



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE COMPUTO



2024

SESIÓN DE LABORATORIO 3: ALGORITMO VORAZ

ALGORITMOS BIOINSPIRADOS

Elaborado por: Flores Estopier Rodrigo

Boleta: 2021630260

Profesor: Rosas Trigueros Jorge Luis

Grupo: 6CV2

Fecha de realización: 21/09/2024

Fecha de entrega: 24/09/2024

Semestre: 25-1

Contenido

1. Marco Teórico	2
2. Desarrollo	3
Problema del Cambio	3
Problema de la mochila 0/1	6
3. Conclusiones	8
4. Bibliografía	9

1. MARCO TEÓRICO

Los algoritmos voraces (o greedy algorithms) son una técnica de diseño de algoritmos que se utilizan para resolver problemas de optimización. La estrategia de un algoritmo voraz es tomar decisiones locales en cada paso del proceso, eligiendo siempre la opción que parece ser la más prometedora en ese momento, con la esperanza de que esta elección lleve a una solución globalmente óptima.

A diferencia de otros enfoques como la programación dinámica, los algoritmos voraces no reconsideran las decisiones previamente tomadas, lo que los hace eficientes en términos de tiempo de ejecución. Sin embargo, esta misma característica puede provocar que no siempre encuentren la solución óptima en todos los problemas, ya que no revisan ni optimizan las decisiones tomadas previamente.

Características

1. **Soluciones parciales:** Se construye la solución paso a paso, tomando decisiones locales que parecen las mejores en cada instante.
2. **Factibilidad:** Cada elección debe ser factible, es decir, debe satisfacer las restricciones del problema.
3. **Selección voraz:** En cada paso, se elige la mejor opción posible en ese momento, basándose en algún criterio.
4. **Solución óptima:** El algoritmo debe garantizar (aunque no siempre lo hace) que la solución final es óptima para el problema global.

Algunos problemas se resuelven de manera eficiente con este enfoque, mientras que, en otros, la solución obtenida puede ser subóptima.

Problema del Cambio

El problema del cambio consiste en determinar la cantidad mínima de monedas necesarias para devolver una cantidad CCC de dinero, dada una serie de denominaciones de monedas. Este problema es particularmente adecuado para la estrategia voraz en ciertos casos, donde el conjunto de denominaciones cumple con ciertas propiedades.

En el algoritmo voraz para el problema del cambio, se selecciona en cada paso la moneda de mayor denominación posible que no exceda la cantidad restante de dinero a devolver.

Luego, se resta esta denominación de la cantidad pendiente y se repite el proceso hasta que se alcance el valor CCC.

Problema de la Mochila 0/1

En la versión 0/1 del problema de la mochila, donde los objetos solo pueden tomarse completamente o no tomarse, el enfoque voraz no siempre da una solución óptima. Este problema requiere un enfoque como la programación dinámica, ya que las decisiones de tomar un objeto completamente o no pueden influir en las elecciones posteriores.

Los algoritmos voraces son útiles en situaciones donde las decisiones locales óptimas conducen a una solución global óptima, como en el caso del problema del cambio con denominaciones adecuadas. Sin embargo, no siempre son la mejor solución para problemas más complejos, como la mochila 0/1, donde las elecciones tempranas pueden afectar negativamente las soluciones futuras.

Estos algoritmos son ampliamente aplicados en problemas de redes, asignación de recursos, y planificación, entre otros. La simplicidad y la eficiencia de los algoritmos voraces los hacen adecuados para una variedad de problemas de optimización cuando se busca una solución rápida y aproximada.

2. DESARROLLO

Se desarrollarán dos algoritmos utilizando las técnicas de algoritmos voraces para encontrar una solución de los problemas de la mochila, y el problema del cambio. De nuevo se utilizará Python en el entorno de desarrollo de Google Colab.

PROBLEMA DEL CAMBIO

Se tomaron en cuenta las siguientes variables:

- d: Estas son las denominaciones de las monedas disponibles.
- N: Es la cantidad total de dinero que se desea cambiar. Problema de la Mochila 0/1.

```
def changeCMP(d,N):
    #Ordenar las denominaciones de mayor a menor
    d.sort(reverse=True)

    #Imprimir las denominaciones ordenadas
    print("Denominaciones ordenadas: ")
    print(d)

    monedas = []

    while N !=0:
        for i in range(0,len(d)):
            if d[i] <= N:
                monedas.append(d[i])
                N = N-d[i]
                break

    #Imprimir la cantidad de monedas de cada denominacion
    print("Cantidad de monedas de cada denominacion: ")
    for i in range(0,len(d)):
        print(d[i],":",monedas.count(d[i]))

d = [1,3,4]
N= 6
changeCMP(d,N)
```

El algoritmo comienza ordenando las denominaciones en orden descendente, ya que la estrategia voraz siempre selecciona la mayor denominación posible en cada paso. Esto asegura que se prioricen las monedas de mayor valor, lo que generalmente minimiza el número de monedas utilizadas (aunque no siempre garantiza la solución óptima en todos los casos).

