

VictorBetancourt-03MIAR_AG1.ipynb - Colaboratory

 colab.research.google.com/drive/1ztFj0oaNhtXbEu0tEkc6PdZ36dA9VDkw



03MIAR_10_A Algoritmos de Optimización

Actividad Guiada 1 (AG1)



Máster en Inteligencia Artificial

Curso 2023-2024. Edición Oct-2023

Profr.: **Raúl Reyero**

Alumno: **Victor David Betancourt Leal**

Repositorio

-  Notebook Colab:
<https://colab.research.google.com/drive/1ztFj0oaNhtXbEu0tEkc6PdZ36dA9VDkw?usp=sharing>
-  Repositorio GitHub:
<https://github.com/vbleal/03MIAR/tree/477d03010bb7de3900c97d07f5fe25a4fecbaa78/AG1>

Índice

1. Divide y Vencerás (Divide and Conquer, DC)
2. Algoritmos Voraces (Greedy Algorithms)
3. Algoritmos con Vuelta Atrás (Backtracking)
4. Programación Dinámica
5. Problema Adicional

Bibliografía

1. Divide y Vencerás (*Divide and Conquer, DC*)

Torres de Hanoi

```
#Torres de Hanoi - Divide y venceras
#####

#####
def Torres_Hanoi(N, desde, hasta):
    #N - N° de fichas
    #desde - torre inicial
    #hasta - torre fina
    if N==1 :
        print("Lleva la ficha desde " + str(desde) + " hasta " + str(hasta))

    else:
        Torres_Hanoi(N-1, desde, 6-desde-hasta)
        print("Lleva la ficha desde " + str(desde) + " hasta " + str(hasta))
        Torres_Hanoi(N-1, 6-desde-hasta, hasta)

Torres_Hanoi(3, 1, 3)
#####
```

Lleva la ficha desde 1 hasta 3
Lleva la ficha desde 1 hasta 2
Lleva la ficha desde 3 hasta 2
Lleva la ficha desde 1 hasta 3
Lleva la ficha desde 2 hasta 1
Lleva la ficha desde 2 hasta 3
Lleva la ficha desde 1 hasta 3

2. Algoritmos Voraces (*Greedy Algorithms*)

Cambio de Monedas

```
#Cambio de monedas - Técnica voraz
#####
SISTEMA = [12, 5 ,2, 1 ]
#####
def cambio_monedas(CANTIDAD,SISTEMA):
    #...
    SOLUCION = [0]*len(SISTEMA)
    ValorAcumulado = 0

    for i,valor in enumerate(SISTEMA):
        monedas = (CANTIDAD-ValorAcumulado)//valor
        SOLUCION[i] = monedas
        ValorAcumulado = ValorAcumulado + monedas*valor

    if CANTIDAD == ValorAcumulado:
        return SOLUCION

    print("No es posible encontrar solucion")
cambio_monedas(15,SISTEMA)

#####

[1, 0, 1, 1]
```

3. Algoritmos con Vuelta Atrás (*Backtracking*)

N Reinas

```
#N Reinas - Vuelta Atrás()
#####

#Verifica que en la solución parcial no hay amenazas entre reinas
#####
def es_prometedora(SOLUCION,etapa):
#####
    #print(SOLUCION)
    #Si la solución tiene dos valores iguales no es valida => Dos reinas en la misma fila
    for i in range(etapa+1):
        #print("El valor " + str(SOLUCION[i]) + " está " + str(SOLUCION.count(SOLUCION[i])) + " veces")
        if SOLUCION.count(SOLUCION[i]) > 1:
            return False

    #Verifica las diagonales
    for j in range(i+1, etapa +1 ):
        #print("Comprobando diagonal de " + str(i) + " y " + str(j))
        if abs(i-j) == abs(SOLUCION[i]-SOLUCION[j]) : return False
    return True

#Traduce la solución al tablero
#####
def escribe_solucion(S):
#####
    n = len(S)
    for x in range(n):
        print("")
        for i in range(n):
            if S[i] == x+1:
                print(" X " , end="")
            else:
                print(" - ", end="")

#Proceso principal de N-Reinas
#####
def reinas(N, solucion=[],etapa=0):
#####
    ### ....
    if len(solucion) == 0:          # [0,0,0...]
        solucion = [0 for i in range(N) ]

    for i in range(1, N+1):
        solucion[etapa] = i
        if es_prometedora(solucion, etapa):
            if etapa == N-1:
                print(solucion)
            else:
                reinas(N, solucion, etapa+1)
        else:
            None

    solucion[etapa] = 0

reinas(8,solucion=[],etapa=0)
```

