Instituto Politécnico Superior
Rosario, Argentina

Algoritmos y Estructuras de Datos avanzadas

2019

6to Informática

Trabajo Práctico

Programación Dinámica y Algoritmos Greedy

Antúnez Joaquín
Quintero Iago

Docente: Juan Manuel Rabasedas

Problema Greedy

Enunciado del problema:

En el campo de la teoría de grafos un matching o conjunto independiente de arcos en un grafo es un conjunto de arcos sin vértices en común. En un grafo con pesos, un matching de peso máximo es aquel que maximiza el peso total de sus arcos.

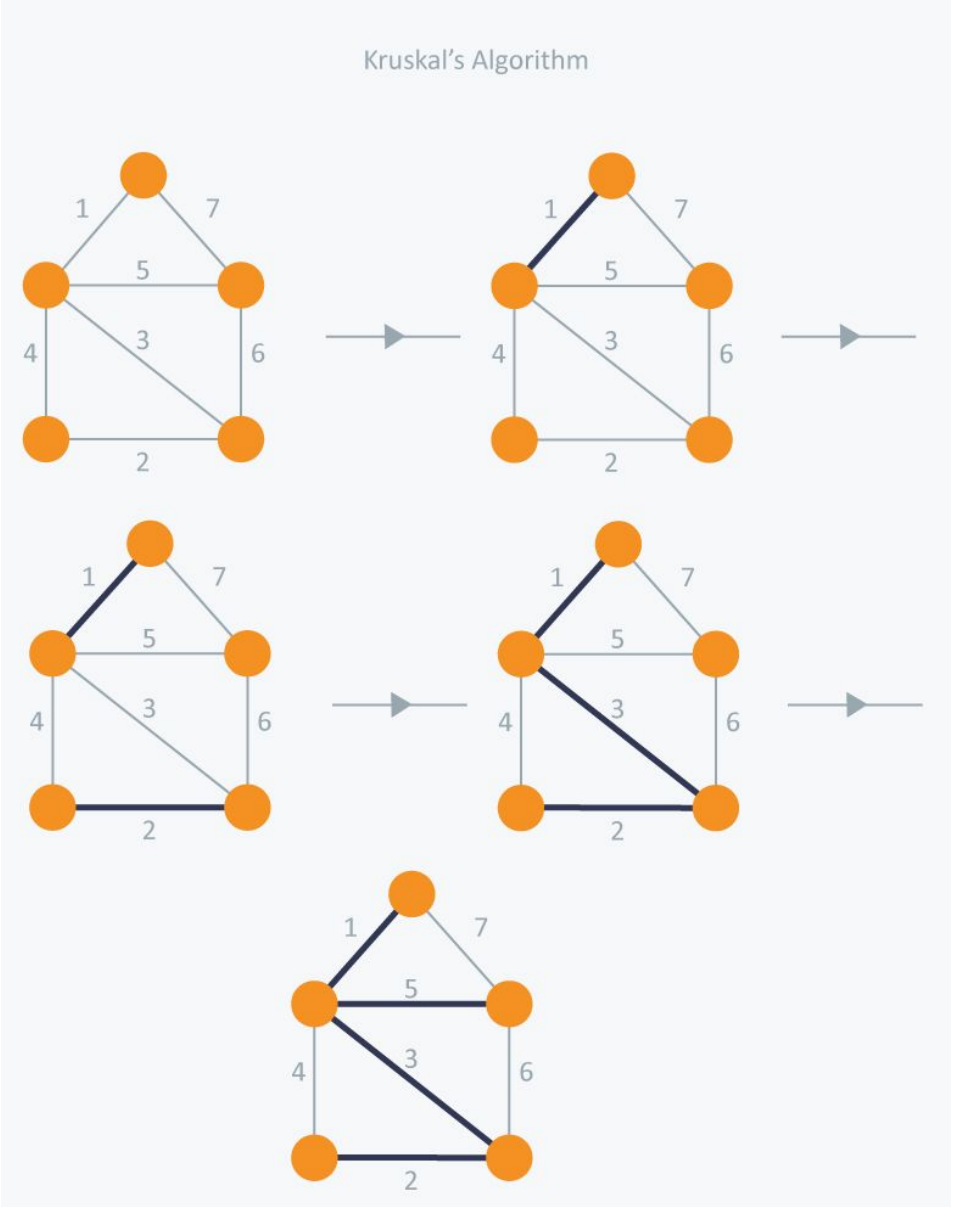
a) Describir un algoritmo greedy que dado un grafo (E, V) intente calcular un matching de peso máximo.
b) ¿Es óptimo el algoritmo anterior? En caso afirmativo, dé una prueba de ello, razonando por el absurdo. En caso negativo, dé un contraejemplo.

Resolución:

Previo a la explicación de cómo resolver el problema del matching de peso máximo, es importante entender cómo funciona el algoritmo de Kruskal.

