Partizionamento di alberi in linguaggio Python

Corso Ottimizzazione Combinatoria (IN440)

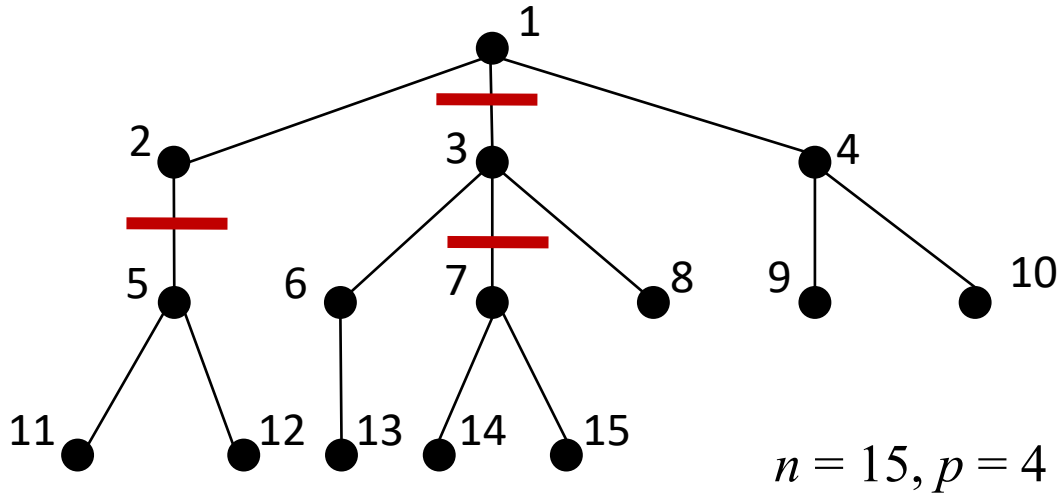
Prof. M. Liverani

Partizionamento di alberi

- Dato un albero T e un intero $k \leq n$ per produrre un partizionamento di T in k sotto-alberi, occorre rimuovere $k-1$ spigoli
- Assegnati dei pesi non negativi ai vertici di T, vogliamo trovare la p-partizione di T che ottimizzi una funzione obiettivo $f(\pi_p)$ il cui valore dipende dal peso delle componenti della partizione π_p
- Si indica con $\mu = (w_1 + w_2 + \ldots + w_n) / k$ il peso medio delle componenti della *p-partizione*
- Algoritmo:
  - ricerca esaustiva della partizione ottima: si generano tutte le p-partizioni di T
  - per ognuna si valuta il valore della funzione obiettivo e si memorizza la partizione corrente se migliora il valore di $f(\pi_p)$, quindi si passa alla partizione «successiva»
  - alla fine del procedimento si visualizza la partizione migliore, quella che ottimizza f al variare di π_p
- Per produrre tutte le partizioni di T occorre generare tutti i sottoinsiemi di $E(T)$ con $k-1$ elementi

Partizionamento di alberi

- Per semplificare l'algoritmo, visto che T è un albero e quindi ogni vertice v di T ha un solo spigolo entrante (u, v), ad eccezione della radice s che non ha alcuno spigolo entrante, possiamo indicare con v ogni spigolo $(u, v) \in E$
- Se $s = 1$ è la radice di T allora l'insieme degli spigoli possiamo indicarlo con $\{2, 3, \ldots, n\}$
- Indichiamo con c_i il vertice radice del sottoalbero che rappresenta la componente i-esima della partizione per $i = 0, 1, \ldots, p - 1$
  - rispetto all'esempio in figura abbiamo:
    $c_0 = 1, c_1 = 5, c_2 = 3, c_3 = 7$



Partizionamento di alberi pesati sui vertici

- Supponiamo di aver assegnato dei pesi interi positivi ai vertici dell'albero: $w_1, \ldots, w_n$
- Il peso di ogni componente della partizione è dato dalla somma dei pesi dei vertici che compongono la componente stessa

$$W_i = \sum w_k \quad \text{per } i = 0, 1, \ldots, p-1$$

- Per calcolare il peso W_i della componente della partizione dell'albero con radice in c_i occorre eseguire una visita del sotto-albero con radice in c_i sommando il peso dei vertici incontrati se questi non sono la radice di altre componenti della partizione

Algoritmo 3 PESOCOMPONENTE(T, i, c, w)

Input: l'albero T, il vertice radice del sotto-albero i, i vertici radice delle componenti della partizione $c_0, \ldots, c_{p-1}$, i pesi w assegnati ai vertici del grafo

Output: il peso della componente con radice in i

```
1:  sia u il vertice i-esimo dell'albero T
2:  sia peso = w_i
3:  sia Q una coda, accoda u in Q
4:  fintanto che Q non è vuota ripeti
5:      sia u il vertice estratto da Q
6:      per v adiacente a u ripeti
7:          se v ≠ c_0, ..., c_{p-1} allora
8:              peso = peso + w_v
9:              accoda v in Q
10:         fine-condizione
11:     fine-ciclo
12: fine-ciclo
13: return peso
```

