Université Libre de Bruxelles

Randomized Algorithms INFO-F413

Assignement Report
Karger's Algorithm

Student :
Achten Alexandre 494484

Professor :
Cardinal Jean

December 2024

Academic year 2024-2025

Contents

1 Introduction

While the best deterministic algorithm to solve the minimum cut problem runs in $\mathcal{O}(nm + n^2 \log n)$ [4], the use of randomized methods can speed things up and find a minimum cut with high probability in only $\mathcal{O}(n^2 \log n)$.
The goal of this report is to experimentally verify the success probability of the Contraction Algorithm 1 and the FastCut Algorithm 2. Then the theoretical time complexity of both algorithms will be verified.

2 Minimum Cut

Let $G(V, E)$ be an undirected multigraph with n vertices and m edges. A multigraph is permitted more than one edge between any given pair of vertices. A cut in G is a partition of the vertices $V = (C, \bar{C})$ into two non-empty sets ; we refer to this as the cut C with the understanding that $\bar{C}$ is $V \backslash C$. The value or size of a cut C is the number of edges crossing the cut, i.e., edges with one end-point in each of the two sets C and $\bar{C}$. A multiple edge will contribute its multiplicity to the value of the cut. A min-cut is a cut of minimum value; the min-cut problem is that of finding a min-cut in an input graph G [6].

2.1 Contraction Algorithm

The contraction algorithm (or Karger's algorithm) is based on the contraction operation. Given an edge (x, y) in a multigraph $G(V, E)$, a contraction of the edge (x, y) corresponds to replacing the vertices x and y by a new vertex z, and for each $v \notin \{x, y\}$ replacing any edge (x, v) or (y, v) by the edge (z, v) ; the rest of the graph remains unchanged. Any multiple edges created are to be retained. The graph obtained by this contraction is denoted by $G/(x, y)$ [3] [6].
The contraction algorithm is presented at the Algorithm 1 [6].

Algorithm 1 Contraction Algorithm

Require: A multigraph $G(V, E)$
Ensure: A cut C
1: $H = G$
2: **while** H has more than 2 vertices **do**
3: Choose an edge (x, y) uniformly at random from the edges in H
4: $F = F \cup \{(x, y)\}$
5: $H = H/(x, y)$
6: **end while**
7: $(C, \bar{C})$ = the sets of vertices corresponding to the two meta-vertices in $H = G/F$
8: **return** $(C, \bar{C})$

The Algorithm 1 is implemented in a different way for the experiments, by just keeping the minimum value of the cut, this to ensure the speed of the program.

In [3], it is proven that the Contraction algorithm produces a particular min-cut in G with probability $\Omega(n^{-2})$. It is also proven in [6], that the Contraction algorithm

can be implemented to run in $\mathcal{O}(n^2)$ on any n-vertex multigraph G.

2.2 Fastcut Algorithm

The approach of FastCut is that instead of using a slower deterministic algorithm, it is better to use two independent invocations of the Algorithm Contract itself on the contracted graph with t vertices. This is because the two repetitions boost the probability of success on the smaller instance, while the cost of the repetition on this instance is not as much as the cost of repeating the entire algorithm ; in fact, this effect multiplies with each successive stage of the recursion. We now specify the algorithm more precisely : first use contractions to reduce the number of vertices to roughly $n/\sqrt{2}$, and then recursively compute the min-cut in the resulting graph ; perform this twice and choose the smaller of the two min-cuts obtained as the final output. The resulting recursive algorithm is summarized in the Algorithm 2 [6].

Algorithm 2 FastCut Algorithm

Require: A multigraph $G(V, E)$
Ensure: A cut C
1: $n = |V|$
2: **if** $n \leq 6$ **then**
3: **return** Min-cut of G by brute-force enumeration
4: **else**
5: $t = \lceil 1 + \frac{n}{\sqrt{2}} \rceil$
6: $H_1 = ContractionAlgorithm(G, t)$, contraction sequences with t vertices
7: $H_2 = ContractionAlgorithm(G, t)$, contraction sequences with t vertices
8: $(C_1, \bar{C_1}) = FastCut(H_1)$
9: $(C_2, \bar{C_2}) = FastCut(H_2)$
10: **return** Minimum value of $\{(C_1, \bar{C_1}), (C_2, \bar{C_2})\}$
11: **end if**

In [6], it is proven that the FastCut algorithm runs in $\mathcal{O}(n^2 \log n)$ and that it finds a min-cut with probability $\Omega(1/\log n)$.

