Efficient computation of partial Gomory-Hu trees

University of Ioannina

Christos Pergaminelis

September 2023

Examination committee

- **Loukas Georgiadis**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Christos Nomikos**, Associate Professor, Department of Computer Science and Engineering, University of Ioannina
- **Leonidas Palios**, Professor, Department of Computer Science and Engineering, University of Ioannina

Dedication

To my parents (George and Anastasia Pergamineli), Cynthia, Antigoni and all the wonderful friends i made in Ioannina.

Acknowledgements

I would like to thank my supervisor, Professor Loukas Georgiadis, for his valuable guidance and support. His introduction to the Gomory-Hu trees and his skillful instructions on topics such as Max flow and Graph Connectivity from his course "Advanced Algorithms and Data Structures", have enriched my understanding of these concepts and how they relate to each other.

Abstract

Consider an undirected unweighted graph, denoted as G=(V, E) where V represents the set of vertices, E represents the set of edges and an integer $k \geq 1$. In this context, our objective is to address the problem of calculating the edge connectivities for all pairs of vertices s,t such that the edge connectivity between them is less than or equal to k. Our goal is the implementation of a data structure which is a weighted tree, denoted as T, in which the nodes correspond to the subsets $V1, V2, ..., Vn$ of a partition of the vertex set V. Notably, the tree possesses a key property: the edge connectivity between any pair of vertices, where one vertex belongs to Vi and the other belongs to Vj, for i≠j, is precisely equal to the weight of the lightest edge present on the path connecting the node Vi and the node Vj within the tree T.

Keywords: Graph Connectivity, Data Structures and Algorithms

Abstract in Greek

Ας θεωρήσουμε ένα μη κατευθυνόμενο άβαρο γράφημα, που συμβολίζεται ως G=(V, E) όπου το V αντιπροσωπεύει το σύνολο των κορυφών, το E αντιπροσωπεύει το σύνολο των ακμών, καθώς και έναν ακέραιο $k \geq 1$. Ο στόχος μας είναι να επιλύσουμε το πρόβλημα του υπολογισμού της συνδεσιμότητας ακμών για όλα τα ζεύγη κορυφών s,t έτσι ώστε η συνδεσιμότητα ακμών μεταξύ τους να είναι μικρότερη ή ίση με k. Η λύση μπορεί να επιτευχθεί με την υλοποίηση μιας δομής δεδομένων ενός έμβαρου δέντρου, που συμβολίζεται ως T, στο οποίο οι κόμβοι αντιστοιχούν στα υποσύνολα $V1, V2, ..., Vn$ ενός διαμερισμού του συνόλου κορυφών V. Συγκεκριμένα, το δέντρο πρέπει να διαθέτει μια βασική ιδιότητα: η συνδεσιμότητα ακμών μεταξύ οποιουδήποτε ζεύγους κορυφών, όπου η μία κορυφή ανήκει στο Vi και η άλλη στο Vj, για $i \neq j$, να είναι ακριβώς ίση με το βάρος της ελαφρύτερης ακμής που υπάρχει στη διαδρομή η οποία συνδέει τον κόμβο Vi και τον κόμβο Vj στο δέντρο T.

Keywords: Συνδεσιμότητα Γραφημάτων, Δομές Δεδομένων και Αλγόριθμοι

Contents

Chapter 1

Introduction

1.1 General

In combinatorial optimization, Gomory-Hu trees are a data structure mostly used in graph theory and network optimization. They were introduced by Ralph E. Gomory and Tomás S. Hu in 1961. These trees provide a compact and simple representation of the minimum cuts in undirected graphs and have a large number of applications. In this work we present and implement a version of an algorithm introduced by Ramesh Hariharan, Telikepalli Kavitha and Debmalya Panigrahi [1] which calculates partial Gomory-Hu trees.

1.2 Basic concepts

In this section, we will delve into fundamental principles essential for this thesis. Graph connectivity contains two definitions, vertex connectivity and edge connectivity. The National Institute of Standards and Technology provides the following definitions:

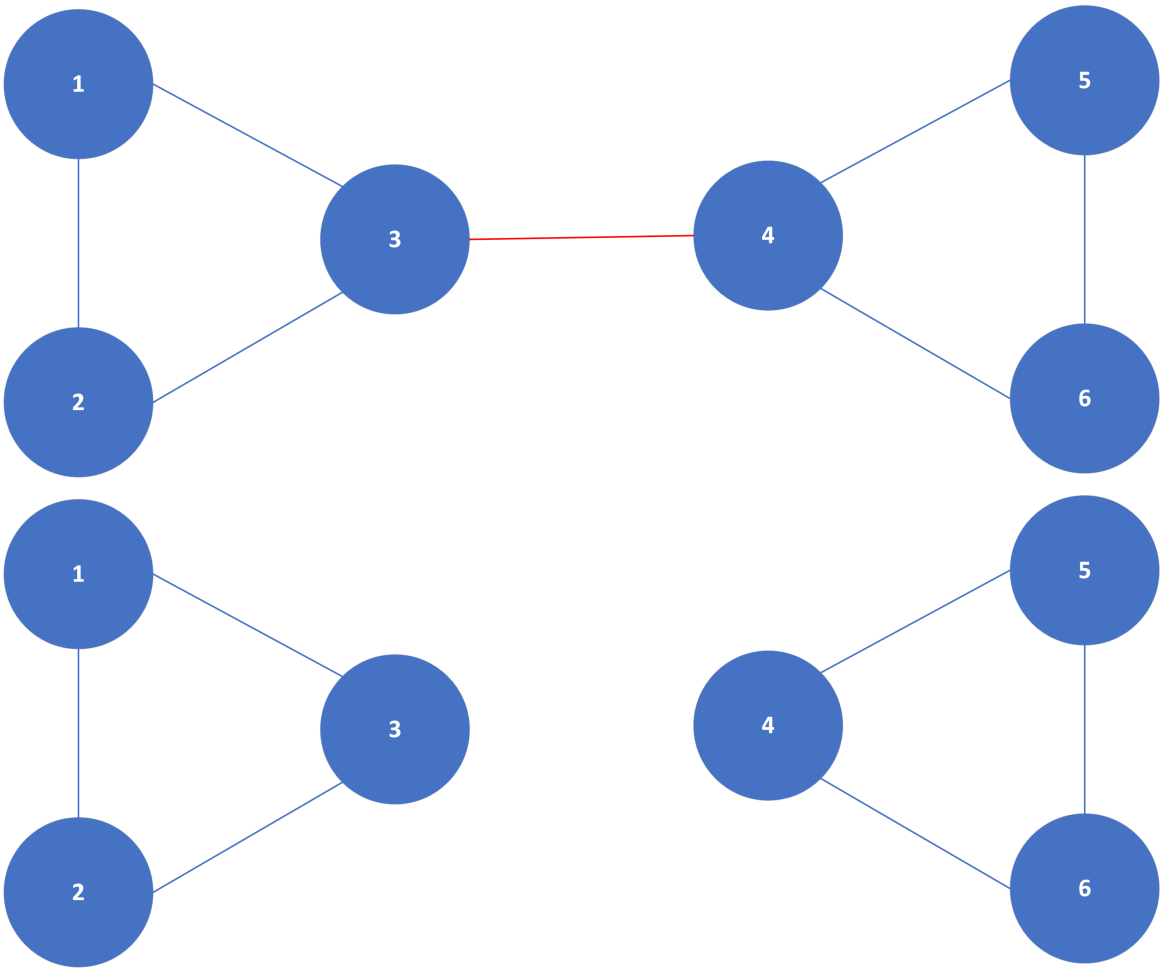
Vertex connectivity: (1) The smallest number of vertices whose deletion causes a connected graph to not be connected. (2) For a pair of vertices s and t in a graph, the smallest number of vertices whose deletion will separate s from t.

Edge connectivity: (1) The smallest number of edges whose deletion will cause a connected graph to not be connected. (2) For a pair of vertices s and t in a graph, the smallest number of edges whose deletion will separate s from t.

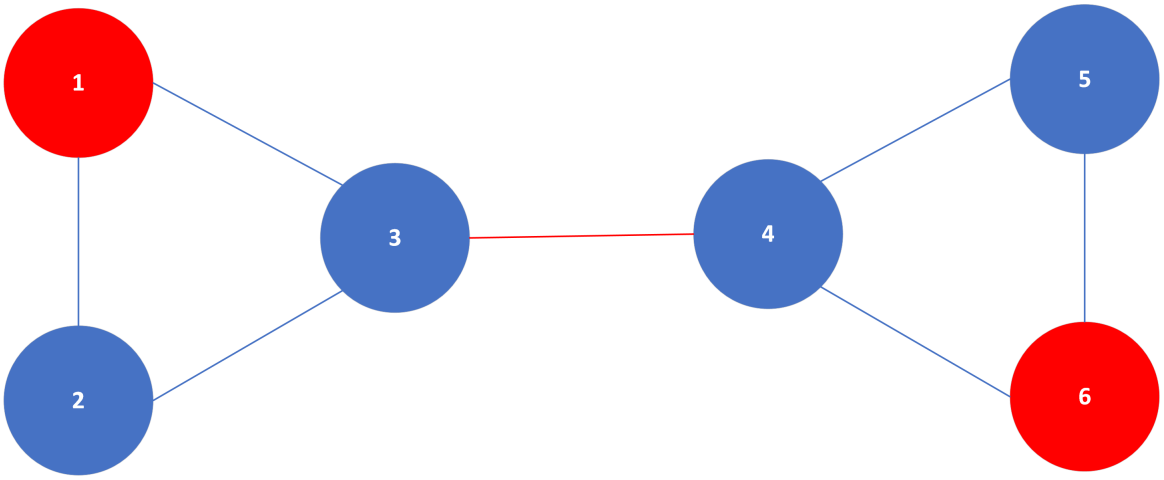
Throughout this thesis, when we refer to connectivity, we specifically imply edge connectivity. We will now delve into the topic of k-edge connectivity and local edge connectivity.

A graph is said to be ”k-edge connected” if, for any set of k-1 or fewer edges removed from the graph, it remains connected, meaning there is still a path between any pair of vertices in the remaining graph. In other words, a graph is k-edge connected if it takes the removal of at least k edges to disconnect the graph.

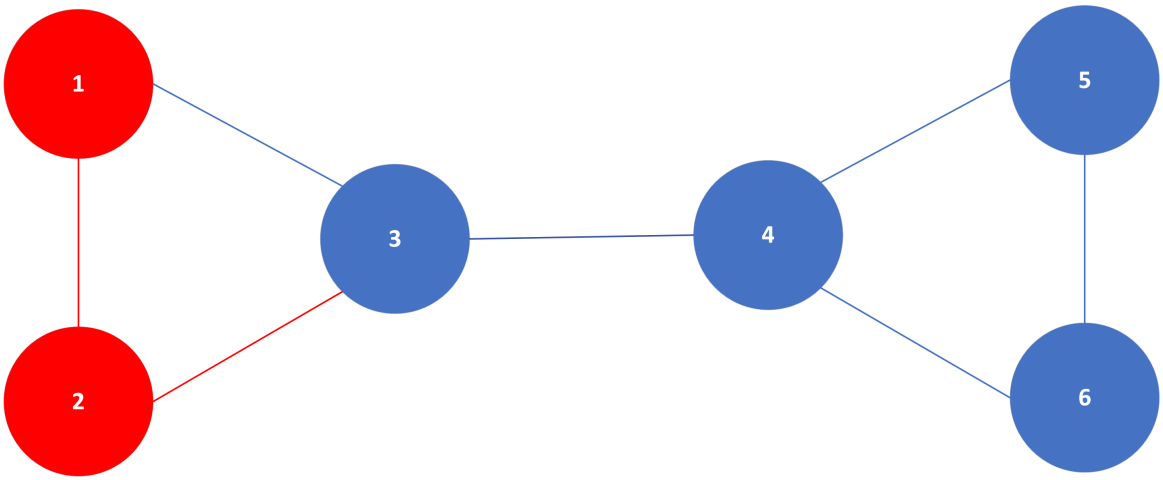
Example: The given graph is clearly 1 edge connected since by removing the edge 3,4 the graph disconnects into two components:



We can now explain local edge connectivity. The local edge connectivity between two vertices, u and v, denoted as λ(u, v), represents the minimum number of edges that need to be removed to disconnect u from v in the graph. If λ(u, v) is equal to k, it means that there are k edge deletions needed to separate u and v. Example: λ(1, 6) is clearly equal to 1, as the removal of edge (3, 4) results in the separation of vertices 1 and 6 into two separate connected components.

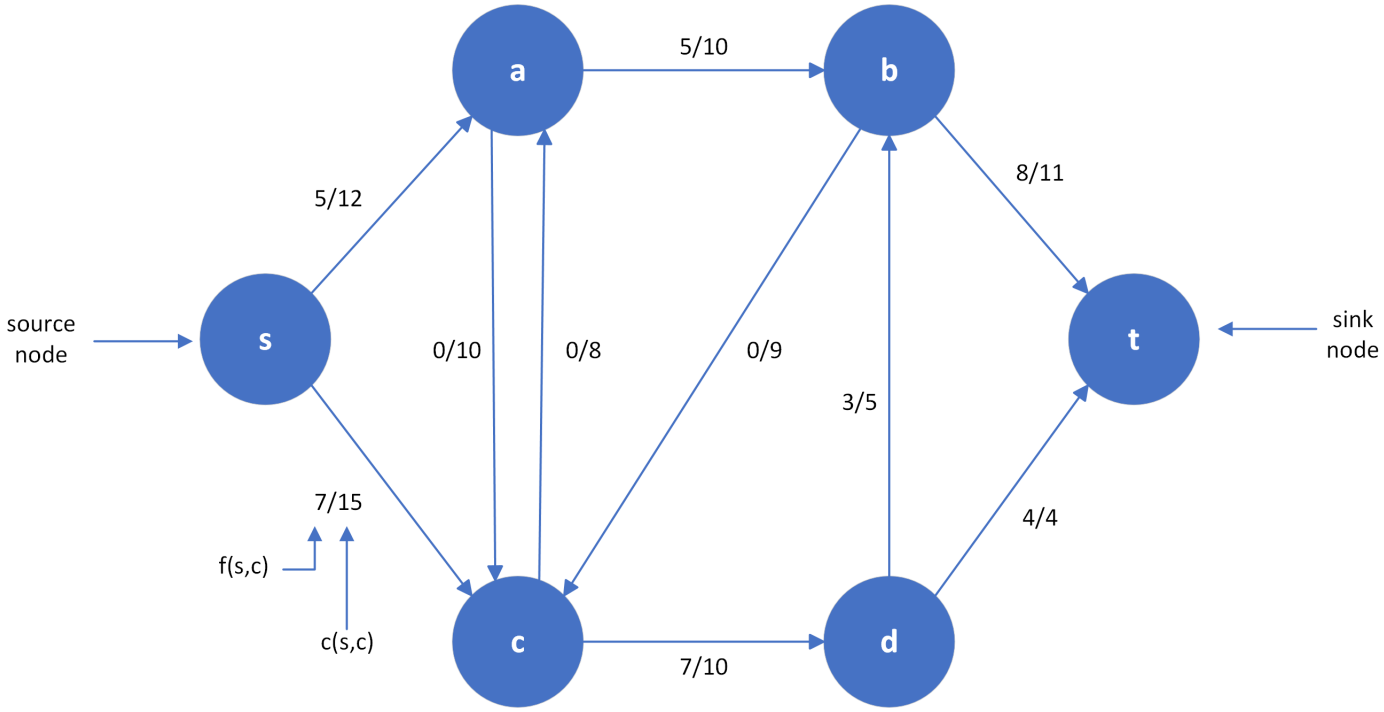


$\lambda(1,2)$ is equal to 2, as the removal of edges (1,2) and (2,3) results in the separation of vertices 1 and 2 into two separate connected components.



At last, we will explain the concept of flow. In graph theory, a "flow" is a function $f : V \times V \to R$ that assigns values to the edges of a directed graph This function represents the movement or transportation of a certain resource, like goods, data, or traffic, from one node to another in the graph. The flow function has the following important properties:

1) Capacity Constraint: $f(u, v) \leq c(u, v)$ for all pairs u,v $\in$ V
2) Antisymmetry: $f(u, v) = -f(v, u)$ for all pairs u,v $\in$ V
3) Flow Conservation: $\sum_{u \in V} f(u, v) = 0$ for every $u \in V - \{s, t\}$
4) Value of flow: $|f| = \sum_{u \in V} f(s, u)$



1.3 The Minimum Cut problem and the Max Flow/Min Cut theorem

To understand Gomory-Hu trees we also need to define the minimum cut:

An (edge) cut of a graph G is a set of edges $C \subseteq E$ such that the deletion of C increases the number of connected components of G.

The minimum cut of an unweighted graph is defined as the minimum number of edges that, when removed, the graph is divided into two sets.

The maximum flow of a graph can be described as follows: A maximum flow is characterized as the highest achievable volume of flow that a graph permits to pass from the source node to the sink node.

Finally, the max-flow/min-cut theorem states that the maximum flow through any network from a given source to a given sink is exactly equal to the minimum sum of a cut. This theorem can be verified using the Ford-Fulkerson algorithm for max flow.

1.4 Gomory-Hu Trees

Given a graph G=(V,E) with a capacity function c, a Gomory-Hu tree T=(V,F) obtained from G is a connected acyclic graph having the same set of vertices V and an edge set F with a capacity function c' verifying the following properties [2]:

1.Equivalent flow tree: for any pair of vertices s and t, $f_{s,t}$ in G is equal to $f_{s,t}$ in T , i.e., the smallest capacity of the edges on the path between s and t in T.

2.Cut property: a minimum cut $C_{s,t}$ in T is also a minimum cut in G.

1.5 Applications

The importance of Gomory-Hu Trees lies in their applications in network optimization. Some of the most common applications are:

1.Network Connectivity: Gomory-Hu Trees provide insight into the connectivity of a graph. The tree's structure reveals the relationships between different components and subgraphs, helping to understand the flow of information or resources in a network.

