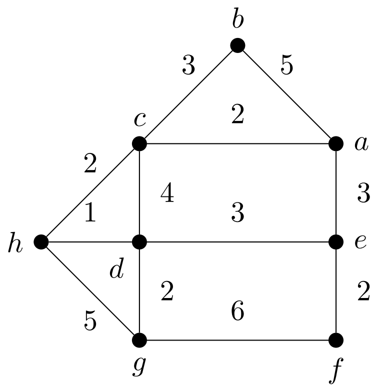
Caleb Logemann
MATH 566 Discrete Optimization
Homework 7

1. A cut of G is *minimal* if there is no cut of G properly contained in it. Prove that the random contraction algorithm returns only minimal cuts.

   Proof. Let G be a connected graph and let the set of edges $E = \{e_1, \dots, e_k\}$ be a cut returned by the random contraction algorithm. Now suppose E is not a minimal cut, that is there exists a proper subset $F \subset E$ that is also a cut of G. Thus there is an edge e, such that $e \in E$ and $e \notin F$. Let E be the cut between the set of vertices X and $\bar{X}$ of G. Since $e \in E$, that implies that $e = \{a, b\}$ where $a \in X$ and $b \in \bar{X}$. Since E was the cut returned by the random contraction algorithm, this implies that at the last step X and $\bar{X}$ were both contracted to a single vertex. If a set of vertices is contracted to a single vertex they must have been connected in the original graph, because the algorithm only contracts along edges. Thus the X and $\bar{X}$ when considered as subgraphs must be connected. Now consider the cut F, since the cut is a subset of E, the sets X and $\bar{X}$ are still connected with edges in F removed. Now consider $u \in X$ and $v \in \bar{X}$, since X is connected there exists a path in G/F from u to a, and since $\bar{X}$ is connected there is a path from b to v. Now since $e \notin F$, there is a path from u tp v. This shows that there is a path from any two vertices in G/F, and thus F is not a cut at all. Since F is not a cut, E must be a minimal cut, showing that the random contraction algorithm does return minimal cuts. ☐

2. Implement Node Identification Minimum Cut Algorithm. Try it on the following graph:



   I implemented the Node Identification algorithm with the following functions.

```
def nodeIdentification(original_graph):
    # original graph is a Sage graph object that is undirected and whose
    # edge_labels are the weights
    graph = original_graph.copy()
    graph.set_vertices(dict((v, {v}) for v in graph.vertices()))
    minimum_cut = None
    minimum_cut_value = None
    while graph.order() > 1:
        legal_ordering = findLegalOrdering(graph)
        cut = graph.get_vertex(legal_ordering[-1])
        cut_value = sum([e[2] for e in graph.edges_incident(
            ↪ legal_ordering[-1])])
        if cut_value < minimum_cut_value or minimum_cut_value == None:
            minimum_cut = cut
            minimum_cut_value = cut_value

        contractVertices(graph, legal_ordering[-1], legal_ordering[-2])

    return (minimum_cut, minimum_cut_value)

def findLegalOrdering(graph):
    vertices = graph.vertices()
    legal_ordering = [vertices[0]]
    vertices_remaining = vertices[1:]
    for i in range(len(vertices_remaining)):
        max_capacity = None
        max_capacity_vertex = None
        for v in vertices_remaining:
            capacity = 0
            for w in legal_ordering:
                if v in graph.neighbors(w):
                    capacity += graph.edge_label(v, w)

            if capacity > max_capacity or max_capacity == None:
                max_capacity = capacity
                max_capacity_vertex = v

        legal_ordering.append(max_capacity_vertex)
        vertices_remaining.remove(max_capacity_vertex)

    return legal_ordering
```

```
def contractVertices(graph, v, w):
    graph.set_vertex(v, graph.get_vertex(v).union(graph.get_vertex(w)))
    for x in graph.neighbors(w):
        if x != v:
            weight = graph.edge_label(w, x)
            if x in graph.neighbors(v):
                graph.set_edge_label(x, v, graph.edge_label(x, v) +
                   ↪ weight)
            else:
                graph.add_edge(v, x, weight)

    graph.delete_vertex(w)

    return None
```

