MCMinCut - Lorenzo Foschi - 4989646

```cpp
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <random>
#include <chrono>
#include <cmath>

// Grafo non pesato
typedef std::vector<std::vector<int>> graph;

// Distribuzione uniforme (meglio del rand)
unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
std::default_random_engine generator(seed);

// Rimozione cappi del grafo
void delCappi(graph& g) {
	for(int i = 0; i < g.size(); ++i)
		g[i][i] = 0;
}

int mincut(const graph& aux) {
	graph g = aux;
	std::uniform_int_distribution<int> distribution(0,g.size()-1);

	// Si accettano input solo senza cappi (problematici)
	delCappi(g);

	for(int i = 0; i < g.size() - 2; ++i) {
		// Scelta randomica dell'arco (esistente) (u, v)
		int u, v;

		do {
			u = distribution(generator);
			v = distribution(generator);
		} while(!g[u][v]);

		for(int j = 0; j < g.size(); ++j) {
			if(g[v][j]) {
				// Gli archi da v ai vari j passano a u
				g[u][j] += g[v][j];
				g[j][u] += g[j][v];
				// Contemporanea liminazione dei precedenti archi da v ai vari j
				// + eliminazione cappi
				g[j][v] = g[v][j] = g[u][u] = 0;
			}
		}
	}
	// Calcolo del mincut candidato
	int c = 0;

	for(int i = 0; i < g.size(); ++i)
		for(int j = 0; j < g.size(); ++j)
			c += g[i][j];
	return c/2;
}
```

```cpp
int main() {
	// Dimensione del grafo, init. e creazione
	int dim = 9;

	graph g(dim, std::vector<int>());
	for(auto& v : g) v.assign(dim, 0);

	g[0][1] = 1;
	g[1][0] = 1;
	g[0][4] = 1;
	g[4][0] = 1;
	g[0][5] = 1;
	g[5][0] = 1;
	g[0][8] = 1;
	g[8][0] = 1;
	g[0][2] = 1;
	g[2][0] = 1;
	g[1][2] = 1;
	g[2][1] = 1;
	g[1][3] = 1;
	g[3][1] = 1;
	g[1][7] = 1;
	g[7][1] = 1;
	g[1][8] = 1;
	g[8][1] = 1;
	g[2][3] = 1;
	g[3][2] = 1;
	g[2][4] = 1;
	g[4][2] = 1;
	g[3][4] = 1;
	g[4][3] = 1;
	g[3][6] = 1;
	g[6][3] = 1;
	g[3][7] = 1;
	g[7][3] = 1;
	g[4][5] = 1;
	g[5][4] = 1;
	g[4][6] = 1;
	g[6][4] = 1;
	g[5][6] = 1;
	g[6][5] = 1;
	g[5][8] = 1;
	g[8][5] = 1;
	g[6][7] = 1;
	g[7][6] = 1;
	g[6][8] = 1;
	g[8][6] = 1;
	g[7][8] = 1;
	g[8][7] = 1;

	// Tentativi e calcolo frequenza
	int n = pow(10, 5);
	int count = 0;
	for(int i = 0; i < n; ++i)
		if(mincut(g) == 4) count++;
```

```cpp
	// Casi favorevoli / possibili
	double p = (double)count / n;

	std::cout << "Freq: " << count << '\n';
	std::cout << "Prob: " << p << '\n';

	// Numero di run utilizzando la relazione nelle slide
	// p di avere almeno 1 volta il minimo --> complemento di avere altro n volte
	// (1-p)^R = (1-0.999) --> R ~= -6.9 / log(1-p)
	// In questo caso con ln(1-0.999) = ln(10^-3) = log(0.001) = -6.9
	double R = log(0.001) / log(1 - p);

	std::cout << "R con formula: " << ceil(R) << '\n';

	// Numero di run calcolando iterativamente la distrib. geom.
	// 0.001 = (1-99%)
	int R2 = 1;
	double comp = 1-p;
	while(comp > 0.001) {
		comp *= (1-p);
		++R2;
	}
	std::cout << "R iterativo: " << R2 << '\n';
}
```

OUTPUT di esempio

```
Freq: 15932
Prob: 0.15932
R con formula: 40
R iterativo: 40
```

Il taglio minimo calcolato per il grafo di Fritsch risulta essere pari a 4.
In questa prova si registrano 15932 occorrenze di 4 come risultato dell'algoritmo di Monte Carlo applicato 10^5 volte, ottenendo una probabilità "p" di 0.15932 (#casi favorevoli / #casi possibili = 10^5).
Si utilizza il suddetto valore "p" per calcolare il numero di run R necessarie per ottenere il taglio minimo con una probabilita del 99.9%.
Cercando la probabilità di trovare almeno una volta il taglio minimo in n tentativi, si considera l'equivalente probabilità di trovare sempre valori diversi dal taglio minimo per gli n tentativi. Si utilizza ora tale probabilità, pari al complemento a 1 del 99%, per calcolare il limite inferiore R:

`(1-p)^R = (1-0.999)` → `applicando log in base 1-p + cambiamento di base` →
→ **`R = ln(0.001)/ ln(1-p)= 40`** `con ln(1-0.999) = ln(10^-3) = log(0.001) ~= -6.9`

Sono dunque necessarie circa 40 iterazioni per essere sicuri al 99.9% di aver trovato il taglio minimo, e questo valore trova conferma nel calcolo iterativo della serie geometrica limitata a `(1-0.999)`.