Trioli Davide – S5316731 a.a. 2022/2023

Analisi e Progettazione di Algoritmi

Esercizio 7.1
Accordo bizantino: Caso minimale di MCByzantineGeneral

Parte 1. *Implementazione e codice*

L'implementazione dell'algoritmo proposto è stata sviluppata interamente in linguaggio *Python*.
Nelle pagine seguenti il codice prodotto, suddiviso per funzionalità implementate.

Sotto: le librerie importate `random` *per l'implementazione della "moneta globale" e* `matplotlib` *per la generazione del grafico relativo alle quantità di accordi trovati in un certo numero di turni. Inizializzate anche diverse variabili necessarie, come quelle per il salvataggio dei messaggi inviati e ricevuti dai varî processi (nell'esempio,* 4 *di cui uno* faulty*) e la soglia di accordo* T come $2f + 1$.

```
import random
import matplotlib.pyplot as plt

############################################################

# global vars
init_vals=[[None,None,None,None], [None,None,None,None],
[None,None,None,None], [None,None,None,None]]
# every xi in init_vals[x1, x2, ..., xn] is like personal memory for process i
# while in reality they can "send" their values by editing the actual value
# in some other process personal memory directly

p_num = len(init_vals) #number of processes ( = 4 in this example)
T=2*1+1 # threshold for maj calculation
global_coin=0 # init of global coin, randomized later
count_rounds=[] # array to keep track of how many rounds needed each run to
find agreement
next_agree=0 # counter used for checking when tally >= T if global_coin ==
maj, next round agreement is reached

############################################################
```

```
def setSentOkValues(init, order):
    #init_vals[order][order]=init
    for i in range(0, p_num):
            #if (i!=order):
            init_vals[i][order]=init
            #e.g. if p1 sends 1 to {p0, p2, p3}, then init_vals[everyone][1]=1


def MCByzantineGeneral(order):

    tally0=init_vals[order].count(0)
    tally1=init_vals[order].count(1)
    maj = 0 if tally0 > tally1 else 1
    tally = tally0 if tally0 > tally1 else tally1

    if(tally >= T):
        global next_agree
        if(global_coin == maj):
            next_agree+=1
        return maj
    else: return global_coin
    #returned value is what to set next iteration


def setSentFaultyValues(order):
    for i in range(0, p_num):
        if (i!=order):
            init_vals[i][order] = 1-init_vals[order][i]
            # set every other bits maliciously to 1 - what they sent

############################################################

tot_runs = 2**10 # tot number of tries to find agreements

print("# # # # # # # # # START # # # # # # # # #")
print("- starting to find", tot_runs, "agreements...")
```

Sopra: le funzioni utilizzate per l'implementazione dell'algoritmo proposto. `setSentOkValues` *è utilizzata per il settaggio dei valori inviati dai processi "onesti" nelle varie zone di memoria predisposte;* `MCByzantineGeneral` *è utilizzata per l'effettiva implementazione dell'algoritmo dell'accordo bizantino randomizzato, a valori inviati già presenti; mentre* `setSentFaultyValues` *è utilizzata come la routine di lavoro per il processo malizioso: imposta i bit inviati a ognuno degli altri processi come l'opposto di ciò che a questo hanno comunicato.*

```
for k in range(0, tot_runs):

    count=0 # counter for num of cycle each run to find agreement
    count_expected=0
    bits_ok=[1,0,0] # starting bits sent by processes pi at first run every
try
    next_agree=0

    while(True):

        global_coin=random.randint(0,1)

        count+=1

        for j in range(0, p_num-1):
            setSentOkValues(bits_ok[j], j)
        # faulty process is always the last one (needs to know what others
sent it)
        setSentFaultyValues(p_num-1)

        for j in range(0, p_num-1):
            bits_ok[j]=MCByzantineGeneral(j)

        if(next_agree > 0):
            count_expected=count
            next_agree=0

        # exit from loop if every bit sent (written inside bits_ok) is the
same for every ok (= not faulty) process (they agree either on 0 or 1)
        if(bits_ok.count(0)==len(bits_ok) or bits_ok.count(1)==len(bits_ok)):
break

    if not (count_expected==count and next_agree == 0):
        # execution arriving here means something wrong happened (should never
display msg below)
        print("- - error! Agreement not reached in round after tally(j) ≥ T
and global coin result equal to maj(j) for a process j")

    count_rounds.append(count)
```