3 Experiments

3.1 Computing environement

The program is compiled with the g++17 compiler, using the -O3 flag. The code is executed on the Ubuntu 22.04.5 LTS on Windows 10 x86_64 OS. The CPU is the Intel i7-1065G7 (8) @ 1.497GHz and the RAM size is 8GB.

3.2 Graphs families

The experiments were realized on different graph families that have different properties. They are detailed in this section.

3.2.1 Complete graphs

A complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge. The complete graph on n vertices is denoted by K_n. An example is presented at Figure 1.

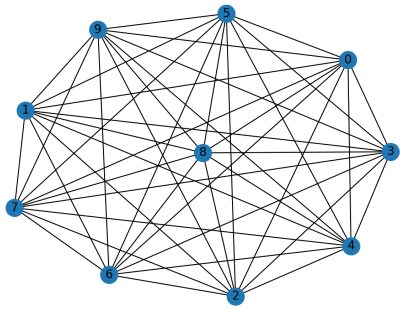

Figure 1: Complete graph K_{10}

3.2.2 Barbell's graphs

The n-barbell graph is the simple graph obtained by connecting two copies of a complete graph K_n by a bridge. It is used here to see if the algorithm handles effectively extreme cases. An example is presented at Figure 2.

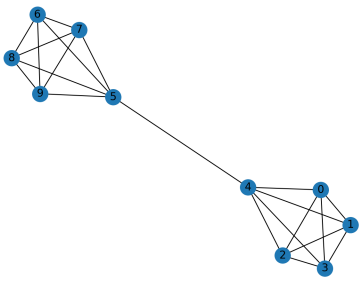

Figure 2: Barbell's graph K_5

3.2.3 Erdos-Rényi graphs

The Erdos-Rényi model is used to generate radom graphs with certain properties. The graph is constructed by connecting nodes randomly. Each edge is included in the graph with probability p (where $0 \leq p \leq 1$), independently from every other edge. The model is written $G(n, p)$.

Because we want a fully connected graph, we use the properties described in [1]. It is said that the graph in $G(n, p)$ will almost surely be connected if $p > \frac{(1+\varepsilon)\ln n}{n}$. In our case, n vary between 10 and 200, which corresponds roughly to $p > 0.02$. To be sure, $p = 0.8$ was chosen to generate the graphs. An example of a random graph generated with this model is presented at Figure 3.

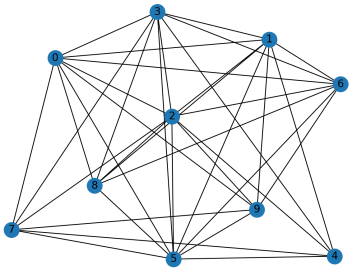

Figure 3: Erdos-Rényi graph $G(10, 0.8)$

The same model was used to generate random weigted graph, by simply assigning a weight $w \in \mathbb{Z}$ where $w \in [1, 10]$ to every edge of the graph.

3.2.4 Watts-Strogatz graphs

The Watts-Strogatz model is used to generate random graphs with small-world properties. It accounts for clustering while retaining the short average path lengths of the Erdos-Rényi model. It does so by interpolating between a randomized structure close to Erdos-Rényi graphs and a regular ring lattice. The graphs are designed by 3 parameters : n the number of vertices, K the mean degree of the vertices (can be interpreted as vertices are joined with its K nearest neighbors in a ring topology) and β a parameter where $0 \leq \beta \leq 1$. The parameters must be $n >> K >> \ln K >> 1$ to generate a graph with n vertices and $\frac{Kn}{2}$ edges. In the experiments, $K = 2\lceil \frac{n}{8} \rceil$ and $\beta = 0.4$. An example is presented at Figure 4.

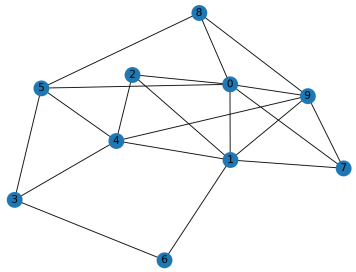

Figure 4: Watts-Strogatz graph $n = 10, K = 4, \beta = 0.4$

Varying β makes possible to interpolate between regular lattice ($\beta = 0$) and structure close to Erdos-Rényi grapg $G(n, p)$ with $p = \frac{K}{n-1}$ at $\beta = 1$. But every vertice will at least have an edge with $\frac{K}{2}$ other vertices.

