the vertices of the input graph, i.e. $V_T = V_G$ and $E_T = \emptyset$. The contracted graph $G' = (V'_G, E'_G, c'_G)$ is initialized as $G' = G$. At each iteration, a node $X \in V_T$ such that $|X| \geq 1$ is selected which is called the *pivot*. The two vertices $s, t \in X$ are selected. The s-t minimum cut

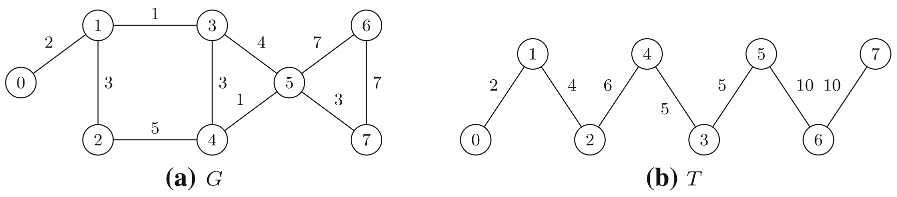

Fig. 1 An example of weighted graph and the corresponding cut tree

between s and t is computed on the contracted graph G' resulting in partition $\{S, \overline{S}\}$ and the cut with value $c(S, \overline{S})$.

Two new nodes X_s, X_t are created in the T tree such that $X_s = \{X \cap S\}$ and $X_t = \{X \cap \overline{S}\}$ as well as a new edge $e = \{X_s, X_t\}$ with $c(e) = c(\{S, \overline{S}\})$. For each edge $e' = \{X, Y\} \in E_T$ incident on X and Y, we check on which side of the s-t minimum cut $\{S, \overline{S}\}$ the vertices in node Y are. In case they are in S then edge $\{X_s, Y\}$ is created, connecting nodes X_s to Y. Otherwise the $\{X_t, Y\}$ edge is created connecting nodes X_t to Y. The algorithm updates the tree in such a way that the set of nodes $V_T = (V_T \setminus \{X\}) \cup \{X_s, X_t\}$ and the set of edges $E_T = E_T \cup \{e\}$.

Then a new pivot $X \in V_T$ is selected such that $|X| > 1$ and the algorithm computes the connected components of $T \setminus X$. The vertices of these connected components are denoted $V_1, V_2, \ldots, V_k$. The new contracted graph $G' = G \setminus X_1, X_2, \ldots, X_k$, where $X_i = \bigcup_{V' \in V_i} V'$ with $1 \leq i \leq k$. A new pair of vertices $s, t \in X$ is then randomly selected and the s-t minimum cut is computed on G'. After the new $\{S, \overline{S}\}$ partition is identified, two new nodes X_s, X_t are created in the tree, in the same way as described above.

The algorithm iteratively computes cuts for $|V| - 1$ pairs of vertices. Each pair is selected from a set of tree nodes which have not been separated by the cuts computed so far. Cuts are computed on contracted graph instances which are built based on the tree in construction and the input graph.

2.2 The Parallel Gomory-Hu Algorithm

The parallel version of the Gomory-Hu algorithm [14] employs the master-slave process communication model. In this strategy the master process is responsible for building the tree and generates s-t minimum cut problem instances which are sent to the slave processes to solve. Contracted graph instances are built by the slaves and minimum cuts computed in these graphs are returned to the master. This strategy requires little synchronization as the slaves compute the s-t minimum cuts without exchanging information with each other, but has the drawback that the computation of some cuts is wasted, as they are invalidated by other cuts computed by other slaves.

Let p be the total number of processes $proc_0,\ proc_1,\ \ldots,\ proc_{p-1}$ which execute the parallel version of the Gomory-Hu algorithm. Initially, each process receives a copy of the input graph $G = (V_G, E_G, c_G)$. The master process ($proc_0$) initializes the tree $T = (V_T, E_T, c_T)$, chooses a pivot $X \in V_T$ such that $|X| \geq 2$. The master then sends a task to the slave which consists of vertex pair $s, t \in X$, and a *partition* which associates vertices of the original graph G with vertices of the contracted graph. After the pair of vertices (s, t) is chosen, the partition is created as follows: each vertex in the pivot is labeled with a unique identifier; these are the only vertices which will not be contracted. Every other vertex will be labeled according to its connected component. After the master has sent the task to each slave $proc_s$ such that $s > 0$ it waits for the results.

