Práctica 2. "Algoritmos probabilisticos y aleatorios"

Edgar Munguia and Izaird Mothelet

[1] ESCOM, Instituto Politécnico Nacional, Ciudad de México 07738, México
[2] CINVESTAV-IPN (Evolutionary Computation Group) Computer Science Department, Ciudad de México 07360, México

Abstract. multi-objective evolutionary algorithms (MOEAs)

Key words: algoritmos aleatorios, método de monte Carlo, algoritmos de las Vegas

1 Introducción

Un algoritmo se dice aleatorizado si usa algún grado de aleatoriedad como parte de su lógica.

En este tipo de algoritmos el tiempo de ejecución o su salida se convierten en variables aleatorias.

Con base en el tiempo de ejecución y la salida de los algoritmos aleatorizados como variables aleatorias, podemos definir dos tipos de algoritmos:

- *Las Vegas:* algoritmos que siempre entregan el resultado correcto, pero cuyo tiempo de ejecución varía (incluso para la misma entrada de datos). Un algoritmo Las Vegas se dice eficiente si su tiempo esperado de ejecución es polinomial para cualquier entrada.
- *Monte Carlo:* algoritmos que pueden entregar resultados incorrectos con una probabilidad acotada. Podemos disminuir esta probabilidad de error repitiendo el algoritmo a expensas del tiempo de ejecución. Un algoritmo Monte Carlo se dice eficiente si su tiempo de ejecución en el peor caso es polinomial para cualquier entrada.

1.1 Método de Monte Carlo aplicado a búsquedas

Los Métodos de Monte Carlo se basan en la analogía entre probabilidad y volumen. Las matemáticas de las medidas formalizan la noción intuitiva de probabilidad, asociando un evento con un conjunto de resultados y definiendo que la probabilidad del evento será el volumen o medida relativa del universo de posibles resultados. Monte-Carlo usa esta identidad a la inversa, calculando el volumen de un conjunto interpretando el volumen como una probabilidad. En el caso más simple, esto significa muestrear aleatoriamente un universo de resultados posibles y tomar la fracción de muestras aleatorias que caen en un conjunto

dado como una estimación del volumen del conjunto. La ley de grandes números asegura que esta estimación converja al valor correcto a medida que aumenta el número de muestras. El teorema del límite central proporciona información sobre la magnitud del probable error en la estimación después de un número finito de muestras.

En un proceso estándar de Monte Carlo tenemos los siguientes pasos:

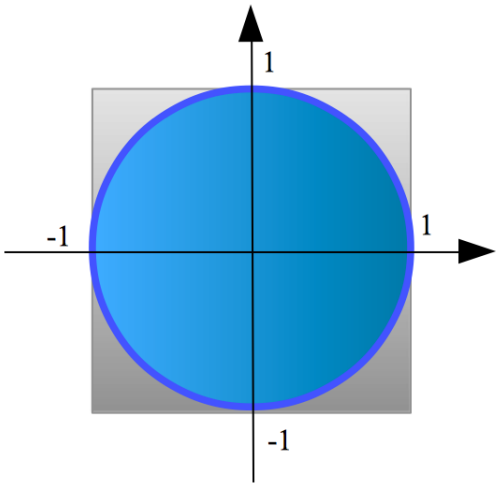
- Se generan un gran número de simulaciones aleatorias desde la posición del tablero para la que se desea encontrar el mejor movimiento siguiente.
- Se almacenan las estadísticas para cada movimiento posible a partir de este estado inicial.
- Se devuelve el movimiento con los mejores resultados generales.

Un ejemplo común de la aplicación de método Monte Carlo es la aproximación del valor de π, para ello tenemos que considerar un círculo unitario (radio=1) dentro de un cuadrado con los lados iguales a 2 (véase la figura 1). Sí escogemos un punto al azar (x, y) donde x y y están definidos en el rango $[-1, 1]$, la probabilidad de que este punto al azar se encuentre dentro del círculo unitario se da como la proporción entre el área el círculo unitario y el cuadrado:

$$P(x^2, y^2 \leq 1) = \frac{A_{circle}}{A_{square}} = \frac{\pi}{4} \tag{1}$$

Si escogemos puntos al azar N veces y M de esas veces el punto cae dentro del círculo unitario, la probabilidad de que un punto al azar caiga dentro de dicho círculo esta dado por:

$$P'(x^2, y^2 \leq 1) = \frac{M}{N} \tag{2}$$



Por lo tanto consideramos la siguiente ecuación para el calculo de π:

$$H(x,y) = \begin{cases} 1 \text{ si } x^2 + y^2 \leq 1 \\ 0 \text{ in other case} \end{cases} \tag{3}$$

1.2 Desarrollo (sección 1. Monte Carlo)

– Pruebe el código en python proporcionado y ejecute pruebas para $N = 100, 1000, 10000, 100000, 1000000$, muestre cuáles son los valores que obtiene para π en cada caso.