Se crea una lista vacía llamada `monedas`, que almacenará las denominaciones de las monedas que se seleccionen para devolver el cambio.

Se ejecuta un bucle `while` que continúa hasta que `N` sea igual a 0, es decir, hasta que se haya devuelto toda la cantidad.

Dentro del bucle, se recorre la lista de denominaciones ordenadas en un bucle `for`.

- Para cada denominación `d[i]`, se verifica si es menor o igual a la cantidad restante `N`. Si lo es, esa moneda se añade a la lista `monedas`.
- Luego, se actualiza el valor de `N` restando el valor de la moneda seleccionada.

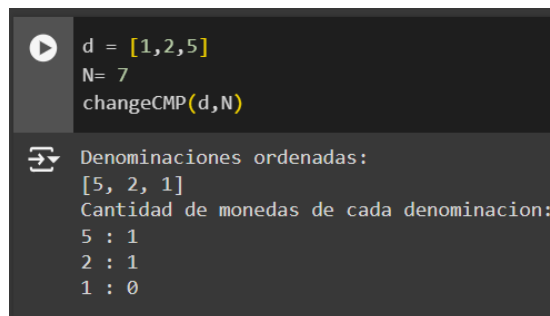
- El break finaliza el bucle for tan pronto como se selecciona una moneda, volviendo a evaluar la nueva cantidad restante N en la siguiente iteración del while.

Una vez que se ha devuelto la cantidad exacta N y el bucle ha finalizado, el código imprime cuántas monedas de cada denominación se usaron.

La función `monedas.count(d[i])` cuenta cuántas veces aparece una denominación `d[i]` en la lista `monedas`, lo que corresponde a la cantidad de veces que se ha usado esa moneda para devolver el cambio.

Ejemplo con solución óptima

En el ejemplo presentado en la Ilustración 1 podemos observar que el algoritmo voraz encuentra la solución óptima, tomando una moneda de 5 y otra de 2 para pagar un total de 7.



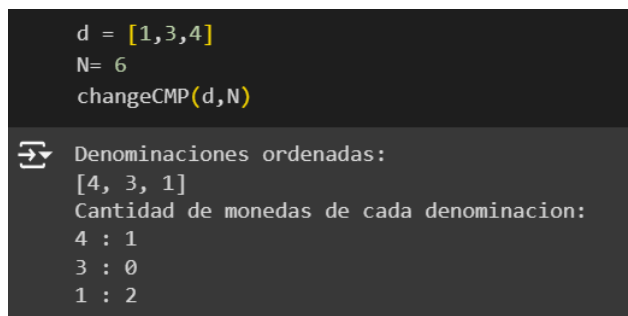
```
d = [1,2,5]
N= 7
changeCMP(d,N)

⇌ Denominaciones ordenadas:
[5, 2, 1]
Cantidad de monedas de cada denominacion:
5 : 1
2 : 1
1 : 0
```

Ilustración 1. Instancia del problema con solución óptima.

Ejemplo con solución no óptima

En el ejemplo presentado en la Ilustración 2 podemos observar que el algoritmo voraz encuentra una solución, sin embargo, esta solución no es la óptima. En este caso toma una moneda de 4 y dos de 1 para pagar 6 con tres monedas. La solución óptima será tomar dos monedas de 3.



```
d = [1,3,4]
N= 6
changeCMP(d,N)

⇌ Denominaciones ordenadas:
[4, 3, 1]
Cantidad de monedas de cada denominacion:
4 : 1
3 : 0
1 : 2
```

Ilustración 2. Instancia del problema con solución no óptima.