A slave $proc_s$ in its turn, after it receives the task from the master, builds a contracted graph G'. The contracted graph construction consists of traversing the list of edges from the input graph and determining for each edge if there is a corresponding edge in the contracted graph. Given the s and t vertices received from the master, $proc_s$ solves a s-t minimum cut problem and returns the result to the master, including s, t and the corresponding s-t minimum cut. The master receives the result from the slave and verifies whether the pair of vertices s and t are in the same node of the tree. In case they are in the same node, the tree is updated. Then in the next step the master chooses a new pivot and sends a new task to the slave.

The master of course keeps receiving and processing other results from all slaves. However, the tree must be updated in a sequential way, as concurrent modifications are not allowed. Thus the master processes one slave result at a time. Some results are discarded because vertices s and t have already been separated by some other result received from another slave.

Algorithm 2 below shows a high level description of the parallel version of the Gomory-Hu algorithm. The *send* and *receive* operations correspond to MPI library communication functions. In the next section the efficient graph contraction strategy is described.

Algorithm 2: Parallel Gomory-Hu Algorithm

```
   input  : $G = (V, E_G, c_G)$ , $proc_j$, $0 \le j < p$ processes
   output: $T = (V, E_T, c_T)$ the cut tree computed from $G$
1  if $proc_j = 0$ then
       // master process
2      $T \leftarrow \{\{V_G\}, \emptyset\}$ ;
3      distribute tasks for all slaves processes;
4      while $|V_T| < |V_G|$ do
5          receive result (s,t,S) from $proc_j$ , such that $\{S, \overline{S}\}$ is a G s-t minimum cut ;
6          if $s, t$ are not separated then
7              update $T$ ;
8          if $T$ has nodes with more than two vertices then
9              send new task to $proc_j$ ;
10
11     return $T$ ;
12 else
       // slave process
13     while receiving tasks do
14         receive task (s,t,partition);
15         if task = end then
16             finish;
17         build contracted graph ;
18         compute the s-t minimum cut between s and t ;
19         send result (s,t,S) for the $proc_0$;
```

3 Efficient Graph Contractions

In [14] a detailed profile of the execution time of the parallel Gomory-Hu algorithm is presented that shows that graph contractions represent the component that requires most of the processing time. Therefore by improving the efficiency of graph contractions it is possible to improve the performance of the algorithm as a whole. However it is no trivial task to design an efficient strategy to implement graph contractions in the context of the parallel Gomory-Hu algorithm.

As mentioned above, graph contractions of the original parallel Gomory-Hu algorithm are executed by slave processes always using the input graph. In this section we specify an efficient graph contraction strategy that makes it possible for the slaves to employ contracted instances from previous steps whenever it is possible, instead of always using the input graph. As mentioned in the previous section, the contractions depend on the partition that is computed after the pivot is chosen. In order to specify the proposed strategy, some definitions follow.

Definition 1 Given graph $G = (V, E)$, we say that $G' = (V', E')$ is a contracted graph from G if:

- The set of vertices V' from G' is a partition from V
- For each $\{u_1, u_2\} \in E$ there is an edge $\{U_1, U_2\} \in E'$ if $u_1 \in U_1$ e $u_2 \in U_2$

Definition 2 Given partitions P and Q of set X , we say that P is a *refinement* of Q if every element in P is a subset of some element of Q .

For example, the partition $\{\{1, 3\}, \{2, 4, 7\}, \{5\}, \{6, 8\}\}$ from $\{1, 2, 3, 4, 5, 6, 7, 8\}$, is a refinement of this other partition $\{\{1, 3, 2, 4, 7\}, \{5, 6, 8\}\}$.

Definition 3 We say that the contracted graph $G'' = (V'', E'')$ is a refinement of $G' = (V', E')$ if V'' is a refinement of V'.

In the parallel Gomory-Hu algorithm, the contractions can be optimized since a contracted graph is a refinement of the graph contracted in the previous step. Furthermore, it is necessary to define a consistent condition to decide which tasks produce contracted graphs that are refinements of others. The lemma below describes this condition. The following notation is employed: $G'(X, T, G)$ is the contracted graph induced by pivot X on the cut tree in construction T from graph G.