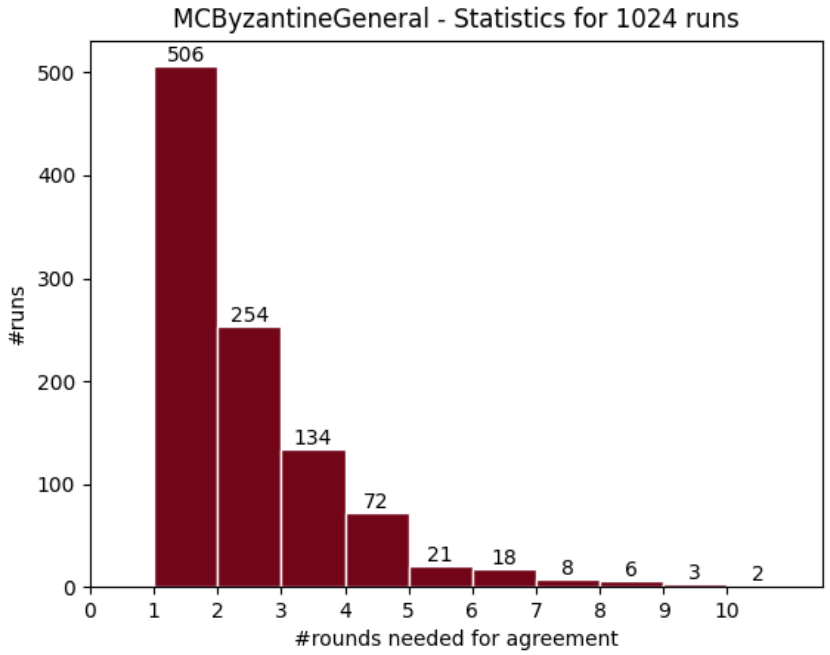
Sopra: l'implementazione di un numero arbitrario di ricerca di accordo tramite algoritmo sopra descritto, in questo esempio 1024*. L'accordo si dice raggiunto quando i bit inviati da tutti i processi affidabili coincidono, sia esso 0 oppure 1. Il controllo relativo al possibile mancato raggiungimento dell'accordo nonostante le condizioni adeguate è discusso più approfonditamente in fondo a pag. 5.*

```
print("- found the agreements, see graph!")
print("# # # # # # # # # # # # # # # # # # # #")

counts,edges,bars=plt.hist(count_rounds, bins=[1,2,3,4,5,6,7,8,9,10,11],
color="#720518", ec="white")
plt.xticks(range(0,11))
plt.bar_label(bars)
plt.xlabel("#rounds needed for agreement")
plt.ylabel("#runs")
plt.title("MCByzantineGeneral - Statistics for " + str(tot_runs) + " runs")
plt.show()
```

Sopra: la generazione dell'istogramma raffigurante il numero di round necessari per trovare l'accordo in base al numero di run effettuate. Sono rappresentate solamente le run che hanno richiesto fino a un massimo di 10 round.

Parte 2. *Risultati e statistiche*



A sinistra: il grafico prodotto dall'esecuzione del codice. Le barre color bordeaux indicano la quantità di iterazioni che hanno richiesto il numero di round indicato nell'asse orizzontale.

Sotto: un breve output di riuscita dell'esecuzione priva di errori.

```
# # # # # # # # START # # # # # # # # #
- starting to find 100000 agreements...
- found the agreements, see graph!
# # # # # # # # # # # # # # # # # # # #
```

Dall'implementazione dell'algoritmo proposto e dall'analisi del grafico prodotto si può osservare sperimentalmente che, chiamando

- r il numero di round necessarî per il raggiungimento dell'accordo;
- R il numero di iterazioni totali;
- n_r la quantità di *run* che hanno richiesto r round per accordarsi, vale:

$$n_r \approx R \cdot \frac{1}{2^r} \text{ con } r = 1,\ 2,\ \ldots,\ 10.$$

In effetti:

- Il lancio della "*global coin*" ha, a ogni turno, $P(0) = P(1) = \frac{1}{2}$.
- Nel caso in cui siano necessari r turni la moneta globale deve essere lanciata r volte, e quindi la probabilità di far concordare tutti i processi affidabili esattamente all'r-esimo round è riconducibile al comportamento di una variabile aleatoria geometrica, avente quindi probabilità di successo uguale a $\left(1 - \frac{1}{2}\right)^{r-1} \cdot \frac{1}{2} = \frac{1}{2^r}$ con $r \geq 1$.
- Applicando questa aspettativa di riuscita a tutte le iterazioni R, ci si aspetta quindi la conclusione di tutti i run in

  $$R \cdot \frac{1}{2^r} \text{ con } r \geq 1$$

  confermando quanto osservato empiricamente.

La conferma dell'esito positivo della ricerca dell'accordo quando

$$(1).\ \ tally(j) \geq T$$

per un qualche processo affidabile j e

$$(2).\ \text{l'esito del lancio della moneta globale coincide con } maj(j)$$

si ha anche sperimentalmente: infatti, ogni volta che si verifica la condizione (1). un processo affidabile aggiorna una variabile globale che tiene conto di quanti processi hanno soddisfatto questa condizione: quando tutti i processi hanno eseguito la loro routine, viene controllato il valore all'interno della suddetta variabile; nel caso in cui sia maggiore di 0, si va a impostare un'aspettativa di successo sul round corrente; successivamente si controlla se per caso questa non venisse rispettata: in questo caso, si va a stampare un messaggio di errore. Dall'output del programma in fondo a pag. 4, si nota come questo non venga mai esplicitato.