Data Structures and Algorithms : Karger's Algorithm

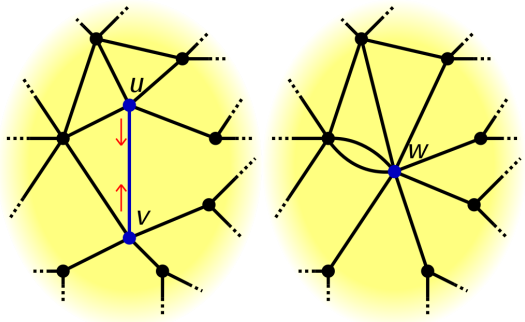
Homework 1

Julien Baudru

October 22, 2021

Introduction

The goal of this first project of the Data Structures and Algorithms course was, on the one hand, to implement the *Contract* and *FastCut* algorithms and, on the other hand, to analyze the results produced by them. These two algorithms allow to compute the minimum cut of a graph **G**, the minimum cut being the minimum number of edges that must be removed from the graph **G** to produce two sets of unconnected vertices. Here we speak about *multigraph* because **G** has parallel edges.



Q1 - Theorem 1

The first theorem is the following : **The running time of the FastCut algorithm is** $O(n^2 \ log \ n)$**.**

The recursion depth of this algorithm is 2 log n as the size of the graphs is reduced by a factor of $1/\sqrt{2}$ between recursive calls. We also know that the recursion stops when the graph has only 6 vertices (noted n). At each loop of the algorithm two subgraphs are created, thus doubling the number of subproblems at each recursion step but reducing the size by a factor of $\sqrt{2}$, so the problem size grows in each recursion step by a same factor of $\sqrt{2}$. Since we know that the recursion depth is 2 log n then the total growth factor is $\sqrt{2}^{2 \ log \ n}$.

Therefore we can calculate an upper bound for the total number of vertices in all the graphs constructed. We denote $V(n)$ the maximum number of vertices in all constructed graphs.

$$V(n) \leq 2n + 2V(\frac{n}{\sqrt{2}}) \tag{1}$$

$$V(n) \leq 2(n + 2^1 \frac{n}{\sqrt{2}} + 2^2 \frac{n}{\sqrt{2}^2} + ...) \tag{2}$$

Reducing the above equation we obtain that the maximum number of vertices $V(n)$ is upper bounded by n^2. And since the recursion depth is 2 log n, we have that this algorithm has a complexity in time of $O(n^2 \ log \ n)$.

Q2 - Coding Contract and FastCut

All data structures and algorithms were written in **Python 3.8.8**. The code written for this project is divided into two files, on the one hand a file *main.py* containing the *Contract* and *FastCut* algorithms as well as the function *Compare* allowing to assign a maximum execution time to the algorithms. On the other hand, the file *graph.py* containing the class **Graph** with different functions explained in detail further on in these pages.

All libraries used for this project are not native to Python so make sure that the following modules are installed on your computer in order to run the program :

- *time*
- *copy*
- *random*
- *math*
- *matplotlib*
- *numpy*
- *itertools*
- *networkx*

To run this program enter the following command in the folder where the *.py* files are located :

```
python main.py
```

For this project, the *NetworkX* library could have been used for contraction but this choice was not made because the way the contraction function works on multigraphs for this library does not correspond exactly to the one defined in the course. That's why I decided to code a graph class from scratch as well as the contraction function. The *Networkx* module has been used here only for the random generation of graphs. The **Graph** class contains the following functions :

- *edges(vertex)* - Return the list of edges connected to a given vertex
- *all_vertice()* - Return all the vertice in the graph
- *all_edges()* - Return all the edges in the graph
- *del_vertice(vertex)* - Delete a given vertex in the graph
- *del_edge(edge)* - Delete a given edge in the graph
- *replace_vertice(src, dest)* - Replace the source vertex by the destination vertex in the graph
- *delete_loop()* - Delete self-loop in the graph
- *get_random_edge()* - Return a edge pseudo-randomly selected in the graph
- *generate_graph()* - Generates different types of random graphs via the *NetworkX* library
- *contract_edge(edge)* - Contract the given edge and return the resulting graph
- *brute_force_min_cut()* - Return the minimal cut from the graph

Contract

```
def contractAlgo(G, t):
    while(len(G.all_vertices()) > t):
        edge = G.get_random_edge()
        G = G.contract_edge(edge)
    return G
```

This function runs until the number of vertices remaining in the graph **G** is greater than t. When this function is called alone the value of t is fixed to 2 but when it is called within the function **fastCutAlgo** (see next section) t is fixed to $\lceil 1 + n/\sqrt{2} \rceil$.