Lemma 1 *The contracted graph $G'(X, T_1, G)$ is a refinement of $G'(Y, T_2, G)$ if and only if pivot X is contained in Y.*

Thus the algorithm chooses a pivot which is contained in the previous , i.e. the pivot of the previous task sent to the slave. This strategy is sufficient to ensure that the new contracted graph is a refinement of that used in the previous step by the same process. Thus, the contracted graph can be built from the contracted graph of the previous step instead of starting from the scratch with the original input graph.

In order to allow the master process to choose the pivot according to lemma 1, it is required to store the previous pivots chosen on previously computed tasks. The node that is chosen to be the pivot is the one with the largest number of vertices among those contained in the previous pivot of the last iteration. This criterion increases the chances that in the future further refinements will be found.

Another optimization strategy which was implemented consists in not executing contractions which will not produce much smaller graphs. In other words, if the contraction does not reduce the number of vertices by some constant k, the contraction is not executed and the contracted graph from the previous step is used to compute the s-t minimum cut. For the experiments we employed $k = 10$.

The Algorithm 3 below corresponds to a specification of the proposed parallel algorithm.

Algorithm 3: Optimized Parallel Gomory-Hu's Algorithm

```
input  : $G = (V, E_G, c_G)$ , $proc_j$, $0 \le j < p$ processes
output: $T = (V, E_T, c_T)$ the cut tree
if $j = 0$ // $proc_j$ is the master process
then
    $processes\_states[j] \leftarrow V$ for all $j$, $1 \le j < p$ ;
    $T \leftarrow \{\{V_G\}, \emptyset\}$ ;
    distributes tasks for all processes;
    while $|V_T| < |V_G|$ do
        receive from $proc_j$ the (s,t,S) result, such that $\{S, \overline{S}\}$ is a G s-t minimum cut ;
        if s and t are in same node X in $V_T$ // tree update
        then
            $X_s \leftarrow X \cap S$;
            $X_t \leftarrow X \cap \overline{S}$;
            $e \leftarrow \{X_s, X_t\}$;
            $c(e) \leftarrow c(S, \overline{S})$;
            foreach $e' = \{X, Y\} \in E_T$ edge incident on X in T do
                if $Y \subseteq S$ then
                    $E_T \leftarrow E_T \cup \{\{X_s, Y\}\} \setminus \{\{X, Y\}\}$;
                else
                    $E_T \leftarrow E_T \cup \{\{X_t, Y\}\} \setminus \{\{X, Y\}\}$;
            $V_T \leftarrow (V_T \setminus \{X\}) \cup \{X_s, X_t\}$;
            $E_T \leftarrow E_T \cup \{e\}$;
        if $|V_T| = |V_G|$ then
            send final message to $proc_j$;
        else
            $(X, has\_to\_refine) \leftarrow$ chose_pivot($T$, $processes\_states[proc_j]$);
            partition $\leftarrow$ build_partition(X, T) ;
            $(s, t) \leftarrow$ chose_st_pair(X);
            $processes\_states[proc_j] \leftarrow X$;
            send task (s,t,partition,has_to_refine) to $proc_j$ process;
    return $T$ ;
else
    // slave process
    while receive tasks do
        receive tasks (s,t,partition,has_to_refine);
        if task = end then
            finish;
        if contraction can reduce the number of vertices of G' then
            if $has\_to\_refine$ then
                $G' \leftarrow$ refine_contracted_graph($G'$, partition) ;
            else
                $G' \leftarrow$ build_contracted_graph($G$, partition) ;
        $S \leftarrow$ minumum_cut($G'$, $s$, $t$) ;
        send result (s,t,S) for $proc_0$ process;
```

Algorithm 3 receives graph $G = (V, E_G, c_G)$ as input and returns the corresponding cut tree $T = (V, E_T, c_T)$. Let p be a number of processes, $proc_0$ is the master process and $proc_j$, $1 \leq j \leq p$ are the slaves. The algorithm initializes the *processes_states* vector with the set of vertices of G in line 3. Then the master process sends tasks with (s, t) vertex pairs for all slaves. The master process stores the results received from the slaves in the loop of line 6. When a result with a (s, t) pair and a $S, \overline{S}$ *s-t* minimum cut is received, the master checks if the s and t vertices are in same tree node, in case they are the tree is updated. Updates are done in following way: take a node $X \in T$ such that $s, t \in X$ then create the new X_s and X_t nodes on the tree such that X_s has vertices $X \cap S$ and X_t has vertices $X \cap \overline{S}$. A new edge $e = \{X_s, X_t\}$ in E_T with $c(e) = c(S, \overline{S})$ is added to T.