3.2.5 Barabasi-Albert graphs

The Barabasi-Albert model is used to generate random scale-free networks using a preferential attachment mechanism. Preferential attachment means that the more connected a vertice is, the more likely it is to receive new edges. Vertices with a higher degree have a stronger ability to grab edges added to the network. The parameters of this model are n the number of vertices and m the number of edges to attach from a new vertices to existing vertices with a probability that is proportional to the number of links that the existing vertices already have. In the experiments, $m = \lceil \frac{n}{2} \rceil$. An example is presented at Figure 5.

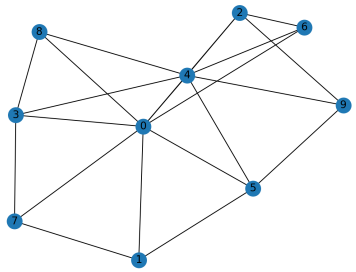

Figure 5: Barabasi-Albert graph $n = 10, m = 3$

3.3 Experimental plan

The graphs used to test the algorithms are generated using the NetworkX library in Python [2]. Once generated, the minimum cut of each graph is calculated with the NetworkX library using the Stoer-Wagner algorithm. This algorithm is deterministic and finds the minimum cut in $\mathcal{O}(nm + n^2 \log n)$ time for undirected, weighted graphs (n the number of vertices and m the number of edges) [4].

The Contraction and FastCut algorithms are written in C++. They take as input

the graph data generated and the precomputed mincut, then compare the precomputed value with the value found by the Contraction and FastCut algorithm to evaluate the success of the algorithm.
Two different experiments were realized :

1. The algorithms are given a budget time $t = 5s$ to run as many times as possible. Then the success probabilities are calculated based on the number of times it could run during the budget time.

2. The algorithms runs a fixed number of times on each graph $iter = 30$, then the average time and success probability are calculated.

4 Results

Both Contraction and FastCut algorithms were tested on all graph families presented in Section 3.2.

