Randomized Algorithms : Homework 2

Romain Markowitch (000540172)

December 23, 2024

Contents

1 Introduction

For the second homework of the course INFO-F413, Randomized algorithms, we were asked to implement two algorithms :

- Contract : contracts the graph until we have a certain number of nodes remaining
- FastCut : improved version of the contract algorithm
- Brute_force : a function called in FastCut. It tries every cut possibility to find the best one.

The goal of the homework is to compare both algorithms in their time to complete their task, and their probability to find the minimal cut.

2 Experimental plan

2.1 Implementations

2.1.1 Graph data structure

To implement graphs, we chose to create 3 classes :

- Edge : It has an id, and two nodes (which it connects).
- Node : It has an id and a list of edges (connecting the node to other nodes).
- Graph : It has a list of nodes and a list of edges to be more efficient time wise. By creating them simultaneously, we can go through them faster (no need to go through all nodes then edges or vice versa). We can also apply the *contract_edge()* that will contract an edge by merging the two nodes it connects.

2.1.2 Contract

The contract algorithm is a simple algorithm that has two possible ways to call it, with only a graph and with a graph and a parameter t. If t is given it will loop until t nodes remains in the graph. Meanwhile if no parameter t is given, then it will stop at 2 remaining nodes.

In the loops it will randomly choose an edge of the graph and then apply the edge contraction on it. Then, it will return the list of edges of the minimum cut found.

This algorithm has a complexity of $O(n^2)$ if we assume that the contraction of an edge is done in $O(n)$.
The problem is that it has a probability of finding the minimum cut of at least $2/(n(n-1))$ where n is the number of nodes. It works well for graphs with a smaller number of nodes than for bigger graphs.

2.1.3 FastCut

FastCut algorithm is an improved version of the Contract algorithm. Its goal is to also find the minimum cut.
It takes a multigraph G and then, if the number of nodes is smaller or equal to 6 it will call the brute_force algorithm (see section 2.1.4). If not it will calculate t :

$$t = \lceil 1 + \frac{n}{\sqrt{2}} \rceil \tag{1}$$

After that, it will call the contract algorithm two times with the parameter t. That will give us the edges to create $H1$ and $H2$. And then we will do a recursive call (with FastCut) on both $H1$ and $H2$ and return the smaller cut of those two results.

The complexity of the FastCut algorithm is $O(n^2 log(n))$. It is worst than Contract algorithm but it increases the probability to find the minimum cut by a lot. Instead of a probability of $2/(n(n-1))$ we have : $\Omega(1/log(n)$ to successfully find the minimum cut. So FastCut is a more reliable algorithm to find the minimum cut.

2.1.4 Brute force algorithm

The algorithm starts by generating all possible subsets of nodes (S) and for each S we create T that is its complement. S cannot be a subset that is empty or a subset with all the nodes. We create both sets by using the combinations function from the library itertools.
Then, it checks the edges that cross the cut (by checking if both nodes of the edge are in different subsets). If the edge crosses the cut, it increases the count and the list of edges for the minimum cut. For each combination of subset we compare the current minimum cut with the temporary best minimum cut and change it if the current one is better.

The complexity of the generation of subset is $O(2^n)$ as we create 2^n subsets of n nodes. And as the algorithm only checks the subsets of 1 to n-1 nodes, we check in total $2^n - 2$ nodes.
Then for each subset it checks every edge to see if it crosses the cut. This is done in $O(E)$ where E is the number of edges of the subsets.
Thus, the total complexity is $O(2^n * E)$. But as this algorithm is only used for 6 nodes or less, its complexity is negligible.

2.2 Tests

2.2.1 Environment

As for the first homework, all experiments were realized on a MacBook Air (2024) with a M3 Apple chip (8-core CPU, 10-core GPU, and 16 GB unified memory), using Python 3.10.11 with libraries (matplotlib, networkx, random, itertools, copy and time).

