Randomized MinCut

In this tutorial we will implement algorithms to compute the mincut of a multigraph. A multigraph G is just a graph where there can be several edges between any pair of vertices. There are no self-loops in a multigraph (any edge connects two *distinct* vertices). Figure 1 shows an example (for instance vertices 'c' and 'g' are connected by two edges). The *degree* of a vertex is the number of edges that are incident to it (taking the multiplicity of edges into account), for instance in Figure 1 the vertex c has degree 4 (and has 3 neighbours). Let V be the set of vertices. For $(S, \bar{S})$ a partition of V into two nonempty sets, the corresponding *cut* is the set of edges that connect a vertex of S to a vertex of $\bar{S}$. The *cutsize* is the number of edges in the cut. A *mincut* of G is a cut of minimal size among all the cuts of G. For instance in the graph of Figure 1 the mincut size is 3 and is obtained by taking the partition $V = \{a, c, d, g\} \cup \{b, e, f, h\}$. The mincut is an important graph parameter (if the mincut size is k, then an adversary can disconnect the graph by cutting k links in the graph ; a graph with small mincut is thus more vulnerable in the context of network disruption).

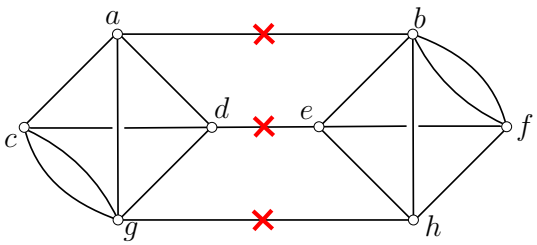

FIGURE 1 – A multigraph and its unique mincut

Download the files `mincut.py`, `MultiGraph.py` and `test_mincut.py`. The file `MultiGraph.py` contains a class `MultiGraph`, of which each instance `M` represents a multigraph. It has an attribute `adj` which is a dictionary associating to each vertex v its neighbors and for each neighbour w gives the number of edges between v and w. More precisely, testing if `x` is a vertex of `M` is done by `if x in M.adj`. If `x`, `y` are two vertices, testing if `x` and `y` are connected by at least one edge is done by `if y in M.adj[x]`, and if true, `M.adj[x][y]` gives the number of edges connecting `x` and `y`. There is also an attribute `deg` such that if x is a vertex of M then `deg[x]` gives the degree of x. The constructor in the class `MultiGraph` receives a list such as `L=[3,[['a','b',1],['c','b',2],['a','c',4]]]`, where $L[0]$ gives the number of vertices, and $L[1]$ gives the list of edges including multiplicities (for instance there are 4 edges connecting `a` and `c`). For instance the list `L_tutorial` given at the beginning of `mincut.py` represents the multigraph of Figure 1 written in this list format.

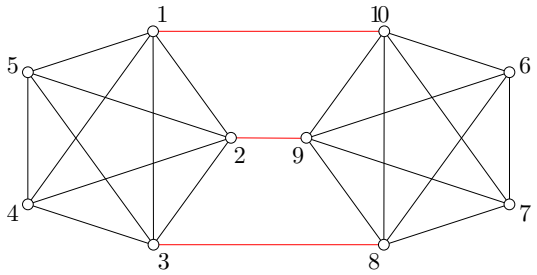

FIGURE 2 – The two-clique graph for $n = 5$.

The function `two_clique_graph(n)` in `test_mincut.py` outputs (written in this list format) the graph on $2n$ vertices, such that there is a clique for vertices with labels in $[1..n]$ (any pair of these

vertices is connected by an edge), there is another clique for vertices with labels in $[n+1..2n]$, and there are additionally $n-2$ edges, which connect vertex i to vertex $2n-i+1$ for i from 1 to $n-2$ (see Figure 2 for an illustration). This graph has mincut size $n-2$ which is uniquely obtained for the partition $\{1..n\} \cup \{n+1..2n\}$. It thus gives a graph with $2n$ vertices (for any $n \geq 1$) for which we know the answer of the mincut problem (this will be convenient to test our algorithms). There is also a constructor in the class `MultiGraph.py` to create a MultiGraph object from the list input format, and a display method.

1 A brute force mincut algorithm

A first naive approach is to run over all possible $2^{n-1}-1$ cuts of M, and return a cut of minimal size.