Next, from line 14 to 18, the master process iterates over all edges $e' = \{X, Y\}$ updating them according with the side of $\{S, \overline{S}\}$ on which each vertex in Y is. If $V_T = V_G$ then the cut tree is built and master waits for the remaining results to send the final messages to all slaves. Otherwise the master chooses a new node X node as pivot for slave $proc_j$ using the *chose_pivot* procedure in line 24. The pivot is the vertex in $processes\_states[j]$ that is contained in the node with largest cardinality. If it is possible to find a pivot in this way, then the master sends the *refine* command to the slave which contracts the graph from the previous step, otherwise the slave will do the contraction using the input graph. After the new pivot is chosen, vector $processes\_state[j]$ is updated and a new task is sent to the process $proc_j$.

In line 32 slave $proc_j$ receives a task from the master. In case when the task has label *end* then the tree is constructed. Otherwise, if the slave receives the *refine* command then it builds the contracted graph G' from the contracted graph of the previous step using the *refine_contracted_graph* procedure. Otherwise the contraction process is one using the input graph with the *build_contracted_graph* procedure. After the contraction is completed, $proc_j$ slave process computes a $\{S, \overline{S}\}$ *s-t* minimum cut on G' and sends the (s, t, S) result to the master.

4 Experimental Results

Experiments were executed with multiple graph families, which are listed on Table 1. The BLOG, PGRI, ROME and GEO are graphs corresponding to real networks: a blog network [15], a power distribution network [16], a network which represents the streets of Rome [17] and a collaborative scientific network [18]. Two graph instances were generated using Erdős–Rényi [19] and Barabási–Albert [20] models. The other instances are synthetic graphs which have been used as *benchmarks* for minimum cut algorithms and cut trees [21–23]. The *noi* graphs are random graphs with the set of vertices divided in k clusters of which the inner edges tend to have higher capacities than the edges connecting vertices of different clusters. The *path* graphs are random graphs which have a path that consists of k edges with high capacity. Every vertex not in the path is randomly connected to the path by a high capacity edge, and all the other edges have lower capacities. Graphs of the *tree* type are similar to the *path* graphs, however a random tree is built so that the first k vertices are connected with high capacity edges.

Table 1 Graph families used in the experiments

Instance	$\|V\|$	$\|E\|$
BA	2000	9995
DCYC	1024	2048
ER	2000	10079
GEO	3621	9461
NOI	1000	99900
PATH	2000	21990
BLOG	1222	16714
PGRI	4941	6594
ROME	3353	8870
TREE	2000	21990

Experiments were executed on a high performance *cluster* which consists of 18 servers based on Intel® Xeon® E5-2670 processors each with with 8 cores, interconnected on a Gigabit Ethernet network.[1]

The algorithm was implemented with MPI using C/C++ and compiled with `gcc` using the max level optimization parameter (`-O3`). Speedups were computed as $S = T_S/T_P$ such that T_S is the time to execute the sequential version of algorithm and T_P is the time to execute the parallel version with P processes. Each experiment was executed 10 times and averages are presented. Next we show the results using *GH-Opt* to refer to the algorithm proposed in this work and *GH* corresponds to the parallel algorithm without efficient graph contractions.

Figures 2, 3 and 4 show the total execution times for both versions of the algorithm using graphs of types *path*, *noi* and *tree* respectively. These graphs were built with 1000 vertices, a density of 20% and a varying k parameter. We note that the proposed algorithm *GH-Opt* reached the best results for all instances.

The fact that the running times – in particular those shown in figures 3 and 4 – are non monotone can be explained by the impact of parameter k on the different graph instances. As mentioned above, the edges within a cluster have high capacities, while the edges between clusters have low capacities. So, for small values of k there are no balanced cuts; then as k increases the cuts become more balanced, thus the contracted graphs become smaller. However, as k becomes too large, there are few balanced cuts and eventually there will be no more balanced cuts.

Figure 5 shows the total execution times for both algorithms using the instances in Table 2. For the majority of instances the proposed GH-Opt got better results. Only with the PGRI and ROME graphs GH-Opt took longer than GH. These graphs do not have balanced cuts and so the GH-Opt algorithm does not have many opportunities

[1] The Cluster is hosted at the Central Laboratory for High Performance Computing (LCPAD) of UFPR, and is sponsored by FINEP through the CT-INFRA/UFPR projects.