2.2.2 Graphs

To have a better understanding of the impacts of the number of nodes and edges on the time taken for the algorithm to find the minimum cut, we have created multiple graph generators (see appendix B.4) where we create :

- Complete graph : A graph where all the nodes are connected to every other nodes.
- Bipartite graphs : Where all the nodes are in one of the two sets and where a node from a set cannot be connected to another node in the same set. With this there is a probability of connecting the new node with the other nodes to make randomly different kinds of graphs.
- Cycle graphs : A 2-connected graph where every node has only two nodes connected to it, the whole forming one big cycle.
- Tree graphs : Create a graph where we have a root and all the nodes have a path (of nodes and edges) connecting them to the root. There are no cycles in the tree.
- Planar graphs : Create a graph where edges cross often. It uses 2 sets of nodes and randomly decides if it connects the new node to each one of the previous ones.
- Multi-graphs : A graph where multiple edges can be present between two nodes.

To see an example of each type, see appendix A for a better understanding.

2.2.3 Implementation

To test all the graphs and retrieve the data for the results, we tested each type of graph with a number of edges or nodes, a certain number of times (between 50 to 100 times) to have a more precise and correct data.
Only the time taken by the algorithm was taken into account. Thus generating different kinds of graphs did not have an impact on the time.

Some of the graphs needed two values one for each subset of nodes, the arbitrary repartition to limit to a certain extent the randomness was the following :
Subset 1 : n//random.randint(1,(n//4)+1) where random.randint return a value between 1 and (n//4)+1.
Subset 2 : n - the number of nodes in subset 1.
Here, the total number of nodes varies to have as much different and bigger graphs as possible.

3 Results

3.1 Time

Below are the graphs representing the time taken by FastCut to try to find the minimal cut. The time only measures the time taken by the algorithm and not the time taken to generate the graphs. Each value has been tested at least 50 times to be more precise. We limited the number of nodes as too much time was required to run the tests efficiently and some create an error : *"RecursionError: maximum recursion depth exceeded while calling a Python object"*.

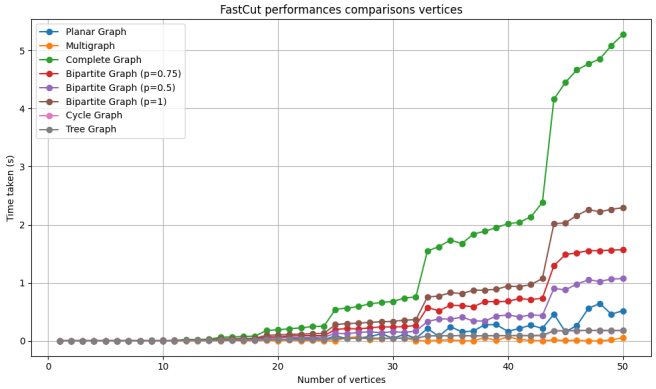

Figure 1: Time comparison to solve with FastCut different types of graphs from 1 node to 50 nodes

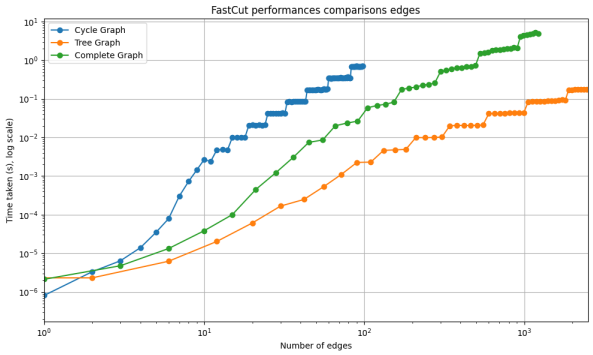

Figure 2: Time comparison to solve with FastCut different types of graphs with various number of edges

3.2 Randomness

When generating the graphs, some generators use random functions to create them. The following figure is there to show it.