4.1 Limited time budget

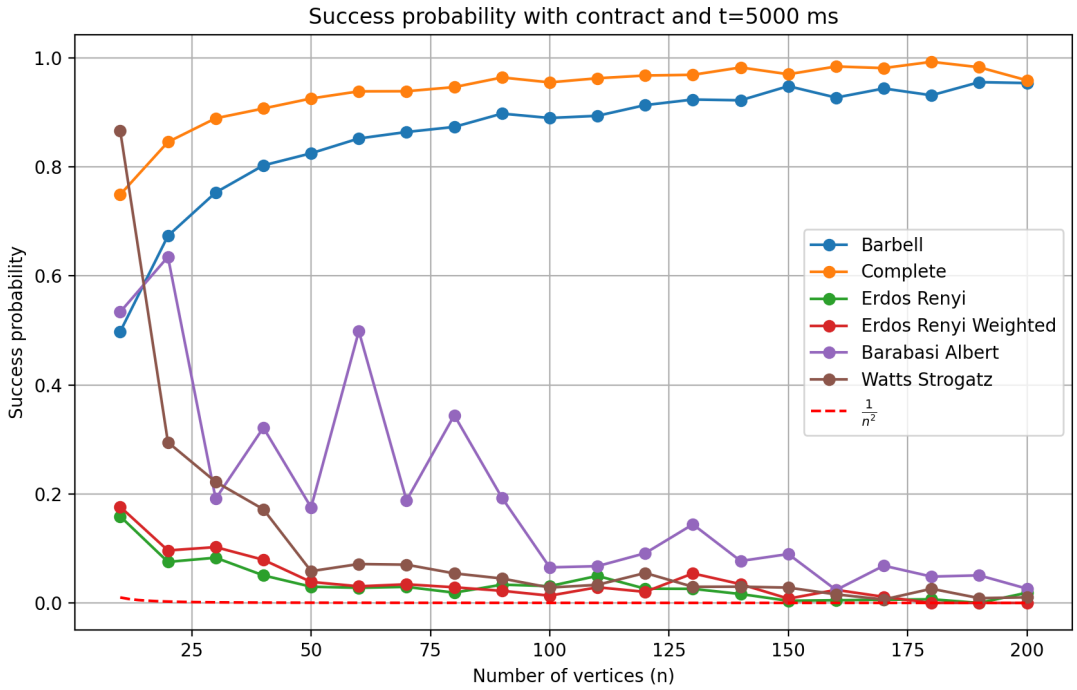

Figure 6: Success probability of Contraction algorithm with time budget $t = 5s$

The theoretical success probability is lower bounded by $\Omega(n^{-2})$ for the Contraction algorithm. In the Figure 6, we observe that the graph family have a huge impact on the probability to find the min-cut, especially for the complete and Barbell graphs. This may be due to the fact that barbell is composed of 2 complete graphs. The min-cut is trivial to find in both cases, in the complete graph it's $mincut = n - 1$ and for barbell it's $mincut = 1$. For the other graph families, they stay above the

bound. We can conclude that the theoretical bound is verified.

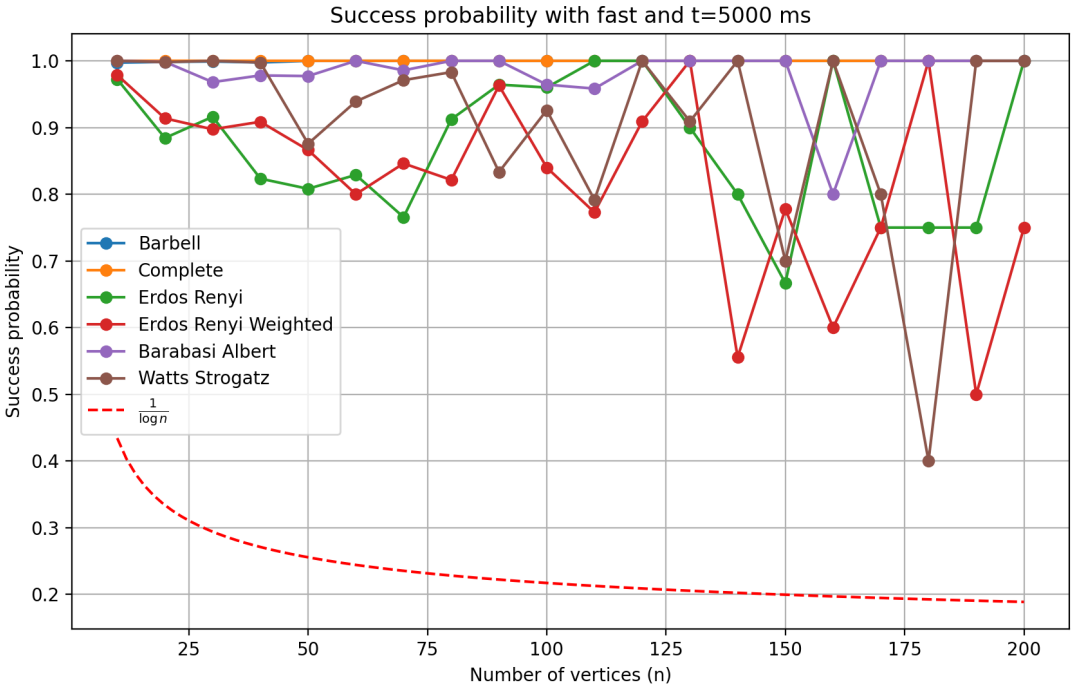

Figure 7: Success probability of FastCut algorithm with time budget $t = 5s$

The theoretical success probability is lower bounded by $\Omega(\frac{1}{\log n})$ for the FastCut algorithm. In the Figure 7, the general success probability is really high, some graph families have a 100% success probability. The explanation is the same as for the Contraction algorithm. We can see that the success probability isn't robust for large number of vertices. This is surely related to the time budget that is the same for both algorithms. The Contraction algorithm has plenty more iterations and the FastCut, due to it's slowness, doesn't have many iterations, which makes the success probability unstable. But the success probability is still higher than that of the Contraction algorithm.

4.2 Fixed number of iterations

To compensate for the problem of the time budget for the FastCut, a fixed number of iterations is defined to allow the algorithms to be compared on another criterion. This also makes it possible to calculate the time complexity.