2.Network Reliability: Gomory-Hu Trees help analyze how reliable networks are. By calculating the minimum cuts, which represent potential failure points, we can assess the resilience and vulnerability of a network. This information is useful in designing robust networks and planning backup strategies.

3.Network Design: Gomory-Hu Trees can assist in designing optimal networks by identifying critical connections. They assist in determining the most efficient placement of links to minimize costs or maximize performance.

4.Network Capacity Planning: Gomory-Hu Trees are valuable in network design and capacity planning problems. They can assist in determining optimal routing and resource allocation strategies, ensuring efficient utilization of network resources while meeting capacity constraints.

1.6 The "at most k-connectivity" problem

The "at most k-connectivity" problem can be expressed as:

Given a graph G and an integer $k \geq 1$, compute the edge connectivities of all pairs of vertices in G whose edge connectivity is less than or equal to k

For small values of k, the problem above is directly related to the problem of representing failure points. The problem described can be solved by calculating a Gomory-Hu tree that includes only the edges with weights that are less than or equal to a given value, denoted as k, without the need to calculate all the edges of the tree. This approach ensures a significantly more efficient solution.

1.7 An efficient solution to the problem

Consider any value of k. Envision contracting all edges in the Gomory-Hu tree T whose weights exceed k. This contraction process establishes a new tree, denoted as T', which includes subsets of vertices denoted as $V1, V2, ..., Vn$. Notably, for any given i, vertices s,t $\in$ Vi if and only if all edges in the path between s and t in the original tree T have weights greater than k. The T' tree is a partial Gomory-Hu tree. Therefore, it can be deduced that the edge connectivity between vertices s and t in graph G, where both s and t belong to the same set Vi for any i, is guaranteed to be at least k + 1 and the s-t min-cut value for any pair s, t that belong to different sets Vi, Vj in the partition $V1, V2, ..., Vn$ is the weight of the lightest edge on the path between the nodes Vi and Vj in the tree T'. Clearly, computing a partial Gomory-Hu tree solves the at most k-connectivity problem.

Chapter 2

Critical Subroutines and Algorithms

Preceding the exposition and implementation of the partial Gomory-Hu algorithm, it is necessary to provide some foundational concepts and algorithms.

2.1 BFS/DFS algorithms

The Breadth-First Search (BFS) algorithm is employed to explore a tree or graph data structure in search of a node that satisfies specified criteria. It commences from the root of the tree or graph and examines all nodes at the current depth level before progressing to nodes at subsequent depth levels with a time complexity of O(|V|+|E|).

Algorithm 1 BFS (G,root,sink)

```
let Q be a queue
Q.enqueue(root)
root ← visited
while Q is not empty do
    v=Q.dequeue
    for all neighbours of v in G do
        if neighbour is not visited then
            Q.enqueue(neighbour)
            mark neighbour as visited
            if neighbour == sink then
                return path
            end if
        end if
    end for
end while
```

Similarly, Depth-first Search (DFS), also known as depth-first traversal, is a recursive algorithm utilized to explore all the nodes within a graph or tree data structure. Traversal refers to the process of visiting each node in the graph, following a depth-first approach, with a time complexity of O(|V|+|E|).

Algorithm 2 DFS (G,root)

```
root ← visited
for all neighbours of s in G do
    if neighbour is not visited then
        DFS(G,neighbour)
    end if
end for
```

2.2 The Ford Fulkerson algorithm and finding the edges of the Min Cut

Given a graph G, a source node s and a sink node t, the Ford-Fulkerson algorithm with a time complexity of O(|E|F) where F is the maximum flow value and the algorithm can be defined as follows:

Algorithm 3 Ford-Fulkerson (G,s,t)

```
max flow ← 0
path ← BFS(G, s, t)
while path exists do
    augment flow in path
    RG ← residual graph
    path ← BFS(RG, s, t)
end while
return flow
```

In order to identify the edges of the minimum cut instead of returning the maximum flow, one can execute a Depth-First Search (DFS) algorithm starting from the source node. During this process, all visited nodes are marked. Subsequently, the edges connecting the marked nodes to the non-marked nodes are collected, forming the set of edges representing the minimum cut in an undirected and unweighted graph. The residual graph (RG) represents the remaining capacity of edges after some flow has been pushed through them.

In RG:

Forward edges represent the remaining capacity of the original edges, indicating how much more flow can be pushed in the same direction.

Backward edges represent the possibility of decreasing the flow on an edge if needed in subsequent iterations of the algorithm.

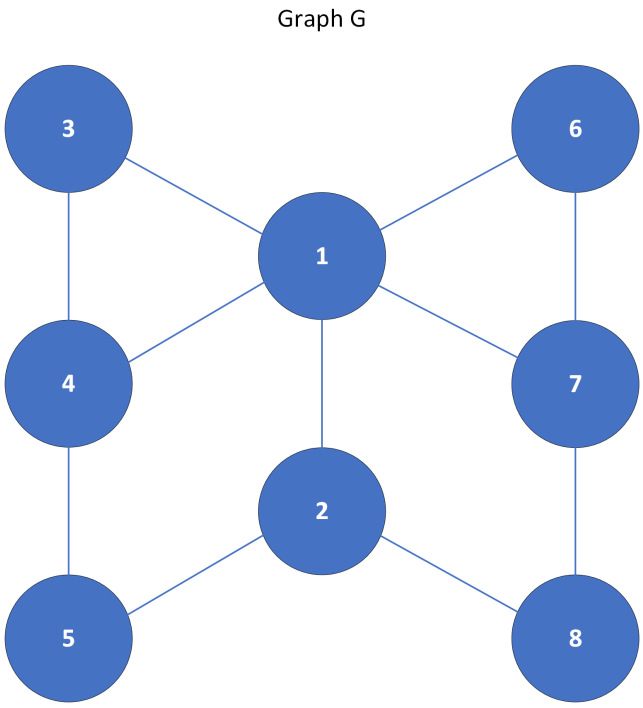
2.3 The Gomory-Hu with contractions algorithm

The "Component contruction Algorithm" is a prevailing method for computing a Gomory-Hu tree, employing solely $|V| - 1$ minimum cuts [5]

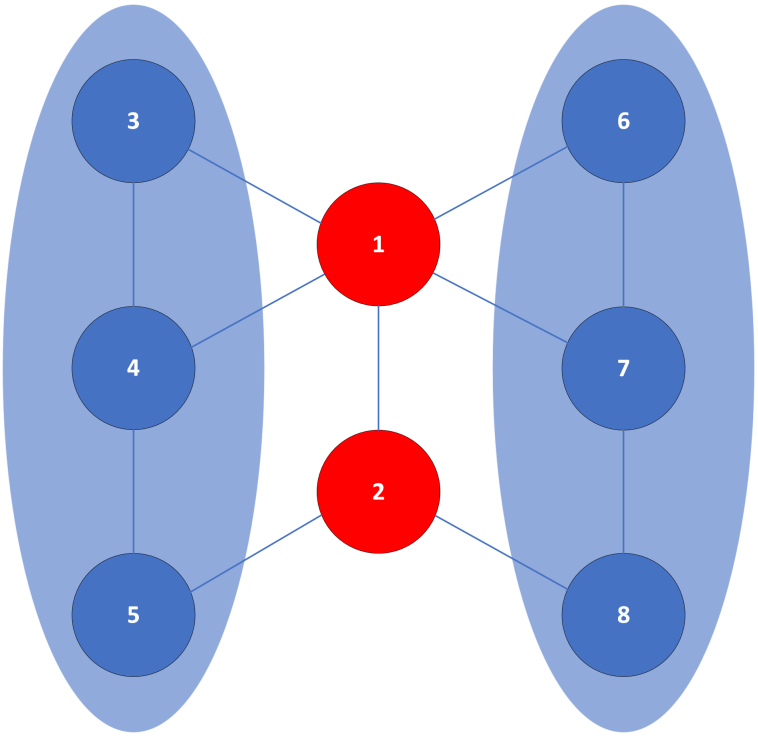
Algorithm 4 Gomory-Hu Algorithm (G)

$V_T \leftarrow V_G$
$E_T \leftarrow \emptyset$
while Choose a X $\in$ V_T with |X|$\geq$2 if such X exists **do**
  Calculate T\X
  for each connected component C=(V_C, E_C) in T\X let S_C=$\cup_{VT \in VC} V_T$. Let S={S_C|C is a connected component in T\X} **do**
    Contract components to form the new graph H
    V_H=X $\cup$ S
    E_H=$E_G|_{X \times X}$ $\cup$ {(u,S_C) $\in$ X$\times$S|(u,v)$\in$ E_G for some v $\in$ S_C} $\cup$ {(S_{C1},S_{C2} $\in$ S$\times$S|(u,v)$\in$$E_G$ for some u $\in$ S_{C1} and v $\in$ S_{C2}}
    c':$V_H \times V_H \rightarrow R^+$ is the capacity function defined as:
    if (u,S_C)$\in$$E_G|_{X \times S}$ **then**
      c'(u,S_C)=$\Sigma_{v \in Sc:(uv) EG}$ c(u,v)
    else if (S_{C1},S_{C2})$\in$$E_G|_{S \times S}$ **then**
      c'(S_{C1},S_{C1})=$\Sigma_{(uv) EG: u \in SC1 \wedge v \in SC2}$ c(u,v)
    else
      c'(u,v)=c(u,v)
    end if
  end for
  Choose two vertices s,t $\in$ X and find the minimum s,t cut (A_H,B_H) in H
  Set A=($\cup_{SC \in AH \cap S} S_C$) $\cup$ ($A_H \cap$X) and B=($\cup_{SC \in BH \cap S} S_C$) $\cup$ ($B_H \cap$X)
  Set V_T=(V_T\X) $\cup$ {A $\cap$ X, B $\cap$ X}
  for each e=(X,Y) $\in$ E_T **do**
    if Y $\subset$ A **then**
      e_H=(A $\cap$ X,Y)
    else
      e_H=(B $\cap$ X,Y)
    end if
  end for
  Set E_T=E_T\{e} $\cup$ {e_H}
  E_T=E_T $\cup$ {(A$\cap$X,B$\cap$X)}
  w((A$\cap$X,B$\cap$X))=c'(A_H,B_H)
end while

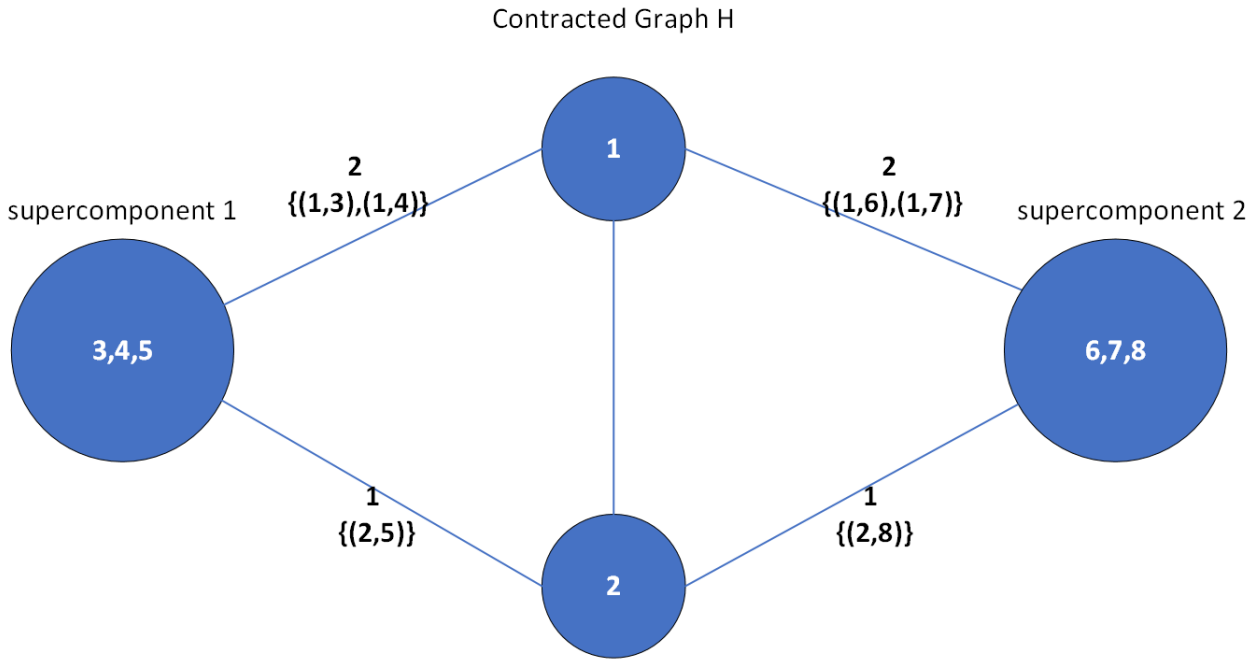
A core concept within the algorithm mentioned above is the idea of contraction. To enhance comprehension, an example of a contraction which creates a graph H, is presented below.



Assume we have a set X containing {1, 2}.



In this specific case, the edge (1, {3, 4, 5}) will be assigned a weight of 2. This weight is derived from the original graph, which contains two edges of weight 1 each (edge (1, 3) and edge (1, 4)), resulting in a combined weight of 2. Similarly, the edge (1, {6, 7, 8}) follows the same principle. On the other hand, the edges (2, {3, 4, 5}) and (2, {6, 7, 8}) will have a weight of 1 each.



The complexity of the Gomory-Hu algorithm depends on the method used to implement it. A commonly used implementation of the minimum cut is based on using maximum flow algorithms, such as the Ford-Fulkerson or Edmonds-Karp algorithm, as a subroutine. The Ford-Fulkerson algorithm has a time complexity of O(|E|F) as stated before. In the context of the Gomory-Hu algorithm, the maximum flow value is bounded by the "maximum max flow", F_{max}. So the time complexity of the Ford-Fulkerson-based Gomory-Hu algorithm is O(|V||E||Fmax).

2.4 The Minimum Steiner Cut problem

The Minimum Steiner Cut problem can be expressed as followed: Given a subset S⊆V, called the Steiner set, the minimum Steiner cut is the size of the smallest cut that splits S into two non-empty components. To calculate the Minimum Steiner cut, an efficient algorithm developed by Cole and Hariharan, refered as **CH algorithm** with a time complexity of $\tilde{O}(|E|+|V|^2k^3)$ (this notation ignores polylogarithmic factors) can be employed, specifically when the graph has undergone preprocessing using the **Nagamochi-Ibaraki (NI) algorithm** with a time complexity of O(|E|). However this algorithm does not yield interesting results for large k, for instance when $k = \Theta(n)$ [3].

2.5 The Nagamochi-Ibaraki (FOREST) algorithm

Lemma:*For a graph G=(V,E), simple or multiple, let F_i=(V,E_i) be a maximal spanning forest in G - $E_1 \cup E_2 \cup \ldots \cup E_{i-1}$ for i = 1,2,...,/E/, where possibly E_i=E_{i+1}=$E_{|E|}$=∅ for some i. Then each spanning subgraph G_i=(V,E_1∪E_2∪...∪E_i) satisfies λ(x,y;G_i)≥min{λ(x,y;G), i} for all x,y ∈ V, where λ(x,y;H) denotes the local graph connectivity between x and y in graph H.*

The Nagamochi-Ibaraki (FOREST) algorithm computes the subsets E_1, E_2,...,$E_{|E|}$ in a single scan

Algorithm 5 FOREST Algorithm (G)

```
set E1=E2=...=E|E|←∅
Label all nodes v∈V and all edges e∈E "unscanned"
r(V)←0
while there exists "unscanned" nodes do
    Choose an "unscanned" node x∈V with the largest r
    for each "unscanned" edge e incident to x do
        Er(y)+1=Er(y)+1∪{e}
        if r(x)==r(y) then
            r(x)=r(y)+1
        end if
        r(y)=r(y)+1
        Mark e as "scanned"
    end for
    Mark x as "scanned"
end while
```

The obtained graph G'(V',E') satisfies $|E'| \leq k|V| - k(k+1)/2 (\leq k(|V|-1))$. By using this FOREST as preprocessing, the time complexity of algorithms for solving other graph problems can be improved, as stated by Hiroshi Nagamochi and Toshihide Ibaraki in [4].

Chapter 3

Partial Gomory-Hu Tree

This chapter aims to provide a comprehensive explanation of the algorithm that constructs the partial Gomory-Hu tree, which serves as a solution to the "at most k-connectivity" problem, given an integer k.

Fact 3.1: *(Submodularity of cuts) If A and B are two subsets of vertices in a graph G and $\delta(X)$ represents the size of the cut $(X, V \setminus X)$, then $\delta(A) + \delta(B) \geq \delta(A \cap B) + \delta(A \cup B)$* Fact 3.1 can now help us understand theorem 3.1

Theorem 3.1: *If $(S, V \setminus S)$ is a minimum s-t cut in G and $u, v \in S$, then there exists a minimum u-v cut $(S^*, V \setminus S^*)$ such that $S^* \subset S$*

3.1 Introduction to the partial Gomory-Hu Tree

The Gomory-Hu tree construction algorithm commences by initializing the cut-tree T with a single node encompassing the entire vertex set. In each step of the algorithm, a node S in T (supervertex) containing more than one vertex is selected, and two vertices, s and t, are chosen from S. The process involves contracting the entire subtree formed by each neighbor of S into a single node. Then, a max flow computation is performed from s to t in this modified graph. According to Theorem 3.1, the resulting minimum s-t cut (denoted as C) is also a minimum s-t cut in the original graph.

Subsequently, in the tree T, the node S is split into two nodes, S1 and S2, based on the cut C. These two nodes are connected by an edge with a weight equal to the size of C. Additionally, all neighboring subtrees of S become neighboring subtrees of either S1 or S2, depending on their location with respect to the cut C. In classical Gomory-Hu trees, this process continues until all nodes in T become singleton sets. The final result is a weighted tree T, with its nodes representing the vertices of V. It is worth noting that T captures all-pairs min-cuts.