N	Valores	Error
100	3.28	4.22%
1000	3.152	0.33%
10000	3.1348	0.22%
100000	3.1414	0.01%
1000000	3.1433	0.05%

Fig. 1. se muestran los resultados de la experimentacin

– Implemente el método de Monte Carlo para resolver integrales. Utilice la siguiente formula:

$$\frac{b-a}{n} \sum_{i=1}^{n} f(u_i(b-a)+a) \tag{4}$$

donde a y b son los límites inferior y superior de una integral, u_i es un valor aleatorio generado entre $[0, 1]$ y f se refiere a la función a integrar.
Considere las siguientes integrales para probar su implementación:

$$f_1(x) = \int_0^1 (1-x^2)^{3/2} \cdot dx \tag{5}$$

$$f_2(x) = \int_{-1}^1 e^{x+x^2} \cdot dx \tag{6}$$

$$f_3(x) = \int_0^2 (1+x^2)^2 \cdot dx \tag{7}$$

$$f_4(x) = \int_0^{2\pi} \frac{1}{cos(x)+2} \cdot dx \tag{8}$$

$$f_5(x) = \int_0^2 log(x) \cdot dx \tag{9}$$

Deberá desarrollar una simulación donde se muestre paso a paso la generación de números aleatorios y su graficación en la función correspondiente. (Ver ejemplo en figura 1.2 y consulte la página web https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration).