Question 1. If a multigraph G has n vertices, and if i is an integer between 1 and 2^n-2 (the binary representation of i has length n and is not made only of 1's nor only of 0's), there is a natural way to associate a cut $(S, \bar{S})$ to i : running over the set of vertices with an instruction `for x in self.adj`, the first x over the iteration is put in S iff i is odd, the second x is put in S iff $i >> 1$ is odd, the third x is put in S iff $i >> 2$ is odd, etc. Complete the method `cutsize(self,i)` of `MultiGraph`, which has to return the output under the format `[c,L]`, where L is the list of vertices of the above described set S (associated to i), and c is the size of the cut for the partition $(S, \bar{S})$.

Question 2. Complete the function `mincut_brute(m)` in `mincut.py`. As in Question 1 we return the output in the format `[c,L]`, where c is here the mincut size, and L gives the list of vertices in S, for some partition $(S, \bar{S})$ realizing a mincut. (Remark : since a cut $(S, \bar{S})$ is equivalent to $(\bar{S}, S)$, it is enough to run a loop over all odd integers i between 1 and 2^n-2). Test your function by running `test_mincut_brute()`.

2 A randomized algorithm

There is a famous randomized mincut algorithm due to Karger that has a very simple description and works in polynomial time. First we have to describe the operation of *contracting an edge* $\{u, v\}$ in a multigraph M. The *contracted multigraph* $M/\{u, v\}$ is obtained from M by merging u with v, keeping u as the name of the merged vertex, and deleting all the edges connecting u to v (that have become self-loops). We say that u has *absorbed* v. See Figure 3 for examples.

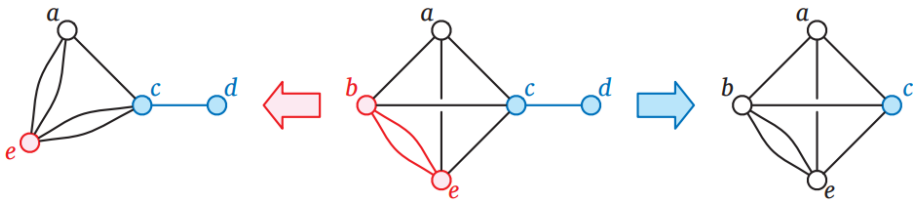

FIGURE 3 – A multigraph M and the two contracted multigraphs $M/\{e, b\}$ and $M/\{c, d\}$.

Question 3. Complete the method `contract(self,i,j)` that contracts an edge $\{i, j\}$ (beware of deleting the self-loops arising from the contraction, and of updating the attributes `deg` and `adj`). Test your method by running `test_contract()`.

A second ingredient in the randomized mincut algorithm is to be able to draw an edge uniformly at random in a multigraph. We will do it from a general-purpose function to draw a random element in a set where each element has a weight.

Question 4. For `dict` a dictionnary whose values are positive integers giving the weights of the elements, implement (at the beginning of `MultiGraph.py`) the function `random_element(dict)` that

returns a random element with probability proportional to the weight of the element. For instance, for `dict={'a':2, 'b':1, 'c':4}`, a call to `random_element(dict)` should return `'a'` with probability 2/7, return `'b'` with probability 1/7, and return `'c'` with probability 4/7. Test your function by running `test_random_element()`.

Question 5. Now, in `MultiGraph.py`, complete the method `random_vertex(self)` that should return a random vertex where the weight of each vertex is its degree. For instance in Figure 1, the sum of the degrees over all vertices is 34 (twice the number of edges), and the vertex `'b'` has degree 5. Hence a call to `self.random_vertex()` should return `'b'` with probability 5/34.

And complete the method `random_edge(self)` that should return an edge `(i,j)` taken uniformly at random (think of a random edge (i, j) as obtained from a random vertex i and then a random neighbour j of i). For instance, in the graph of Figure 1, there are 17 edges in total, 2 edges between `'c'` and `'g'`, and one edge between `'a'` and `'d'`. Hence for a call to `self.random_edge()`, the probability that the output is `('c','g')` or `('g','c')` is 2/17, and the probability that the output is `('a','d')` or `('d','a')` is 1/17. Test your method by running `test_random_edge()`.

For M a multigraph with n vertices, a random cut of M is obtained as follows : for i from 1 to $n-2$ we choose an edge uniformly at random in the current multigraph (which has $n-i+1$ vertices) and contract it. At the end there remain two vertices a, b and a bunch of $k \geq 1$ edges connecting them. This naturally corresponds to a partition $(S, \bar{S})$ of cutsize k (S consists of a and all the vertices that have been absorbed by a).