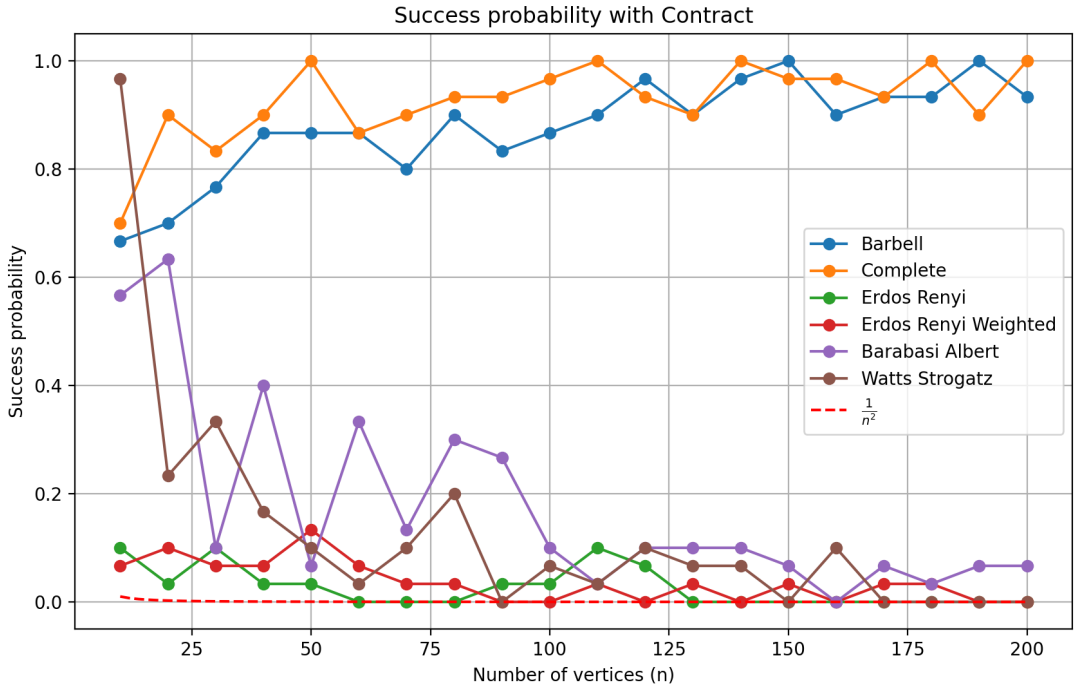

Figure 8: Success probability of Contraction algorithm with $iter = 30$

In comparison, Figure 8 is less stable than Figure 6.