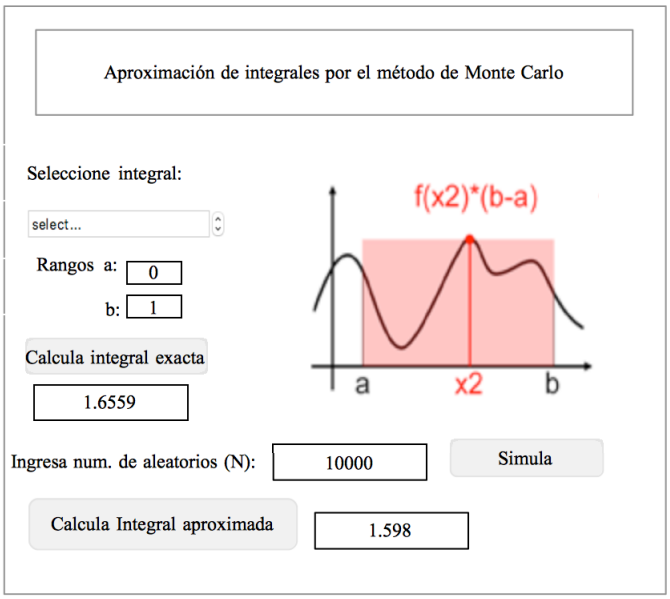


Fig. 2. Ejemplo de interfaz gráfica para simular aproximación a integrales

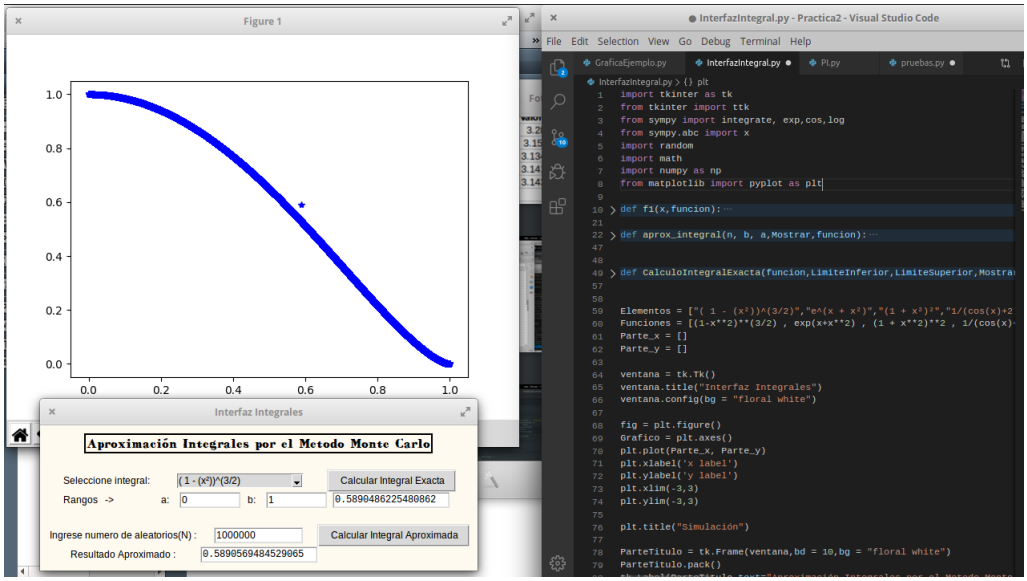


Fig. 3. Interfaz gráfica para simular aproximación a integrales que se realizó

```
import tkinter as tk
from tkinter import ttk
from sympy import integrate, exp,cos,log
from sympy.abc import x
import random
import math
import numpy as np
from matplotlib import pyplot as plt

def f1(x,funcion):
    if funcion == 0:
        return ( 1 - x**2 )**( 3/2 )
    elif funcion == 1:
        return math.exp( x + x**2 )
    elif funcion == 2:
        return (1 + x**2 )**2
    elif funcion == 3:
        return 1/(math.cos(x) + 2)
    elif funcion == 4:
        return math.log(x)

def aprox_integral(n, b, a,Mostrar,funcion):
```

```
    Parte_x = []
    Parte_y = []
    sum  = 0
    for i in range(0, n):
        ui = random.uniform(0, 1)
        operacion = ui*(b-a) + a
        resultadoFuncion = f1(operacion,funcion)
        sum  = sum + resultadoFuncion
        Parte_x.append(operacion)
        Parte_y.append(resultadoFuncion)

    resultado = (b-a)/n * sum

    Mostrar.config(state="normal")
    Mostrar.delete(1.0, tk.END)
    Mostrar.insert(tk.END,resultado)
    Mostrar.config(state="disabled")

    Parte_x.append(resultado)
    Parte_y.append(resultado)

    plt.cla()
    plt.plot(Parte_x,Parte_y,"b*")
    plt.show()


def CalculoIntegralExacta(funcion,LimiteInferior,LimiteSuperior,Mostrar):

    resultado = float(integrate(Funciones[funcion],
     (x,float(LimiteInferior),float(LimiteSuperior))))

    Mostrar.config(state="normal")
    Mostrar.delete(1.0, tk.END)
    Mostrar.insert(tk.END,resultado)
    Mostrar.config(state="disabled")


Elementos = ["( 1 - ( x ))^(3/2)","e^(x + x )","(1 + x )  ",
"1/(cos(x)+2)","log(x)"]
Funciones = [(1-x**2)**(3/2) , exp(x+x**2) , (1 + x**2)**2 ,
 1/(cos(x)+2) ,log(x)]
Parte_x = []
Parte_y = []

ventana = tk.Tk()
```

```
ventana.title("Interfaz Integrales")
ventana.config(bg = "floral white")

fig = plt.figure()
Grafico = plt.axes()
plt.plot(Parte_x, Parte_y)
plt.xlabel('x label')
plt.ylabel('y label')
plt.xlim(-3,3)
plt.ylim(-3,3)

plt.title("Simulaci n")

ParteTitulo = tk.Frame(ventana,bd = 10,bg = "floral white")
ParteTitulo.pack()
tk.Label(ParteTitulo,text="Aproximaci n Integrales por