Nevertheless, for the specific purpose of the at most k-connectivity problem, constructing the entire Gomory-Hu tree becomes unnecessary. **Instead, a partial Gomory-Hu tree is constructed to selectively capture only those s-t min-cuts whose size is less than or equal to k. This optimization allows for a more efficient representation of the graph, focused solely on the relevant edges required for the at most k-connectivity analysis.**

To construct the Partial Gomory-Hu tree, a modification is required in the Gomory-Hu with contractions algorithm. Instead of utilizing the standard min cut algorithm, we shall employ a custom function. This function takes three inputs: the contracted graph H, the set X as the Steiner set, and an integer k. The objective is to determine whether the contracted graph H contains a minimum cut of value less than k.

If the minimum cut with a value less than k exists in the contracted graph H, the algorithm proceeds as usual. However, if no minimum cut with a value less than k is found, the supervertex currently being processed in the tree T will be included appropriately in the Tree T without further processing. Subsequently, the algorithm will move on to process the next supervertex of the Tree.

In summary, the modification ensures that the partial Gomory-Hu tree is created selectively based on the presence of minimum cuts with values less than k in the contracted graph H. If such cuts are identified, the algorithm continues to find additional relevant s-t min-cuts. Conversely, when no cuts meeting the condition are discovered, the algorithm directly incorporates the supervertex into the tree without additional processing and proceeds to the next step in the tree construction process.

Algorithm 6 Partial Gomory-Hu tree algorithm with parameter k

Initialize the tree T to a single supervertex, which contains the entire vertex set
Initialize the queue Q to the queue containing only the initial supervertex
while the Q is not empty **do**
    delete the first element of Q, call this element S
    delete the element from the Tree
    call the contraction algorithm and produce the graph H
    if there is a minimum cut with value c, with c $\leq$ k **then**
        let S1 and S2 be the two components that S is split into by the above cut; update T by splitting the supervertex S into supevertices S1 and S2 and introduce an edge of weight c between S1 and S2
        set the neighbors of S1 and S2 in the tree T appropriately
        insert the supervertices S1 (similarly S2) in the queue Q if S1 (similarly S2) contains more than one vertex.
    else
        reinsert the supervertex into the tree
    end if
end while

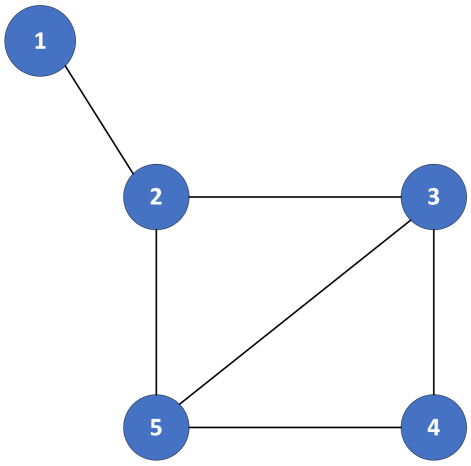
The notion of the partial Gomory-Hu tree introduces a significant advantage by selectively revealing edges exclusively from the Gomory-Hu tree that possess values lower than or equal to 'k'. When 'k' is small, the resulting tree is more compact, simplifying the process of determining the values for each s,t cut (where the cut value is smaller than 'k') within the graph.

When 'k' surpasses the highest value among the maximum minimum cuts, the partial Gomory-Hu tree becomes identical to the traditional Gomory-Hu tree.

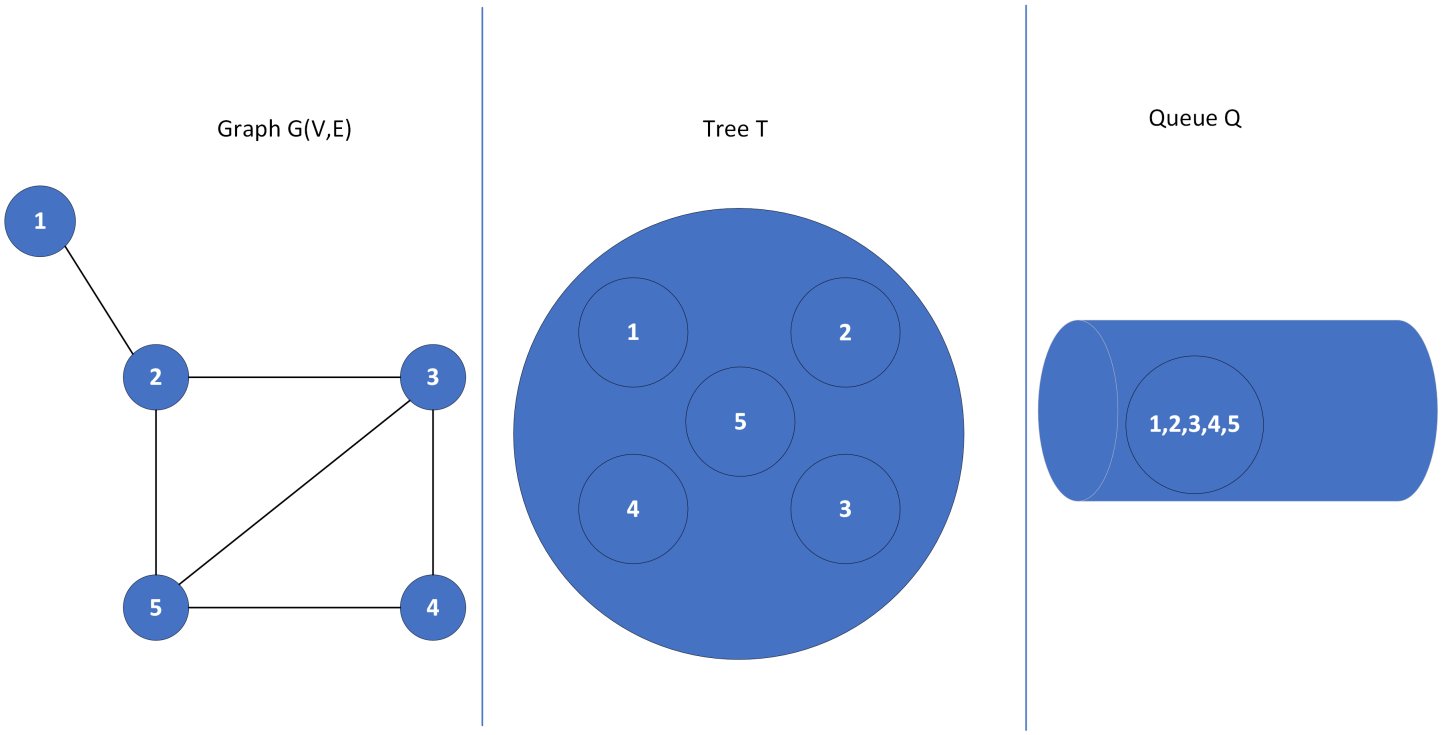
The next section will present a demonstration of the algorithm for different values of 'k'.

3.2 Showcasing the algorithm

Below is a depiction of a graph denoted as G(V,E) **(unweighted edges represent an edge of weight 1)**.



Considering the graph provided above, we will analyze the algorithm across various values of k with a very basic example. We will begin by examining the algorithm for the case where k=2.

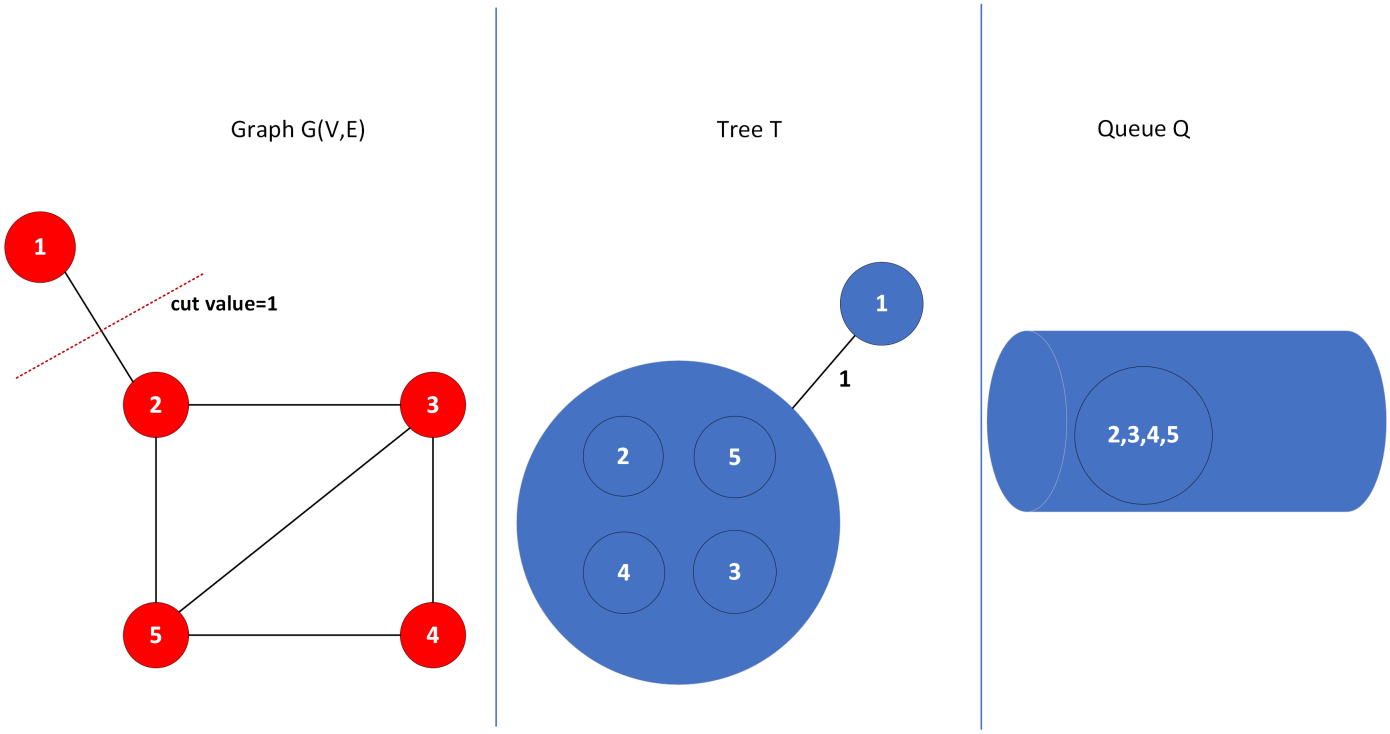


Initiating the algorithm, we initialize the Tree T into a single supervertex which contains all the vertices of the graph G, and the queue Q will be initialized to contain the single supervertex.

During the initial iteration, the vertices from the supervertex dequeued from the queue are chosen to form a set (X). A minimum cut is then determined between vertices 1 and 2, yielding a value of 1, which is indeed smaller than the given threshold 'k'. Consequently, the connected components emerge as A={2,3,4,5} and B={1}. As a result, the new supervertices for Tree T are computed as the intersection of A and X, denoted as A∩X, and the intersection of B and X, denoted as B∩X. These

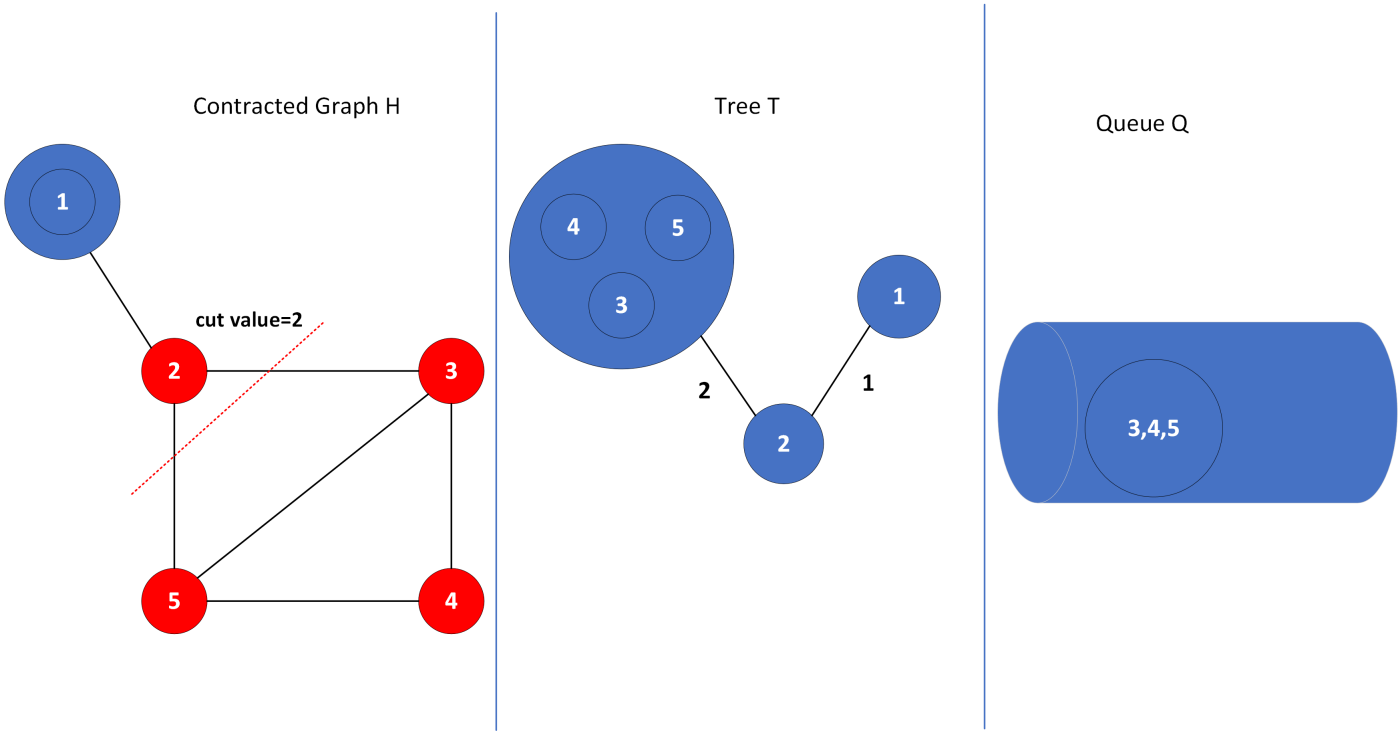
two supervertices are interconnected by an edge with a weight of 1.

Among the newly formed supervertices, supervertex1 containing vertices {2,3,4,5} and supervertex2 containing vertex {2} are under consideration. Supervertex1 is enqueued into Q since it contains multiple vertices, leading to its incorporation into the tree as well. In contrast, supervertex2 is added to the tree but is not included in the queue due to containing only a single vertex.



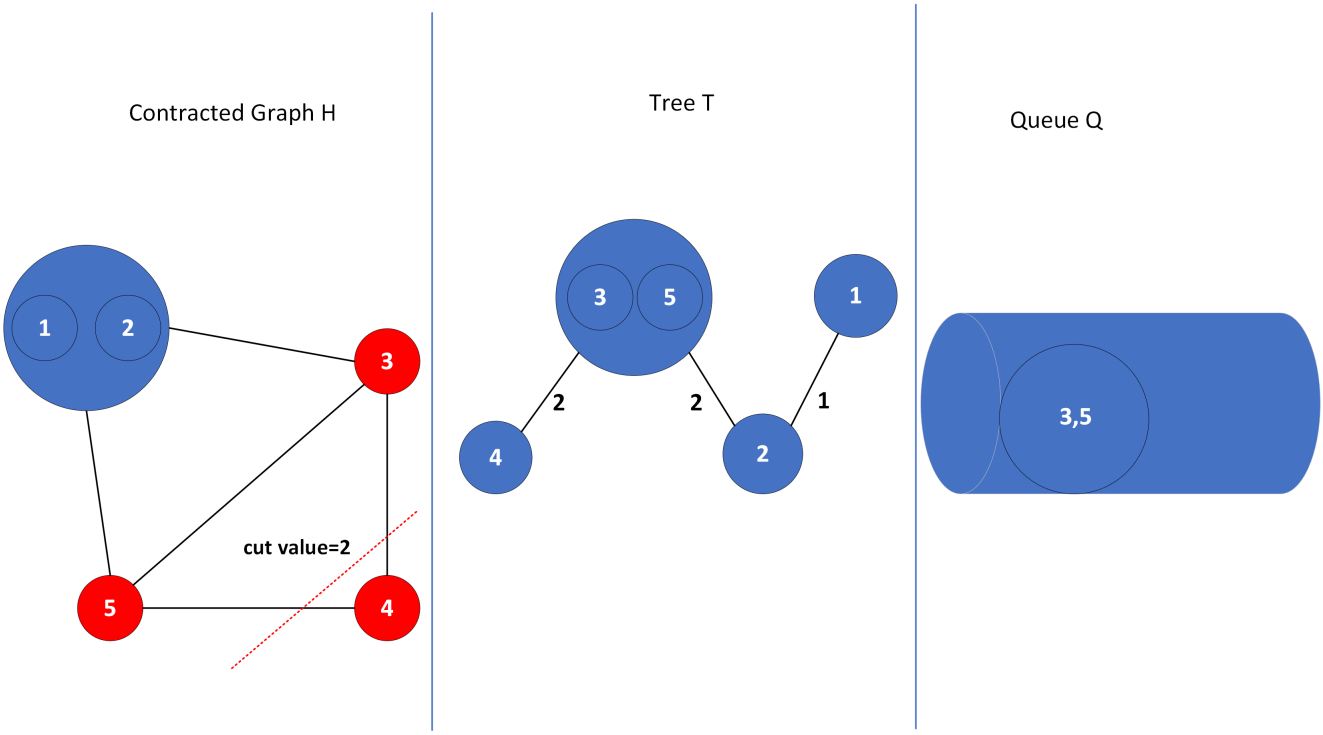
In the second iteration, our initial step involves dequeuing the foremost supervertex from the queue. Subsequently, we eliminate this supervertex from the Tree. The vertices within the residual connected components serve as the new supercomponents for the contracted graph. Alternatively, we can observe that the vertices extracted from the original graph G also establish the supercomponents of the new contracted graph. For instance, in this scenario, vertex 1 forms a supercomponent within the recently contracted graph. The collection of vertices within the chosen supervertex becomes our newly defined set X.