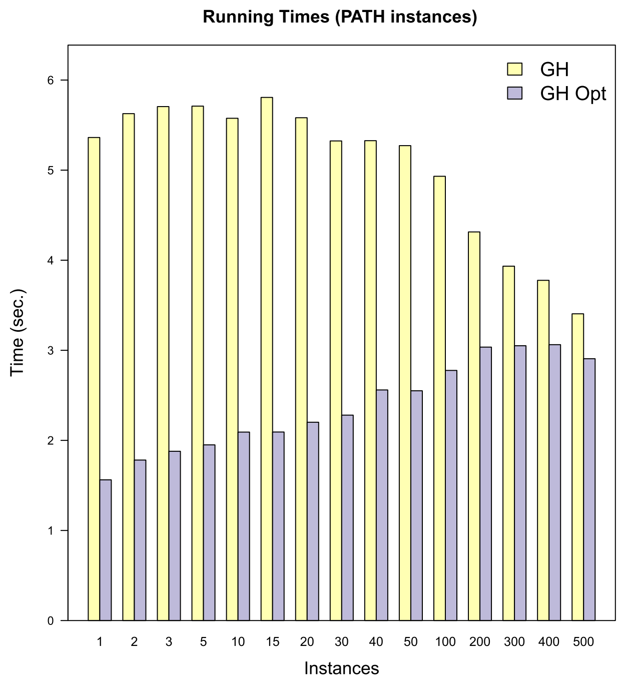

Fig. 2 Total execution times for the GH and GH-Opt algorithms for *path* graphs

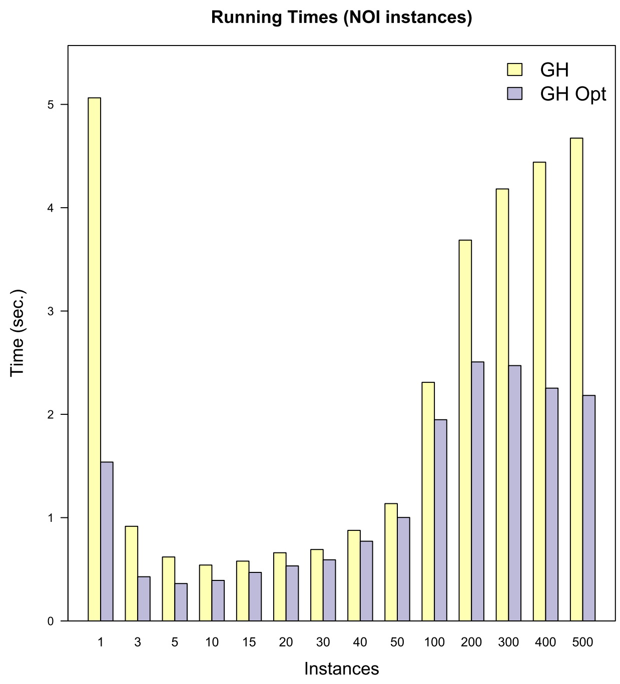

Fig. 3 Total execution times for the GH and GH-Opt algorithms for *noi* graphs

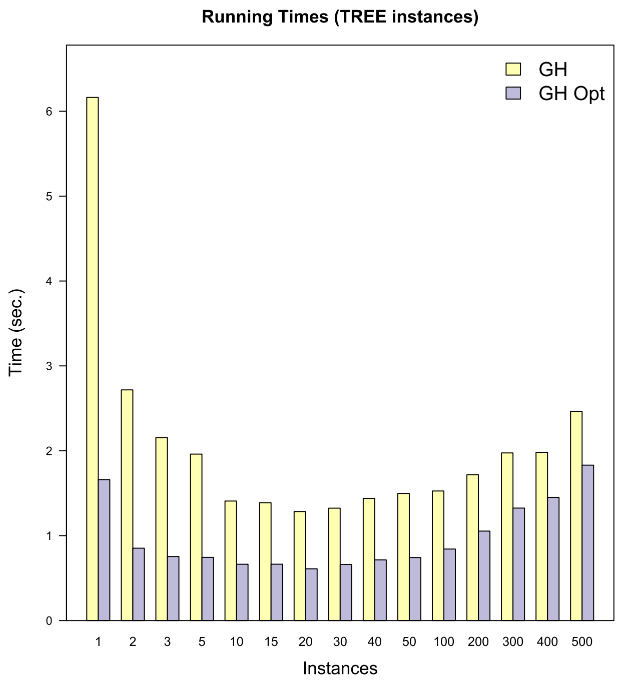

Fig. 4 Total execution times for the *GH* and *GH-Opt* algorithms for the *tree* graph