[1, 5, 8, 6, 3, 7, 2, 4]
[1, 6, 8, 3, 7, 4, 2, 5]
[1, 7, 4, 6, 8, 2, 5, 3]
[1, 7, 5, 8, 2, 4, 6, 3]
[2, 4, 6, 8, 3, 1, 7, 5]
[2, 5, 7, 1, 3, 8, 6, 4]
[2, 5, 7, 4, 1, 8, 6, 3]
[2, 6, 1, 7, 4, 8, 3, 5]
[2, 6, 8, 3, 1, 4, 7, 5]
[2, 7, 3, 6, 8, 5, 1, 4]
[2, 7, 5, 8, 1, 4, 6, 3]
[2, 8, 6, 1, 3, 5, 7, 4]
[3, 1, 7, 5, 8, 2, 4, 6]
[3, 5, 2, 8, 1, 7, 4, 6]
[3, 5, 2, 8, 6, 4, 7, 1]
[3, 5, 7, 1, 4, 2, 8, 6]
[3, 5, 8, 4, 1, 7, 2, 6]
[3, 6, 2, 5, 8, 1, 7, 4]
[3, 6, 2, 7, 1, 4, 8, 5]
[3, 6, 2, 7, 5, 1, 8, 4]
[3, 6, 4, 1, 8, 5, 7, 2]
[3, 6, 4, 2, 8, 5, 7, 1]
[3, 6, 8, 1, 4, 7, 5, 2]
[3, 6, 8, 1, 5, 7, 2, 4]
[3, 6, 8, 2, 4, 1, 7, 5]
[3, 7, 2, 8, 5, 1, 4, 6]
[3, 7, 2, 8, 6, 4, 1, 5]
[3, 8, 4, 7, 1, 6, 2, 5]
[4, 1, 5, 8, 2, 7, 3, 6]
[4, 1, 5, 8, 6, 3, 7, 2]
[4, 2, 5, 8, 6, 1, 3, 7]
[4, 2, 7, 3, 6, 8, 1, 5]
[4, 2, 7, 3, 6, 8, 5, 1]
[4, 2, 7, 5, 1, 8, 6, 3]
[4, 2, 8, 5, 7, 1, 3, 6]
[4, 2, 8, 6, 1, 3, 5, 7]
[4, 6, 1, 5, 2, 8, 3, 7]
[4, 6, 8, 2, 7, 1, 3, 5]
[4, 6, 8, 3, 1, 7, 5, 2]
[4, 7, 1, 8, 5, 2, 6, 3]
[4, 7, 3, 8, 2, 5, 1, 6]
[4, 7, 5, 2, 6, 1, 3, 8]
[4, 7, 5, 3, 1, 6, 8, 2]
[4, 8, 1, 3, 6, 2, 7, 5]
[4, 8, 1, 5, 7, 2, 6, 3]
[4, 8, 5, 3, 1, 7, 2, 6]
[5, 1, 4, 6, 8, 2, 7, 3]
[5, 1, 8, 4, 2, 7, 3, 6]
[5, 1, 8, 6, 3, 7, 2, 4]
[5, 2, 4, 6, 8, 3, 1, 7]
[5, 2, 4, 7, 3, 8, 6, 1]
[5, 2, 6, 1, 7, 4, 8, 3]