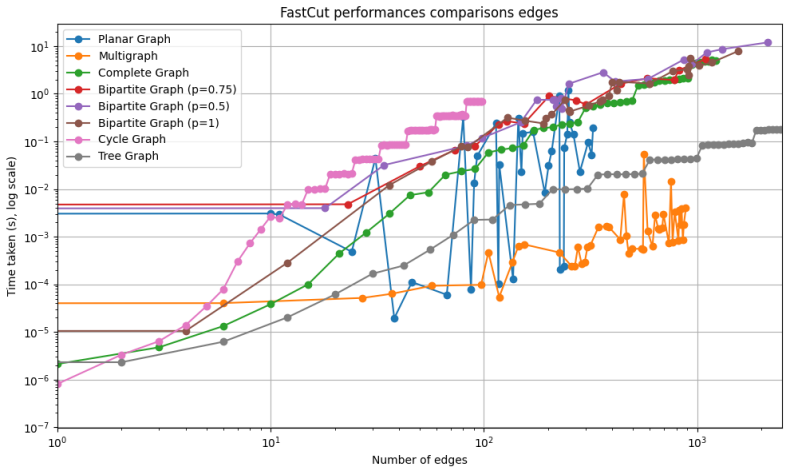

Figure 3: Effect of randomness in the time Comparison for FastCut with different numbers of edges

3.3 Probability of finding the minimum cut

The following data are the results of the number of times each algorithm successfully returns the right number of edges for the minimal cut out 25 tries. The tests are easily implemented for complete graphs, as the minimal cut is $n-1$ and where n is the number of nodes. We limit the number of nodes because with more nodes it will raise the *"RecursionError: maximum recursion depth exceeded while calling a Python object"*.

	Success rate to find the minimal cut	
Complete graph	**Contract(G,2)**	**FastCut(G)**
2 nodes	100%	100%
5 nodes	96%	100%
10 nodes	88%	100%
20 nodes	84%	100%
30 nodes	80%	100%
40 nodes	92%	100%
50 nodes	84%	100%
60 nodes	88%	100%
70 nodes	84%	100%
80 nodes	88%	100%

Table 1: Success rate for each algorithm on a complete graph with different numbers of nodes

4 Analyze

4.1 Time

As we can see on the Figure 1 where we increase the number of nodes without checking the number of edges, simpler graph structures like cycles (2-connected graph) and trees take less time to find the minimal cut than more complex graph structures like complete graphs or bipartite graphs. It is due to the fact that the simpler structures have less edges and thus less operations to do at each edge contraction.
Moreover, the complete graph which is the more complex and dense graph structure in our case, takes the most time to find the minimal cut.
In addition, the bipartite graphs are a perfect example. P is the probability of adding an edge between a new node and the previous ones. Thus the higher P is, the more edges the graph will have and that will lead FastCut to take more time to find the minimal cut. And that is in fact what we can observe where the bipartite graph with $P = 1$ is slower than $P = 0.75$ which is slower than $P = 0.5$.

We can also observe that for most of the graphs there are stages, where only increasing by 1 the number of nodes, adds a huge time to the process of finding the cut.

In the other hand, in the Figure 2, we removed the time taken by graphs using randomness (see section 3.2 for their values) and focused on the more regular ones. We can see that as for Figure 1 the more dense (higher number of edges) a graph is, the more time FastCut will take to find the minimal cut. Thus we can conclude that FastCut is more optimized for less dense graphs like trees.
Moreover, we can see that cycle graphs, which are less dense than complete graphs take more time. This is due to the fact that at each edge contraction we only remove one edge and never more (as it is a 2-connected graph). So we can determine that the more edges are removed each edge contraction, the faster the algorithm will be. That is why the multi-graph is fast when it is not a simple graph structure.

4.2 Randomness

As we can see in Figure 3, the randomness influences the number of edges in the graph. This can be seen by looking at multi-graphs, planar graphs and bipartite graphs where their time taken depending on the number of edges fluctuates a lot.
This is due to the fact that a graph with x edges and a lot of nodes and a graph with x edges but less nodes will not take the same time for FastCut to try to find the minimal cut. The reason is that if two graphs have the same number of edges, the one with less nodes will be processed faster as there will be more edges to remove at each edge contraction. Thus the number of edges will decrease faster and so less operations will be done in the next contractions.

This can also be seen in the bipartite graphs, as explained in section 4.1. Where the greater the probability to add an edge between the new node and the previous ones, the greater the time will be to find the minimal cut, as there are more edges. But we can still see that it depends on each new bipartite graph as two of them with the

same parameters will never be the same (except for $P = 1$).