Question 6. In `mincut.py` complete the function `random_cut(m)` that outputs a random cut of the multigraph `m`, after contracting $n-2$ random edges (as described above). The format of the output should again be $[c, L]$ where c is the cutsize and L is the list of vertices in one of the two parts of the partition $(S, \bar{S})$ for the cut. In order to return L you should maintain at each step a dictionary `partition` such that for every x not already absorbed, `partition(x)` is the list of vertices that have been already absorbed by x (at the end, `partition` will just have two keys `a,b` corresponding to the two remaining vertices, and one can return L as `[a]+partition[a]`). To test your code, you can run `test_random_cut()`.

As we show at the end of the file, the probability that `random_cut(m)` outputs a mincut of a multigraph `m` with n vertices is at least $\frac{2}{n(n-1)}$. As seen in class, we can boost the success probability by repeating the process k times, and output a cut of smallest size among the k experiments. Then the probability of outputting a mincut becomes at least $1-(1-\frac{2}{n(n-1)})^k$. Hence for any fixed $e \in (0,1)$, this probability is guaranteed to be at least $1-e$ for $k = \lceil \log(e)/\log(1-2/(n(n-1))) \rceil$ (which is $O(n^2 \ln(1/e))$).

Question 7. Complete the function `mincut_karger(L,e)` that receives as input a multigraph (in the list format) and a real $e \in (0,1)$, and returns a cut of the multigraph, such that the returned cut is a mincut with probability at least $1-e$. To test your code you can run `test_mincut_randomized()`.

Remark. There is a clever improvement on the Karger algorithm due to Karger and Stein. Instead of contracting edges till having two vertices, one runs in parallel two independent scenarios of edge-contractions, where in each instance one stops when there remains about $n/\sqrt{2}$ vertices (say $1+\lceil n/\sqrt{2} \rceil$). Then one calls the algorithm recursively for each instance, so one gets two cuts (one for each instance). The one of minimal size is then returned. Due to recursivity, this amounts to having a kind of binary tree of contraction scenarios, such that one has a success trial if a mincut is found in at least one scenario (path from the root to a leaf in the binary tree of scenarios). This dramatically increases the probability of success of a trial, from $\Theta(1/n^2)$ to $\Theta(1/\log(n))$. Moreover, the threshold $n/\sqrt{2}$ to stop the contractions before the recursive calls is well suited so that the complexity of each trial remains of the same order (quadratic in n) as in the classical Karger algorithm.

Proof that `random_cut(M)` outputs a mincut with probability at least $\frac{2}{n(n-1)}$. We fix a mincut $C = (S, \bar{S})$ (where C is the set of edges between S and $\bar{S}$) in a multigraph M with n vertices. We are

going to prove that `random_cut(M)` outputs C with probability at least $\frac{2}{n(n-1)}$. Let k be the number of edges in C. Let $e_1, \ldots, e_{n-2}$ be the random edges (that are contracted) chosen at the successive steps. It is easy to get convinced that `random_cut(M)` outputs C iff none of the edges e_i is in C. We are going to give a lower bound to the probability that this happens.

A first observation is that each vertex v has degree at least k (indeed if we take $S = \{v\}$ then this gives a cut of size degree(v)). Hence the total vertex degree is at least nk, so that there are at least $nk/2$ edges. Hence at the first step, the chosen edge e_1 is in C with probability $\frac{k}{\#(\text{edges})} \leq \frac{2}{n}$, so that it is not in C with probability at least $\frac{n-2}{n}$. Moreover, if e_1 is not in C, then it is easy to prove by contradiction that C is still a mincut of M/e_1. Let $i \in [1..n-2]$. Conditioned on the fact that none of $e_1, \ldots, e_{i-1}$ is in C, the probability that e_i is not in C is at least $\frac{n-i-1}{n-i+1}$ (indeed, if we call M_i the graph obtained from M after contracting $e_1, \ldots, e_{i-1}$, it has $n-i+1$ vertices, C is a mincut of M_i, and we can apply to M_i the same argument we have applied to M). Hence the probability that none of $e_1, \ldots, e_{n-2}$ is in C is at least

$$\frac{n-2}{n} \cdot \frac{n-3}{n-1} \cdots \frac{2}{4} \cdot \frac{1}{3} = \frac{2}{n(n-1)}.$$