Following this, a minimum cut with a value of 2 is discovered between vertices 2 and 3. Since the cut value matches 'k', the algorithm proceeds to compute the updated A∩X and B∩X sets, positioning them suitably within the tree structure. Subsequently, two new supervertices emerge: supervertex1, which includes vertices 3, 4, and 5, and supervertex2, which includes vertex 2. While supervertex1 is added to the queue, supervertex2 remains unqueued due to having only one element.

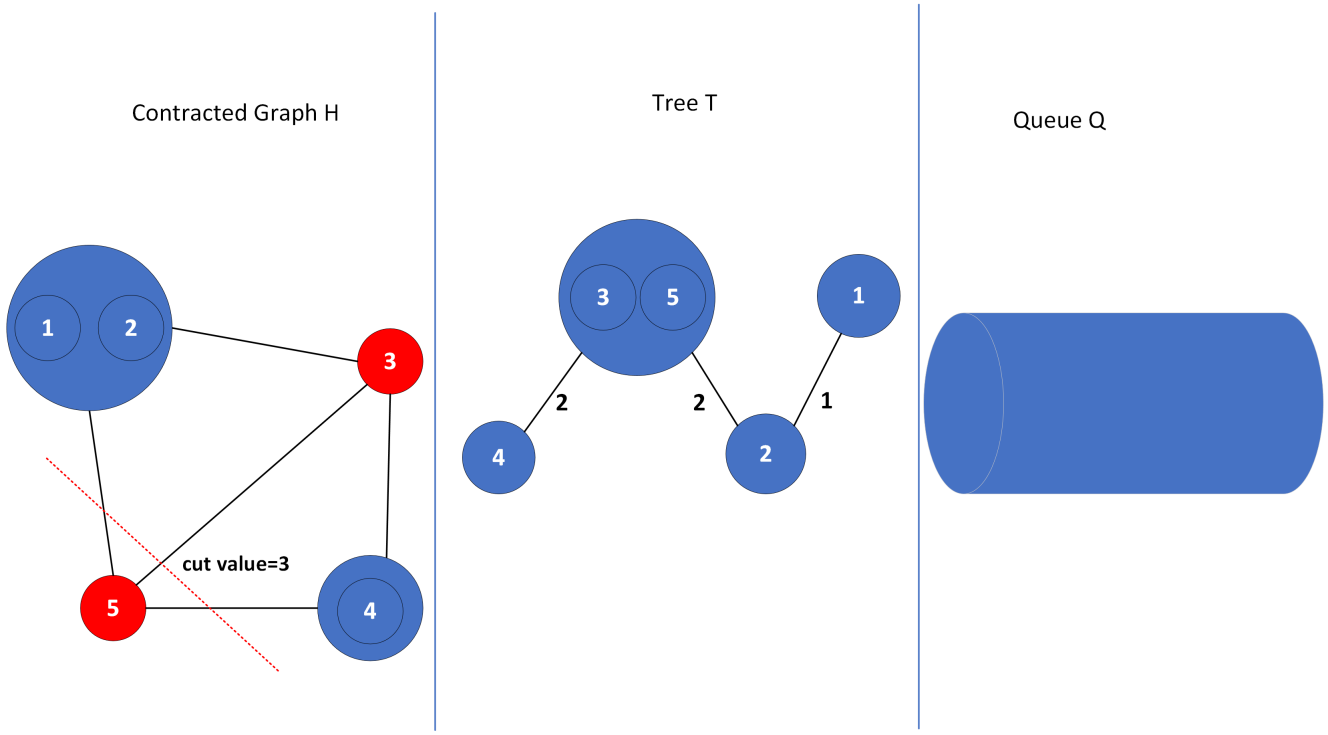


Moving on to the third iteration, we once more dequeue the initial element from Q, establishing X as the set {3, 4, 5}. Subsequently, we compute the new contracted graph H, followed by determining the minimum cut between vertices 3 and 4 within H. This cut has a value of 2, aligning precisely with 'k' so the cut can proceed.

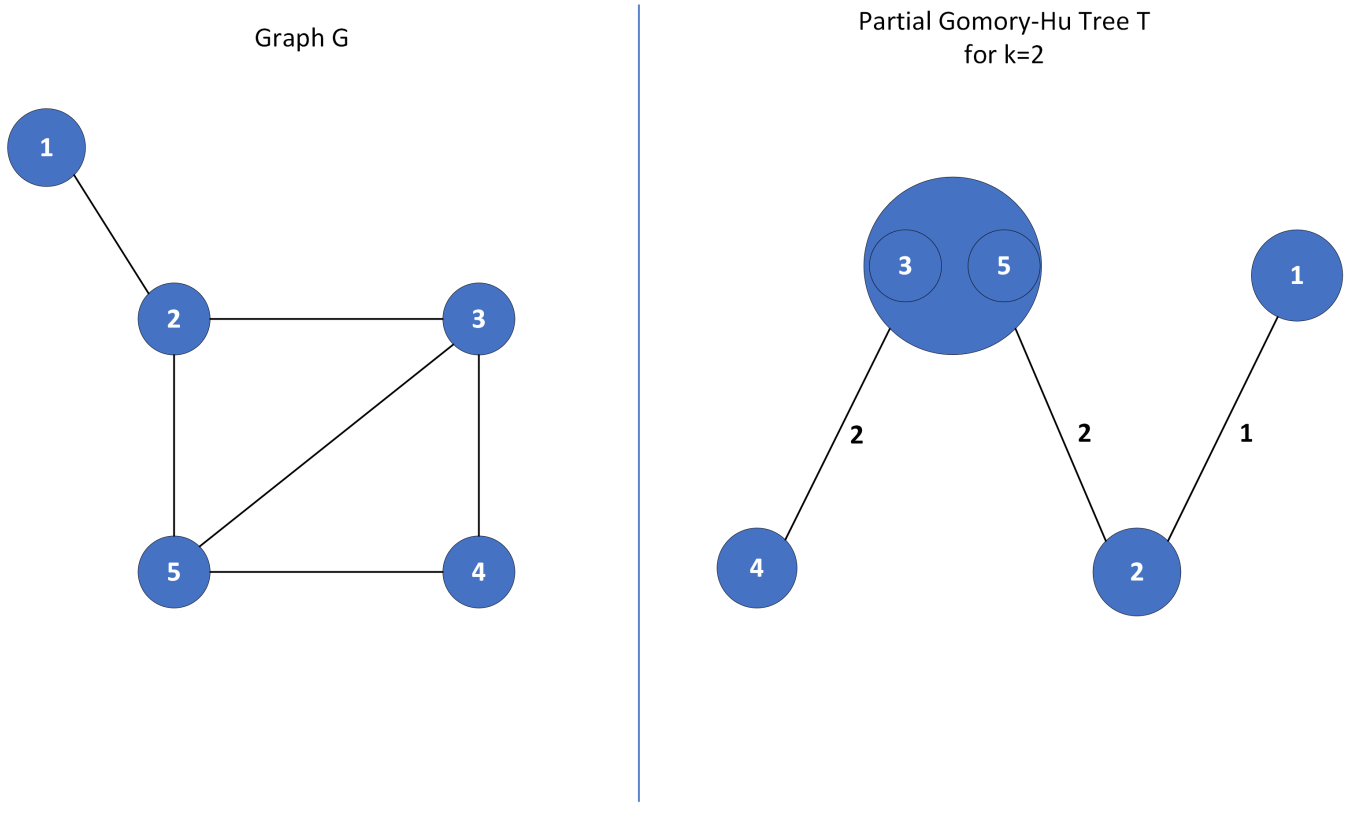
Subsequently, we proceed to compute the updated sets A∩X and B∩X. This leads us to create the two new supervertices, supervertex1 which includes vertices 3, 5 and supervertex2 which includes 4. Since supervertex2 consists of only one element, it is not included in the queue for enqueuing. Finally we add the supervertices to the tree appropriately we an edge of weight 2.



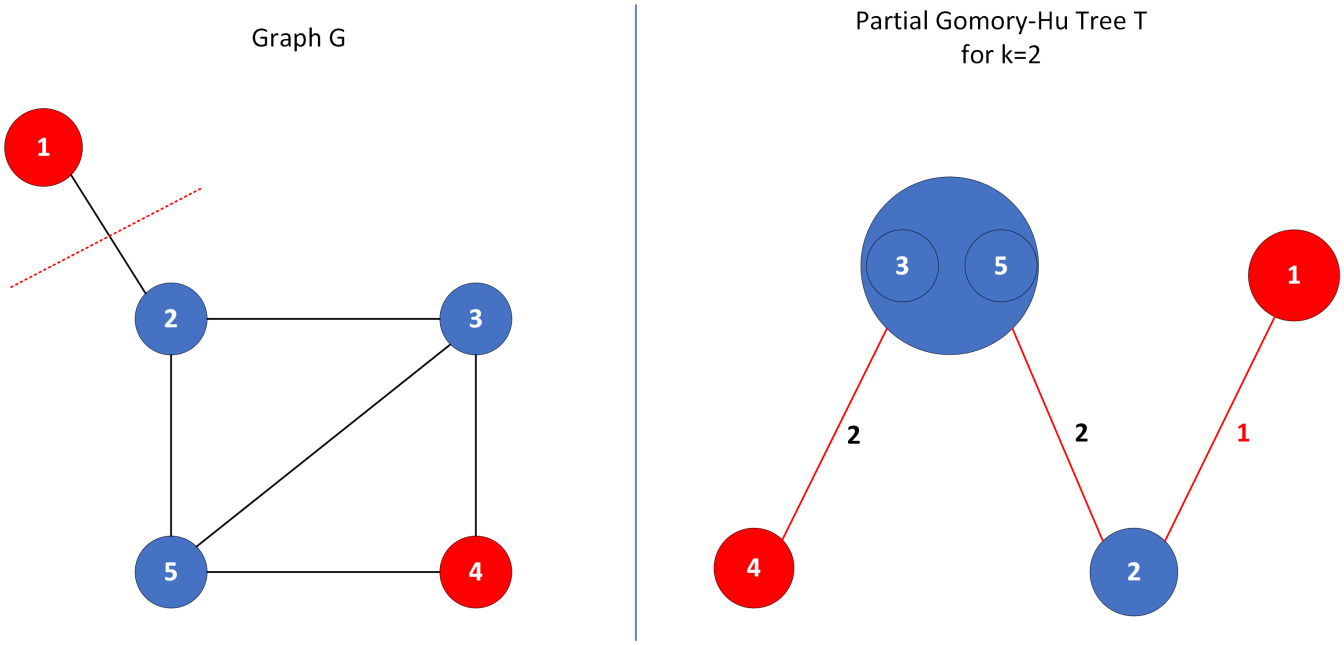
Moving to the final iteration, we dequeue the final supervertex, compute X as 3, 5, and generate the revised contracted graph H. Within H, we evaluate the lone remaining cut, situated between vertices 3 and 5, with a value of 3. Since this value exceeds 'k', the cut cannot progress. Consequently, the supervertex is once again inserted into the Tree, and no elements are enqueued into Q.



With the absence of additional elements in Q, the algorithm terminates and delivers the partial Gomory-Hu tree, depicted below for 'k'=2.

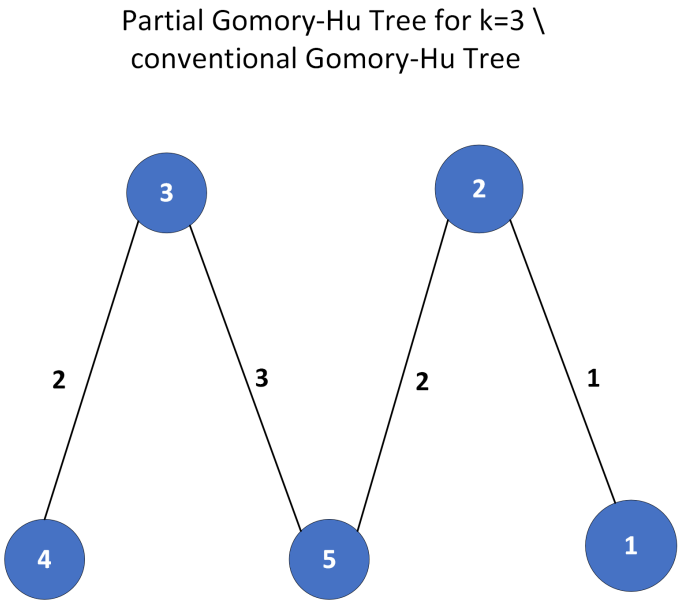


Now, we can begin to examine certain attributes of Gomory-Hu trees. For instance, in the following graph, the minimum cut between vertices 1 and 4 holds a value of 1. This can be verified by identifying the lowest weight along the path connecting vertex 1 to vertex 4, which also turns out to be 1.



Furthermore, it's evident that the Partial Gomory-Hu tree effectively addresses "the at most k connectivity" problem for k=2. This is apparent from our observation that the tree provides the connectivities that are at most 2 for all pairs of vertices within the original graph. When considering the scenario where k=3, the cut involving vertices 3 and 5 becomes relevant. Consequently, vertices 3 and 5 are incorporated into the tree, connected by an edge with a weight of 3. At this point, the partial Gomory-Hu tree transforms into an identical representation of a conventional Gomory-Hu tree

Generally if the value of 'k' is equal or greater than the value of the greatest cut in the graph, the Partial Gomory-Hu becomes identical to the conventional Tree.



3.3 A more efficient algorithm

In this section we would like to mention one more algorithm.This efficient algorithm has been proposed by Ramesh Hariharan, Telikepalli Kavitha and Debmalya Panigrahi for the computation of Partial Gomory-Hu trees using the CH (Cole-Hariharan) algorithm to calcucate the minimum Steiner cuts instead of calculating minimum cuts. The CH algorithm comprises two steps: the first step involves the preprocessing method of Nagamochi-Ibaraki, followed by the utilization of a modified version of Gabow's algorithm. The simpler of the two algorithms mentioned in the paper[1] can be seen below with a time complexity of $\tilde{O}(|E|+|V|^2k^3)$ (this notation ignores polylogarithmic factors).

Algorithm 7 An algorithm for partial Gomory-Hu tree with parameter k for graph G=(V,E)

Initialize the tree T to a single node, which is the entire vertex set V
Initialize the queue Q to the queue containing only one element, which is the set V
while the queue Q is not empty **do**
    delete the first element of Q, call this element S
    call the minimum Steiner cut algorithm with the set S as the Steiner set
    if the value, c, of this minimum Steiner cut is $\leq$ k **then**
        let S1 and S2 be the two components that S is split into by the above cut; update T by splitting the node S into nodes S1 and S2 and introduce an edge of weight c between S1 and S2
        set the neighbors of S1 and S2 in the tree T appropriately
        insert the node S1 (similarly, S2) in the queue Q if S1 (resp., S2) contains more than one vertex
    end if
end while

Algorithm 7 does not use the compaction routine as stated above. Instead, it uses the CH algorithm to calculate the minimum Steiner cut for the provided Steiner set. In our context, this set S comprises the vertices contained within the supervertices.

The CH algorithm is a constructive proof of the following theorem:
Given an Eulerian directed graph G and any vertex r, there exist k edge disjoint directionless trees T_1, T_2, ..., T_k rooted at r such that each vertex v in G appears exactly con(v) times over all the trees and has in-degree exactly con(v) over these trees, where con(v) is the edge connectivity of r to v and $k = max_{u \neq v} \{con(v)\}$.

More details about the CH algorithm appears in the cited reference [3].

Chapter 4

Implementation in Python

Within this chapter, our focus will be directed towards exploring the code implementation of the partial Gomory-Hu algorithm using contractions. This algorithm will rely on two fundamental data structures. The compacted graph H will be represented using an adjacency matrix, while the Partial Gomory-Hu tree will be represented using an adjacency list.

4.1 Implementation of min cut

To begin, we will develop a script that reads a text file containing the graph's edges and constructs an adjacency matrix.

```
def create_graph(filename):
    with open(filename) as f:
        edges = []
        lines = f.readlines()
        for line in lines[1:]:
            parts = line.strip().split()
            edges.append([int(parts[0])-1, int(parts[1])-1])
    size = len(set([n for e in edges for n in e]))
    adjacency = [[0] * size for _ in range(size)]
    for sink, source in edges:
        adjacency[sink][source] = 1
        adjacency[source][sink]=1
    return adjacency
```

Listing 4.1: Create inital graph

To initialize, we will generate a script capable of calculating the minimum cut of a graph, using two specified input nodes and the graph itself (source node, sink node, adjacency matrix).

The following code segment implements the minimum cut through the utilization of the Ford-Fulkerson algorithm. This algorithm necessitates the employment of the Breadth-First Search (BFS) algorithm to identify augmenting paths during each iteration. Additionally, the Depth-First Search (DFS) algorithm is essential to identify the edges that are cut by the first part of the algorithm. This code segment will play a crucial role in the main algorithm.

```
def bfs(graph, s, t,parent):
    visited = [False]*len(graph)
    queue = []
    queue.append(s)
    visited[s] = True
    while queue:
        node = queue.pop(0)
        for ind, val in enumerate(graph[node]):
            if visited[ind] == False and val > 0:
                queue.append(ind)
                visited[ind] = True
                parent[ind] = node
                if ind == t:
                    return True
    return False


def dfs(graph, s, visited,mark):
        visited[s] = mark
        for i in range(len(graph)):
            if graph[s][i] > 0 and visited[i]!=mark:
                dfs(graph, i, visited,mark)


def find_cuts(org_graph,visited,symbol):
    min_cuts=[]
    for i in range(len(visited)):
        if (visited[i]=="F"):
           for j in range(len(org_graph[i])):
               if (org_graph[i][j]>0 and visited[j]==symbol):
                   min_cuts.append([i,j])
    return min_cuts


def minCut(graph, source, sink):
    original_graph = [i[:] for i in graph]
    parent = [-1] * (len(graph))
    max_flow = 0
    while bfs(graph,source, sink, parent):
        path_flow = float("Inf")
        s = sink
        while (s != source):
            path_flow = min(path_flow,graph[parent[s]][s])
            s = parent[s]
        max_flow += path_flow
        v = sink
        while (v != source):
            u = parent[v]
            graph[u][v] -= path_flow
            graph[v][u] += path_flow
            v = parent[v]
    visited=len(graph) * ["F"]
    dfs(graph,source,visited,"S") #border of source side
    mincut=find_cuts(original_graph,visited,"S")
    return [mincut,max_flow]
```

Listing 4.2: Minimum cut

The algorithm commences by identifying an augmenting path from the source node to the sink node through Breadth-First Search (BFS). Subsequently, it updates the capacities of the edges by pushing flow towards the sink node. Once the sink node becomes unreachable, the initial phase of the algorithm concludes, yielding the maximum flow/minimum cut value. The next challenge involves identifying the edges that have been cut. To address this, we employ the Depth-First Search (DFS) algorithm, as previously mentioned, to label all edges reachable from the source node. Following this labeling, we pinpoint the edges linking marked nodes to unmarked ones. These identified edges constitute our set of minimum cut edges.

