Robert Munné Mas Boris Llona Alonso
GM2

Primera pràctica

Algorísmia i Complexitat

Robert Munné Mas 79274677W
Boris Llona Alonso 48053932D

7 d'abril de 2019
2018-2019

ESCOLA POLITÈCNICA SUPERIOR
UNIVERSITAT DE LLEIDA

Robert Munné Mas Boris Llona Alonso
GM2

Plantejament de la pràctica:

Per a realitzar el que se'ns descriu en l'enunciat cal primer posar-se a pensar com abordar la implementació els problemes que ens podem trobar.
Vam analitzar diverses opcions per a detectar si s'havia produït una modificació en el nostre text. Al no tenir el text original no podem fer una funció que compari els dos textos fins trobar si hi ha o no discrepancia. Per tant vam tenir que pensar una manera de afegir una marca que tingui dependència amb tot el text i que si aquest era modificat mitjançant la marca ho puguem saber. Miran diverses opcions ens vam trobar amb una de facil implementació. L'algorisme **check sum** funciona sumant els bits i aplicant el complement a 1 per a generar una clau. Un cop enviat el missatge es sumen tots els bits inclosa la clau i s'aplica el complement a 1 de nou. Si el resultat és 0, no hi ha hagut modificació. Tot i la senzillesa d'aquest algorisme la taxa de fallada es bastant gran per tant vam decidir seguir mirant opcions per augmentar la fiabilitat.

CRC és l'algorisme que s'empra principalment en la detecció d'errors en telecomunicacions, aquest és més complex pero la seva fiabilitat augmenta exponencialment. El seu funcionament es basa en, a partir d'una seqüència en binari i una clau realitzar la divisió binària per tal d'obtenir el seu residu.
Un cop s'aplica el soroll la secuencia binaria del nostre missatge canvia (Nosaltres passem a binari el caràcter ASCI que llegim, sempre que no sigui ja un 0 o 1). Procedim a dividir aquesta nova seqüència adjuntant el residu obtingut anteriorment al final entre la nostra clau. Si la sequencia és idèntica a la del primer cas (és a dir no s'ha realitzat modificació) el residu de d'aquesta nova divisió serà igual a 0.
Aquesta divisió binària esmentada prèviament actua com una porta XOR, és a dir:

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	0

XOR

Altres aspectes a destacar de la nostra implementació abans de analitzar els algorismes son la lectura dels fixers. Es passen per paràmetre, per tant cal controlar els arguments que passem a l'hora de cridar a l'executable. Si no hi ha els suficients esperarà a que introduïm manualment l'adreça de l'arxiu per l'entrada estàndard.

Cal destacar també que llegirem bite per bite per tal de poder fer les transformacions adients per el correcte funcionament de crc.

Detecció de posició del error i correcció:

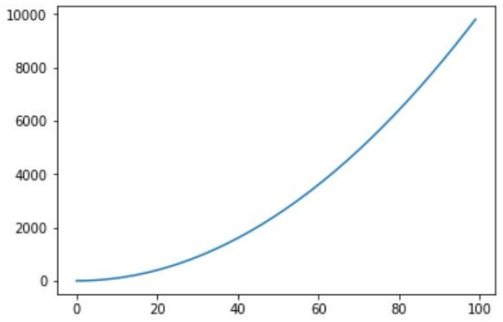
Realitzar la correcció i trobar la posició del error ja es una tasca una mica més complexa si no tenim el arxiu original per a comparar. Existeixen diversos algorismes com el de **Hamming** que comprova les posicions múltiples de 2^n basant se en la paritat que molt resumidament controla si el numero de multiples en aquella "n" es parell o imparell. Cada bit al dependre de multiples "n" si canvia la paritat en algun camp podem trobar exactament on i quin és l'error. Tot i la eficiencia d'aquest mètode es complex d'implementar i te diverses versions per si volem controlar un màxim de 1 o més canvis.