Kruskal es un algoritmo de la teoría de grafos para encontrar un árbol recubridor mínimo en un grafo conexo y ponderado. Es decir, busca un subconjunto de aristas que, formando un árbol, incluyen todos los vértices y donde el valor de la suma de todas las aristas del árbol es el mínimo. Para esto el algoritmo va eligiendo las aristas en orden de menor a mayor peso y solo las añade al grafo resultante, de menor a mayor peso, si al agregarla no se forma un ciclo, si se formara un ciclo se descarta y se continúa con la siguiente arista.

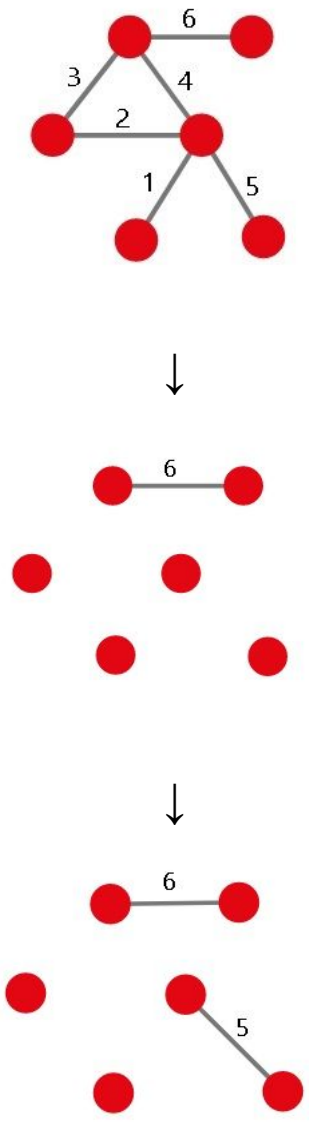
Ejemplo de cómo trabaja Kruskal:



Como se ve en la imágen, del paso 3 al paso 4 la arista de menor peso es la de peso 4, pero como al agregarla al grafo resultante esto da lugar a que se genere un ciclo, la pasamos por alto y continuamos con la siguiente. Lo mismo pasa luego con las aristas de peso 6 y peso 7.

Para resolver el problema y encontrar dado un grafo ponderado su matching de peso máximo debemos implementar para ello un algoritmo que trabaja similarmente a Kruskal. En vez de ordenar las aristas de menor a mayor e iniciar por la arista de menor peso, ordenaremos las aristas de mayor a menor e iniciaremos por la arista de mayor peso, y no analizaremos al momento de agregar una arista si es que se forma un ciclo, sino que debemos revisar que las aristas no posean vértices en común.

Ejemplo del funcionamiento del algoritmo:



Matching de 11.

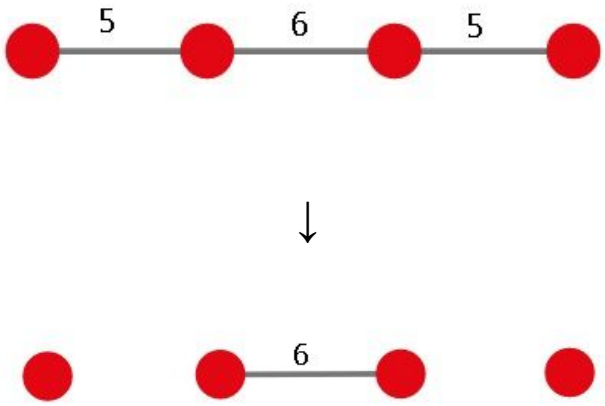
La arista siguiente es la arista de peso 4, pero como esta comparte vértice con la arista de peso 5 la cual ya está en el grafo resultante la omito y continúo con la siguiente. Lo mismo va a pasar en este caso con todas las aristas restantes por lo que ya obtuve el matching de peso máximo del grafo dado.

Implementación del algoritmo en Haskell:

```
mergeSort []          = []
mergeSort [a]         = [a]
mergeSort a           = merge (mergeSort firstFew) (mergeSort lastFew)
                where firstFew = take ((length a) `div` 2) a
                      lastFew = drop ((length a) `div` 2) a


merge a []            = a
merge [] b            = b
merge ((a,b,c):as) ((x,y,z):bs)
  | c > z     = (a,b,c):(merge as ((x,y,z):bs))
  | otherwise = (x,y,z):(merge ((a,b,c):as) bs)


match [] v            = []
match ((a,b,w):es) v  = if ((elem a v) || (elem b v))
                           then match es v
                           else (a,b,w):(match es ([a,b] ++ v))

matching e            = match (mergeSort e) []
```

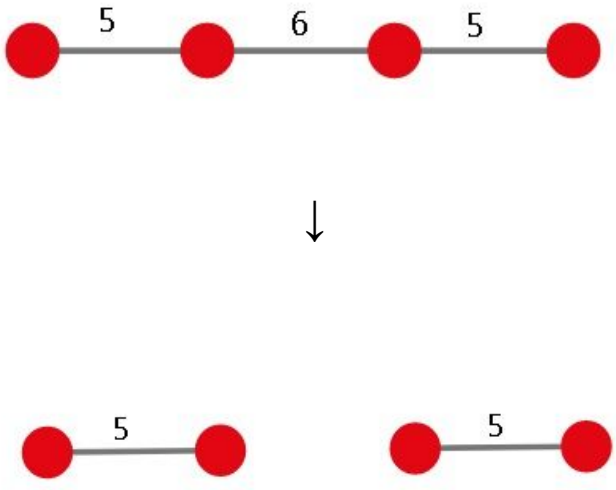
No óptimo:

En el siguiente ejemplo podemos comprobar que el algoritmo greedy *NO* es óptimo.
Dado el siguiente grafo y aplicando el algoritmo recién explicado obtenemos:



Obtenemos un matching de 6.

