Trabajo Práctico **DINÁMICA** Y **GREEDY** – Instituto Politécnico Superior

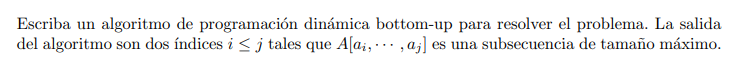
Profesor: *Juan Manuel Rabasedas*

Integrantes: *Luciano Perez e Iván Aquilia*

Dinámica: Ejercicio 7

Texto

Descripción generada automáticamente



Explicación:

Primero definiremos la formulación matemática con una metodología Top-Down:

Defino ||(i,j)|| como:

||(i,j)|| = j - i



(i,j) Si i+r y aj-r

SC(A[ai…aj]) SC(A[ai+1…aj]) Si ||SC(A[ai+1…aj])|| ||SC(A[ai…aj-1])||

SC(A[ai…aj-1]) En otro caso

Lo que pasa aquí es que la función SC recibe un arreglo, si cumple la condición de que la suma de elementos internos es menor igual a ambos elementos extremos o si estamos en un caso de un arreglo de uno o dos elementos, se retornan los índices que delimitan la subsecuencia máxima en el array original.

De lo contrario, el array se recortará en uno de los dos extremos, dependiendo de cual contenga a la subsecuencia máxima dentro de él.

Esto provocará que la función SC se vaya sumergiendo cada vez más profundo en el array a medida que recorta extremos, hasta que sea trivial solucionar el problema, y retorne los índices en cuestión para que estos se vayan desenvolviendo dentro de los llamados recursivos del cálculo entre cual es el máximo largo de secuencia encontrado.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Luego mediante código con metodología Bottom-Up implementamos este algoritmo en Python.

Primero creamos la matriz de listas en la cual vamos a ir guardando los largos en el primer índice de la lista y el valor de la suma interna (sin contar los números en los que estoy parado) en el segundo índice, el uso de una matriz se decidió ya que necesitábamos tener alguna manera de poder acceder a posiciones específicas inmediatas que hayamos calculado anteriormente, además, el uso de arreglos bidimensionales nos permitió movernos de forma favorable dentro del mismo (en nuestro caso en forma de ventanas de valores que van aumentando de tamaño) así de este modo podiamos asegurarnos que anteriormente ya se había calculado un valor para la casilla previa a la que estábamos necesitando. Por lo tanto, obtenemos acceso a subestructuras óptimas las cuales solucionan el problema al fin de cuentas de forma óptima global, junto con el uso de problemas solapados que en nuestro caso es la necesidad de saber si la suma interna cumple con la condición del ejercicio. Esto nos reduce el costo considerablemente a menos de O(n2) ya que usamos menos de la mitad de la matriz, siendo que originalmente el problema por fuerza bruta rondaba el O(n x 2n).

Esta matriz va a poseer una fila y una columna extra repleta de [0,0] para poder comparar luego con las celdas de los límites de la matriz. El resto de las filas y columnas corresponden a el arreglo y se llenan originalmente de [0,0], exceptuando los casos triviales de longitud 1 y 2, los cuales van a ser los responsables de que esta metodología Bottom-Up sea posible y eficiente. Respectivamente estos se llenan de [1,0], y de [2,0].

El índice de la fila en la matriz representa donde comienza la subsecuencia y el de la columna donde termina.

Implementación en Python:

*#Función que calcula el largo máximo entre 2 elementos del array y retorna dicho elemento.*

**def** maxdevuelvelist(lista, lista2):

**if** lista[0] >= lista2[0]:

**return** lista

**return** lista2

*#Input del programa.*

entrada = input("Por favor, ingresa una secuencia de números separados por espacios: ")

A = [int(num) **for** num **in** entrada.split()]

i = 1

j = len(A)

*# Género la matriz de listas repleta de [0,0] añadiendo 1 columna y fila más para poder realizar comparaciones de subproblemas luego.*

matriz = [[list([0, 0]) **for** \_ **in** range(j + 1)] **for** \_ **in** range(j + 1)]

*# Asumo y calculo los subproblemas triviales de forma lineal. Calculo los largos y el valor de la suma interna de los problemas base.*