Un altre mètode de correcció i detecció d'error molt atractiu tant per la seva senzillesa com per la seva fiabilitat es crear una **matriu** i sumar els valors de cada columna i cada fila, comparant aquests dos camps en una posició matr[i][j] concreta sabrem si ha canviat respecte els altres valors o no, així doncs podrem detectar exactament la posició i arreglar-ho(recordant que la suma de aquella columna/fila - els altres valors = número Correcte).
El pseudocodi de la implementació de la matriu, càlcul de la suma de totes les seves components (x,y) i escritura a l'arxiu es el següent:

```
SumX, SumY = createMatrix()
writteSums(SumX,SumY)
createMatrix():
        obrir arxiu
        llegir següent lletra
        mentres hi hagi lletres
                digits.afegir(lletra)
                llegir següent lletra
```

```
x = arrelquadrada(longitud(digits))
y = arrelquadrada(longitud(digits))

matriu = [][]
mentres  i <0 :
        mentres j<0:
                si hi ha digits:
                        matriu[i][j] = digits[0]
                        digits = digits[1:]
                        j++
                i++
```

Per a crear dues llistes que continguin els valors de la suma de cada component vertical i horitzontal simplement caldria recorrer la matriu, suposant el mateix cost, i sumar els valors a la posició corresponent tal i com es troba implementat a l'arxiu python.



Podem apreciar que trobem dos bucles, un dins d'un altre, per tant tenim el cost O(N^2) que se'ns demana a l'enunciat.

Disseny recursiu i transformacions:

Per a realizar el informe de la pràctica i el posterior anàlisis dels costos cal diferenciar els algorismes segons la seva implementació: Recursiva, Recursiva final, Iterativa. Per tant a continuació descriurem el pseudo codi del algorisme de detecció implementat (CRC).
Per al anàlisi dels costos de cada algorisme cal tenir en compte diversos aspectes: Primer que per a tenir una visió real de la mitja dels costos cal realitzar diverses medicions, si calculem per a molt poques iteracions podrem tenir costos bastant diferenciats i que no ens donaran una visió general de l'algorisme.

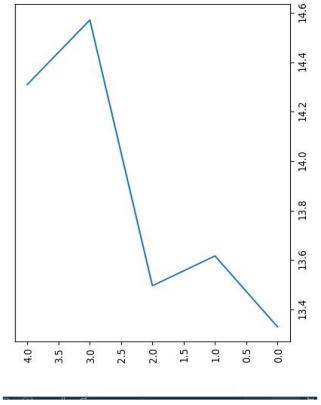


figura 1

figura1: Medició de costos per a 5 paraules diferents (generades amb plumbus) a la versió iterativa.

Cal tenir en compte també que per a comprovar el cost cal diverses paraules a transmetre generades per plumbus.py, per tant cada iteració del nostre analisis de costos afegirà el cost d'executar plumbus.py.

Per últim i per veure la relació del cost en paraules petites o més grans hem modificat el plumbus.py (sols per l'analisis) per tal que cada vegada que el cridem generi una paraula de llargada 1 fins a 100.

Per analitzar els costos de els algorismes implementats hem creat una funció que analitza els temps amb la llibreria timeit i els representa en una gràfica amb matplotlib.

costos():

```
cost = 0
costIT = []
IT = []
for i in range(0, 100):
   system('python plumbus.py '+str(i))
   inputoutputfiles()
   cost += timeit.timeit("iterative()", setup="from __main__ import
iterative")
   costIT.append(cost)
   IT.append(i)
```

```
plt.plot(IT,costIT)
plt.show()
```