4.3 Probability of finding the minimum cut

We can see in Table 1 that the success rate for Contract algorithm is lower than FastCut. The reason is due to their probability to find the minimal cut, as it is most of the time not 100%.
As explained in their implementation sections (sections 2.1.2 and 2.1.3), the probability of Contrast is : $2/(n(n-1))$ meanwhile the probability of FastCut is : $\Omega(1/log(n))$.
This means that for a $n = 100$, the probability that Contrast finds the minimum cut would be :
$(2/(100 * 99)) = 2.02 * 10^{-4}$. In the other hand, The lower bound of FastCut would be : $1/log(100)$ which is around 0.3 and 0.5.
We can deduce from the formulas that the more nodes we have in the graphs, the greater the probability to not find the minimum cut increases. And it increases faster for the Contrast algorithm. This explains why, Contrast has a worst success rate than Fastcut.

In contrast to their success rates, Contrast is faster than FastCut with a complexity of $O(n^2)$ where we can contract an edge in $O(n)$. And Fastcut is done in $O(n^2 log(n))$ which is a bit slower but it guarantees better results at the end.

Moreover, we can make FastCut a bit faster by changing the Brute_force algorithm, called when we have 6 or less nodes, by $Contrast(G, 2)$ without losing too much of its success rate. The reason is that the complexity of Brute_force is $O(2^n * E)$ where E is the number of edges, is much greater than the complexity of Contrast. By example, with $n = 6$ instead of $2^6 * E = 64 * E$ we would have $6^2 = 36$. And the probability to not find the minimal cut with Contract and less than 6 nodes in the graph is very low. After some tests, the success rate of FastCut with Contract(G,2) instead of Brute_force was also 100% for each number of nodes tested above.

5 Conclusion

In conclusion, we can see some differences between Contract (algorithm) and FastCut.
If we want speed but not 100% of accuracy, we can use Contract. It works well for really small graphs (2 to 6 nodes). But if we want accuracy, FastCut is better. It is slower due to the repetitions but guarantees better results for the minimal cut, due to its much better probability to find the minimal cut. In addition, FastCut works for small and big number of nodes.
Moreover, Fastcut is faster than the Brute_force algorithm while having nearly the same accuracy.

Thus, we can conclude that FastCut is the best choice and the most effective algorithm to find the minimal cut of a graph.

Appendix

A Graph examples

Here, you will find some examples of the graphs generated.