[5, 2, 8, 1, 4, 7, 3, 6]
[5, 3, 1, 6, 8, 2, 4, 7]
[5, 3, 1, 7, 2, 8, 6, 4]
[5, 3, 8, 4, 7, 1, 6, 2]
[5, 7, 1, 3, 8, 6, 4, 2]
[5, 7, 1, 4, 2, 8, 6, 3]
[5, 7, 2, 4, 8, 1, 3, 6]
[5, 7, 2, 6, 3, 1, 4, 8]
[5, 7, 2, 6, 3, 1, 8, 4]
[5, 7, 4, 1, 3, 8, 6, 2]
[5, 8, 4, 1, 3, 6, 2, 7]
[5, 8, 4, 1, 7, 2, 6, 3]
[6, 1, 5, 2, 8, 3, 7, 4]
[6, 2, 7, 1, 3, 5, 8, 4]
[6, 2, 7, 1, 4, 8, 5, 3]
[6, 3, 1, 7, 5, 8, 2, 4]
[6, 3, 1, 8, 4, 2, 7, 5]
[6, 3, 1, 8, 5, 2, 4, 7]
[6, 3, 5, 7, 1, 4, 2, 8]
[6, 3, 5, 8, 1, 4, 2, 7]
[6, 3, 7, 2, 4, 8, 1, 5]
[6, 3, 7, 2, 8, 5, 1, 4]
[6, 3, 7, 4, 1, 8, 2, 5]
[6, 4, 1, 5, 8, 2, 7, 3]
[6, 4, 2, 8, 5, 7, 1, 3]
[6, 4, 7, 1, 3, 5, 2, 8]
[6, 4, 7, 1, 8, 2, 5, 3]
[6, 8, 2, 4, 1, 7, 5, 3]
[7, 1, 3, 8, 6, 4, 2, 5]
[7, 2, 4, 1, 8, 5, 3, 6]
[7, 2, 6, 3, 1, 4, 8, 5]
[7, 3, 1, 6, 8, 5, 2, 4]
[7, 3, 8, 2, 5, 1, 6, 4]
[7, 4, 2, 5, 8, 1, 3, 6]
[7, 4, 2, 8, 6, 1, 3, 5]
[7, 5, 3, 1, 6, 8, 2, 4]
[8, 2, 4, 1, 7, 5, 3, 6]
[8, 2, 5, 3, 1, 7, 4, 6]
[8, 3, 1, 6, 2, 5, 7, 4]
[8, 4, 1, 3, 6, 2, 7, 5]

escribe_solucion([1, 5, 8, 6, 3, 7, 2, 4])

X	-	-	-	-	-	-	-
-	-	-	-	-	-	X	-
-	-	-	-	X	-	-	-
-	-	-	-	-	-	-	X
-	X	-	-	-	-	-	-
-	-	-	X	-	-	-	-
-	-	-	-	-	X	-	-
-	-	X	-	-	-	-	-

4. Programación Dinámica

Viaje por el Río

```

#Viaje por el rio - Programación dinámica
#####

TARIFAS = [
[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]
]

#999 se puede sustituir por float("inf")

#Calculo de la matriz de PRECIOS y RUTAS
#####
def Precios(TARIFAS):
#####
    #Total de Nodos
    N = len(TARIFAS[0])

    #Inicialización de la tabla de precios
    PRECIOS = [ [9999]*N for i in [9999]*N]
    RUTA = [ [""]*N for i in [""]*N]

    for i in range(0,N-1):
        RUTA[i][i] = i           #Para ir de i a i se "pasa por i"
        PRECIOS[i][i] = 0       #Para ir de i a i se se paga 0
        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] )
                    RUTA[i][j] = k           #Anota que para ir de i a j hay que pasar por k
            PRECIOS[i][j] = MIN

    return PRECIOS,RUTA
#####

PRECIOS,RUTA = Precios(TARIFAS)
#print(PRECIOS[0][6])

print("PRECIOS")
for i in range(len(TARIFAS)):
    print(PRECIOS[i])

print("\nRUTA")
for i in range(len(TARIFAS)):
    print(RUTA[i])

#Determinar la ruta con Recursividad
def calcular_ruta(RUTA, desde, hasta):
    if desde == hasta:
        #print("Ir a :" + str(desde))
        return ""
    else:
        return str(calcular_ruta( RUTA, desde, RUTA[desde][hasta])) + \
            ',' + \
            str(RUTA[desde][hasta] \
            )

```



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

5. Problema Adicional

Puntos más cercanos



Planteamiento del Problema

- **Supuesto:**

Dado un conjunto de puntos, el objetivo es encontrar los dos puntos más cercanos.

