Branch: master ▾ 03MAIR-Algoritmos-de-optimizacion / AG2 / AG2_Isabel_Vazquez.ipynb

Find file Copy path

404isabel Creado con Colaboratory ab51341 38 seconds ago

1 contributor

528 lines (528 sloc) | 17.5 KB

Raw Blame History

Open in Colab

Isabel Vázquez - AG2
Actividad Guiada2
Url: https://github.com/404isabel/03MAIR-Algoritmos-de-optimizacion/tree/master/AG2

In [0]:
```
import math
import random
from time import time
def calcular_tiempo(f):

    def wrapper(*args, **kwargs):
        inicio = time()
        resultado = f(*args, **kwargs)
        tiempo = float(time() - inicio)
        print("\r\n Tiempo de ejecución para algoritmo: "+"{0:.25f}".format(tiempo))
        return resultado

    return wrapper

N=10000
LISTA_2D = [ (random.randrange(1,N*10),random.randrange(1,N*10))  for _ in range(N)]
#print(LISTA_2D)
```

In [0]:
```
def distancia(A,B):
  if type(A) is int or type(A) is float:
    return abs(B-A)
  else:
    return math.sqrt(sum([(A[i]-B[i])**2 for i in range(len(A))]))
```

In [177]:
```
distancia([1,3],[2,5])
```

Out[177]:
```
2.23606797749979
```

Puntos más cercanos por fuerza bruta

In [178]:
```
#Fuerza bruta
def distancia_fuerza_bruta(L):
  mejor_distancia = float("infinity") #Modificación, se cambia mejor_distancia por float("infin
ity")

  A,B = (),()
  tamLista=len(L) #Modificación:Se mete tamLista en una variable, en vez de calcularlo 2 veces
  for i in range(tamLista):
    for j in range(i+1,tamLista):
      distanciaActual=distancia(L[i], L[j]) #Modificación:Se mete el cálculo de la distancia en
 una variable
      if distanciaActual<mejor_distancia:
        A,B=L[i], L[j]
        mejor_distancia=distanciaActual
  return [A,B]


#Modificación: se engloba la función en lanzarFuerzaBruta, para poder calcular el tiempo,
#y posteriormente poder reutilizar distancia_fuerza_bruta por separado, sin que calcule el tiem
po
@calcular_tiempo
def lanzarFuerzaBruta(L):
  distancia_fuerza_bruta(L)

lanzarFuerzaBruta(LISTA_2D[:100])
```

```
Tiempo de ejecución para algoritmo: 0.014406681060791015625000
```

Puntos más cercanos divide y vencerás

In [179]:
```
def distancia_divide_y_venceras(L):
  #Si hay pocos, resolvemos por fuerza bruta
  if len(L)<10:
    return distancia_fuerza_bruta(L)

  #Corrección (se incluye este fragmento de código ya que no funcionaba con listas de 1 dimensi
ón, fallaba por lambda x:x[0], y en el caso de ser de una dimensión x
  #no era un array,sino un valor numérico.
  #Además englobo todo en un else, reestructurando código
  else:
```

```
    if type(L[0]) is int or type(L[0]) is float:
       L=sorted(L)
    else:
      L=sorted(L, key=lambda x:x[0])

    LISTA_IZQ = L[:len(L)//2]
    LISTA_DER = L[len(L)//2:]

    PUNTOS_LISTA_IZQ = distancia_divide_y_venceras(LISTA_IZQ)
    PUNTOS_LISTA_DER = distancia_divide_y_venceras(LISTA_DER)
    return distancia_fuerza_bruta(PUNTOS_LISTA_IZQ+PUNTOS_LISTA_DER)

@calcular_tiempo
def LANZA(L):
  return distancia_divide_y_venceras(L)

#Prueba con el mismo número de elementos empleado para el algoritmo de fuerza bruta
print("----- Con 100 elementos:")
SOL1=LANZA(LISTA_2D[:100])
print(SOL1)
#Prueba con 1 dimensión
print("\r\n ----- Con 1 dimensión:")
SOL=LANZA([25,44,38,11,32,90,33])
print(SOL)
print("\r\n------- Con 10000 elementos:")
SOL=LANZA(LISTA_2D)
print(SOL)
```

```
----- Con 100 elementos:

 Tiempo de ejecución para algoritmo: 0.001838684082031250000000000
[(86411, 76254), (86910, 75747)]

 ----- Con 1 dimensión:

 Tiempo de ejecución para algoritmo: 0.000027179718017578125000000
[32, 33]

------- Con 10000 elementos:
```

```
 Tiempo de ejecución para algoritmo: 0.11564326286315917968750000
[(38446, 91081), (38450, 91085)]
```

