**Protocollo Montecarlo - Lorenzo Foschi - 4989646**

#include <fstream>

#include <random> // Per il random engine

#include <vector>

#include <chrono> // Per il seed del random\_gen

#include <unordered\_map> // Per l’output (k, v)

typedef std::vector<bool> bits;

typedef std::vector<bits> matrix;

typedef std::unordered\_map<int, int> mappa;

// Distribuzione uniforme (meglio del rand)

unsigned seed = std::chrono::system\_clock::now().time\_since\_epoch().count();

std::default\_random\_engine **generator**(seed);

// Ritorna il valore con più occorrenze

int **maj**(const bits& v) {

int c1 = 0;

int c0;

// Per ogni bit dell’array bits, conto gli 1

for (auto bit : v) c1 += bit;

// Il complemento sono le occorrenze di 0

c0 = v.size() - c1;

return (c1 > c0);

}

// Ritorna il numero di occorrenze di maj

int **tally**(const bits& v) {

int c1 = 0;

int c0;

for (auto bit : v) c1 += bit;

c0 = v.size() - c1;

return (c1 > c0) ? c1 : c0;

}

/\*

Nota sullo standard utilizzato per la matrice:

4 righe i (la 4a è il malizioso quindi non memorizzata);

ogni colonna j è il bit ricevuto al proc. i dal proc. j

[1, 1, 0, 0],

[1, 1, 0, 0],

[1, 1, 0, 1],

[...]]

\*/

int **main**() {

int r = pow(2, 10); // Round

int t = 1; // Maliziosi

int p = 2 \* t + 1; // Affidabili

mappa rounds; // Array di rounds

bits **b**(4); // b (bit in trasmissione) di dimensione 4

// Distribuzione uniforme valori

std::uniform\_int\_distribution<int> distribution(0, 1);

for (int i = 0; i < r; ++i) {

// Creazione matrice con split 2 a 1 (e diverso per i maliziosi)

matrix m = { {1, 1, 0, 0}, {1, 1, 0, 0}, {1, 1, 0, 1} };

int cround = 0; // Contatore turni

while (true) {

bool coin = distribution(generator);

cround++;

for (int j = 0; j < p; ++j) {

// Calcolo di maj e tally

int mag = maj(m[j]);

int tl = tally(m[j]);

// Se ho raggiunto il consenso (sopra tally)…

// …metto maj, sennò "toss a coin"

b[j] = (tl >= p ? mag : coin);

}

// Inserimento dei valori in ricezione

for (auto& v : m) v = b;

// Impostazione valori del malizioso (per ogni riga,

// …set dell’ultima colonna : 1 - sent)

for (int j = 0; j < p; ++j) m[j].back() = !b[j];

// Controllo terminazione

if (m[0][0] == m[0][1] && m[0][1] == m[0][2]) break;

}

rounds[cround]++;

}

// stampo valori su file per plottare in py

std::ofstream out;

out.open("output.txt");

for (const auto& [n, fqz] : rounds)

out << n << " " << fqz << '\n';

out.close();

}

**# PARTE IN PY per il PLOTTING dei round per ogni run**

from matplotlib import pyplot as plt

data = {}

# importo i risultati ottenuti dalla mappa in cpp

with open("output.txt") as fd:

for l in fd:

# coppie chiave-valore con numero\_round-frequenza

(k, v) = l.split()

data[int(k)] = int(v)

# creazione grafico

run = 2 \*\* 10

rounds = max(data.keys()) + 1

# dati teorici per il confronto (accordo raggiunto in r round ~= R/(2^r))

x = list(range(1, rounds))

y = [run / 2 \*\* r for r in range(1, rounds)]

plt.title("Protocollo MonteCarlo: ${2^{​​​​​​10}​​​​​​}​​​​​​$ run")

plt.plot(list(data.keys()), list(data.values()), "b", label="empirica")

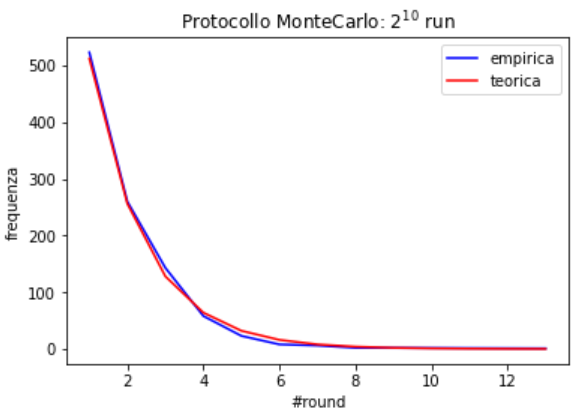
plt.plot(x, y, "r", label="teorica")

plt.legend()

plt.xlabel("#round")

plt.ylabel("frequenza")

plt.show()



*Per l’implementazione si sceglie di scrivere un codice in cpp per il protocollo Montecarlo e un modulo Python che, presi in input le coppie (k, v) = (round, frequenza di convergenze per il round k), produca un grafico per il confronto. Per i processi si utilizza una matrice, la cui convenzione è riportata nei commenti del codice cpp in prima pagina, e per memorizzare l’output si utilizza una unordered map in cui si registrano le occorrenze. Nel main del programma si segue lo pseudocodice proposto, utilizzando le due funzioni ausiliarie per il calcolo dei valori di maj e tally.*

Si sono messi in confronto il grafico dei risultati teorici, cioè ottenuti tramite la formula R/2r, e il grafico dei risultati empirici misurati tramite l’algoritmo in cpp. Si osserva una forte correlazione tra i due (ad esempio in questa prova si sono calcolati 523 casi in cui l’accordo si è raggiunto al primo round; risultato che combacia con il valore 512 che si ottiene dalla formula teorica). Tale osservazione trova conferma nel valore atteso della convergenza dell'algoritmo, che per conseguenza del lancio della moneta nel caso di disaccordo risulta essere ½, cioè la probabilità di convergenza dei processi affidabili sul valore di maggioranza. Al round successivo si otterrà dunque una probabilità di ¼, e così via seguendo la serie geometrica.