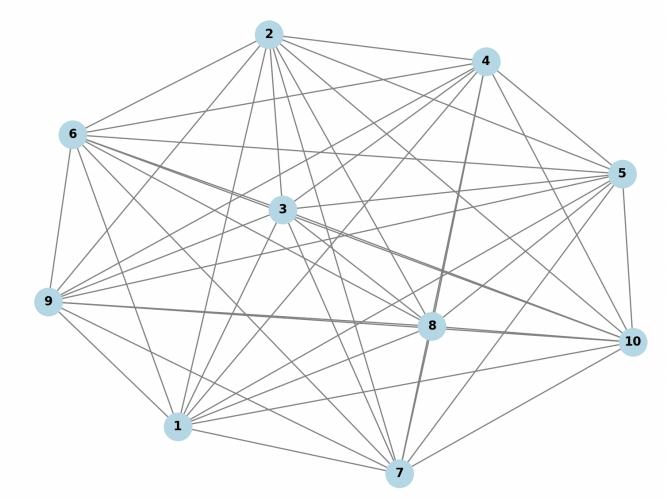

Figure 4: Example of a complete graph with 10 nodes

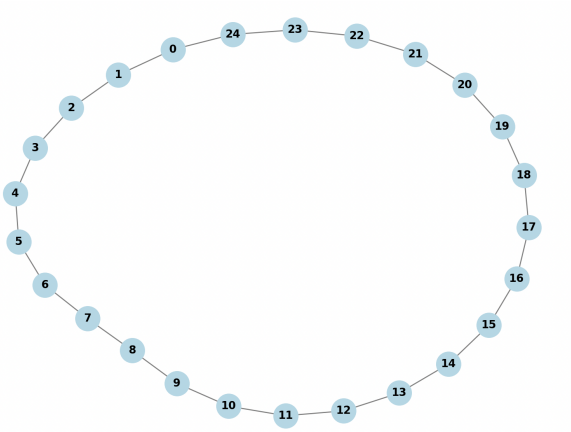

Figure 5: Example of a cycle graph with 25 nodes

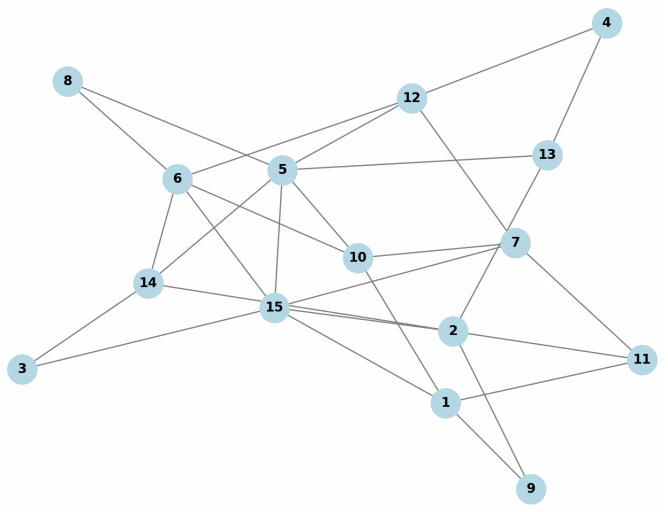

Figure 6: Example of a bipartite graph with subset 1 : 7, subset 2 : 8 and a probability of 0.5 to connect the new node to previous nodes