FastCut

```
1  def fastCutAlgo(G):
2      if(len(G.all_vertices()) <= 6):
3          edges = G.brute_force_min_cut()
4          return edges
5      else:
6          t = int(1 + (len(G.all_vertices()) / math.sqrt(2))) + 1
7          H1 = contractAlgo(G, t)
8          H2 = contractAlgo(G, t)
9          edges1 = fastCutAlgo(H1)
10         edges2 = fastCutAlgo(H2)
11         if(len(edges1)/2 < len(edges2)/2):
12             return edges1
13         else:
14             return edges2
```

Above the length of the minimum cut is divided by two in line 13 because the **contraAlgo** function returns the edges in the cut in both directions (e.g. $[B, C], [C, B]$).

Bruteforce

The main idea behind the bruteforce function used to find the minimal cut of the graph **G** is the following. First, we generate all possible combinations of two subsets of vertices S1 and S2. (Ex: If $G = \{A, B, C\}$ then $S1 = \{A\}$ and $S2 = \{B, C\}$ or $S1 = \{B\}$ and $S2 = \{A, C\}$ or $S1 = \{C\}$ and $S2 = \{A, B\}$). Then, for each combination we look at how many edges connect the two subsets, which gives us one of the possible cuts. Finally we retain the cut with the smallest number of edges.

Compare

The **compare** function in *main.py* is used to answer the question in the next section. This function takes as argument a graph **G** and a time interval **maxtime**, so it will consecutively run the *Contract* and *FastCut* algorithms as many times as possible for the given time interval. This compare function is itself run several times with graphs having a different number of vertices in order to produce the summary diagrams presented below.

Q3 - Comparison of algorithms

Theorem 2

The second theorem is the following : **The FastCut algorithm succeeds in finding a minimum cut with probability** $\Omega(1/log\ n)$**.**

Analysis of the success probabilities of the two algorithms

The probability of success of both algorithms is calculated as follows : Before launching the algorithm, we calculate the minimal cut of **G** via the *Bruteforce* function, then each time an iteration of the algorithm gives a cut equal to the one calculated by the *Bruteforce* we consider this as a success. The probability of success will therefore be given by $number\_of\_successes/number\_of\_iterations$.

For each diagram appearing later in these pages, the red line represents the function $y = 1/log\ n$ with n being the number of vertices in the graph **G**.

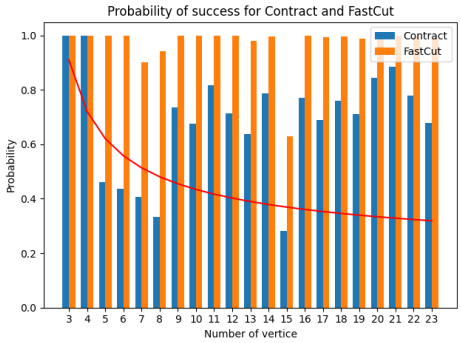
For both algorithms, each of the following graph families was tested with an increasing number of vertices (up to 24 vertices max., this limit of the number of vertices has been set arbitrarily because the time taken to generate all the diagrams is not negligeable) :

- *Barabasi-Albert graphs*
- *Complete graphs*
- *Multipartite complete graphs*

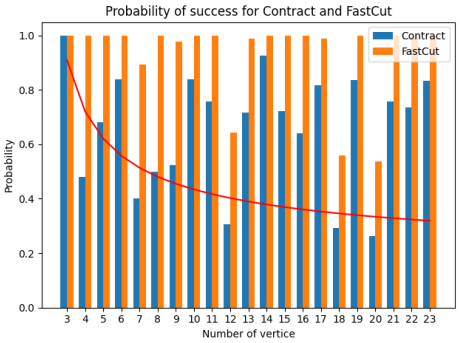
- *Turan graphs*
- *Wheel graphs*
- *Barbell graphs*
- *Random graphs*