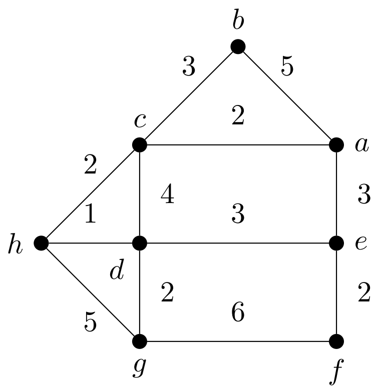
The following script uses these function to run the Node Identification algorithm on the graph given for this problem.

```
M = matrix([
    (0, 5, 2, 0, 3, 0, 0, 0),
    (5, 0, 3, 0, 0, 0, 0, 0),
    (2, 3, 0, 4, 0, 0, 0, 2),
    (0, 0, 4, 0, 3, 0, 2, 1),
    (3, 0, 0, 3, 0, 2, 0, 0),
    (0, 0, 0, 0, 2, 0, 6, 0),
    (0, 0, 0, 2, 0, 6, 0, 5),
    (0, 0, 2, 1, 0, 0, 5, 0)])
graph = Graph(M, weighted=True)
graph.relabel(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
load('nodeIdentification.sage')
load('contractVertices.sage')
s = nodeIdentification(graph)
minimum_cut = s[0]
minimum_cut_value = s[1]
print("minimum␣cut:␣" + str(minimum_cut))
print("minimum␣cut␣value:␣" + str(minimum_cut_value))
```

The output of this script is

```
minimum cut: set(['h', 'g', 'f'])
minimum cut value: 7
```

3. Implement Random Contraction Algorithm. Try it on the following graph. Run it many times and based on your experiment conclude what is the probability that that your algorithm succeeds on this graph.



I implemented the random contraction algorithm with the following function. This function makes use of the contractVertices function, which was shown in the previous exercise, but is ommited here.

```
import numpy as np
def randomContraction(original_graph, numTrials):
    # original graph is a Sage graph object that is undirected and whose
    # edge_labels are the weights
    # numTrials is the number of times the random contraction algorithm
       ↪ should
    # be run before returning the minimum cut found
    minimum_cut = None
    minimum_cut_value = None
    for i in range(numTrials):
        graph = original_graph.copy()
        graph.set_vertices(dict((v, {v}) for v in graph.vertices()))
        while graph.order() > 2:
            total_weight = sum(graph.edge_labels())
            probabilities = [e/total_weight for e in graph.edge_labels()
               ↪ ]
            edgeNum = np.random.choice(graph.size(), p=probabilities)
            edge = graph.edges()[edgeNum]

            # contract found edge
            v = edge[0]
            w = edge[1]
            contractVertices(graph, v, w)

        cut = graph.get_vertices().values()[0]
        vertices = graph.vertices()
        cut_value = graph.edge_label(vertices[0], vertices[1])

        if cut_value < minimum_cut_value or minimum_cut_value == None:
            minimum_cut = cut
            minimum_cut_value = cut_value

    return (minimum_cut, minimum_cut_value)
```

The following script uses these function to run the randomContraction algorithm on the graph given for this problem.

```
M = matrix([
    (0, 5, 2, 0, 3, 0, 0, 0),
    (5, 0, 3, 0, 0, 0, 0, 0),
    (2, 3, 0, 4, 0, 0, 0, 2),
    (0, 0, 4, 0, 3, 0, 2, 1),
    (3, 0, 0, 3, 0, 2, 0, 0),
    (0, 0, 0, 0, 2, 0, 6, 0),
    (0, 0, 0, 2, 0, 6, 0, 5),
    (0, 0, 2, 1, 0, 0, 5, 0)])
graph = Graph(M, weighted=True)
graph.relabel(['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h'])
load('randomContraction.sage')
load('contractVertices.sage')
s = randomContraction(graph, 100)
minimum_cut = s[0]
minimum_cut_value = s[1]
print("minimum␣cut:␣" + str(minimum_cut))
print("minimum␣cut␣value:␣" + str(minimum_cut_value))

p = dict()
for i in range(5, 51, 5):
    p[i]=0
    for j in range(500):
        s = randomContraction(graph, i)
```

```
            if s[1]==7:
                p[i]+=1

# convert to percentages
p = dict((i, j/500.0) for (i, j) in p.items())
print p
```