Funció Recursiva:

funcio xor_recorsiu(a, b) :
len(a) == 0 or len(b) == 0 → return ‘ ’
si a[0] == b[0] → result = ‘0’
si no: result = ‘1’
return result + xor_recorsiu(a[1:], b[1:])

funcio divisio_recursiva(divident, divisor, llarg=””, temp=””)
si no llarg → llarg = len(divisor)
si no temp → temp = divident[0: llarg]
si llarg < len(divident):
si temp[0] == ‘1’ → temp = xor_recorsiu(divisor, temp) + divident [llarg]
si no → temp = xor_recorsiu(‘0’*llarg, temp) + divident [llarg]
llarg++
divisio_recursiva(divident, divisor, llarg, temp)
si no:
si temp[0] == ‘1’ → return xor_recorsiu(divisor, temp)
si no → return xor_recorsiu(‘0’*llarg, temp)
return divisio_recursiva(divident, divisor, llarg, temp)
funcio ajustarDades_recorsiu(data, clau):
l_clau = len(clau)
afegir0s = data+’0’*(len(clau)-1)
residu = divisio_recursiva(afegir0s,clau)
return residu

funcio plumbusToBinary_recursiu():
binary = ‘ ’
byte = llegir següent lletra
si byte is null → return binary
si no → digits.afegir(lletra)
si byte == ‘0’ or byte == ‘1’ → binary = str(byte)
si no → binary = str(bin(ord(byte))[2:])
return binary + plumbusToBinary_recursiu()

funcio writeSums_recursiu(X,Y):
si no X → return 0
digits.afegir(str(X[0])
si no Y → return 0
digits.afegir(str(Y[0])

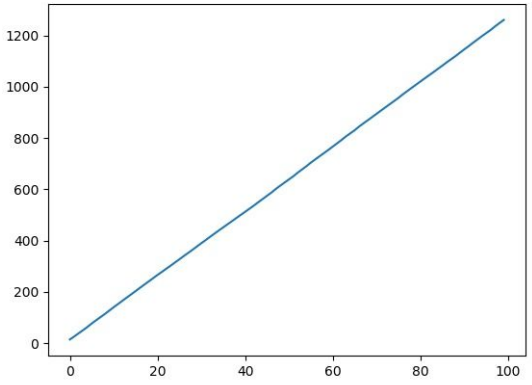
```
    return writeSums_recursiu(X[1:],Y[1:])
funció  recursive():
    data = plumbusToBinary_recursiu()
    dividentIResidu = ajustarDades_recursiu(data, clau)
    escriureAarxiu(dividentIResidu)
```

Funció Iterativa:

```
funció  iterative():
    data = plumbusToBinary()
    dividentIResidu = ajustarDades(data, clau)
    escriureAarxiu(dividentIResidu)

funció  plumbusToBinary():
    obrir arxiu
    llegir següent lletra
    mentres hi hagi lletres
        digits.afegir(lletra)

        binary = binary + bin(ord(lletra))
        llegir següent lletra
    retornar binary #Text de l'arxiu original en binary

 funció ajustarDades(data, clau):
    afegir0s = data+'0'*(len(clau)-1)
    residu = divisio(afegir0s,clau)
    retorna residu

funció divisio(divident,divisor):
    temporal = divident[0:len(divisor)]
    mentres len(divisor)<len(divident)
        si temporal[0] == '1'
            tmp = xor(divisor, tmp) + divident[llarg]
        si temporal[0] =='0' #Vol dir que el bit de mes a l'esquerra es un 0
            tmp = xor('0' * llarg, tmp) + divident[llarg]
        llarg++

      si tmp[0] == '1':
          tmp = xor(divisor, tmp)
      sinó:
```

```
        tmp = xor('0' * llarg, tmp)
        residu = tmp
        retorna residu

funció xor(a,b):  #Actua com una porta xor
        resultat = []
        mentre i<len(b)
                si a[i]==b[i]:
                        resultat.afegir('0')
                sinó:
                        resultat.afegir('1')
        retorna ' '.juntar(resultat)
```

El cost del algorisme de detecció iteratiu en notació big oh és: O(N), ho podem comprovar tan llegint l'algorisme com representant els costos de cada iteració en una gràfica fent ús de les llibreries matplotlib i timeit.



A la gràfica podem apreciar el cost linear O(n) on l'eix de les y es el cost i l'eix de les x el nombre de paraules que genera el plumbus.

Per últim comentar que el cost teòric és O(n) on n es la llargada de paraules a analitzar, on el millor cas es que hi hagi 1 paraula.