**for** k **in** range(i, len(A) + 1):

**for** p **in** range (0, len(A) + 1):

**if** p==0:

matriz[k][p][0] = 0

**elif** k == p:

matriz[k][p][0] = 1

**elif** k == p - 1 **or** p == k - 1:

matriz[k][p][0] = 2

*#Declaro un mayor muy chico y mi respuesta final de índices.*

mayor = -1

respuesta = ()

*# Evito calcular nuevamente los problemas ya calculados usando ventanas a partir de tamaño 3 y extiendo la longitud*

*# y actualizo el valor de su suma interna de la secuencia anterior si se cumple la condición que se calcula*

*# basada en una memoizacion de los casos anteriores, tengo en cuenta también los casos de longitud 3 ya que*

*# estos son tratados de manera distinta debido a su longitud de suma interna igual a 0 (no tienen elementos entre medio).*

**for** tamaño **in** range(3, j + 1):

**for** k **in** range(1, j - tamaño + 2):

p = k + tamaño - 1

**if** tamaño != 3:

**if** matriz[k+1][p-1][1] + A[k] + A[p-2] <= A[k-1] **and** matriz[k+1][p-1][1] + A[k] + A[p-2] <= A[p-1]:

matriz[k][p][0] = matriz[k+1][p-1][0] + 2

matriz[k][p][1] = matriz[k+1][p-1][1] + A[k] + A[p-2]

**else**:

listamaxima = maxdevuelvelist(matriz[k][p-1], matriz[k+1][p])

matriz[k][p][0] = listamaxima[0]

matriz[k][p][1] = listamaxima[1]

**else**:

**if** A[k] <= A[k-1] **and** A[k] <= A[p-1]:

matriz[k][p][0] = matriz[k+1][p-1][0] + 2

matriz[k][p][1] = A[k]

**else**:

listamaxima = maxdevuelvelist(matriz[k][p-1], matriz[k+1][p])

matriz[k][p][0] = listamaxima[0]

matriz[k][p][1] = listamaxima[1]

**if** matriz[k][p][0] > mayor:

mayor = matriz[k][p][0]

**if** mayor > 2:

respuesta = (k-1, p-1)

**else**:

respuesta = (k-1, p-2)

*#Output del problema.*

**print**(f"Los índices que marcan los extremos de la subsecuencia más larga que cumple con las condiciones son: {respuesta}")

Greedy: Ejercicio 5

Texto

Descripción generada automáticamente

Explicación:

En este caso no se necesita formulación matemática recursiva por lo que simplemente pasaremos a explicar el funcionamiento Greedy y fundamentación del código implementado nuevamente en Python.

La base de nuestro algoritmo se basa en una decisión Greedy bastante obvia y simple para el problema que se nos plantea, el cuál es encontrar un matching máximo dentro de un grafo cualquiera. Esto lo logramos con una metodología sencilla, agarrar siempre la arista de mayor peso disponible para luego eliminarla junto con sus vértices de las opciones válidas para tomar en la siguiente iteración, así, hasta que no queden más aristas para tomar y se llegue al final del algoritmo. Esto provee una solución óptima en la mayoría de los casos que se presenten, ya que es la idea más primitiva de un matching máximo, quedarse siempre con la mayor disponibilidad pase lo que pase, y así ir construyendo de manera local una solución global.

De todos modos, hay casos vagos en los que este algoritmo no es del todo óptimo, ya que es superado por otros los cuales no son Greedy, o, si lo son, estos cumplen en el caso que el nuestro no cumple, pero viceversa. ¿A qué se debe esto y por qué no tiene una solución óptima? es porque el caso del problema de un matching de grafos máximo (al igual que el caso similar del coloreo de un grafo) son problemas NP (Non-deterministic Polynomial time), lo cual significa que no existe ningún algoritmo en las matemáticas que pueda resolverlo de forma óptima el 100% de las veces, pero puede aproximarse con metodologías (por ejemplo, Greedy) y llegar a un caso muy cercano al óptimo en tiempos polinomiales, ya que si se intentara resolver de manera global y abarcativa debería utilizarse una búsqueda de patrones exhaustivos y pesados los cuales pueden llegar a tiempos exponencialmente gigantescos en los peores casos. De este modo, nuestro algoritmo que siempre elige la arista máxima y la agrega al matching es una solución muy buena pero no perfecta debido a esta característica.