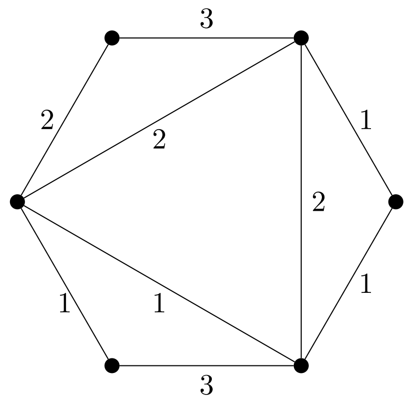
The output of this script is shown below. As it can be seen with a hundred trials the algorithm finds the minimum cut.

```
minimum cut: set(['a', 'c', 'b', 'e', 'd'])
minimum cut value: 7
```

Also the script computes an experimental probability that the algorithm finds the correct minimum cut, based on the number of trials that are run. Note the interesting fact that at just 20 trials the probability of finding the right cut is over 90%. This is much higher than the lower bound given in the notes.

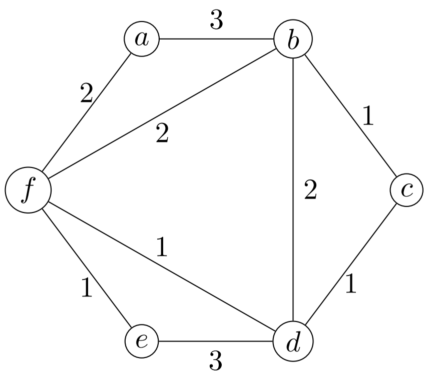
number of trials	success percentage
1	0.126
5	0.438
10	0.668
15	0.834
20	0.918
25	0.956
30	0.972
35	0.980
40	0.992
45	0.992
50	0.998

4. Construct Gomory-Hu Tree for the following graph using the algorithm from the class. Numbers on edges correspond to capacities. Show Show steps after every new cut.

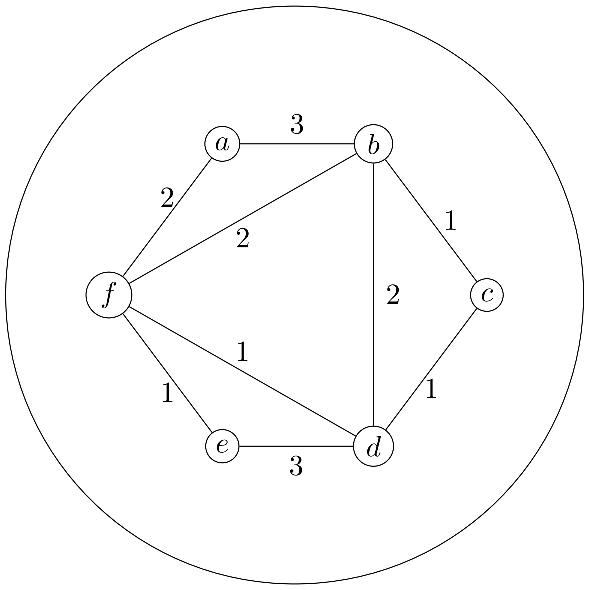


Check your answer with Sage using method `Graph.gomory_hu_tree()`. Do not implement it yourself, just run it.