el Metodo Monte Carlo", borderwidth=2, relief="solid",
font=("HVD Bodedo",10),bg = "floral white").pack()

ParteEntradas = tk.Frame(ventana, bd = 10,bg = "floral white")
ParteEntradas.pack()

tk.Label(ParteEntradas,text="Seleccione integral: ",
bg = "floral white").grid(row = 0,column = 0)
Opciones = ttk.Combobox(ParteEntradas,state = "readonly",
values = Elementos , width = 20)
Opciones.grid(row = 0,column = 1,columnspan= 4 )

tk.Label(ParteEntradas,text="Rangos -> ",bg = "floral white").
grid(row = 1,column = 0,sticky = 'W')
LimiteInferior = tk.Entry(ParteEntradas,width = 10)

tk.Label(ParteEntradas,text="a: ",bg = "floral white",width = 4).
grid(row = 1,column = 1)
LimiteInferior.
grid(row = 1,column = 2)
tk.Label(ParteEntradas,text="b: ",bg = "floral white",width = 4).
grid(row = 1,column = 3)
LimiteSuperior = tk.Entry(ParteEntradas,width = 10)
LimiteSuperior.grid(row = 1,column = 4)

tk.Button(ParteEntradas, text=" Calcular Integral Exacta", height = 1,
 activebackground = "blue", activeforeground = "White",
 command = lambda: CalculoIntegralExacta(Opciones.current(),
 LimiteInferior.get(),LimiteSuperior.get(),
```

```
TextoIntegralExacta)).grid(row = 0,column = 5)
TextoIntegralExacta = tk.Text(ParteEntradas,width = 20,height = 1)
TextoIntegralExacta.insert(tk.END,"Solucion ")
TextoIntegralExacta.config(state = "disabled")
TextoIntegralExacta.grid(row = 1,column = 5)

ParteSimulacion = tk.Frame(ventana,bd = 10,bg = "floral white")
ParteSimulacion.pack()
tk.Label(ParteSimulacion,text="Ingrese numero de aleatorios(N) : ",
bg = "floral white").grid(row = 0,column = 0)
N = tk.Entry(ParteSimulacion,width = 15)
N.grid(row = 0,column = 1)
tk.Button(ParteSimulacion, text=" Calcular Integral Aproximada",
height = 1, activebackground = "blue", activeforeground = "White",
command = lambda: aprox_integral(int(N.get()),float(LimiteSuperior.
get()),float(LimiteInferior.get()),
TextoIntegralAproximado,Opciones.current())).grid(row = 0,column = 2)

tk.Label(ParteSimulacion,text="Resultado Aproximado : ",
bg = "floral white").grid(row = 1,column = 0)
TextoIntegralAproximado = tk.Text(ParteSimulacion,width = 20,height = 1)
TextoIntegralAproximado.insert(tk.END,"Solucion ")
TextoIntegralAproximado.config(state="disabled")
TextoIntegralAproximado.grid(row=1,column=1)

ventana.mainloop()
```