Se podría modificar el algoritmo para que en vez de tomar siempre la mayor, tome aristas en base a una relación costo-beneficio basado en una métrica calculada por qué tantos vértices dejaría inutilizables el hecho de tomar esa arista contra el peso que agrega de forma proporcional al matching final, así de este modo acercarse a un resultado un poco mas optimo en “teoría”, pero esto tiene algunos problemas. Por ejemplo, el hecho de estar tomando decisiones en base a una métrica que depende de la comparación de variables externas al entorno local en cada iteración (como una métrica en base a las demás posibles opciones) dejaría de hacer que el algoritmo sea pura y netamente Greedy, ya que perdería la característica de obviar por completo los aspectos generales y simplemente tomar una decisión basada puramente en la situación local sin tener en cuenta el futuro ni el pasado dentro del algoritmo. Además, otro ejemplo de problema de hacer esto es que si implementamos de esta manera la búsqueda del matching, este nuevo algoritmo seria mas optimo en ciertos casos pero dejaria de serlo en otros casos en los que el original planteado por nosotros seguiría siendo una mejor opción la cual otorgaría el matching máximo el 100% de las veces.

Por lo que se ve que es una situación en la que se elaboró un algoritmo lo más neutro y óptimamente balanceado teniendo en cuenta la característica del problema.

A continuación se deja un caso en el que nuestro algoritmo no encuentra la solución óptima, ya que esta estaría dada por un algoritmo no Greedy.

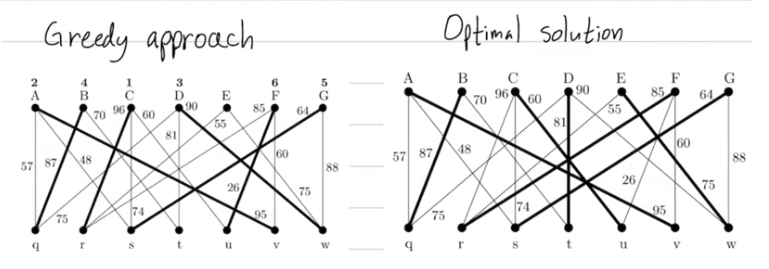


Imagen sacada de internet sobre un caso en el que los algoritmos Greedys de matching de grafos son vencidos (en algunos casos muy concretos) por otros algoritmos no Greedys que toman decisiones más parametrizadas y específicas.

Implementación en Python:

*# Descarta las aristas que estén conformadas por vertices ya saturados/ocupados*

**def** descartarAristas(a, b, grafo):

nuevo\_grafo = []

**for** tupla **in** grafo:

**if** tupla[0] != a **and** tupla[0] != b **and** tupla[2] != a **and** tupla[2] != b:

nuevo\_grafo.append(tupla)

**return** nuevo\_grafo

*# Input del programa*

Grafo = [("q",57,"A"),("q",87,"B"),("q",75,"D"),("r",96,"C"),("r",55,"E"),("r",85,"F"),("s",48,"A"),("s",74,"C"),("s",64,"G"),("t",70,"B"),("t",81,"D"),("u",60,"C"),("u",26,"F"),("v",95,"A"),("v",60,"F"),("w",90,"D"),("w",75,"E"),("w",88,"G")]

*# Declaracion del conjunto de arcos con mayor peso total*

Matching = []

*# Declaración del arreglo del que elegiremos las aristas para el matching*

AristasDisponibles = Grafo

*# Mientras haya aristas para añadir al matching:*

**while** len(AristasDisponibles) != 0:

mayor = None

*# Recorremos todas las aristas buscando la de mayor peso*

**for** tupla **in** AristasDisponibles:

**if** mayor **is** None **or** tupla[1] > mayor[1]:

mayor = tupla

*# Tras recorrer todo el array añadimos la arista de mayor peso al matching*

Matching.append(mayor)

*# Descartamos las aristas que tengan uno de sus vértices ya ocupados por el último agregado al matching*

AristasDisponibles = descartarAristas(mayor[0], mayor[2], AristasDisponibles)

**print**(Matching)