- **Guía para aprendizaje:**

1. Suponer en 1D, es decir, una lista de números, como [3403, 4537, 9089, 9746, 7259, ...].
2. Primer intento: Utilizar el método de fuerza bruta para encontrar la pareja de puntos más cercanos.
3. Calcular la complejidad del método de fuerza bruta y considerar si se puede mejorar.
4. Segundo intento: Aplicar el método Divide y Vencerás para intentar mejorar la complejidad del algoritmo.
5. Calcular la complejidad del método Divide y Vencerás y determinar si hay mejoras.
6. Extender el algoritmo a 2D, trabajando con puntos definidos por pares de números, como [(1122, 6175), (135, 4076), (7296, 2741)...].
7. Extender el algoritmo a 3D.



Análisis

1D

1D con Fuerza Bruta

Para la lista en 1D, el método de fuerza bruta implicaría comparar cada punto con todos los demás puntos para encontrar la distancia mínima.

1D con Divide y Vencerás

En 1D, este algoritmo implicaría dividir la lista de números ordenada en dos mitades, encontrar el par de puntos más cercanos dentro de cada mitad y luego encontrar el par más cercano que cruza el punto medio.

2D

En 2D, además de dividir los puntos, tendría que considerarse un área alrededor de la línea divisoria donde los puntos más cercanos podrían residir en diferentes mitades.

3D

En 3D, lo anterior se extiende a un volumen alrededor del plano divisorio.



Solución

```
import math
```

1D

1D con Fuerza Bruta

```
# 1D: Fuerza Bruta
def closest_pair_brute_force(points):
    min_distance = float('inf')
    closest_points = None
    for i in range(len(points) - 1):
        for j in range(i + 1, len(points)):
            distance = abs(points[i] - points[j])
            if distance < min_distance:
                min_distance = distance
                closest_points = (points[i], points[j])
    return closest_points, min_distance
```

La complejidad computacional de este enfoque es $O(n^2)$, donde n es el número de puntos en la lista. Ahora, se procederá a intentar mejorar esta complejidad utilizando el enfoque de Divide y Vencerás.

Es decir:

1. Se itera sobre cada punto de la lista.
2. Para cada punto, se itera nuevamente sobre todos los otros puntos para comparar las distancias.
3. Esto significa que para cada uno de los n puntos, se hacen $n-1$ comparaciones (en la práctica, es aproximadamente n comparaciones porque $n-1$ es casi igual a n para grandes valores de n).

Esto da lugar a un bucle anidado, donde la cantidad de operaciones es proporcional a $n \cdot n$, que se simplifica como $O(n^2)$. Este es un crecimiento cuadrático, lo que significa que si se duplica el número de puntos, el tiempo que toma resolver el problema se multiplica por 4.

1D con Divide y Vencerás

```
# 1D: Divide y Vencerás
def closest_pair_divide_and_conquer(points):
    if len(points) <= 3:
        return closest_pair_brute_force(points)
    mid = len(points) // 2
    left_points = points[:mid]
    right_points = points[mid:]
    left_pair, left_distance = closest_pair_divide_and_conquer(left_points)
    right_pair, right_distance = closest_pair_divide_and_conquer(right_points)
    closest_pair, closest_distance = (left_pair, left_distance) if left_distance < right_distance else (right_pair, right_distance)
    return closest_pair, closest_distance
```

La complejidad computacional de este método es $O(n \cdot \log(n))$, debido principalmente al paso de ordenamiento. La parte de dividir y vencer realmente opera en $O(n)$, pero como se necesita ordenar la lista al principio, esa sería la complejidad dominante.