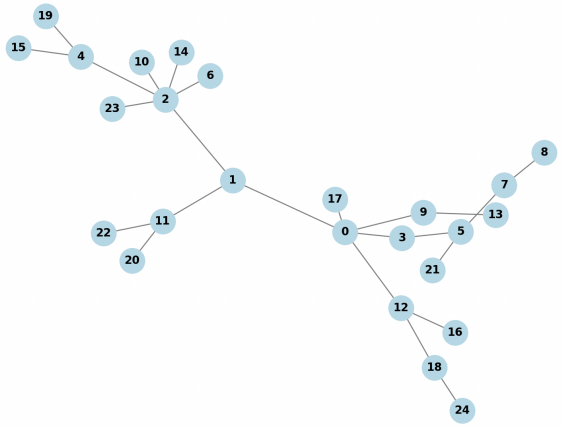

Figure 7: Example of a tree with 25 nodes

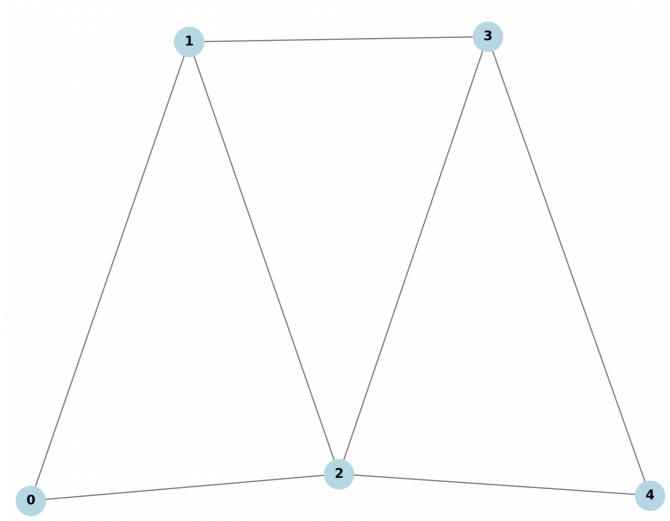

Figure 8: Example of a planar graph with 5 nodes and 7 edges

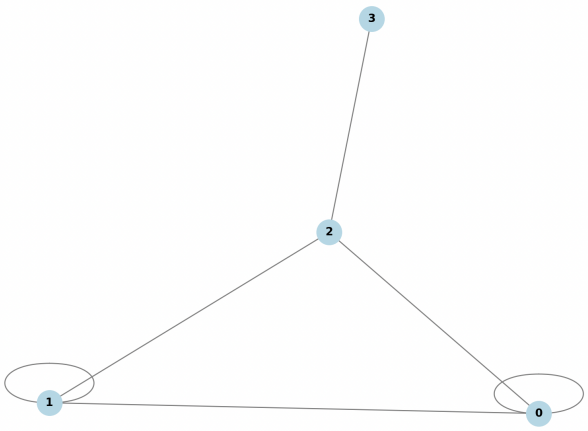

Figure 9: Example of a multi-graph with 4 nodes and 6 edges

B Code

B.1 Graph data structure

Here, you will find the implementation of the Edge, Node and Graph classes :

```python
class Edge:
    """ Class representing an edge in a graph """
    def __init__(self, id, start_node, end_node):
        """ Initialize an edge with an id, start node, and end node """
        self.id : int = id
        self.start_node : Node = start_node
        self.end_node : Node = end_node

    def get_other_node(self, node):
        """ Get the other node connected to the edge """
        if node == self.start_node:
            return self.end_node
        else :
            return self.start_node

    def create_loops(self, node1, node2):
        """ Check if the edge creates a self-loop with two nodes"""
        return (self.start_node == node1 and self.end_node == node2) or
            (self.start_node == node2 and self.end_node == node1)
```

Figure 10: Code for the Edge class

```python
class Node:
    """ Class representing a node in a graph """
    def __init__(self, id):
        """ Initialize a node with an id and an empty list of edges """
        self.id : str = id
        self.edges : list[Edge] = []

    def get_edges(self):
        """ Get the edges connected to the node """
        return self.edges

    def add_edge(self, edge):
        """ Add an edge to the node """
        self.edges.append(edge)

    def set_edges(self, edges):
        """ Set the edges of the node """
        self.edges = edges

    def get_neighbors(self):
        """ Get the neighbors (nodes connected to current one by an edge) of the
            node """
        neighbors :list[Node] = []
        for edge in self.edges:
            neighbors.append(edge.get_other_node(self))
        return neighbors
```

Figure 11: Code for Node class

```python
class Graph:
    """ Class representing a graph """
    def __init__(self, edges = None):
        """ Initialize a graph with an empty list of nodes and edges """
        self.nodes : list[Node] = []
        self.edges : list[Edge] = []
        if edges is not None:  # Create a graph from a list of edges if provided
            self.create_graph_from_edges(edges)


    def create_graph_from_edges(self, edges):
        """ Create a graph from a list of edges by adding nodes and edges """
        for edge in edges:
            if edge.start_node not in self.nodes:
                self.nodes.append(edge.start_node)
            if edge.end_node not in self.nodes:
                self.nodes.append(edge.end_node)
            if edge not in self.edges:
                self.edges.append(edge)

    def get_nb_nodes(self):
        """ Get the number of nodes in the graph """
        return len(self.nodes)

    def add_node(self, node):
        """ Add a node to the graph """
        self.nodes.append(node)


    def set_edges(self):
        """ Set the edges of the graph from the nodes"""
        for node in self.nodes:
            for edge in node.get_edges():
                if edge.id not in self.edges:  # a retravailler
                    self.edges.append(edge)
```

```python
def contract_edge(self, edge):
    """ Contract an edge by merging its two nodes and deleting the edge(s)
        connecting them"""
    edges_to_update : list[Edge] = []  # Edges to be updated (to connect to the
        new node)
    edges_to_remove : list[Edge] = []  # Edges to be removed ("self-loops")
    node1 : Node = edge.start_node
    node2 : Node = edge.end_node
    for e in self.edges:  # Identify edges connected to the nodes being
        contracted
        if e.start_node in (node1, node2) or e.end_node in (node1, node2):
            if e.create_loops(node1, node2):  # Self-loop check
                edges_to_remove.append(e)
            else:  # Edge to be updated (to connect to the new node)
                edges_to_update.append(e)

    new_node = Node(node1.id + "$" + node2.id)  # Create a new node (contracted
        node)
    for e in edges_to_update:  # Update edges to connect to the new node
        if e.start_node in (node1, node2):
            e.start_node = new_node
        if e.end_node in (node1, node2):
            e.end_node = new_node

    for e in edges_to_remove:  # Remove self-loops from edges and update graph
        structure
        self.edges.remove(e)

    new_node.set_edges(edges_to_update)  # Set the edges of the new node
    self.nodes.append(new_node)  # Add the new node to the graph
    self.nodes.remove(node1)  # Remove the old nodes
    self.nodes.remove(node2)
```