Mi aportación para divide y vencerás

In [180]:

```
#En este algoritmo, parto de un array ordenado
#Voy dividiendo el algoritmo en dos de forma sucesiva y calculando las distancias
#de la matriz 1 y la matriz 2, así como sus elementos adyacentes
#es decir el último de matriz 1 y el primero de matriz 2
#Me quedo con la menor distancia obtenida
def divideVencerasMatriz(matriz):

  if(len(matriz)<2):
    return matriz

  elif(len(matriz)==2):
    punto1=matriz[0]
    punto2=matriz[1]
    matriz=[punto1,punto2]
    return matriz
  else:

    resultado=[0,0]

    #Divido en dos partes la matriz ya ordenada
    mitad=int(len(matriz)/2)
    matriz1=matriz[0:mitad]
    matriz2=matriz[mitad:]

    #distancia de las mitades
    #Compruebo la distancia del último elemento de la matriz 1 y del primer elemento de la matr
iz 2)
    distanciaTG=distancia(matriz1[mitad-1],matriz2[0])

    #Ejecuto de manera recursiva el algoritmo para la primera y la segunda parte de forma suces
iva
    minI=divideVencerasMatriz(matriz1)
    minD=divideVencerasMatriz(matriz2)
```

```
    #Si minI tiene dos elementos y minD también, compruebo la distancia
    #que hay entre el último elemento del primer array y el primer elemento del segundo

    #Si sólo tiene un elemento cojo como distanciaT la primera distancia obtenida:distanciaTG
    #Para más de 1 dimensión, no es suficiente con coger el último elemento y el primero, hay q
ue comparar los dos pares
    if(len(minI)>1 and len(minD)>1):
      distanciaT=distancia(minI[1],minD[0])
    else:
      distanciaT=distanciaTG

    #Compruebo la distancia de los elementos de minI y lo guardo en distanciaI, si sólo hay un
 elemento
    #cojo lo que hay en distanciaT
    if(len(minI)==1):
      distanciaI=distanciaT
    else:
      distanciaI=distancia(minI[0],minI[1])

    #Compruebo la distancia de los elementos de minD, y lo guardo en distanciaD, si sólo hay un
 elemento
    #cojo lo que hay en distanciaT
    if(len(minD)==1):
      distanciaD=distanciaT
    else:
      distanciaD=distancia(minD[0],minD[1])


    #Por último miro cual de todas las distancias es la menor, y la guardo en la variable resul
tado
    if(distanciaI<distanciaD and distanciaI<distanciaT and distanciaI<distanciaTG):
        resultado=[minI[0],minI[1]]
    elif(distanciaD<distanciaI and distanciaD<distanciaT and distanciaD<distanciaTG):
        resultado=[minD[0],minD[1]]
    elif(distanciaT<distanciaI and distanciaT<distanciaD and distanciaT<distanciaTG):
        resultado=[minD[0],minD[1]]
    elif(distanciaTG<distanciaI and distanciaTG<distanciaD and distanciaTG<distanciaT):
        resultado=[matriz1[mitad-1],matriz2[0]]
    elif(distanciaI==0 and len(minI)>1):
```

```
        resultado=[minI[0],minI[1]]
        return resultado
    elif(distanciaD==0 and len(minD)>1):
        resultado=[minD[0],minD[1]]
        return resultado
    else:
        resultado=[matriz1[mitad-1],matriz2[0]]


    return resultado


@calcular_tiempo
def divideVencerasMatrizT(matriz):
  if type(matriz[0]) is int or type(matriz[0]) is float:
       matriz=sorted(matriz)
  else:
      matriz=sorted(matriz, key=lambda x:x[0])
  return divideVencerasMatriz(matriz)


#Prueba con 1 dimensión
print("\r\n ---- Con 1 dimensión:")
print(divideVencerasMatrizT([25,44,38,11,32,90,33]))
print("\r\n ---- Con 10000 elementos:")
print(divideVencerasMatrizT(LISTA_2D))
```

```
 ---- Con 1 dimensión:

 Tiempo de ejecución para algoritmo: 0.000024080276489257812500
0
[32, 33]

 ---- Con 10000 elementos:

 Tiempo de ejecución para algoritmo: 0.061123609542846679687500
0
[(38446, 91081), (38450, 91085)]
```