Nótese que:

1. Antes de aplicar el método, la lista de puntos se ordena, lo cual tiene una complejidad temporal de $O(n \cdot \log(n))$ para algoritmos de ordenamiento eficientes como **Merge Sort** o **Quick Sort**.
2. El método divide y vencerás divide la lista de puntos en mitades hasta llegar a un caso base que se resuelve con fuerza bruta. Esto sigue una progresión logarítmica, dividiendo el conjunto de datos en cada paso. El número de divisiones que se pueden hacer es proporcional a $\log(n)$.
3. En cada nivel de división, se hacen comparaciones que en total suman n operaciones (por ejemplo, comparar los puntos más cercanos en la franja central). Debido a que esto se hace en cada nivel del árbol de recursión, se multiplica por $\log(n)$, lo que da una complejidad de las operaciones de división y conquista de $O(n \cdot \log(n))$.

La razón por la cual la complejidad dominante es $O(n \cdot \log(n))$ y no solo $O(n)$, a pesar de que el divide y vencerás por sí solo sería $O(n)$, es porque el paso de ordenamiento es necesario y su complejidad no puede ser ignorada. Es el paso más costoso y, por lo tanto, define la complejidad general del algoritmo.

2D

Distancia Euclideana en 2D

```
# Distancia Euclideana en 2D
def euclidean_distance_2d(point1, point2):
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2)
```

2D con Fuerza Bruta

```
# 2D con Fuerza Bruta
def closest_pair_brute_force_2d(points):
    min_distance = float('inf')
    closest_points = None
    for i in range(len(points) - 1):
        for j in range(i + 1, len(points)):
            distance = euclidean_distance_2d(points[i], points[j])
            if distance < min_distance:
                min_distance = distance
                closest_points = (points[i], points[j])
    return closest_points, min_distance
```

2D con Divide y Vencerás

```

# 2D con Divide y Vencerás
def closest_pair_dc_2d(points):
    if len(points) <= 3:
        return closest_pair_brute_force_2d(points)

    mid = len(points) // 2
    left_points = points[:mid]
    right_points = points[mid:]
    left_pair, left_distance = closest_pair_dc_2d(left_points)
    right_pair, right_distance = closest_pair_dc_2d(right_points)
    closest_pair, closest_distance = (left_pair, left_distance) if left_distance < right_distance else (right_pair, right_distance)

    strip_points = [point for point in points if abs(point[0] - points[mid][0]) < closest_distance]
    strip_points.sort(key=lambda point: point[1])

    for i in range(len(strip_points)):
        for j in range(i+1, len(strip_points)):
            if strip_points[j][1] - strip_points[i][1] < closest_distance:
                distance = euclidean_distance_2d(strip_points[i], strip_points[j])
                if distance < closest_distance:
                    closest_distance = distance
                    closest_pair = (strip_points[i], strip_points[j])
            else:
                break

    return closest_pair, closest_distance

```

3D

Distancia Euclídeana en 3D

```

# Distancia Euclídeana
def euclidean_distance_3d(point1, point2):
    return math.sqrt((point1[0] - point2[0])**2 + (point1[1] - point2[1])**2 + (point1[2] - point2[2])**2)

```

3D con Fuerza Bruta

```

# 3D con Fuerza Bruta
def closest_pair_brute_force_3d(points):
    min_distance = float('inf')
    closest_points = None
    for i in range(len(points) - 1):
        for j in range(i + 1, len(points)):
            distance = euclidean_distance_3d(points[i], points[j])
            if distance < min_distance:
                min_distance = distance
                closest_points = (points[i], points[j])
    return closest_points, min_distance

```

3D con Divide y Vencerás

```
# 3D con Divide y Vencerás
def closest_pair_dc_3d(points):
    if len(points) <= 3:
        return closest_pair_brute_force_3d(points)

    mid = len(points) // 2
    left_points = points[:mid]
    right_points = points[mid:]
    left_pair, left_distance = closest_pair_dc_3d(left_points)
    right_pair, right_distance = closest_pair_dc_3d(right_points)
    closest_pair, closest_distance = (left_pair, left_distance) if left_distance < right_distance else (right_pair, right_distance)

    strip_points = [point for point in points if abs(point[0] - points[mid][0]) < closest_distance]
    strip_points.sort(key=lambda point: (point[1], point[2]))

    for i in range(len(strip_points)):
        for j in range(i+1, min(i + 7, len(strip_points))):
            distance = euclidean_distance_3d(strip_points[i], strip_points[j])
            if distance < closest_distance:
                closest_distance = distance
                closest_pair = (strip_points[i], strip_points[j])

    return closest_pair, closest_distance
```