Figure 12: Code for the Graph class

B.2 Contract

Here, you will find the code for the Contract algorithm. It is based on the pseudo code from the

```python
import random
from graph import Graph, Edge

def contract(G : Graph, t = None) -> list[Edge]:
   """ Contract edges of a graph until it reaches a 2 or t nodes.
   The goal is to find the minimum cut of the graph G """
   min_nb : int = 2 if t == None else t  # min number of nodes
   while G.get_nb_nodes() > min_nb:
      edge : Edge = random.choice(G.edges)
      G.contract_edge(edge)
      if G.get_nb_nodes() == min_nb: # If the number of nodes remaining is reached
         return G.edges
   return G.edges
```

Figure 13: Code for the Contract algorithm

B.3 FastCut

Here you will find the code for the FastCut algorithm and the Brute_force algorithm. FastCut is based on the pseudo code from the Chapter 10 of the textbook Randomized algorithms by Motwani and Raghava. And the Brute force is based on the fact that every enumeration must be tested.

```python
import math
from contract import contract
from graph import Graph, Edge
from copy import deepcopy
from itertools import combinations

def fastCut(G: Graph) -> list[Edge]:
    """ Compute the minimum cut of a graph using the fastCut algorithm and returns
        the min-cut"""
    n = G.get_nb_nodes()
    if n <= 6:
        return minCutBruteForce(G) # min cut by brute force
    else:
        t :int =  math.ceil(1 + (n/math.sqrt(2)))
        g1 : Graph = deepcopy(G)
        g2 : Graph = deepcopy(G)
        H1 = Graph(contract(g1, t))  # Contract the graph G until it reaches t nodes
        H2 = Graph(contract(g2, t))
        cut1 : list[Edge] = fastCut(H1)  # Recursively call fastCut on the two
            subgraphs
        cut2 : list[Edge] = fastCut(H2)
        return cut1 if len(cut1) <= len(cut2) else cut2  # Return the min-cut of
            the two subgraphs
```

Figure 14: Code for the FastCut algorithm

```python
def minCutBruteForce(graph: Graph) -> list[Edge]:
    """
    Compute the minimum cut of a small graph using brute force and returns the
        min-cut
    It uses the combinatorial approach to find the minimum cut.
    """
    nodes = graph.nodes
    n = len(nodes)
    min_cut_edges = []
    min_cut_value = float('inf')

    # Generate all possible subsets S of nodes (except empty and full)
    for i in range(1, n):  # Subset sizes from 1 to n-1
        for subset in combinations(nodes, i):
            S = set(subset)
            T = set(nodes) - S  # The complement subset
            # Calculate the edges crossing the cut
            cut_edges = []
            for edge in graph.edges:
                if (edge.start_node in S and edge.end_node in T) or
                    (edge.start_node in T and edge.end_node in S):
                    cut_edges.append(edge)
            # Update the minimum cut if this one is better
            if len(cut_edges) < min_cut_value:
                min_cut_value = len(cut_edges)
                min_cut_edges = cut_edges

    return min_cut_edges
```

Figure 15: Code for the Brute_Force algorithm

B.4 Graph generators

The graphs generator have been created with the knowledges from the course INFO-F521, Graph theory and the help of Chat GPT to adjust some errors.
Some of the generators use randomness to create graphs.

```python
from graph import Graph, Node, Edge
import random


def complete_graph(n):
    nodes = [Node(id=str(i)) for i in range(1, n+1)]  # Create n nodes with string IDs

    # Create edges (complete graph: every pair of nodes is connected)
    edges = []
    edge_id = 1
    for i in range(len(nodes)):
        for j in range(i + 1, len(nodes)):
            edge = Edge(id=edge_id, start_node=nodes[i], end_node=nodes[j])
            edges.append(edge)
            nodes[i].add_edge(edge)
            nodes[j].add_edge(edge)
            edge_id += 1

    # Create the graph
    graph = Graph()
    for node in nodes:
        graph.add_node(node)
    for edge in edges:
        graph.edges.append(edge)
    return graph



def bipartite_graph(set1_size, set2_size, edge_probability=0.5):
    """
    Generates a random bipartite graph with two disjoint sets of nodes.

    :param set1_size: The number of nodes in the first set
    :param set2_size: The number of nodes in the second set
    :param edge_probability: The probability that an edge exists
    between a node in set 1 and a node in set 2
    """
    # Create nodes for both sets
    set1 = [Node(str(i)) for i in range(1, set1_size + 1)]
    set2 = [Node(str(i + set1_size)) for i in range(1, set2_size + 1)]

    # Create edges between nodes in set1 and set2 with a certain probability
    edges = []
    for node1 in set1:
        for node2 in set2:
            if random.random() < edge_probability:  # Edge probability
                edge_id = len(edges) + 1
                edge = Edge(edge_id, node1, node2)
                edges.append(edge)
                node1.add_edge(edge)
                node2.add_edge(edge)

    # Create the graph
    bipartite_graph = Graph(edges)
    return bipartite_graph
```