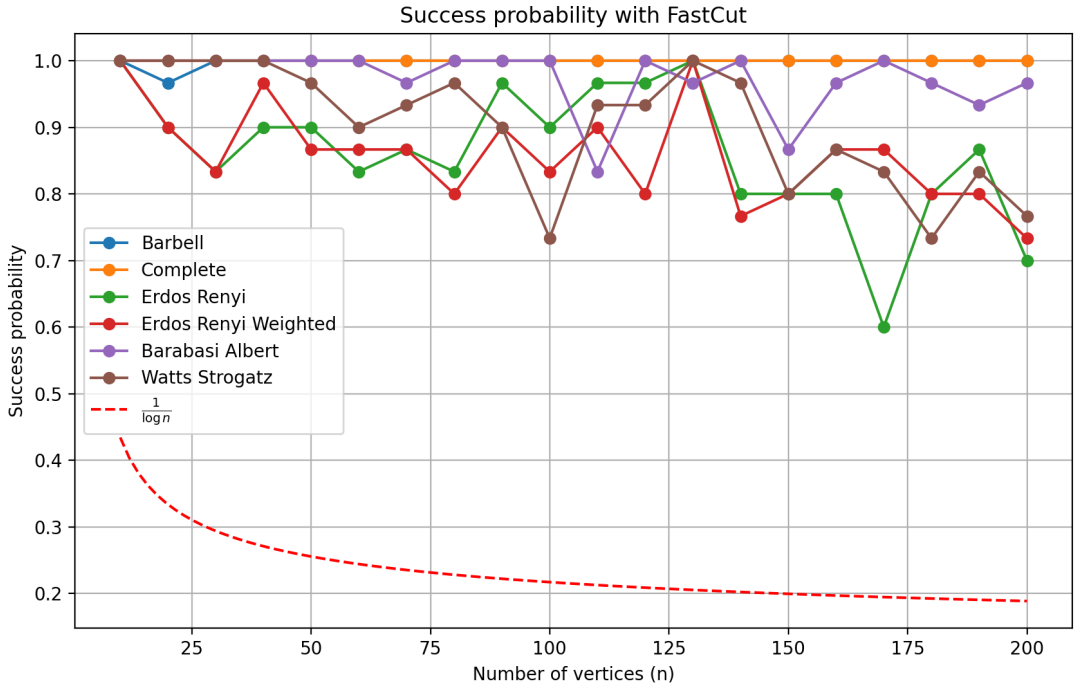

Figure 9: Success probability of FastCut algorithm with $iter = 30$

The Figure 9 have a more stable success probability for large number of vertices than Figure 7.

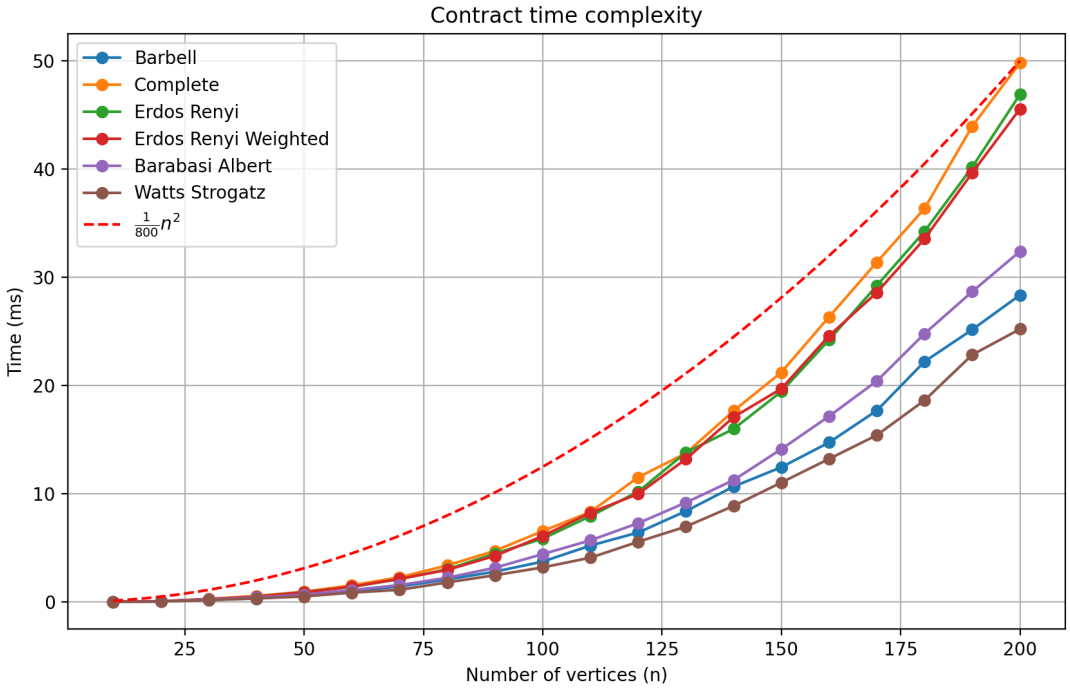

Figure 10: Time complexity of Contraction algorithm

In Figure 10, the difference between graph families is related to the number of edges. The selection of a random edge counts all the edges every time, so it has an negative impact on the time complexity. A better use of data structures could solve this issue. The theoretical bound is verified anyway.