Ejemplo

```
# Ejemplo de uso
points_1d = [3403, 4537, 9089, 9746, 7259]
points_2d = [(1122, 6175), (135, 4076), (7296, 2741)]
points_3d = [(1122, 6175, 285), (135, 4076, 1235), (7296, 2741, 567)]

# Llamar a las funciones con los conjuntos de puntos
closest_1d, distance_1d = closest_pair_divide_and_conquer(sorted(points_1d))
closest_2d, distance_2d = closest_pair_dc_2d(points_2d)
closest_3d, distance_3d = closest_pair_dc_3d(points_3d)

# Imprimir resultados
print("Puntos más cercanos en 1D:", closest_1d, "con distancia:", distance_1d)
print("Puntos más cercanos en 2D:", closest_2d, "con distancia:", distance_2d)
print("Puntos más cercanos en 3D:", closest_3d, "con distancia:", distance_3d)
```

```
Puntos más cercanos en 1D: (9089, 9746) con distancia: 657
Puntos más cercanos en 2D: ((1122, 6175), (135, 4076)) con distancia:
2319.4762339804215
Puntos más cercanos en 3D: ((1122, 6175, 285), (135, 4076, 1235)) con distancia:
2506.4855874311347
```

Usando Números Aleatorios y el Algoritmo de Divide y Vencerás

En este apartado, se aplican las funciones creadas anteriormente para 1D, 2D, y 3D, pero usando **números (puntos) aleatorios** como inputs.

```
import random
```

1D

```
# Generar puntos aleatorios para 1D
lista_1d_aleatoria = [random.randrange(1, 10000) for x in range(1000)]

#closest_1d, distance_1d = closest_pair_divide_and_conquer(sorted(lista_1d_aleatoria)) # Siempre da cero si
closest_1d, distance_1d = closest_pair_divide_and_conquer(lista_1d_aleatoria)
print("Puntos más cercanos en 1D", closest_1d, "con distancia:", distance_1d)
```

Puntos más cercanos en 1D (9388, 9399) con distancia: 11

2D

```
# Generar puntos aleatorios para 2D
lista_2d_aleatoria = [(random.randrange(1, 10000), random.randrange(1, 10000)) for x in range(1000)]

# Ejemplo en 2D con puntos aleatorios
closest_2d, distance_2d = closest_pair_dc_2d(lista_2d_aleatoria)
print("Puntos más cercanos en 2D:", closest_2d, "con distancia:", distance_2d)
```

Puntos más cercanos en 2D: ((6601, 4914), (6596, 4922)) con distancia:
9.433981132056603

3D

```
# Generar puntos aleatorios para 3D
lista_3d_aleatoria = [(random.randrange(1, 10000), random.randrange(1, 10000), random.randrange(1, 10000))
```

```
# Ejemplo en 3D con puntos aleatorios
closest_3d, distance_3d = closest_pair_dc_3d(lista_3d_aleatoria)
print("Puntos más cercanos en 3D:", closest_3d, "con distancia:", distance_3d)
```

Puntos más cercanos en 3D: ((7006, 3101, 7686), (6879, 3058, 7532)) con distancia:
204.1910869749216



Bibliografía

1. Bratley P. & Brassard, G. (2000). ***Fundamentos de Algoritmia***. Prentice Hall.
2. Guerequeta, R. & Vallecillo, A. (1998). ***Técnicas de Diseño de Algoritmos***. Servicio de Publicaciones de la Universidad de Málaga.
3. Lee (2007). ***Introducción al Diseño y Análisis de Algoritmos***. McGraw Hill.