4.2 Implementation of other utility algorithms

Within this section, we will explore the implementation of additional utility algorithms, including the implementation of the connected components algorithm and a pseudo Steiner cut routine.

```
def connected_components(graph):
    n = len(graph)
    visited = set()
    components = []

    def dfs(node, component):
        visited.add(node)
        component.append(node)
        for neighbour in range(n):
            if graph[node][neighbour] and neighbour not in visited:
                dfs(neighbour, component)

    for node in range(n):
        if node not in visited:
            component = []
            dfs(node, component)
            components.append(component)

    return components


def remove_vertex(graph,vertex):
    for i in range(len(graph)):
        for j in range(len(graph)):
            if i==vertex and graph[i][j]==1:
                graph[i][j]=0
            if j==vertex and graph[i][j]==1:
                graph[i][j]=0
    return graph

def pseudo_steinerCut(graph,steinerSet,k):
    s_cut=None
    cut_value=None
    for i in range(len(steinerSet)):
        for j in range(i, len(steinerSet)):
            if (i != j):
                s_cut, cut_value = min_cut.minCut(deepcopy(graph), steinerSet[i],
     steinerSet[j])
```

```
            if cut_value<=k:
                return s_cut,cut_value
    return None,None
```

Listing 4.3: Utilities

The connected components routine accepts a graph as input and generates a list containing the connected components in the form of lists. It accomplishes this task by utilizing the Depth-First Search (DFS) algorithm. For each unvisited node, the routine employs the DFS algorithm to identify the connected component. This process continues until all nodes are visited, and the resultant list encompasses all the identified components.

The pseudo Steiner cut algorithm presents a comparatively less efficient approach to implementing the minimum Steiner cut algorithm. Given an integer 'k' and a set 'S', this algorithm identifies a minimum cut with a value below 'k' among the nodes within 'S'. This ensures that the cut remains viable. Notably, the algorithm refrains from returning any cuts in cases where no suitable cuts can be found.

The remaining "remove vertex" routine is a straightforward procedure. It accepts a vertex and an adjacency matrix as inputs. The routine locates the specified vertex and eliminates its associated edges from the list. This function will be employed within the contraction function at a later stage.

4.3 Implementation of the contraction algorithm

Within this chapter, our focus will be directed towards the primary algorithm. Given the extensive nature of the implementation, we will divide it into distinct sections for comprehensive coverage. Firstly, we will discuss the concept of the "supervertex." A supervertex is an object that has four distinct attributes. These attributes include an integer "id" (each supervertex will have a unique id), a list of vertices, another list named "neighbours" to account for adjacent vertices, and a third list named "connectivity." The "connectivity" list maintains a one-to-one correspondence with the "neighbours" attribute and has the weights that establish connections between supervertices. The object also possesses a "to string" function, which displays the attribute values when called.

```
class SuperVertex:

    def __init__(self, id, vertices=None, neighbours=None,connectivity=None):
        self.id = id
        self.vertices = vertices if vertices else []
        self.neighbours = neighbours if neighbours else []
        self.connectivity = connectivity if connectivity else []

    def __str__(self):
        return str(f"{self.id}, {self.vertices}, {self.neighbours}, {self.
    connectivity}")
```

Listing 4.4: Utilities

Subsequently, we will delve into the function responsible for implementing the contraction algorithm. This function accepts a graph and the dequeued supervertex,

as previously discussed, as its inputs. The algorithm proceeds by extracting the vertices from the supervertex to form the set X, as outlined earlier. Following this, the function employs the "remove vertex" function to eliminate the vertices within X from the graph. Utilizing the connected components routine, it determines the connected components of the modified new graph. The function utilizes the supervertex objects to construct the corresponding supercomponents, integral to the construction of the contraction graph. This newly created graph incorporates the supercomponents along with the vertices of X. The weights of the edges linking the supercomponents with the vertices of X are then calculated. Ultimately, the function returns the resultant graph, referred to as graph H.

The algorithm is captured by the code below.

```
def contract(graph, superVertex: SuperVertex):
    cpGraph = deepcopy(graph)
    for v in superVertex.vertices:
        cpGraph = utilites.remove_vertex(cpGraph, v)
    connectedComponents = utilites.connected_components(cpGraph)
    connectedComponents = [connectedComponent for connectedComponent in
    connectedComponents if connectedComponent[0] not in superVertex.vertices]
    n = len(superVertex.vertices) + len(connectedComponents)
    newGraph = [[0]*n for _ in range(n)]
    superVertexList = []

    for v in superVertex.vertices:
        superVertexList.append(v)

    for connectedComponent in connectedComponents:
        newSuperVertex = SuperVertex(id=len(superVertexList), vertices=
    connectedComponent, neighbours=[], connectivity=0)
        totalNeighbours = []
        totalConnectivity = []
        for v1 in superVertex.vertices:
            neighbours = []
            for v2 in connectedComponent:
                if graph[v1][v2] == 1:
                    neighbours.append(v1)
            totalConnectivity.append(len(neighbours))
            totalNeighbours.append(v1)
        newSuperVertex.neighbours = totalNeighbours
        newSuperVertex.connectivity = totalConnectivity
        superVertexList.append(newSuperVertex)


    for vertex_id, vertex in enumerate(superVertexList):
        if isinstance(vertex, SuperVertex):
            i = vertex.id
            for neighbour, connectivity in zip(vertex.neighbours, vertex.
    connectivity):
                newGraph[i][superVertexList.index(neighbour)] = connectivity
                newGraph[superVertexList.index(neighbour)][i] = connectivity
        else:
            for v_id, v in enumerate(superVertexList):
                if not isinstance(v, SuperVertex):
                    newGraph[vertex_id][v_id] = graph[vertex][v]
                    newGraph[v_id][vertex_id] = graph[v][vertex]
```

```
steinerlist=[]
for vertex in range(len(superVertex.vertices)):
    steinerlist.append(superVertexList.index(superVertex.vertices[vertex]))
return newGraph,steinerlist,superVertexList
```

Listing 4.5: Contraction

4.4 Implementation of the Partial Gomory-Hu Tree algorithm

In this chapter, we will address the core of our approach: the Partial Gomory-Hu tree algorithm. To begin, a counter be initialized to 1 and we will establish an initial supervertex named "supervertex0." This supervertex will carry an id of the value of the counter and encompass all the vertices within the graph. Naturally, during initialization, both the "neighbours" and "connectivity" lists will remain empty.

Subsequently, supervertex0 will be enqueued into the queue Q, while the partial Gomory-Hu tree T, adhering to the earlier discussion, will be represented as an adjacency list. At the initiation stage, this adjacency list will remain empty.

In the initial iteration, the contraction algorithm is unnecessary since the initial graph remains uncontracted. To identify the first iteration, we can verify if the counter equals 1. In this initial iteration, our objective is to compute a minimum cut with a value lower than 'k'. Upon locating a suitable cut separating the graph in two components, component A and component B, two supervertices named supervertex1 and supervertex2 will be established.

In this first iteration, supervertex1 will be assigned an id equal to the counter value (it will inherit id of the father supervertex) and will contain the vertices of A∩X. Conversely, supervertex2 will receive an id of counter+1 and will contain the vertices of B∩X. A∩X and B∩X can be calculated by the intersec function which calculates and returns the intersection of two lists. Finally If the size of the newly formed supervertices is greater than 1, these supervertices will be enqueued and placed into the Tree connected to each other.

When not in the initial iteration, the process becomes more intricate. Firstly, the supervertex (our new supervertex0) must be dequeued from Q and removed from Tree T. Subsequently, the contraction algorithm is called to return the contracted graph H. Following this, the algorithm searches for a minimum cut with a value below k. In the absence of such a cut, the supervertex will be reinserted into the tree; however, it won't be enqueued back into Q.

If a cut with a value lower than k is present, two new supervertices will be generated. These supervertices will, again, adopt distinct ids: supervertex1 will take on the id of the parent supervertex, while supervertex2 will be assigned an id of count+1.

The next challenge involves effectively establishing connections between the new supervertices and the tree. We now need to address three distinct cases.

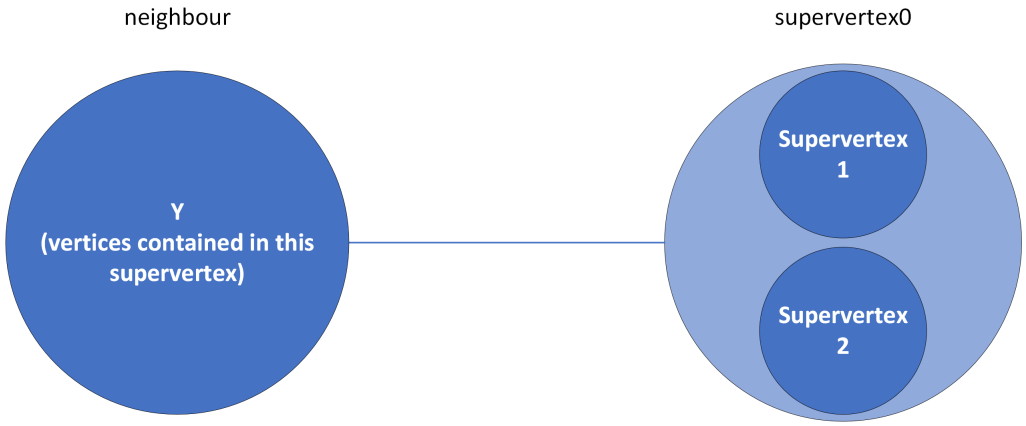
To determine the suitable neighbours for the newly formed supervertices (supervertex1 and supervertex2) originating from supervertex0, we firstly have to state that supervertex1 will include the vertices of set A∩X, while supervertex2 will incorporate the vertices of set B∩X. Now, considering the vertices of each supervertex

adjacent to supervertex0 (which we'll denote as set Y), we need to assess the following "if-else statement".

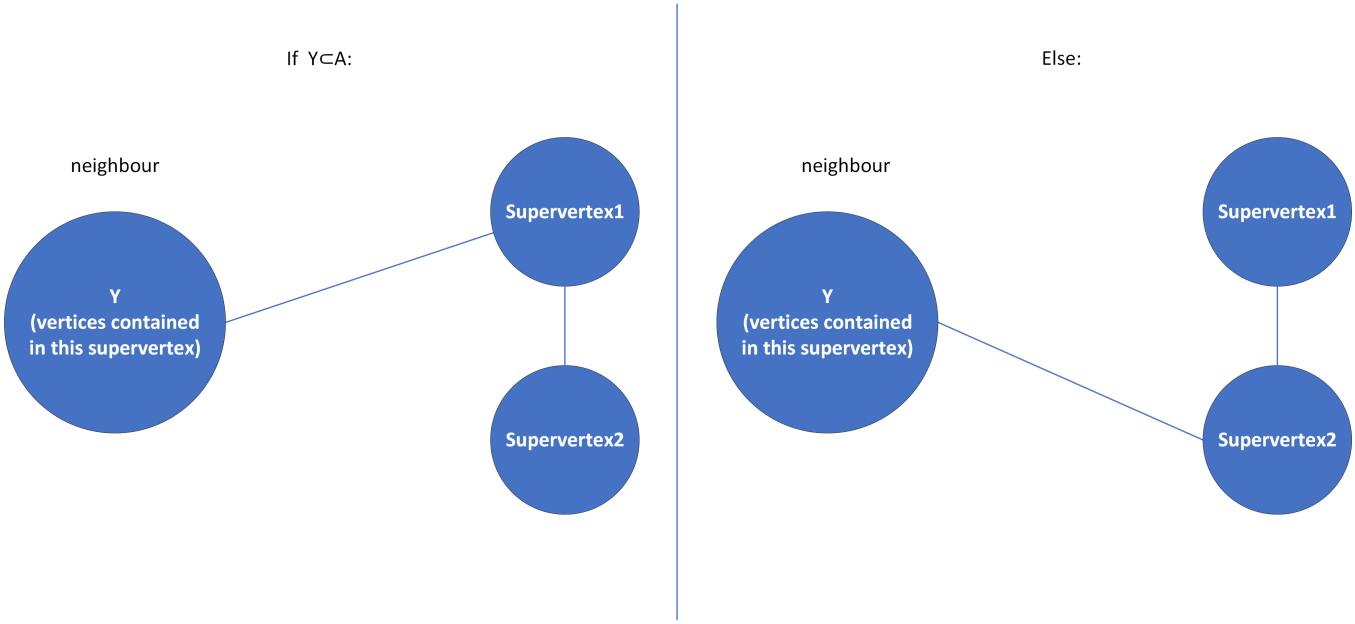
If Y is a subset proper of A ($Y \subset A$), we establish a connection between supervertex1 and the neighbouring supervertex that includes Y. This connection is assigned a weight equivalent to the connection between supervertex0 and the aforementioned neighbouring supervertex. Else we establish a connection between supervertex2 and the neighbouring supervertex. This connection bears a weight matching the one between supervertex0 and the said neighbouring supervertex.

Lastly, we connect supervertex1 and supervertex2 with an weighted edge equal to the value of the minimum cut.

As an illustration, during our iteration through the neighbours of supervertex0, we are inspecting the supervertex labeled "neighbour," which contains the vertex set Y:



We will now proceed to examine the previously mentioned if-else statement.



Lastly, if a newly created supervertex contains more than one vertex, it is enqueued as well, excluding its placement in the tree.

The provided code includes all the previously discussed aspects and is supplemented by supporting functions that aid in locating and removing supervertices based on their respective ids within the Tree.

```
def intersect(A, X):
    I = []
    for item in A:
        items = item if isinstance(item, list) else [item]
        I += [item for item in items if item in X]
    return I

def find_supervertices(T,in_id):
    for t in T:
        if t.id==in_id:
            return t.vertices

def connect(T,neigh,id,conn):
    for t in T:
        if t.id==neigh:
            t.neighbours.append(id)
            t.connectivity.append(conn)

def setup(graph):
    Vset=[]
    for i in range(len(graph)):
        Vset.append(i)
    return Vset

def delete_supervertex(T,target:SuperVertex):
    for neigh in target.neighbours:
        for t in T:
            if (t.id==neigh):
                for i,neighbour in enumerate(t.neighbours):
                    if neighbour==target.id:
                        t.neighbours.remove(target.id)
                        del t.connectivity[i]
    return T

def check_subset(listA, listB):
    setB = set(listB)
    for element in listA:
        if element in setB:
            return True
    return False

def del_Tpid(T,id): #delete instance of supervertex from every neighbour
    for t in T:
        if id in t.neighbours:
            t.neighbours.remove(id)

def partialGH1(graph,k):
    Vset=setup(graph) #vertex set
    count=1 #supervertex id counter
    superVertex0=SuperVertex(id=count, vertices=Vset, neighbours=[], connectivity
    =[]) #first cell
    T=[superVertex0] #GM tree
    Q=[superVertex0] #init Q
    while Q: #while Q not empty
        superVertex0=Q.pop(0) #pop first elem
        X=superVertex0.vertices #get vertices of supervertex
        cpgraph=deepcopy(graph)
```

```
    if len(X)==len(graph):
        T.remove(superVertex0)
        H=deepcopy(graph)
        cut,cutValue=utilites.pseudo_steinerCut(H,X,k)
        if cutValue!=None:
            for edge in cut:
                H[edge[0]][edge[1]] = 0  # cut the edges
                H[edge[1]][edge[0]] = 0  # cut the edges
            cc = utilites.connected_components(H)
            A = cc[0]
            B = cc[1]
            A_intersec_X = intersect(A, X)
            B_intersec_X = intersect(B, X)
            superVertex1 = SuperVertex(id=count+1, vertices=B_intersec_X,
neighbours=[superVertex0.id],connectivity=[cutValue])
            count+=1
            superVertex2 = SuperVertex(id=superVertex0.id, vertices=
A_intersec_X,neighbours=superVertex0.neighbours + [superVertex1.id],
connectivity=superVertex0.connectivity + [cutValue])
            T.append(superVertex1)
            T.append(superVertex2)
            if len(superVertex1.vertices) > 1:
                Q.append(superVertex1)
            if len(superVertex2.vertices) > 1:
                Q.append(superVertex2)
    else:
        H = deepcopy(graph)
        newGraph, steinerlist,superVertexList=contract(H,superVertex0)
        cut, cutValue = utilites.pseudo_steinerCut(newGraph, steinerlist,k)
        if cutValue!=None:
            T.remove(superVertex0)
            delete_supervertex(T, superVertex0)
            for edge in cut:
                newGraph[edge[0]][edge[1]] = 0  # cut the edges
                newGraph[edge[1]][edge[0]] = 0  # cut the edges
            A, B = utilites.connected_components(newGraph)
            A = [superVertexList[a] for a in A]
            B = [superVertexList[b] for b in B]
            temp = []
            for sv in A:
                temp += sv.vertices if isinstance(sv, SuperVertex) else [sv]
            A = temp
            temp = []
            for sv in B:
                temp += sv.vertices if isinstance(sv, SuperVertex) else [sv]
            B = temp
            A_intersec_X = intersect(A, X)
            B_intersec_X = intersect(B, X)
            superVertex1=SuperVertex(id=-1, vertices=[], neighbours=[],
connectivity=[])
            superVertex2 = SuperVertex(id=-1, vertices=[], neighbours=[],
connectivity=[])

            for i,supervertex_neighbour in enumerate(superVertex0.neighbours)
:
                neighbour_vertices = find_supervertexes(T,
supervertex_neighbour)
                if check_subset(neighbour_vertices,A)==True:
```

```
                    superVertex1 = SuperVertex(id=superVertex0.id, vertices=
    A_intersec_X, neighbours=superVertex1.neighbours+[supervertex_neighbour],
    connectivity=superVertex1.connectivity+[superVertex0.connectivity[i]])
                    connect(T, supervertex_neighbour, superVertex1.id,
    superVertex0.connectivity[i])
                else:
                    superVertex2 = SuperVertex(id=count+1, vertices=
    B_intersec_X, neighbours=superVertex2.neighbours+[supervertex_neighbour],
    connectivity=superVertex2.connectivity+[superVertex0.connectivity[i]])
                    connect(T, supervertex_neighbour, superVertex2.id,
    superVertex0.connectivity[i])
            if (superVertex1.id == -1):
                superVertex1 = SuperVertex(id=superVertex0.id, vertices=
    A_intersec_X, neighbours=[], connectivity=[])
            if (superVertex2.id == -1):
                superVertex2 = SuperVertex(id=count+1, vertices=B_intersec_X,
     neighbours=[], connectivity=[])
            count+=1
            superVertex1.neighbours.append(superVertex2.id)
            superVertex1.connectivity.append(cutValue)
            superVertex2.neighbours.append(superVertex1.id)
            superVertex2.connectivity.append(cutValue)
            T += [superVertex1, superVertex2]
            if len(superVertex1.vertices) > 1:
                Q.append(superVertex1)
            if len(superVertex2.vertices) > 1:
                Q.append(superVertex2)
    return T

def format(T):
    for t in T:
        for i in range(len(t.vertices)):
            t.vertices[i]=t.vertices[i]+1
    return T
```

Listing 4.6: Partial Gomory Hu

The ”setup” function increments the ids of each vertex by +1. This adjustment is necessary as the vertex ids will be decreased by 1 when loaded into the adjacency matrix through the ”create graph” routine and its sole purpose is for visualization purposes.