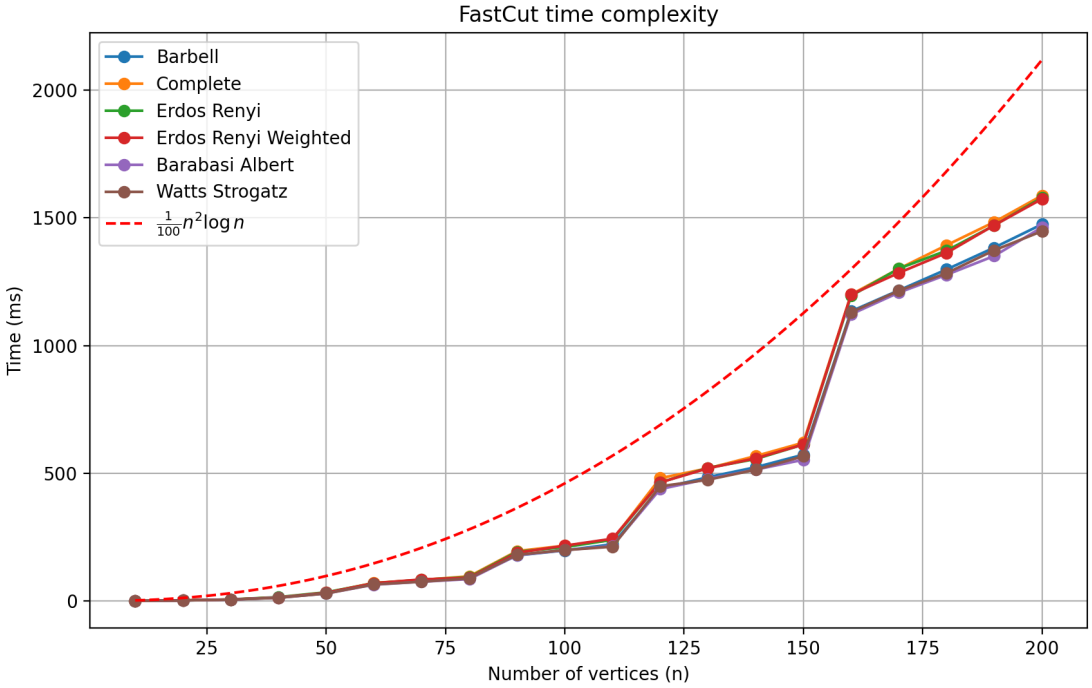

Figure 11: Time complexity of FastCut algorithm

In the Figure 11, the bound is verified.

5 Conclusion

While the Contraction algorithm is very fast, the FastCut algorithm have a much higher success probability. In real life applications, the FastCut algorithm will be preferred due to it's high success probability.

A Implementation of Graph Class

The Graph class allows to easily work with graphs. It stores the graph in a adjacency matrix form. The multiplicity of an edge is represented by an integer weight on the edge. Github project

```
class Graph {
public:
    vector<vector<int>> adjMatrix;
    Graph(int vertices); //Creates empty graph with vertices
    /**
    * @brief Add an edge to the graph
    * @param v The first vertex
    * @param w The second vertex
    */
    void addEdge(int v, int w);
    /**
    * @brief Add an edge to the graph with weight
    * @param v The first vertex
    * @param w The second vertex
    * @param weight The weight of the edge
    */
    void addEdge(int v, int w, int weight);
    int getVertices() const;
    void setMinCut(int mincut);
    int getMinCut() const;
    void setSubsetCut(set<int> subsetCut1, set<int> subsetCut2);
    set<int> getSubsetCut1() const;
    set<int> getSubsetCut2()const;
    /**
    * @brief Contract two vertices
    * @param u The first vertex
    * @param v The second vertex
    */
    void contract(int u, int v);
    /**
    * @brief Print the graph in the standard output in adjacency matrix f
    */
    void printGraph() const;

private:
    int vertices;
    int mincut;
    set<int> subsetCut1;
    set<int> subsetCut2;
};
Graph::Graph(int vertices) { //constructor
    this->vertices = vertices;
    adjMatrix.resize(vertices, vector<int>(vertices, 0));
}
```

```
void Graph::addEdge(int v, int w) {
    adjMatrix[v][w] = 1;
    adjMatrix[w][v] = 1;
}

void Graph::addEdge(int v, int w, int weight) {
    adjMatrix[v][w] = weight;
    adjMatrix[w][v] = weight;
}

void Graph::setMinCut(int mincut) {
    this->mincut = mincut;
}

void Graph::setSubsetCut(set<int> subsetCut1, set<int> subsetCut2) {
    this->subsetCut1 = subsetCut1;
    this->subsetCut2 = subsetCut2;
}

void Graph::contract(int u, int v) {
    for (int i = 0; i < vertices; i++) {
        adjMatrix[u][i] += adjMatrix[v][i]; //add the values
        //of the second vertex to the first vertex
        adjMatrix[i][u] += adjMatrix[i][v];
    }
    adjMatrix[u][u] = 0; //we don't want self loops
    for (int i = 0; i < vertices; i++) { //rearrange the matrix
        adjMatrix[i][v] = adjMatrix[vertices - 1][i]; //copy the last
        //row to the second vertex
        adjMatrix[v][i] = adjMatrix[i][vertices - 1];
    }
    adjMatrix[v][v] = 0; //we don't want self loops
    vertices--;
}

void Graph::printGraph() const{
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            cout << adjMatrix[i][j] << " ";
        }
        cout << endl;
    }
}
```