Viaje por el rio

In [181]:
```
TARTEAS = [
```

```
[0,5,4,3,999,999,999],
[999,0,999,2,3,999,11],
[999,999, 0,1,999,4,10],
[999,999,999, 0,5,6,9],
[999,999, 999,999,0,999,4],
[999,999, 999,999,999,0,3],
[999,999,999,999,999,999,0]
]

#Viaje por el rio
def Precios(TARIFAS):
  N = len(TARIFAS[0])

  PRECIOS = [ [999]*N for i in[9999]*N]
  RUTA = [ [""]*N for i in[9999]*N ]



  for i in range(N-1):
    for j in range(i+1,N):
      MIN = TARIFAS[i][j]
      RUTA[i][j] = i

      for k in range(i,j):
        if PRECIOS[i][k]+TARIFAS[k][j]<MIN:
          #MIN = min(MIN, PRECIOS[i][k]+TARIFAS[k][j])
          MIN=PRECIOS[i][k]+TARIFAS[k][j] #Modificación: guardo en MIN (PRECIOS[i][k]+TARIFAS
[k][j]), en lugar de guardar el min de los valores
          RUTA[i][j] = k
      PRECIOS[i][j]=MIN
  return PRECIOS,RUTA

PRECIOS,RUTAS = Precios(TARIFAS)
print(PRECIOS)
print("\r\n")
print(RUTAS)
```

```
[[999, 5, 4, 3, 8, 8, 11], [999, 999, 999, 2, 3, 8, 7], [999, 999, 999, 1, 6, 4, 7], [999, 999,
999, 999, 5, 6, 9], [999, 999, 999, 999, 999, 999, 4], [999, 999, 999, 999, 999, 999, 3], [999,
999, 999, 999, 999, 999, 999]]
```

```
[['', 0, 0, 0, 1, 2, 5], ['', '', 1, 1, 1, 3, 4], ['', '', '', 2, 3, 2, 5], ['', '', '', '', 3,
3, 3], ['', '', '', '', '', 4, 4], ['', '', '', '', '', '', 5], ['', '', '', '', '', '', '']]
```

In [174]:
```
#Pintar la ruta
def calcular_ruta(RUTAS, desde, hasta):
  if desde == hasta:
    #print("Ir a :" + str(desde))
    return desde
  else:
    return str(calcular_ruta(RUTAS, desde, RUTAS[desde][hasta])) +  ',' + str(RUTAS[desde][hast
a])

print("\nLa ruta es:")
calcular_ruta(RUTAS, 0,6)
```

```
La ruta es:
```

Out[174]:
```
'0,0,2,5'
```

In [0]:
```
#Órden de complejidad n**3
```