4.5 Implementation of the Nagamochi-Ibaraki algorithm

In the concluding segment of this chapter, we will discuss the implementation of the Nagamochi-Ibaraki algorithm. This algorithm serves as the preprocessing step for the partial Gomory-Hu Tree, as outlined in [1], and its functioning aligns with the concepts discussed in Chapter 2, section 2.6 .

```
def preprocess(graph):
    countE=0
    NI_markedV=[] #label unmarked vertices
    NI_markedE=deepcopy(graph) #label unmarked edges
    for i in range(len(NI_markedE)):
```

```
        for j in range(len(NI_markedE)):
            if NI_markedE[i][j]==1:
                NI_markedE[i][j]="UE" #unvisited edge
                countE+=1
    for k in range(len(graph)):
        NI_markedV.append("UV") #unvisited vertex
    E = [[] for _ in range(countE//2)]
    return [NI_markedE,NI_markedV,E]


def choose(NI_vertices,r_list):
    dummy_lst=[0]*len(NI_vertices)
    for i in range(len(NI_vertices)):
        if NI_vertices[i]=="UV":
            dummy_lst[i]=r_list[i]
        else:
            dummy_lst[i]=0
    res=dummy_lst.index(max(dummy_lst))
    return res

def FOREST(NI_graph,NI_vertices,E):
    r_list=[0]*len(NI_vertices)
    while ("UV" in NI_vertices):
            v=choose(NI_vertices,r_list)#index of vertex with largest r
            for ind in range(len(NI_graph[v])):
                if NI_graph[v][ind]=="UE":
                    E[r_list[ind]+1].append([v,ind])
                    if r_list[v]==r_list[ind]:
                        r_list[v]+=1 #r(x)=r(x)+1
                    r_list[ind]+=1 #r(y)=r(y)+1
                    NI_graph[v][ind]=0 #unmark edge
                    NI_graph[ind][v]=0 #unmark edge
            NI_vertices[v]=0
    forest = [ele for ele in E if ele != []] #filter empty lists
    return forest

def format(graph):
    NI_graph,NI_vertices,E=preprocess(graph)
    res = FOREST(NI_graph, NI_vertices, E)
    for i in range(len(res)):
        for j in range(len(res[i])):
            for k in range(len(res[i][j])):
                res[i][j][k]+=1
    return res
```

Listing 4.7: Nagamochi Ibaraki

Once more, the setup function aids us by increasing the id of each vertex by +1 due to the create graph function. The algorithm's resulting trees can be observed through a list referred to as the forest, which contains nested lists representing individual trees. These nested lists store the edges of the respective trees.

Chapter 5

Results and Visualization

5.1 Visualization with Python and NetworkX

Initially, we will explore the code responsible for visualizing the initial graph G and the Partial Gomory-Hu tree. For this purpose, we will utilize the "NetworkX" external library.

```
def find_supervertex(T,in_id):
    for t in T:
        if t.id==in_id:
            return t.vertices

def visualiazeTree(T):
    G=nx.Graph()
    for t in T:
        for i in range(len(t.neighbours)):
            str_vertices=str(t.vertices)
            str_neightbours=str(find_supervertex(T,t.neighbours[i]))
            G.add_edge(str_vertices,str_neightbours,weight=t.connectivity[i])
    elarge = [(u, v) for (u, v, d) in G.edges(data=True) if d["weight"] > 0.5]
    esmall = [(u, v) for (u, v, d) in G.edges(data=True) if d["weight"] <= 0.5]
    pos = nx.spring_layout(G, seed=8)
    nx.draw_networkx_nodes(G, pos, node_size=700)
    nx.draw_networkx_edges(G, pos, edgelist=elarge, width=6)
    nx.draw_networkx_edges(G, pos, edgelist=esmall, width=6, alpha=0.5,
    edge_color="b", style="dashed")
    nx.draw_networkx_labels(G, pos, font_size=30, font_family="sans-serif")
    edge_labels = nx.get_edge_attributes(G, "weight")
    nx.draw_networkx_edge_labels(G, pos, edge_labels)
    ax = plt.gca()
    ax.margins(0.08)
    plt.axis("off")
    plt.tight_layout()
    plt.savefig("results/Partial Gomory-Hu Tree")
    plt.show()
    B = nx.adjacency_matrix(G)
    return B

def visualiazeGraph(txt):
    G = nx.Graph()
    with open(txt, 'r') as file:
        lines = file.readlines()
```

```
        edges = [line.strip().split() for line in lines[1:]]
    for edge in edges:
        G.add_edge(edge[0],edge[1])
    nx.draw(G, with_labels=True, font_weight='bold')
    plt.savefig("results/Initial Graph")
    plt.show()
    A = nx.adjacency_matrix(G)
    return A

def main():
    txt="input.txt"
    k=2 #k for partial GM tree
    graph = create_graph.create_graph(txt)
    preformatT = algorithm11.partialGH1(graph,k)
    T = algorithm11.format(preformatT)
    visualiazeGraph(txt)
    visualiazeTree(T)
    return 1
```

Listing 5.1: Visualization

Within the provided code, we encounter two functions: "visualizeGraph" and "visualizeTree." These functions play a pivotal role in visually representing the initial graph and the partial Gomory-Hu tree, which were computed using the implementation outlined in the previous chapter. The main function simply reads the input from the txt file and calls the "visualizeGraph" and "visualizeTree" functions. The "find supervertex" function accepts the supervertex's id as input and returns the contained vertices.

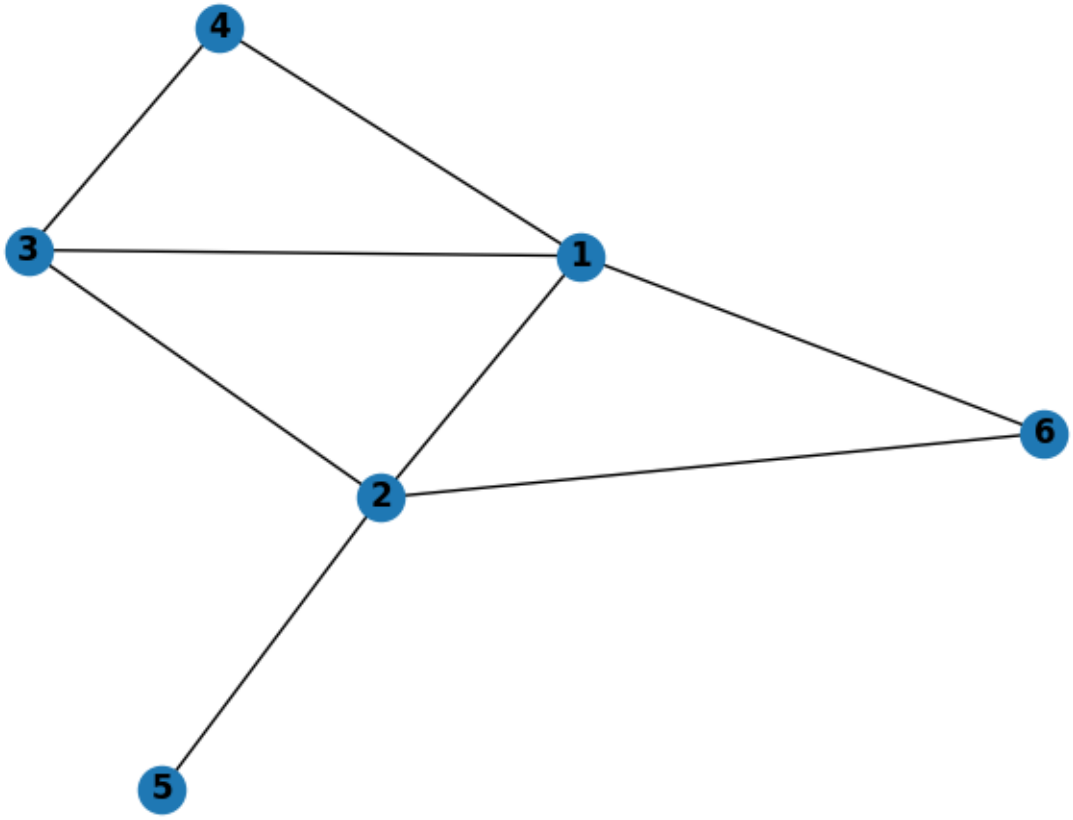
```
###############
1 2
1 3
2 1
2 3
3 4
4 1
4 3
5 2
6 1
6 2
```

Listing 5.2: Example txt

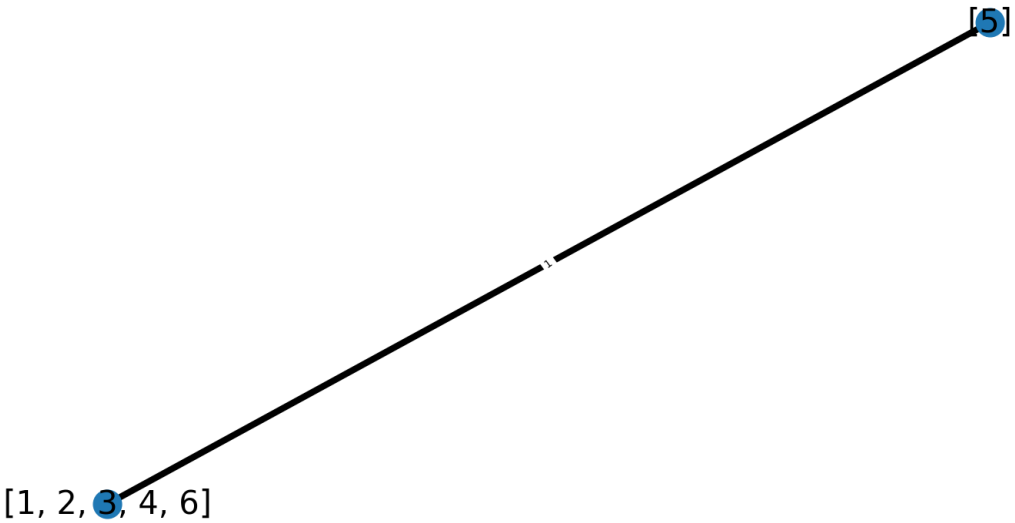
Presented below are examples of graphs along with their corresponding partial Gomory-Hu trees, each computed for different values of k.

5.2 Results

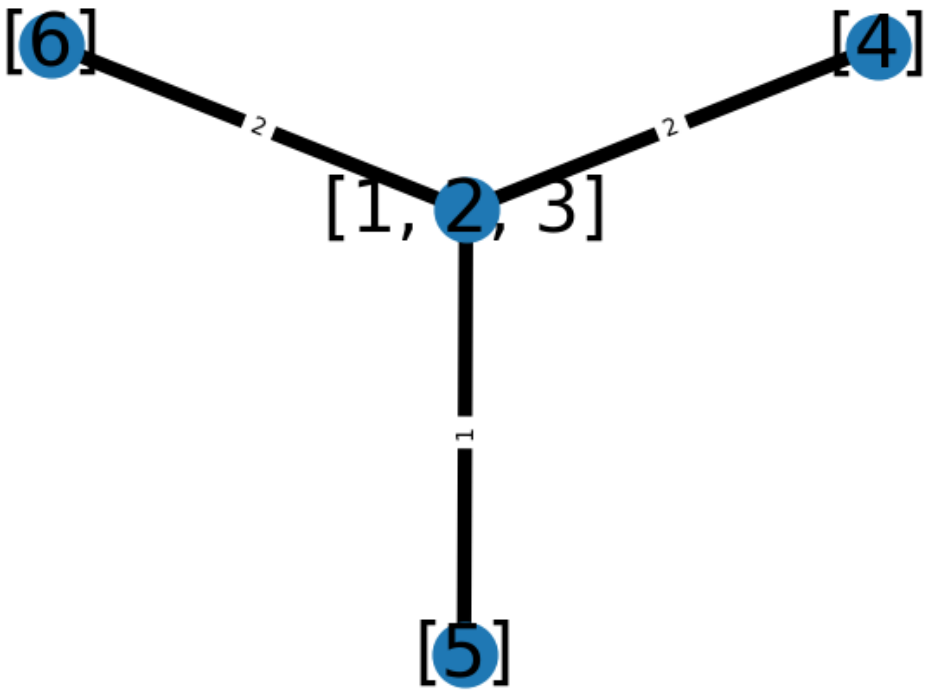
In this section, we will examine various inputs for the algorithm alongside their respective outputs, all evaluated for different values of k.
Input graph (graph of example of Listing 5.2: Example txt):



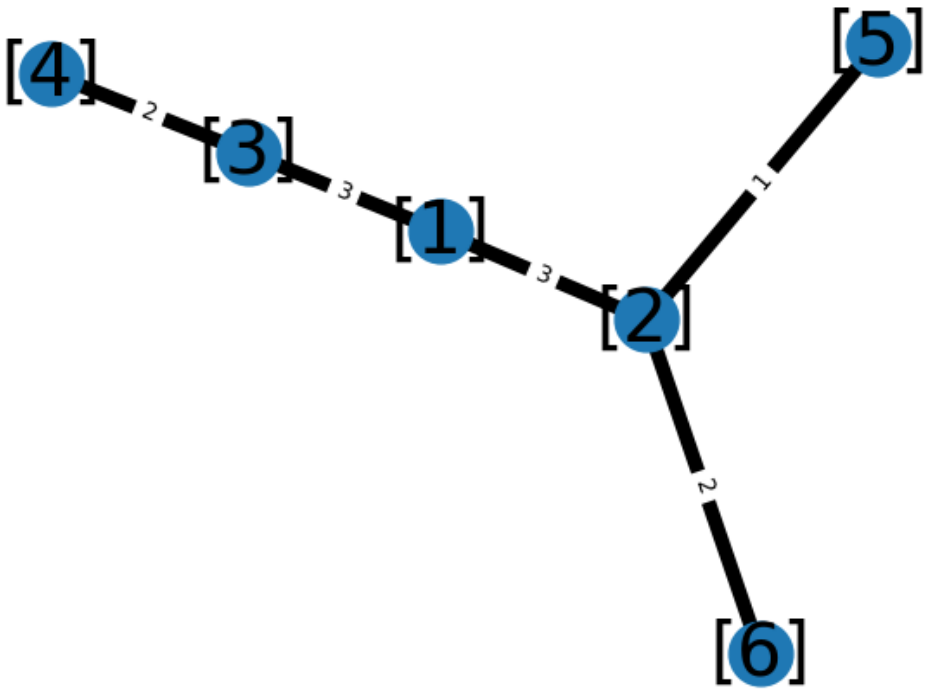
Partial Gomory-Hu Tree for k=1:



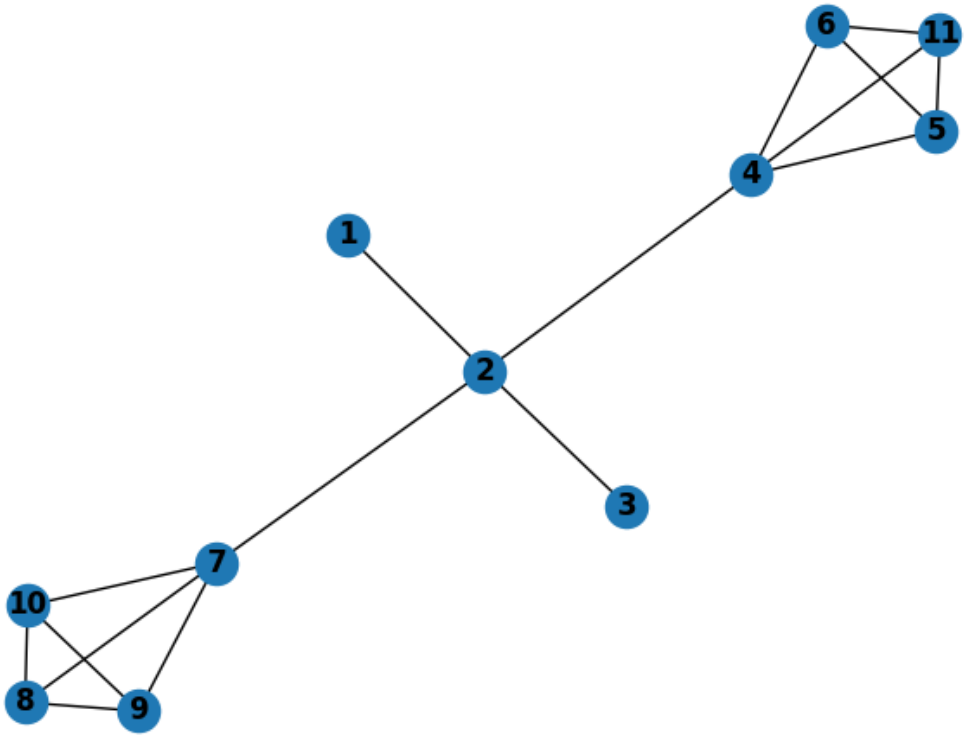
Partial Gomory-Hu Tree for k=2:



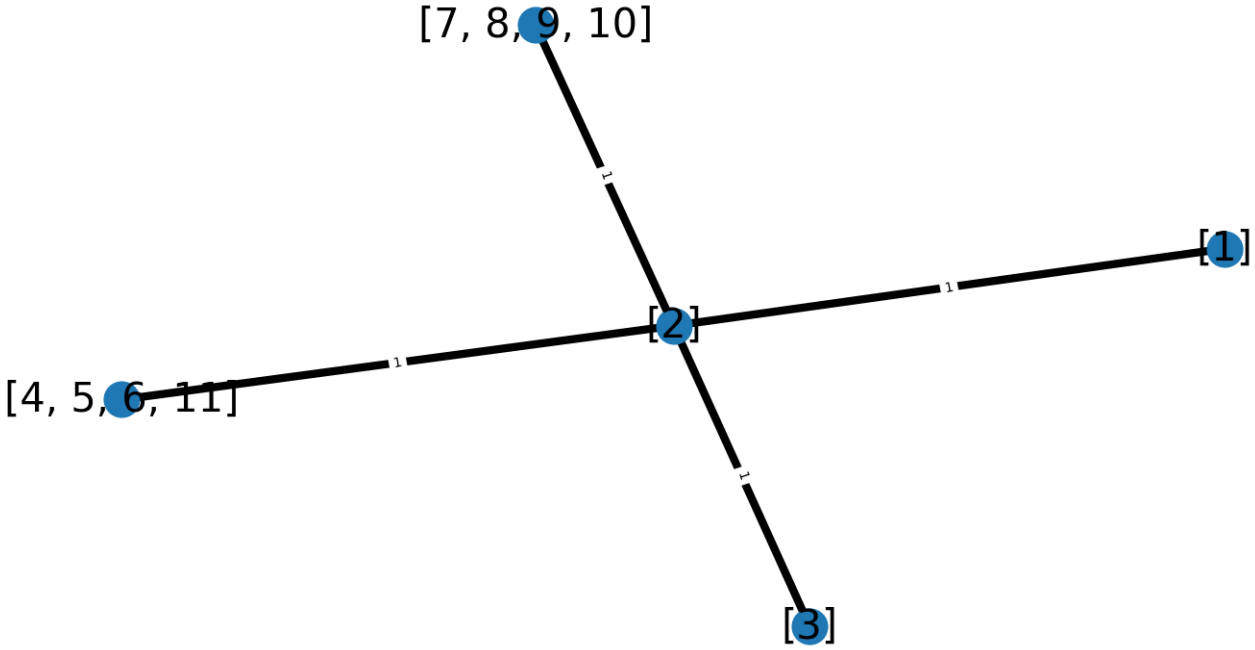
Partial Gomory-Hu Tree for k=3:



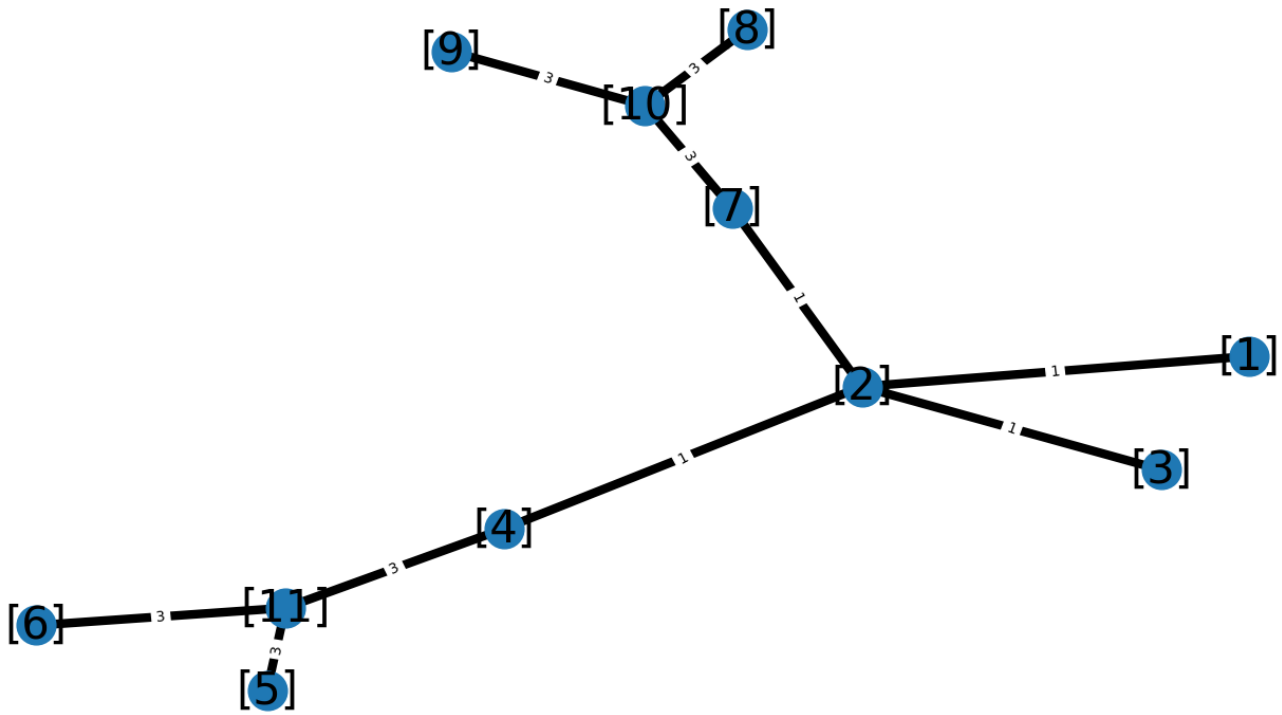
New input graph:



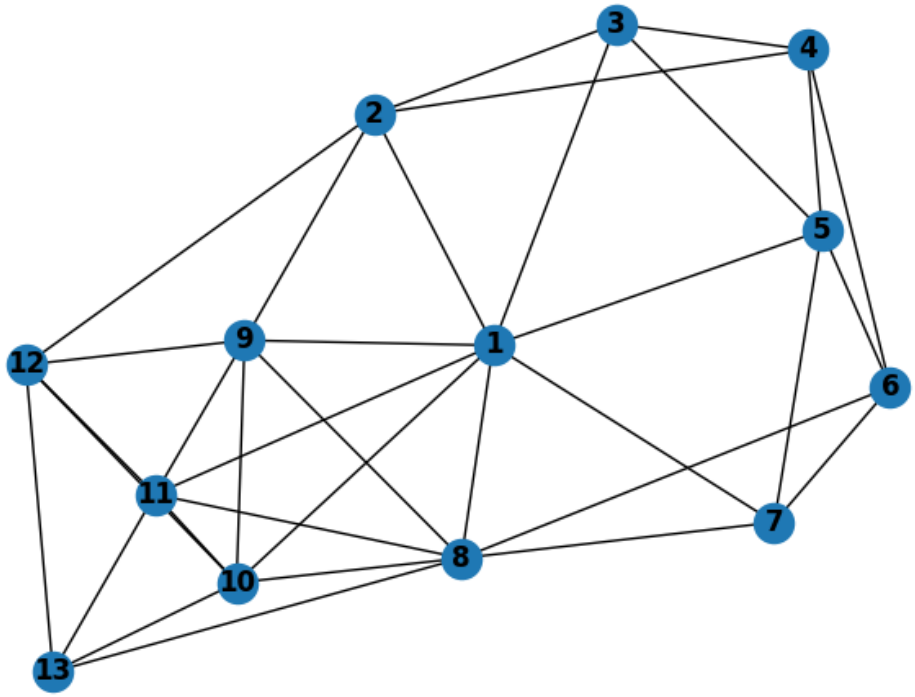
Partial Gomory-Hu Tree for k=1 and k=2:



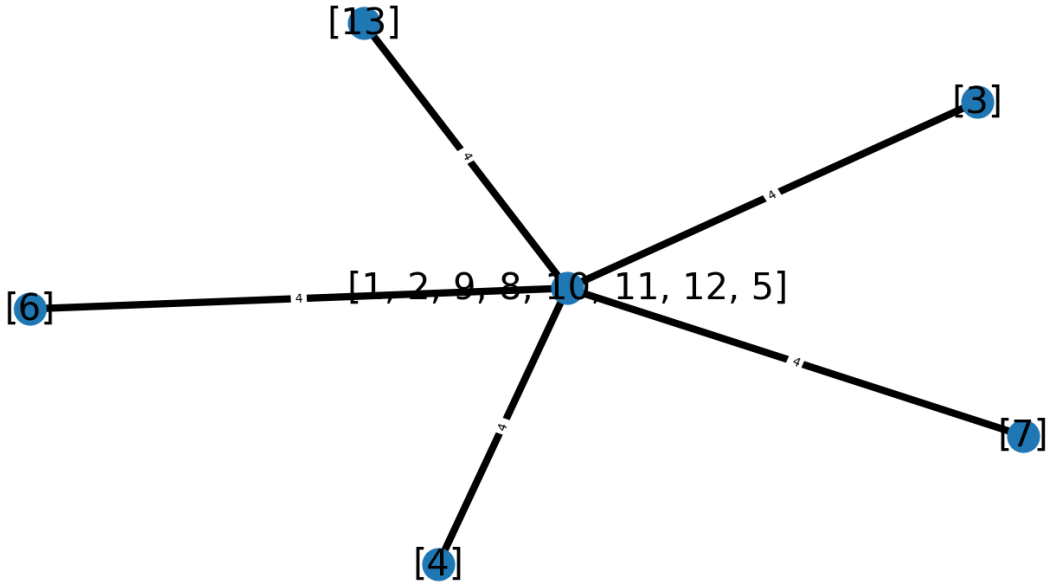
Partial Gomory-Hu Tree for k=3:



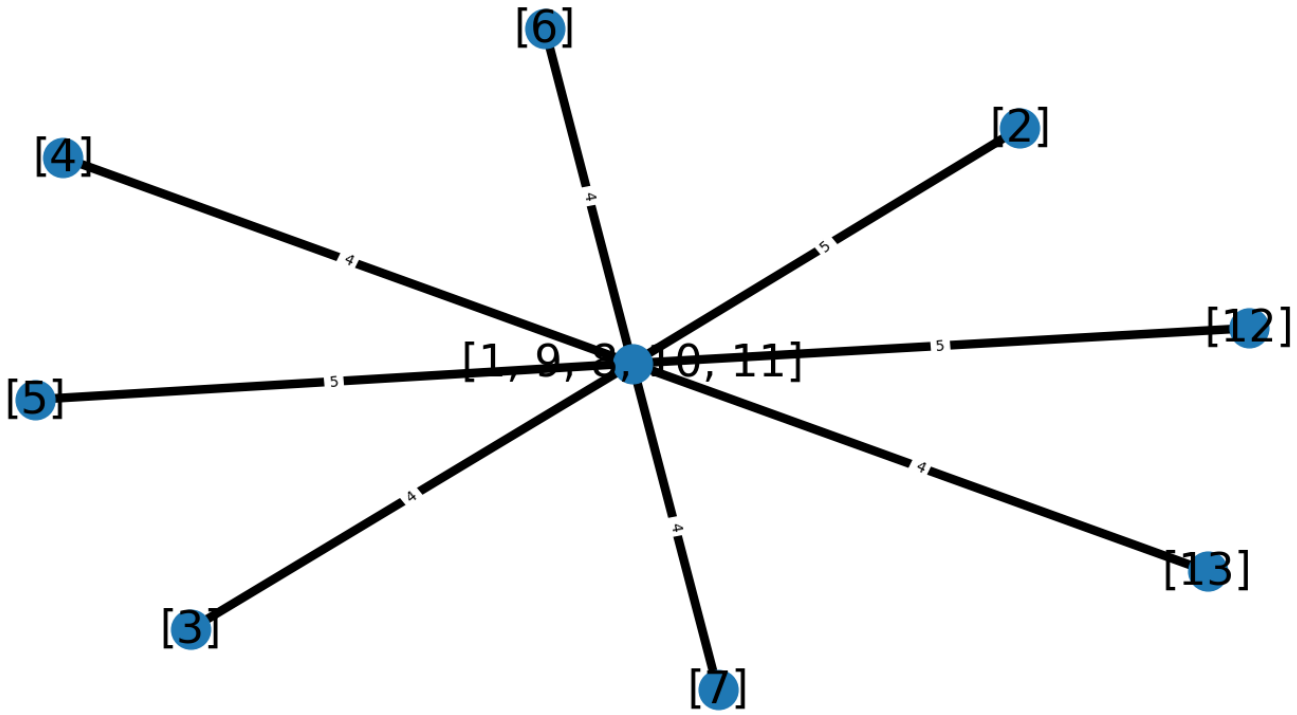
New input graph: (For values of k below 4, no partial Gomory-Hu tree exists)



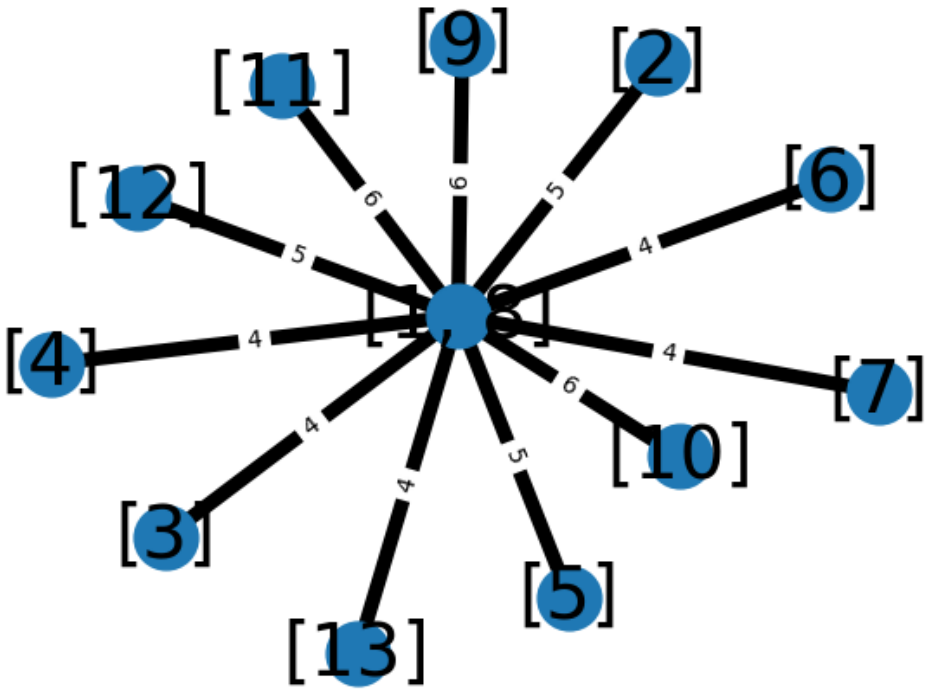
Partial Gomory-Hu Tree for k=4:



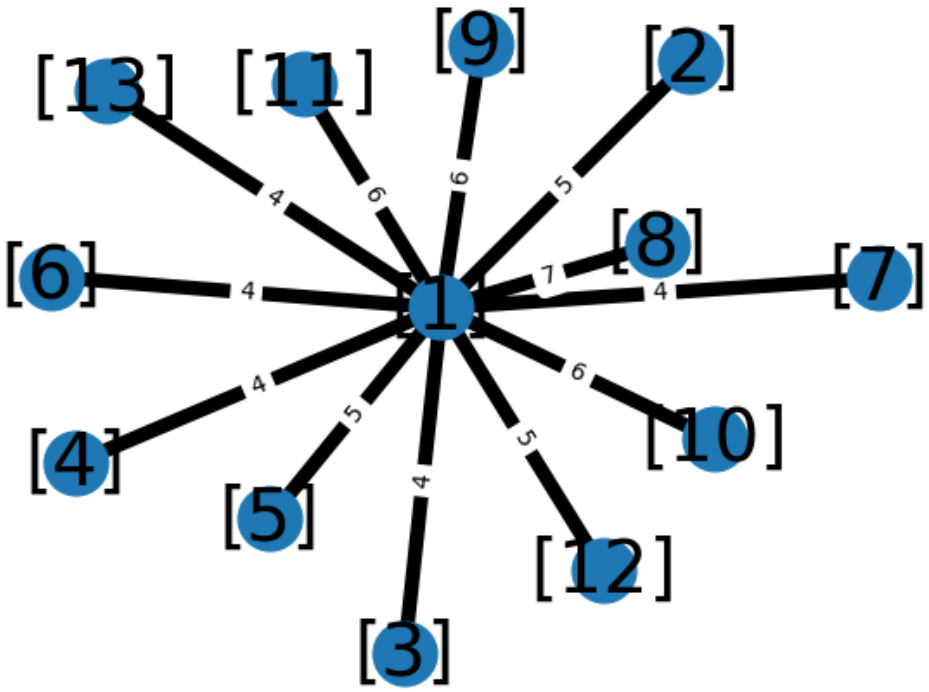
Partial Gomory-Hu Tree for k=5:



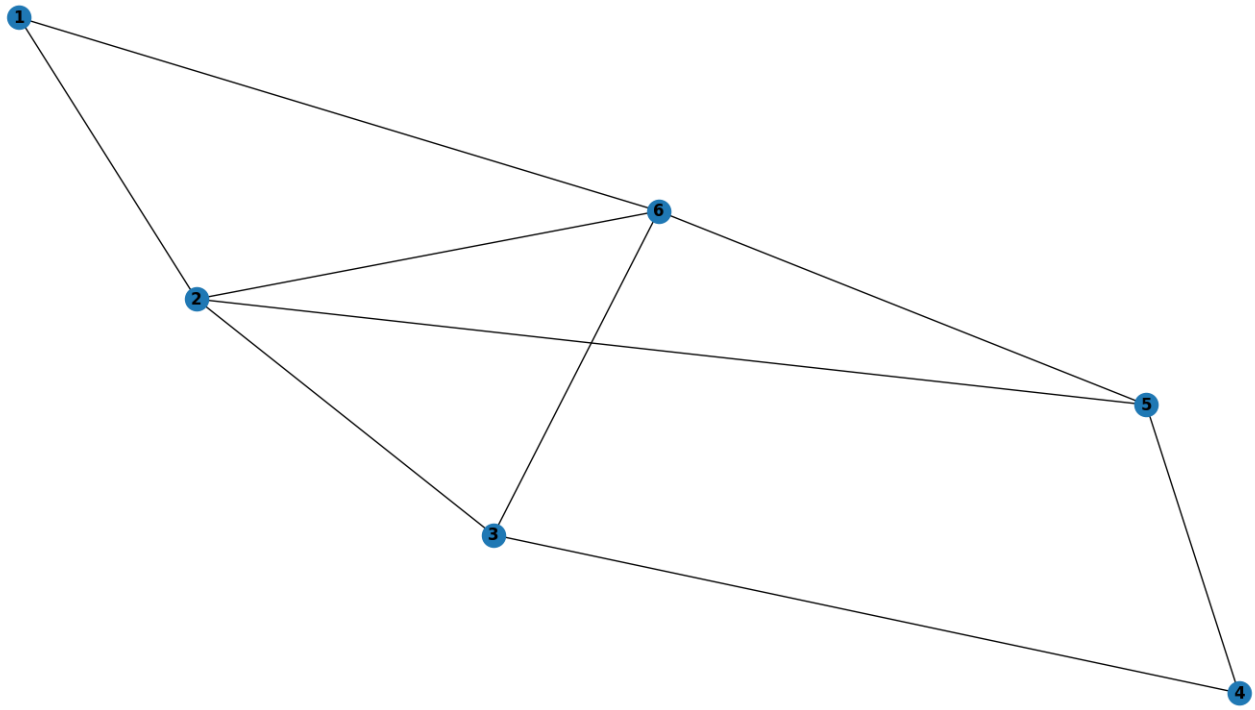
Partial Gomory-Hu Tree for k=6:



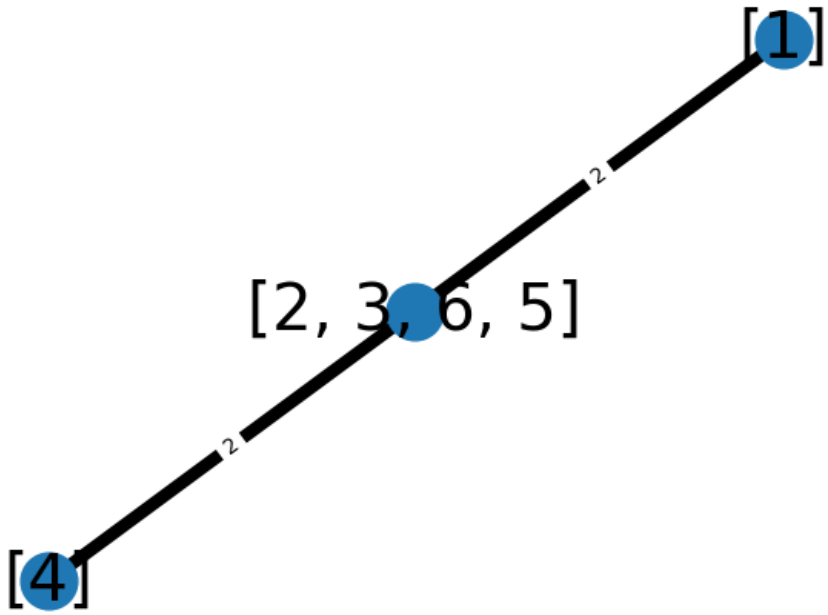
Partial Gomory-Hu Tree for k=7:



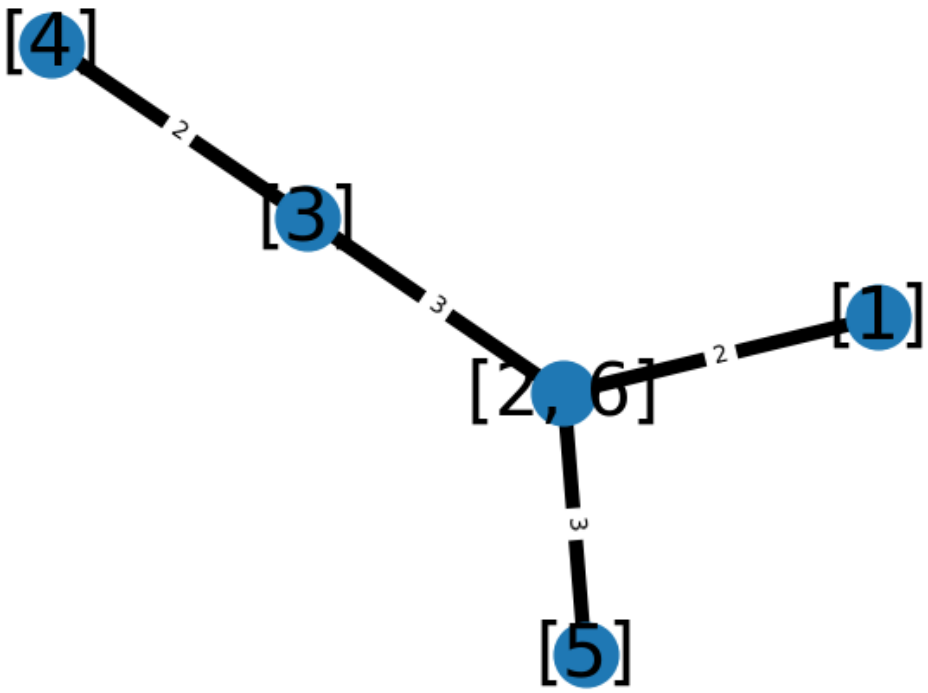
New input graph:



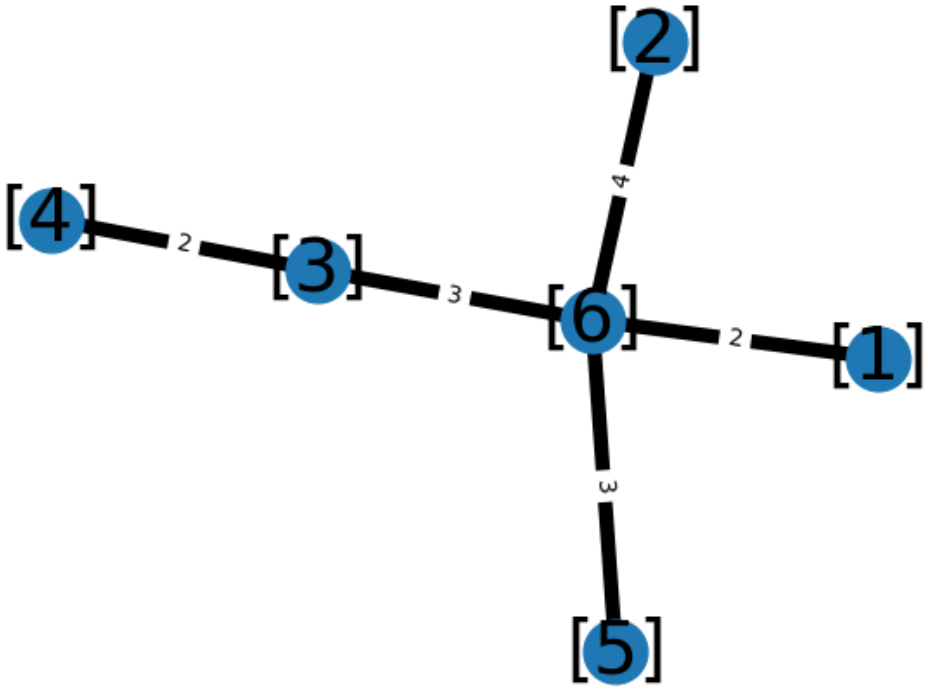
Partial Gomory-Hu Tree for k=2:



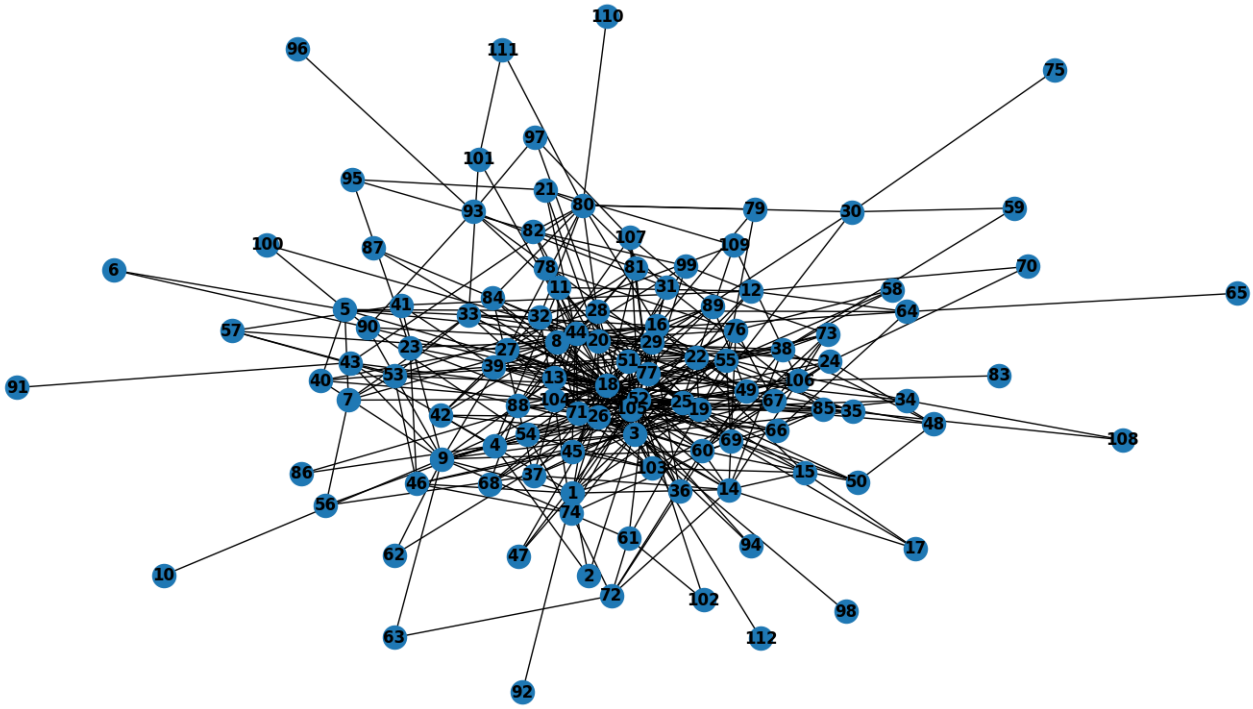
Partial Gomory-Hu Tree for k=3:



Partial Gomory-Hu Tree for k=4:



New input graph:



Partial Gomory-Hu Tree for k=15:

For sizable graphs, like the one presented above, it becomes apparent that the visualizer does not effectively assist in understanding the critical points.

To address this issue, we can simply print the tree before returning, revealing the ensuing structure:

```
2, [1], [3], [3]
1, [0], [4, 3], [1, 13]
4, [91], [1], [1]
6, [9], [3], [1]
5, [3], [7, 3], [2, 9]
7, [85], [5], [2]
8, [12], [3], [10]
9, [40], [3], [4]
10, [50], [3], [15]
12, [38], [3], [9]
11, [11], [13, 3], [2, 7]
13, [69], [11], [2]
14, [15], [3], [12]
15, [24], [3], [15]
16, [105], [3], [7]
17, [26], [3], [11]
18, [48], [3], [10]
19, [81], [3], [6]
20, [103], [3], [9]
21, [59], [3], [13]
```

```
22, [84], [3], [6]
24, [35], [3], [6]
23, [31], [25, 3], [2, 12]
25, [86], [23], [2]
26, [25], [3], [14]
28, [46], [3], [3]
27, [33], [29, 3], [2, 5]
29, [107], [27], [2]
30, [61], [3], [2]
31, [30], [3], [6]
32, [21], [3], [10]
33, [54], [3], [13]
34, [6], [3], [6]
35, [7], [3], [11]
36, [68], [3], [12]
37, [18], [3], [14]
38, [32], [3], [10]
40, [106], [3], [6]
39, [42], [41, 3], [1, 6]
41, [90], [39], [1]
42, [66], [3], [7]
43, [57], [3], [3]
44, [67], [3], [4]
45, [23], [3], [8]
46, [34], [3], [7]
47, [101], [59], [2]
48, [49], [3], [5]
50, [56], [3], [4]
51, [110], [49], [2]
49, [79], [51, 53, 3], [2, 1, 12]
53, [109], [49], [1]
54, [75], [3], [10]
52, [37], [55, 3], [2, 10]
55, [58], [52], [2]
56, [36], [3], [7]
57, [76], [3], [12]
58, [87], [3], [10]
59, [60], [47, 3], [2, 5]
61, [13], [3], [12]
60, [71], [62, 3], [2, 6]
62, [62], [60], [2]
63, [14], [3], [6]
64, [102], [3], [10]
65, [44], [3], [7]
66, [27], [3], [15]
68, [20], [3], [5]
67, [4], [69, 3], [2, 7]
69, [5], [67], [2]
70, [99], [80], [2]
71, [39], [3], [5]
73, [80], [3], [8]
72, [63], [74, 3], [1, 4]
74, [64], [72], [1]
75, [70], [3], [12]
76, [83], [3], [5]
77, [98], [3], [5]
78, [41], [3], [8]
79, [100], [3], [3]
```

```
80, [89], [70, 3], [2, 7]
81, [53], [3], [6]
82, [73], [3], [7]
83, [45], [3], [6]
84, [19], [3], [9]
85, [22], [3], [5]
86, [94], [3], [3]
87, [52], [3], [7]
88, [28], [3], [13]
90, [96], [3], [3]
89, [92], [91, 3], [1, 5]
91, [95], [89], [1]
93, [55], [3], [3]
92, [29], [94, 3], [1, 4]
94, [74], [92], [1]
95, [72], [3], [7]
96, [65], [3], [6]
97, [108], [3], [4]
98, [16], [3], [3]
99, [77], [3], [3]
100, [47], [3], [5]
101, [78], [3], [3]
102, [88], [3], [6]
103, [10], [3], [4]
104, [82], [3], [1]
105, [93], [3], [2]
106, [97], [3], [1]
3, [2, 8, 17, 104, 51, 43], [2, 1, 6, 5, 8, 9, 10, 12, 11, 14, 15, 16, 17, 18,
    19, 20, 21, 22, 24, 23, 26, 28, 27, 30, 31, 32, 33, 34, 35, 36, 37, 38, 40,
    39, 42, 43, 44, 45, 46, 48, 50, 49, 54, 52, 56, 57, 58, 59, 61, 60, 63, 64,
    65, 66, 68, 67, 71, 73, 72, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86,
    87, 88, 90, 89, 93, 92, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105,
    106, 107], [3, 13, 1, 9, 10, 4, 15, 9, 7, 12, 15, 7, 11, 10, 6, 9, 13, 6, 6,
    12, 14, 3, 5, 2, 6, 10, 13, 6, 11, 12, 14, 10, 6, 6, 7, 3, 4, 8, 7, 5, 4, 12,
     10, 10, 7, 12, 10, 5, 12, 6, 6, 10, 7, 15, 5, 7, 5, 8, 4, 12, 5, 5, 8, 3, 7,
     6, 7, 6, 9, 5, 3, 7, 13, 3, 5, 3, 4, 7, 6, 4, 3, 3, 5, 3, 6, 4, 1, 2, 1, 1]
107, [111], [3], [1]
```

Listing 5.3: Example2 txt

As previously noted in chapter 4, the initial column represents the supervertex ids, the second column represents the enclosed vertices, the third column represents the adjacent supervertices, and the fourth column represents the edge weight values.

References

[1] Ramesh Hariharan, Telikepalli Kavitha, Debmalya Panigrahi *Efficient Algorithms for Computing All Low s-t Edge Connectivities and Related Problems*

[2] Koutris Paraschos and Vasileios Syrgkanis *GOMORY-HU TREES: THEORY AND APPLICATIONS*

[3] Richard Cole, Ramesh Hariharan *A Fast Algorithm for Computing Steiner Edge Connectivity*

[4] Hiroshi Nagamochi, Toshihide Ibaraki *A Linear-Time Algorithm for Finding a Sparse k-Connected Spanning Subgraph of a k-Connected Graph*

[5] Dan Gusfield *VERY SIMPLE METHODS FOR ALL PAIRS NETWORK FLOW ANALYSIS*