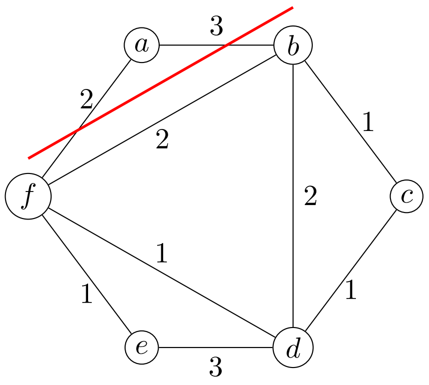
First in order to run the Gomory Hu algorithm, I will label the vertices.



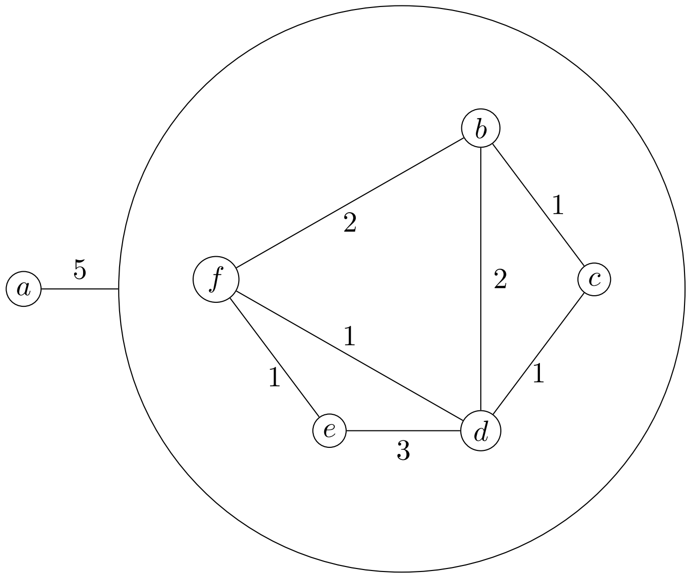
First start with a tree that is $T = (\{V\}, \emptyset)$. This tree looks like.



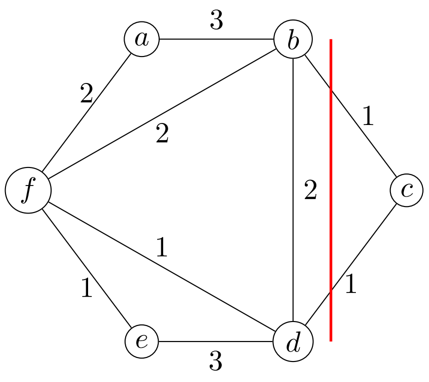
Now I will chose to find the minimum a-b cut, which corresponds to finding the minimum a-b flow in the following graph. The minimum cut found is shown in red.



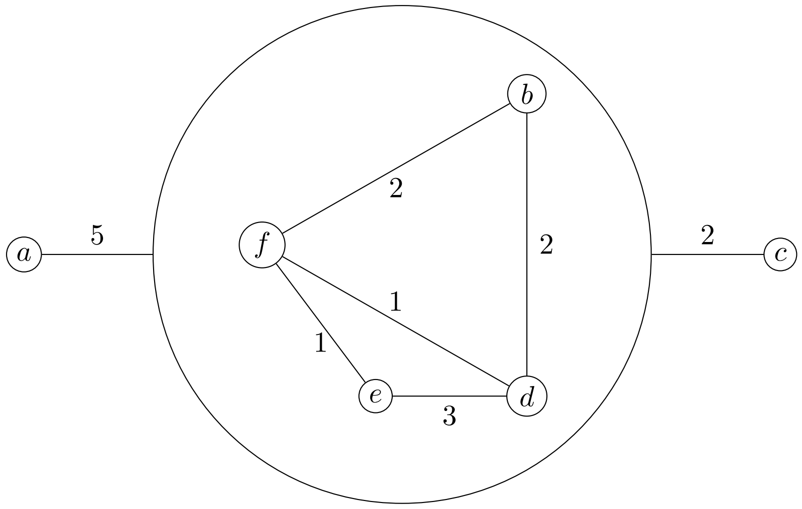
Using this cut the new Gomory Hu tree is