Con este contraejemplo queda comprobado que el algoritmo greedy descrito no es óptimo pues se puede obtener un matching mayor (10) del mismo grafo inicial.



Problema de Dinámica

Enunciado del problema:

Algunas normas tipográficas establecen que las líneas deben contener entre 40 y 60 caracteres. Supongamos que no queremos partir las palabras y que conocemos los tamaños (incluidos los signos de puntuación y espacios en blanco adyacentes) s1, s1, ...sN de las N palabras de un párrafo de M líneas. Si el objetivo es minimizar la longitud de la línea más larga del párrafo, escribe un algoritmo de programación dinámica que minimice la longitud de la línea más larga de un párrafo.

Para esto:
a) completa la definición de la función *f*(n, m) que devuelva la longitud mínima de la línea más larga cuando las n primeras palabras se distribuyen en m líneas.

$$f(n, m) = \begin{cases} 0 & n = 0 \\ \Sigma_{n=1}^{N} sn & m = 1 \\ \ldots\ldots\ldots & \text{en otro caso} \end{cases}$$

b) Utiliza programación dinámica para implementar eficientemente la función anterior.

Resolución:

Sea n la cantidad de palabras
Sea m la cantidad de líneas

Casos:

- n = 0

  Si n = 0, es decir, la cantidad de palabras es igual a 0, aplico la función *f* a un n=0 y a un m=x, sea cual sea la cantidad de líneas, mi resultado va a ser 0 pues al no tener nada que distribuir la longitud mínima de la línea más larga es 0.

  $f(0,m) = 0$

- m = 1
  Si m = 1, es decir, la cantidad de líneas es igual a 1, voy a tener que colocar todas las palabras en 1 línea, la longitud mínima de la línea más larga será la sumatoria de todas las palabras. Recordemos que las palabras se identifican con un número entero que significa su largo.

  $f(n,1) = \Sigma_{i=1}^{n} s_i$

- Casos restantes

Suponemos que *k* palabras forman la línea *m*, la línea más larga podría ser *m* o alguna de las anteriores

- En el caso de *m* la longitud está dada por

  $\Sigma^{n}_{i=n-k+1} s_i$

  siendo *n - k + 1* la primera palabra que va en el renglón actual

- En el caso de *m - 1* la longitud máxima está dada por aplicar la función *f* pero tomando como parámetro lo siguiente

  f(n-k, m-1)

  m - 1 pues es la línea anterior
  n - k pues es la cantidad de palabras total - la cantidad de palabras de la línea siguiente (*m*)

Dado que no podemos asegurar que el valor de *k* sea el óptimo:

$$f(n,m) = \min_{k=1..n} \max\{ f(n-k, m-1), \Sigma^{n}_{i=n-k+1} s_i \}$$

Es decir, *el mínimo valor del resultado* de:

probar con k = 1 hasta k = n en:

obtener el *máximo* entre:

- la línea anterior con *cantidad de palabras n - k*
- la sumatoria de las palabras que van desde *n - k + 1* hasta n

Implementación del algoritmo en Haskell:

```
import Data.Matrix

largoAux p f [] d          = 0
largoAux p f (s:ss) d      =  if (d >= p && d <= f)
                    then s + (largoAux p f ss (d+1))
                    else (largoAux p f ss (d+1))

pals = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]

len                        = length pals + 1
ren                        = 5
initial                    = zero ren len

largo p f                  = largoAux p f pals 1

setUnRenglon 1 mat         = setElem (largo 1 1) (1,2) mat
setUnRenglon n mat         = setUnRenglon (n-1) (setElem (largo 1 (n-1)) (1,
n) mat)
first                      = setUnRenglon len initial

kcalc 1 m n mat                 = min (1000) (max (getElem (m-1) (n) mat)
(largo n n))
kcalc k m n mat                 = min (kcalc (k-1) m n mat) (max (getElem
(m-1) (n-k+1) mat) (largo (n-k+1) n))

rencalc 2 n mat                 = setElem (kcalc n 2 (n) (mat)) (2,(n+1))
(mat)
rencalc m n mat            = setElem (kcalc n m (n) (rencalc (m-1) n mat))
(m,(n+1)) ((rencalc (m-1) n mat))

wcalc 1 m mat              = rencalc m 1 mat
wcalc n m mat              = rencalc m (n-1) (wcalc (n-1) m mat)


-- wcalc len ren first
```