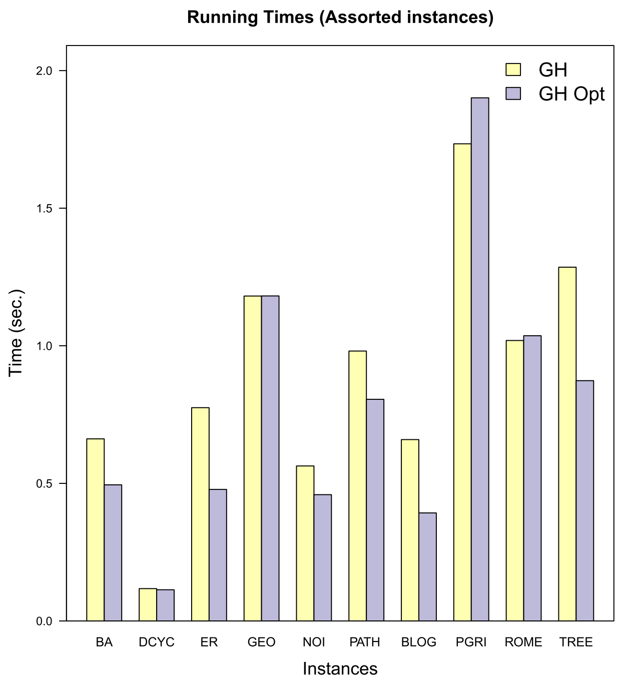

Fig. 5 Total execution times for GH and GH-Opt algorithms for the graphs on Table 1

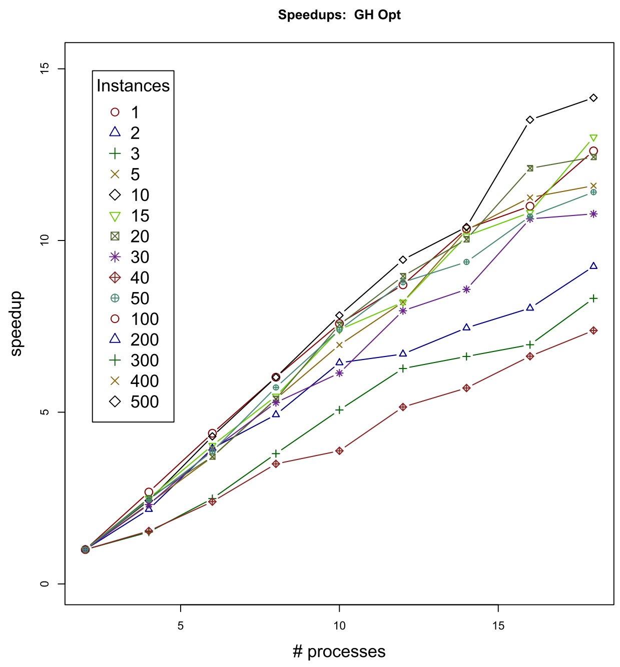

Fig. 6 Speedup of GH-Opt for *noi* graphs with 2000 vertices

to optimize contractions. For the GEO and DCYC graphs the time to execute both algorithms are exactly the same.

Figure 6 shows the *speedup* of the proposed algorithm using *noi* graphs with 2000 vertices and density of 20% and a varying k. For up to 18 hosts, which is the maximum number of hosts we had available to run our experiment, the speedups were roughly linear.

Figure 7 shows the average time taken by both algorithms contracting graphs. It is possible to confirm that the proposed strategy does have a positive impact on the total time to build cut trees.

In order to measure the proportion of the total time that is spent with graph contractions in the new algorithm we executed a batch of experiments for the *noi* graphs and report the results in Table 2. It is possible to see that in the original algorithm, graph contractions correspond to about half of the execution time. On the other hand the proposed optimized graph contractions saved up to 26% of that proportion.