PROBLEMA DE LA MOCHILA 0/1

Se tomaron en cuenta las siguientes variables:

- **capacidad:** La capacidad total de la mochila (peso máximo que puede soportar).
- **pesos:** Una lista que contiene los pesos de los objetos.
- **valores:** Una lista que contiene los valores de los objetos.

```
def mochila01(capacidad, pesos, valores):  
    #Crear tupla  
    items = list(zip(pesos, valores))  
    #Ordenar tupla por mayor valor  
    items.sort(key=lambda x: x[1], reverse=True)  
  
    #Imprimir tupla ordenada  
    print("Tupla ordenada: ")  
    print(items)  
  
    itemsTomados = []  
    valorTotal = 0  
  
    for i in range(0, len(items)):  
        if items[i][0] <= capacidad:  
            capacidad = capacidad - items[i][0]  
            itemsTomados.append(items[i])  
            valorTotal = valorTotal + items[i][1]  
  
    #Imprimir items tomados  
    print("Items tomados: ")  
    print(itemsTomados)  
  
    #Imprimir valor total  
    print("Valor total: ")  
    print(valorTotal)
```

Primero, se combina cada peso con su valor correspondiente en una lista de tuplas, donde cada tupla tiene la forma (peso, valor).

Se ordenan las tuplas según el valor de los objetos, de mayor a menor, utilizando la función `sort()` y la clave `key=lambda x: x[1]`, que accede al segundo elemento de la tupla (el valor del objeto).

Esta estrategia sigue el enfoque voraz de priorizar los objetos con mayor valor, intentando maximizar el valor total en cada paso

El código recorre cada objeto en la lista `items` y evalúa si su peso (primer valor de la tupla) es menor o igual a la capacidad restante de la mochila.

- Si el objeto cabe en la mochila (su peso es menor o igual a la capacidad), se selecciona:
 - Se resta el peso del objeto a la capacidad restante.
 - Se añade el objeto a la lista `itemsTomados`.
 - Se incrementa el `valorTotal` sumando el valor del objeto seleccionado.
- Este proceso continúa hasta que la capacidad de la mochila ya no permite añadir más objetos.

Ejemplo con solución optima

En el ejemplo presentado en la Ilustración 3 podemos observar que el algoritmo voraz encuentra la solución óptima, maximizando el valor de la mochila dentro de la capacidad establecida, tomando los objetos con valor 120 y 100, con un peso de 50.

```
pesos = [10, 20, 30]
valores = [60, 100, 120]
capacidad = 50

mochila01(capacidad, pesos, valores)
```



```
↔ Tupla ordenada:
[(30, 120), (20, 100), (10, 60)]
Items tomados:
[(30, 120), (20, 100)]
Valor total:
220
```

Ilustración 3. Instancia del problema con solución óptima.

Ejemplo con solución no óptima

En el ejemplo presentado en la Ilustración 4 podemos observar que el algoritmo voraz no es capaz de encontrar la solución óptima, observamos su solución es tomar el objeto de mayor valor primero, el de 120 con un peso de 40. Sin embargo, la solución óptima esta en tomar los dos objetos de valor 70 y 60, con pesos de 20 y 15 respectivamente, maximizando así el valor a 130.

```
▶ pesos = [15, 20, 40]
  valores = [60, 70, 120]
  capacidad = 50

  mochila01(capacidad, pesos, valores)

⇌ Tupla ordenada:
  [(40, 120), (20, 70), (15, 60)]
  Items tomados:
  [(40, 120)]
  Valor total:
  120
```

Ilustración 4. Instancia del problema con solución no óptima.

3. CONCLUSIONES

Los resultados mostraron que el enfoque voraz es eficiente en términos de tiempo de ejecución y simplicidad, permitiendo obtener soluciones rápidamente en problemas de optimización.

En el problema del cambio, observamos que el algoritmo voraz encuentra la solución óptima cuando las denominaciones de las monedas están bien estructuradas, pero puede fallar al ofrecer una solución subóptima en sistemas de denominaciones particulares. Esto resalta una limitación importante del enfoque voraz: no siempre garantiza la solución óptima.

Por otro lado, en el problema de la mochila 0/1, el enfoque voraz selecciona primero los objetos con mayor valor, pero no considera cómo esta elección afecta a las combinaciones posibles con otros objetos. Si bien en algunos casos logra encontrar la

solución óptima, en otros, la elección de un objeto con mayor valor en una etapa temprana puede llevar a una solución subóptima.

Una observación importante es que los algoritmos voraces funcionan bien cuando las decisiones locales conducen a soluciones globalmente óptimas, pero no siempre son aplicables a todos los problemas. Para casos donde el enfoque voraz falla, sería más adecuado emplear otros métodos, como la programación dinámica, que garantizan la solución óptima en problemas más complejos.

4. BIBLIOGRAFÍA

Greedy algorithm. (s/f). Programiz.com. Recuperado el 21 de septiembre de 2024, de

<https://www.programiz.com/dsa/greedy-algorithm>

Greedy algorithms. (2024, febrero 7). GeeksforGeeks.

<https://www.geeksforgeeks.org/greedy-algorithms/>