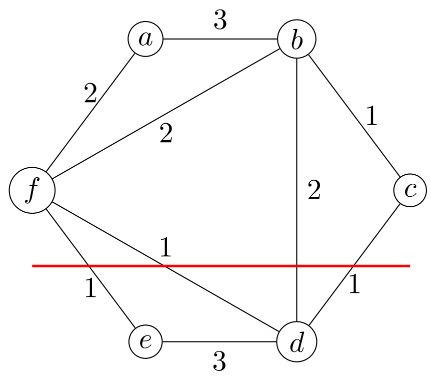
Now I will find the minimum *b*-*c* cut in the following graph. The minimum cut found is shown in red.



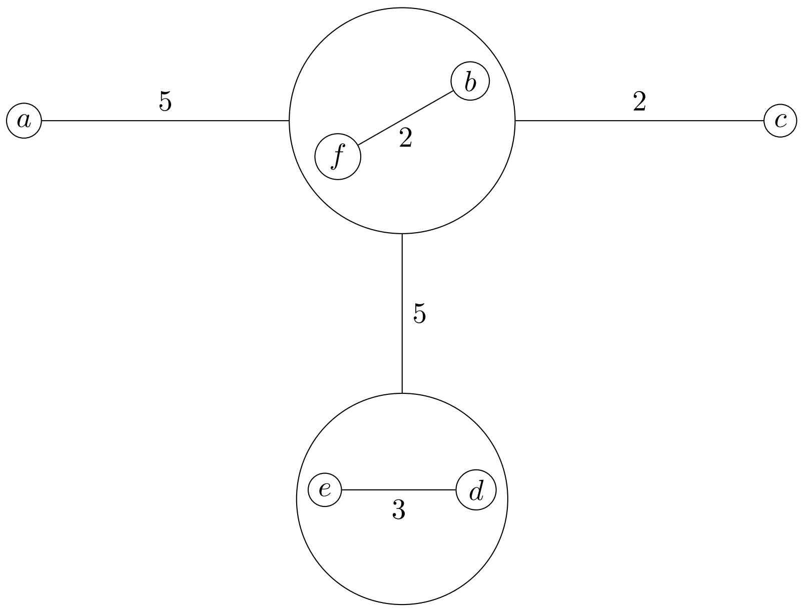
Using this cut the new Gomory Hu tree is



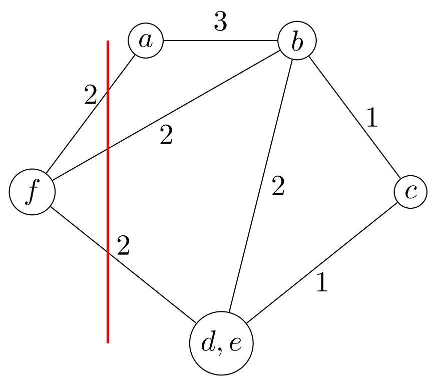
Now I will find the minimum *b*-*d* cut in the following graph. The minimum cut found is shown in red.



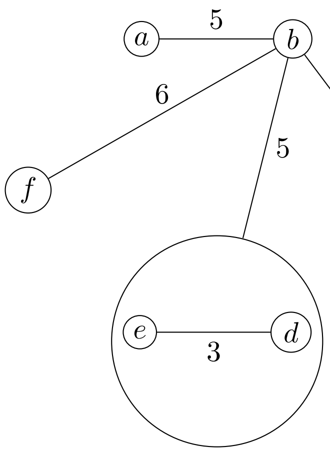
Using this cut the new Gomory Hu tree is shown below.



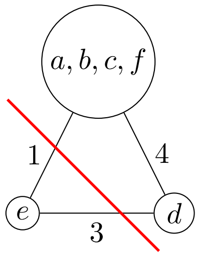
Now I will find the minimum b-f cut in the following graph, where e and d are contracted.