Figure 16: Complete and bipartite graphs generators

```python
def multigraph(n, m):
    """
    Generates a multigraph with n nodes and m random edges, allowing multiple edges between nodes

    :param n: Number of nodes in the graph
    :param m: Number of edges in the multigraph
    :return: A Graph object representing the multigraph
    """
    nodes = [Node(str(i)) for i in range(n)]
    edges = []

    for i in range(m):
        start_node = random.choice(nodes)
        end_node = random.choice(nodes)
        # Ensure there are multiple edges (if needed)
        edge = Edge(i, start_node, end_node)
        start_node.add_edge(edge)
        end_node.add_edge(edge)
        edges.append(edge)

    graph = Graph(edges)
    return graph

def planar_graph(n, m):  # n = number of nodes, m = number of edges
    """
    Generates a planar graph by starting with a tree and adding edges to ensure planarity.

    :param n: Number of nodes in the graph
    :param m: Number of edges in the planar graph
    :return: A Graph object representing the planar graph
    """
    nodes = [Node(str(i)) for i in range(n)]
    edges = []

    # Start by creating a tree (a connected acyclic graph)
    for i in range(1, n):
        start_node = nodes[i - 1]
        end_node = nodes[i]
        edge = Edge(i, start_node, end_node)
        start_node.add_edge(edge)
        end_node.add_edge(edge)
        edges.append(edge)

    # Add additional edges while maintaining planarity
    while len(edges) < m:
        start_node = random.choice(nodes)
        end_node = random.choice(nodes)
        if start_node != end_node:
            # Add the edge only if it doesn't create a cycle
            edge = Edge(len(edges), start_node, end_node)
            start_node.add_edge(edge)
            end_node.add_edge(edge)
            edges.append(edge)

    graph = Graph(edges)
    return graph
```

Figure 17: Multigraphs and planar graphs generators

```
def tree_graph(n):
    """
    Generates a random tree with n nodes.
    :param n: Number of nodes in the tree.
    """
    # Create an empty graph
    tree = Graph()  # Create an empty graph

    # Add the first node
    root = Node("0")
    tree.add_node(root)

    # Add subsequent nodes
    for i in range(1, n):
        parent_node = random.choice(tree.nodes)  # Pick a random parent node
        new_node = Node(str(i))
        edge_id = len(tree.edges)  # Edge id as the current number of edges
        edge = Edge(edge_id, parent_node, new_node)

        parent_node.add_edge(edge)  # Add edge to the parent node's edge list
        new_node.add_edge(edge)  # Add edge to the new node's edge list
        tree.add_node(new_node)  # Add the new node to the tree
        tree.set_edges()

    return tree

def cycle_graph(n):
    """
    Generates a cycle graph with n nodes.

    :param n: Number of nodes in the cycle graph
    :return: A Graph object representing the cycle graph
    """
    nodes = [Node(str(i)) for i in range(n)]
    edges = []

    for i in range(n):
        start_node = nodes[i]
        end_node = nodes[(i + 1) % n]  # Ensures that the last node connects back to the first node
        edge = Edge(i, start_node, end_node)
        start_node.add_edge(edge)
        end_node.add_edge(edge)
        edges.append(edge)

    graph = Graph(edges)
    return graph
```

Figure 18: Cycles and trees graphs generators