And for each of these variations, the algorithms were run with the following time intervals : *1 sec, 0.5 sec, 0.1 sec, 0.05 sec, 0.01 sec* and *0.005 sec*. For the sake of readability, only the diagrams for the 1 second and 0.05 second time intervals are shown here.

Barabasi-Albert graphs

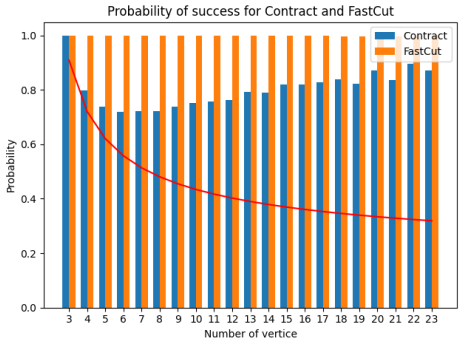


(a) timemax = 1

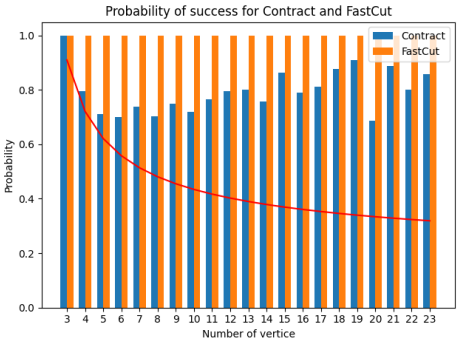


(b) timemax = 0.05

Complete graphs

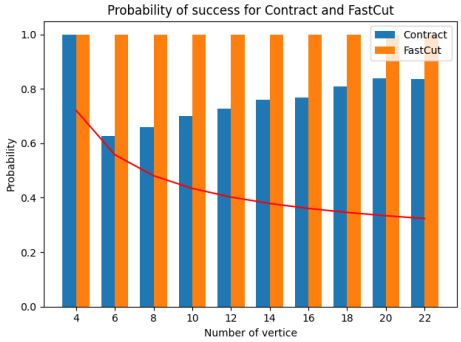


(a) timemax = 1

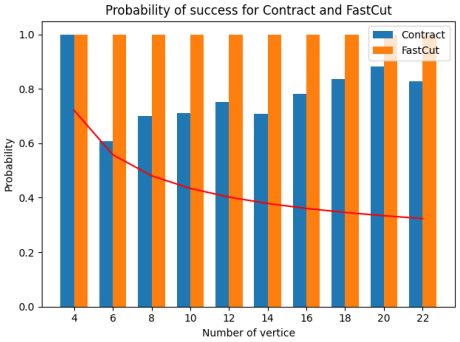


(b) timemax = 0.05

Multipartite complete graphs

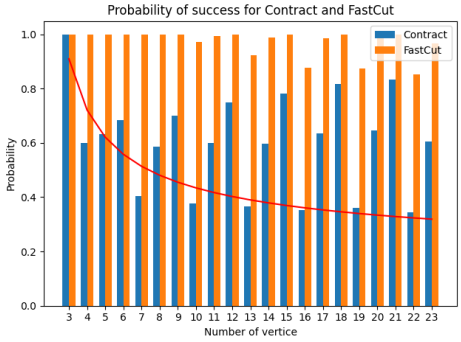


(a) timemax = 1

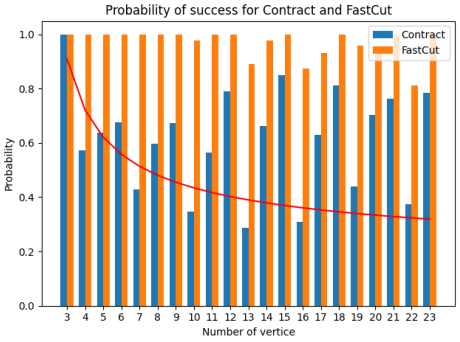


(b) timemax = 0.05

Turan graphs

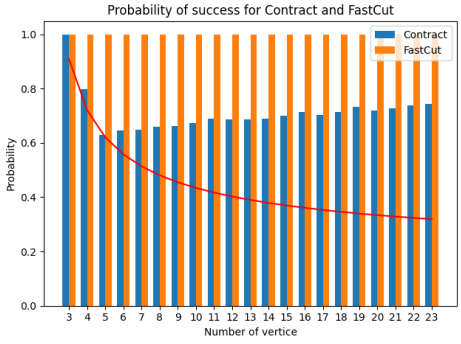


(a) timemax = 1

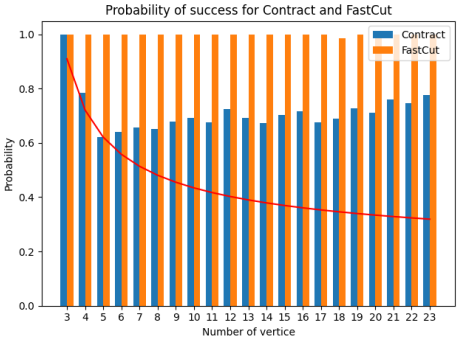


(b) timemax = 0.05

Wheel graphs

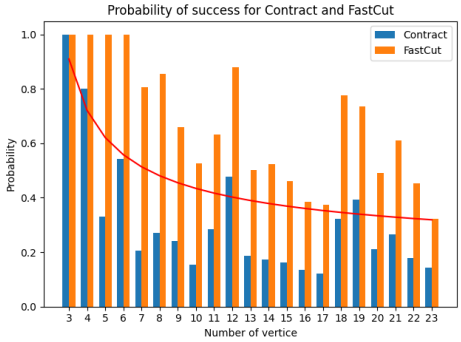


(a) timemax = 1

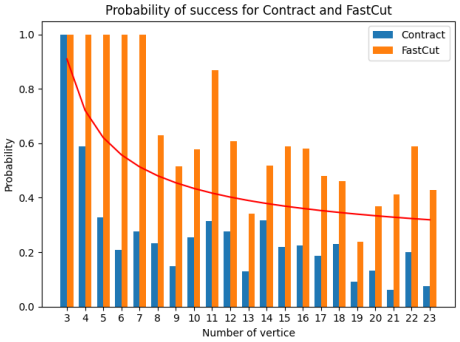


(b) timemax = 0.05

Random graphs



(a) timemax = 1



(b) timemax = 0.05

Conclusion

We notice that, in general, for all the families of graphs tested during the experiments, the success probability of the *FastCut* algorithm is higher than the probability of success of the *Contract* algorithm. This is not the case for the *Barbell graphs* with time intervals less than or equal to 0.05 seconds. Moreover, we notice that when the time interval given to the algorithms is too short, the calculation of probabilities is biased because they do not always have time to run more than once. Thus, if they find the minimal cut the first time, their probability of success will be fixed at 1, but in practice this is not necessarily the case.

We notice that for some family of graphs tested, after a few iterations with a rather high probability, the probability of success of the *Contract* algorithm drops and then seems to increase progressively with the number of vertices in the graph. This is the case for *Complete graphs* (multipart or not) and *Wheel graphs*. Besides, we notice an interesting phenomenon for the success probability of the *Contract* algorithm for *Turan graphs*. Indeed, for $n = 2$, this probability is fixed at 1 but for the following iterations it seems to oscillate in three steps.

Regarding the second theorem of this project, we notice that the success probability of the *FastCut* algorithm never goes below $1/log\ n$ for any n. This confirms that the probability of success of this algorithm is bounded from below by this value. The diagram for the *Random graphs* with a time interval of 1 second shows the tendency for the probability of success to follow the function in red. We note an exception for the family of *Random graphs* with a time interval smaller than or equal to 0.05 seconds, this may be due to the pseudo-random factor involved in the generation of these graphs.