1.3 Desarrollo (sección 2. Las Vegas)

Para poner en práctica el algoritmo las vegas se implementaran dos ejercicios:

1. Algoritmo de Quick-sort. Modifique el algoritmo de Quick sort implementado en la práctica 1. Por cada llamada recursiva seleccione al azar el pivote, compare si hay mejora respecto a su versión anterior para el caso promedio y reporte el tiempo de ejecución para los siguientes casos $n = 1000, 2000, 3000, \ldots, 10,000$ (n es el número de elementos en un arreglo).
2. El prolema de las 8 reinas. Se requiere generar una solución correcta al problema de las 8 reinas de acuerdo a las siguientes reglas.

CODIGO QUICK–SORT

```
import random
import timeit
```

```
def quicksort(arr, start, stop):
    if(start < stop):

        # pivotindex is the index where
        # the pivot lies in the array
        pivotindex = partitionrand(arr, start, stop)

        # At this stage the array is partially sorted
        # around the pivot. separately sorting the
        # left half of the array and the right half of the array.
        quicksort(arr , start , pivotindex)
        quicksort(arr, pivotindex + 1, stop)

# This function generates random pivot, swaps the first
# element with the pivot and calls the partition function.
def partitionrand(arr , start, stop):
    randpivot = random.randrange(start, stop)
    arr[start], arr[randpivot] = arr[randpivot], arr[start]
    return partition(arr, start, stop)


def partition(arr,start,stop):
    pivot = start # pivot
    i = start - 1
    j = stop + 1
    while True:
        while True:
            i = i + 1
            if arr[i] >= arr[pivot]:
                break
        while True:
            j = j - 1
            if arr[j] <= arr[pivot]:
                break
        if i >= j:
            return j
        arr[i] , arr[j] = arr[j] , arr[i]

best_case = []
worst_case = []
rand_case = []
for i in range(10):
    f = open('best_case/b_' + str(i+1) + "000", 'r')
    x = f.read().split()
    best_case.append(x)
```

```
    f.close()

for i in range(10):
    f = open('rand_case/rand_' + str(i+1) + "000", 'r')
    x = f.read().split()
    rand_case.append(x)
    f.close()

for i in range(10):
    f = open('worst_case/w_' + str(i+1) + "000", 'r')
    x = f.read().split()
    worst_case.append(x)
    f.close()


f= open("data","w+")
for i in range(10):
    start= 0
    end = 0
    start = timeit.timeit()
    quicksort(best_case[i],0, len(best_case[i])-1 )
    end = timeit.timeit()
    f.write(str(end) + ", ")
    print(end - start)

f.write("\n")

for i in range(10):
    start = timeit.timeit()
    quicksort(worst_case[i],0, len(worst_case[i])-1 )
    end = timeit.timeit()
    f.write(str(end) + ", ")
    print(end - start)
f.write("\n")

for i in range(10):
    start = timeit.timeit()
    quicksort(rand_case[i],0, len(rand_case[i])-1 )
    end = timeit.timeit()
    f.write(str(end) + ", ")
    print(end - start)


# array = [10, 7, 8, 9, 1, 5]
# start = timeit.timeit()
```

```
# quicksort(array, 0, len(array) - 1)
# end = timeit.timeit()

# print(array)

print(end - start)



f.close()

print(end - start)
```

TIEMPOS QUICK–SORT

–Mejor Caso

N	Tiempo
100:	0.005853399999978137
200:	0.005635400000755908
300:	0.005375199998525204
400:	0.007245799999509472
500:	0.005091899998660665
600:	0.005041200000050594
700:	0.0055608000002393965
800:	0.005019500000344124
900:	0.005019799998990493
1000:	0.005130800000188174

–Peor Caso

N	Tiempo
100:	0.005725000000893488
200:	0.005246299999271287
300:	0.005106699998577824
400:	0.005185100000744569
500:	0.00503509999907692
600:	0.005808799998703762
700:	0.005068500000561471
800:	0.005053700000644312
900:	0.005033600000388105
1000:	0.005040399999415968

–Caso Random

N	Tiempo
100:	0.005251299999144976
200:	0.005269900000712369
300:	0.005247099999905913
400:	0.005210300001635915

```
500:    0.0050253999997948995
600:    0.005389100000684266
700:    0.005251699998552795
800:    0.00606819999848085
900:    0.005101500000819215
10000:  0.0056162000000767875,
```

El problema de las 8 reinas consiste en colocar las piezas en un tablero de ajedrez sin que se amenacen entre sí. En el juego de ajedrez la reina amenaza a aquellas piezas que se encuentren en su misma fila, columna o diagonal.

Para resolver este problema emplearemos un esquema de algoritmo Las vegas. Se supondrá que hay un cierto número de piezas colocadas en forma correcta (sin atacarse) y se generará el resto de las piezas al azar. Considera las siguientes configuraciones predefinidas e implemente un algoritmo que genere las otras reinas de manera aleatoria.

CODIGO QUEEN

```
class NQueens:
    """Generate all valid solutions for the n queens puzzle"""
    def __init__(self, size):
        # Store the puzzle (problem) size and the number of valid solutions
        self.size = size
        self.solutions = 0
        self.solve()

    def solve(self):
        positions = [-1] * self.size
        self.put_queen(positions, 0)
        print("Found", self.solutions, "solutions.")

    def put_queen(self, positions, target_row):
        # Base (stop) case - all N rows are occupied
        if target_row == self.size:
            self.show_full_board(positions)
            self.solutions += 1
        else:
            # For all N columns positions try to place a queen
            for column in range(self.size):
                # Reject all invalid positions
                if self.check_place(positions, target_row, column):
                    positions[target_row] = column
                    self.put_queen(positions, target_row + 1)

    def check_place(self, positions, ocuppied_rows, column):
        for i in range(ocuppied_rows):
            if positions[i] == column or \
```

```
                positions[i] - i == column - ocuppied_rows or \
                positions[i] + i == column + ocuppied_rows:

                return False
        return True

    def show_full_board(self, positions):
        for row in range(self.size):
            line = ""
            for column in range(self.size):
                if positions[row] == column:
                    line += "Q "
                else:
                    line += ". "
            print(line)
        print("\n")

NQueens(8)
```