5 Conclusions

In this work we introduced new version of the parallel Gomory-Hu algorithm that employs an efficient strategy to compute graph contractions. As graph contractions have been shown to dominate the execution time for computing cut trees, investigating strategies to improve its performance is relevant work. The proposed strategy allows

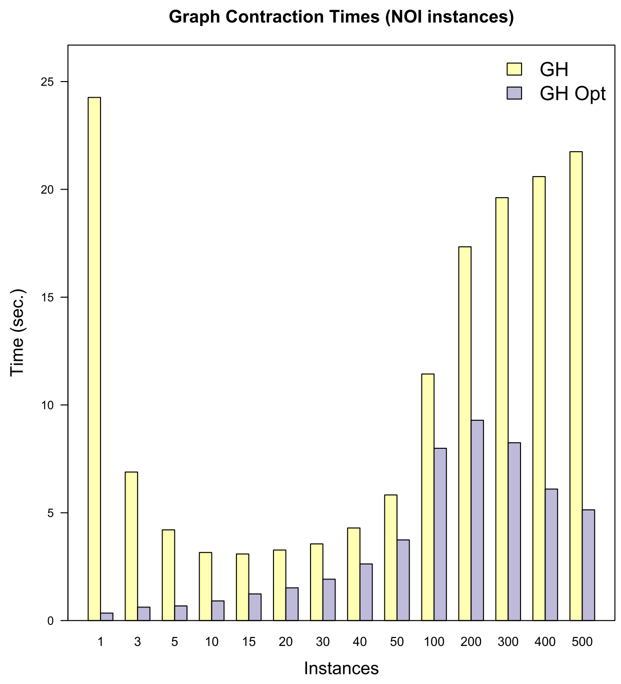

Fig. 7 Average time spent contracting graphs for both GH and GH-Opt

Table 2 Proportion of the total running time used to contract for the *noi* graphs

Instance (k)	Original	Optimized	Improvement
1	53%	54%	−2%
2	52%	49%	6%
3	54%	46%	15%
5	50%	37%	26%
10	42%	32%	24%
15	46%	35%	24%
20	45%	35%	22%
30	47%	38%	19%
40	46%	39%	15%
50	50%	42%	16%
100	51%	47%	8%
200	52%	52%	0%
300	52%	51%	2%
400	52%	52%	0%
500	52%	53%	−2%

previously computed contracted graph instances to be employed whenever possible, thus avoiding always computing contractions on the original input graph. In terms of resources required to be implemented, the master process maintains information on tasks sent to the slaves, storing results for possibly being used in the future. The slaves, in their turn, need to take decisions about whether to use or not a previously contracted graph instance. The strategy was implemented with MPI and executed on a high performance cluster at UFPR. The speedup was evaluated, including a comparison between the algorithm with and without the efficient graph contraction strategy. Experiments were executed for multiple types of synthetic and real graphs. The results show that speedups are roughly linear for the most instances, and that the proposed strategy does improve the performance of the algorithm.

Future work includes designing a parallel version of the master process, since it can become a bottleneck as the number of processes running the algorithm grows. Furthermore, we believe that it is relevant to implement the graph contraction strategy with the OpenMP library, which allows sharing the efforts among multiple local *threads* running on multiple cores. Finally, we can investigate whether our parallel strategies can be applied to or can be improved by new results such as those presented in [24,25].

Acknowledgements This work was partially supported by the Brazilian Research Council CNPq, Grants 311451/2016-0 and 428941/2016-8.