B Implementation of random edge selection

The random edge selection is something not trivial to realize. The weight of every edge is summed and then a random number is generated between 1 and the maximum value of the weights. This allows to take into account the mulitplicity (weight) of the edges when selecting randomly. Though this is not optimal, since the number of edges and their weight is recalculated every time, it is simple to understand how it works. Github project

```
Edge chooseRandomEdge(Graph& graph) {
    vector<Edge> edges;
    // Populate the edges vector with all edges and their weights
    int vertices = graph.getVertices();
    for (int i = 0; i < vertices; ++i) {
        for (int j = 0; j < i; ++j) { //Only the lower
            //triangle of the matrix
            if (graph.adjMatrix[i][j] != 0) {
                edges.push_back({i, j, graph.adjMatrix[i][j]});
            }
        }
    }
    // Calculate the total weight
    int totalWeight = 0;
    for (const auto& edge : edges) {
        totalWeight += edge.weight;
    }
    int randomWeight;
    // Generate a random number in the range [0, totalWeight)
    if (totalWeight == 0) {
        return Edge{-1, -1, 0};
    }
    else{
        randomWeight = rand() % totalWeight;
    }
    // Select the edge based on the random weight
    int cumulativeWeight = 0;
    Edge selectedEdge;
    for (const auto& edge : edges) {
        cumulativeWeight += edge.weight;
        if (randomWeight < cumulativeWeight) {
            selectedEdge = edge;
            break;
        }
    }
    return selectedEdge;
}
```

C Implementation of Contraction algorithm

Github project

```
int contractAlgorithm(Graph& graph, int t) {
    int vertices = graph.getVertices();
    while (vertices > t) {// until t vertices left (default t=2)
        Edge edge = chooseRandomEdge(graph);
        graph.contract(edge.u, edge.v); //inside the number of
                                        //vertices is decremented
        vertices--; //nb of vertices is decremented
        //for the loop (can access the number of vertices)
    }
    // Calculate the value of the mincut to return
    int mincut = 0;
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < i; j++) { //Only the lower triangle
                                      of the matrix
            mincut += graph.adjMatrix[i][j];
        }
    }
    return mincut; //Divide by 2 because we are counting
    twice the edges (matrix symmetric)
}
```

D Implementation of FastCut algorithm

Github project

```
int fastCut(Graph& graph) {
    int n = graph.getVertices();
    if(n <= 6) { // if the number of vertices is less than 6
        return bruteForceMinCut(graph);
    }
    else{
        int t = ceil(1 + n/sqrt(2));
        Graph originalGraph = graph; //Save the original graph to avoid
        copying it again
        contractAlgorithm(graph, t);
        int cutVal1 = fastCut(graph);

        graph = originalGraph;
        contractAlgorithm(graph, t);
        int cutVal2 = fastCut(graph);

        return min(cutVal1, cutVal2);
    }
}
```

References

[1] Rényi A. Erdos P. "On Random Graphs I". In: *Publicationes Mathematicae* (1959).

[2] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. 2008.

[3] David R. Karger. "Global min-cuts in RNC, and other ramifications of a simple Min-Cut Algorithm". In: *Stanford* (1992).

[4] Frank Wagner Mechthild Stoer. "A Simple Min-Cut Algorithm". In: *Freie Universitat Berlin* (1997).

[5] Eli Upfal Michael Mitzenmacher. *Probability and Computing : Randomized Algorithms and Probabilistic Analysis*. en. 2005.

[6] Prabhakar Raghavan Rajeev Motwani. *Randomized Algorithms*. en. Press Syndicate of the University of Cambridge, 1995.