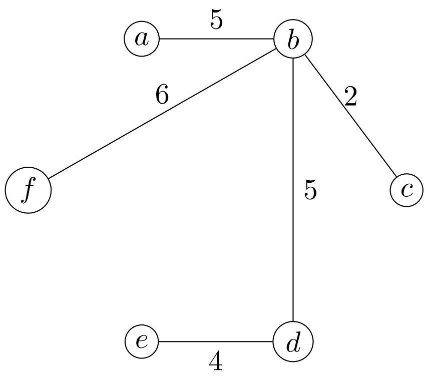
Using this cut the new Gomory Hu graph is



Lastly I will find the minimum d-e cut in the following contracted graph.



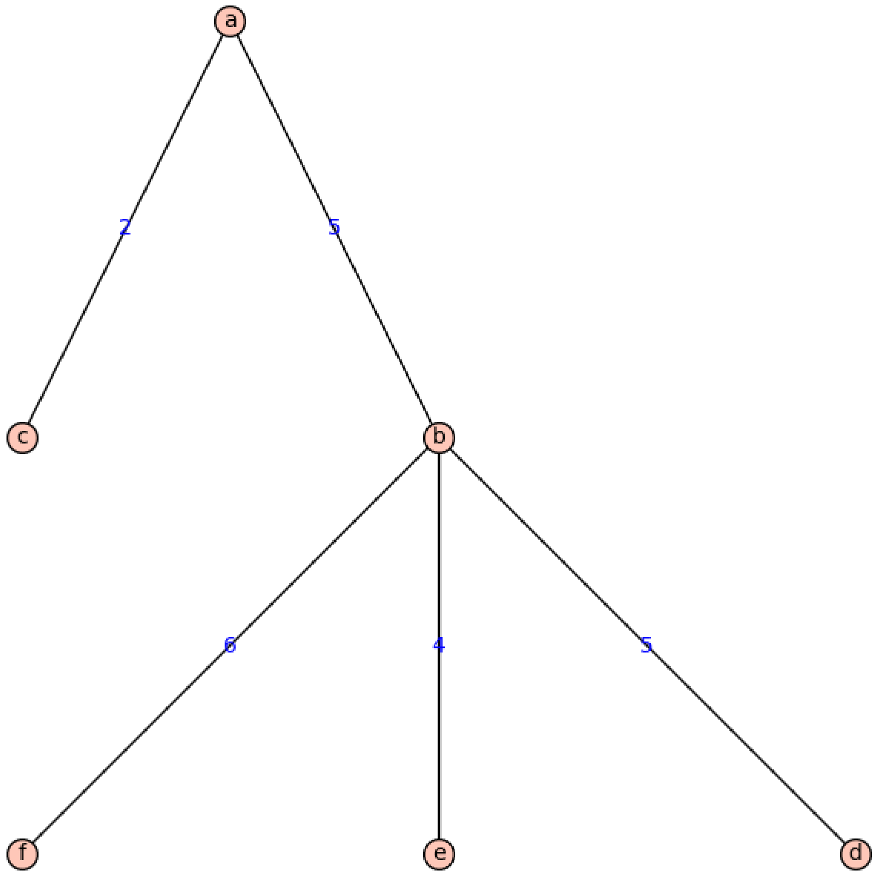
Thus the full Gomory Hu tree is



I checked this in Sage with the following script.

```
m = matrix([
     (0, 3, 0, 0, 0 ,2),
     (3, 0, 1, 2, 0, 2),
     (0, 1, 0, 1, 0, 0),
     (0, 2, 1, 0, 3, 1),
     (0, 0, 0, 3, 0, 1),
     (2, 2, 0, 1, 1, 0)])
graph = Graph(m, weighted=True)
graph.relabel({0:'a', 1:'b', 2:'c', 3:'d', 4:'e', 5:'f'})
gomory_hu_tree = graph.gomory_hu_tree()
gomory_hu_tree.plot(edge_labels=True, layout='tree').show()
```

The output of this script is the following image of the Gomory Hu tree.



This graph has a different structure but it has the same weights As the Gomory Hu tree is not unique I believe this confirms the tree that I arrived at.