References

1. Nagamochi, H., Ibaraki, T.: Algorithmic Aspects of Graph Connectivity. Algorithmic Aspects of Graph Connectivity. Cambridge University Press, Cambridge (2008)
2. Duarte Jr, E., Santini, R., Cohen, J.: Delivering packets during the routing convergence latency interval through highly connected detours. In: 2004 International Conference on Dependable Systems and Networks, pp. 495–504 (2004)
3. Engelberg, R., Könemann, J., Leonardi, S., Naor, J.: Cut problems in graphs with a budget constraint. In: Correa, J.R., Hevia, A., Kiwi, M. (eds.) LATIN 2006: theoretical informatics. Lecture notes in computer science, vol. 3887, pp. 435–446. Berlin Heidelberg, Springer (2006)
4. Saran, H., Vazirani, V.V.: Finding k cuts within twice the optimal. SIAM J. Comput. **24**(1), 101–108 (1995)
5. Mitrofanova, A., Farach-Colton, M., Mishra, B.: Efficient and robust prediction algorithms for protein complexes using Gomory–Hu trees. In: Pacific Symposium on Biocomputing, pp. 215–226 (2009)
6. Tuncbag, N., Salman, F.S., Keskin, O., Gursoy, A.: Analysis and network representation of hotspots in protein interfaces using minimum cut trees. Proteins Struct. Funct. Bioinform. **78**(10), 2283–2294 (2010)
7. Kamath, K.Y., Caverlee, J.: Transient crowd discovery on the real-time social web. In: Proceedings of the 4th ACM International Conference on Web Search and Data Mining, WSDM ’11, pp. 585–594. ACM (2011)
8. Backstrom, L., Dwork, C., Kleinberg, J.: Wherefore art thou r3579x?: anonymized social networks, hidden patterns, and structural steganography. In: Proceedings of the 16th International Conference on World Wide Web, WWW ’07, pp. 181–190. ACM (2007)
9. Kim, C.-B., Foote, B.L., Pulat, P.: Cut-tree construction for facility layout. Comput. Ind. Eng. **28**(4), 721–730 (1995)
10. Jermaine, C.: Computing program modularizations using the k-cut method. In: Sixth Working Conference on Reverse Engineering, 1999. Proceedings. pp. 224–234 (1999)
11. Saha, B., Mitra, P.: Dynamic algorithm for graph clustering using minimum cut tree. In: ICDM Workshops 2006. 6th IEEE International Conference on Data Mining Workshops, 2006, pp. 667–671 (2006)

12. Gomory, R.E., Hu, T.C.: Multi-terminal network flows. J. Soc. Ind. Appl. Math. **9**(4), 551–570 (1961)
13. Gusfield, D.: Very simple methods for all pairs network flow analysis. SIAM J. Comput. **19**(1), 143–155 (1990)
14. Cohen, J., Rodrigues, L.A., Duarte Jr., E.P.: Parallel cut tree algorithms. J. Parallel Distrib. Comput. **109**, 1–14 (2017)
15. Adamic, L.A., Glance, N.: The political blogosphere and the 2004 u.s. election: Divided they blog. In: Proceedings of the 3rd International Workshop on Link Discovery, pp. 36–43. ACM (2005)
16. Watts, D.J., Strogatz, S.H.: Collective dynamics of 'small-world' networks. Nature **393**(6684), 440–442 (1998)
17. Storchi, G., Dell'Olmo, P., Gentili, M.: Road network of the city of rome. In: 9th DIMACS Implementation Challenge—Shortest Paths. Available at http://www.dis.uniroma1.it/challenge9/download.shtml (1999). Accessed 11 Dec 2019 (1999)
18. Batagelj, V., Mrvar, A.: Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data. Accessed 11 Dec 2019
19. Bollobás, B.: Random Graphs, 2nd edn. Cambridge University Press, Cambridge (2001). (**Cambridge Books Online**)
20. Albert, R., Barabási, A.-L.: Statistical mechanics of complex networks. Rev. Mod. Phys. **74**, 47 (2002)
21. Nagamochi, H., Ono, T., Ibaraki, T.: Implementing an efficient minimum capacity cut algorithm. Math. Program. **67**(1–3), 325–341 (1994)
22. Chekuri, C.S., Goldberg, A.V., Karger, D.R., Levine, M.S., Stein, C.: Experimental study of minimum cut algorithms. In: Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '97, pp. 324–333. Society for Industrial and Applied Mathematics, Philadelphia (1997)
23. Goldberg, A.V., Tsioutsiouliklis, K.: Cut tree algorithms. In: Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '99, pp. 376–385. Society for Industrial and Applied Mathematics, Philadelphia (1999)
24. Anari, N., Vazirani, V. V.: Planar graph perfect matching is in NC. In: Proceedings of the 59th IEEE Annual Symposium on Foundations of Computer Science, FOCS'2018, pp. 650–661 (2018)
25. Abboud, A., Krauthgamer, R., Trabelsi O.: New algorithms and lower bounds for all-pairs max-flow in undirected graphs In: Proceedings of the XXth ACM-Siam Symposium on Discrete Algorithms, SODA'2020 (2020)