Calcolo esaustivo della partizione ottima

- Per calcolare in modo esaustivo la partizione ottima per una determinata funzione obiettivo $f(\pi_p)$ occorre:
  1. generare la prima combinazione $c_0, c_1, ..., c_{p-1}$ dell'insieme $\{1, 2, ..., n\}$ ponendo $c_i = i + 1$, per $i = 0, 1, ..., p - 1$
  2. calcolare il primo valore della funzione obiettivo f per tale partizione π_p; tale valore è inizialmente considerato come il valore ottimo f^* per la funzione obiettivo
  3. generare una combinazione $c_0, c_1, ..., c_{p-1}$ dell'insieme $\{1, 2, ..., n\}$ mantenendo fisso $c_0 = 1$; ogni combinazione rappresenta l'insieme di vertici che sono la radice dei sottoalberi della partizione di T (il vertice $v = 1$ radice dell'albero T è sempre la radice di una delle componenti, per questo $c_0 = 1$ per ogni combinazione)
  4. calcolare il valore della funzione obiettivo $f(\pi_p)$ per la partizione πp e confrontarlo con il valore f^* fino ad ora considerato ottimo: se π_p migliora il valore di f si memorizza questo nuovo valore di f come valore ottimo
  5. si torna al passo 3 fino a quando non sono state generate tutte le combinazioni di $\{1, 2, ..., n\}$ in classi di p elementi (mantenendo il primo fisso: $c_0 = 1$)
  6. si visualizza in output il valore ottimo f^* per la funzione obiettivo e la partizione che ha consentito di raggiungere tale valore

Calcolo delle combinazioni dei vertici-radice

- Per calcolare le combinazioni dei vertici radice dei sotto-alberi che costituiscono le componenti delle partizioni di T possiamo adattare un algoritmo standard per il calcolo delle combinazioni di p elementi scelti nell'insieme $\{1, 2, ..., n\}$
- In particolare, siccome dobbiamo fissare per ogni combinazione $c_0 = 1$, calcoleremo le combinazioni di $p - 1$ elementi nell'insieme $\{2, 3, ..., n\}$

Algoritmo 4 COMBINAZIONI(T, p, w)

Input: l'albero T con n vertici e il numero p di componenti della partizione

Output: tutte le combinazioni $c_0, \ldots, c_{p-1}$ di $V(T)$ in classi di p elementi, con $c_0 = 1$

1: sia $i = p - 1$
2: **fintanto che** $i \geq 1$ **ripeti**
3: $i = p - 1$
4: **fintanto che** $i \geq 1$ e $c_i = n - (p - i) + 1$ **ripeti**
5: $i = i - 1$
6: **fine-ciclo**
7: **se** $i \geq 1$ **allora**
8: $c_i = c_i + 1$
9: **per** $j = i + 1, \ldots, p - 1$ **ripeti**
10: $c_j = c_{j-1} + 1$
11: **fine-ciclo**
12: **fine-condizione**
13: **per** $j = 0, \ldots, p - 1$ **ripeti**
14: $W_j =$ PESOCOMPONENTE(T, c_j, c, w)
15: **fine-ciclo**
16: valuta la funzione obiettivo f per la partizione di T definita da $c_0, c_1, \ldots, c_{p-1}$: se migliora il valore di f memorizza la partizione c e il valore f^* della funzione
17: **fine-ciclo**

Mettiamo insieme tutti i pezzi

```
from in440 import *

def f(W):
  return(max(W)-min(W))

def pesoComponente(T, i, c, w):
  u = T.getVertex(i)
  peso = w[i]
  Q = Queue()
  Q.enqueue(u)
  while not Q.isEmpty():
    u = Q.dequeue()
    for v in u.getConnections():
      if v.getId() not in c:
        peso = peso + w[v.getId()]
        Q.enqueue(v)
  return(peso)

T = Graph()
n = int(input("N. vertici: "))
k = int(input("N. max figli: "))
p = int(input("N. componenti: "))
randomTree(T, n, k)
printGraph(T)
```

```
w = np.zeros((n+1), dtype=int)
s = 0
for i in range(1, n+1):
  w[i] = randint(1, 100)
  s = s + w[i]
mu = s/p

print("W: ", w)
print("Media:", mu)

c = [i+1 for i in range(p)]
W = [0 for i in range(p)]
for i in range(p):
  W[i] = pesoComponente(T, c[i], c, w)

Fstar = f(W)
cmin = [c[i] for i in range(p)]
Wmin = [W[i] for i in range(p)]
cont = 1
```

```
i = p-1
while i >= 1:
  cont = cont + 1
  i = p-1
  while i>=1 and c[i] == n-(p-i)+1:
    i = i-1
  if i>=1:
    c[i] = c[i]+1
    for j in range(i+1, p):
      c[j] = c[j-1]+1
  for j in range(p):
    W[j] = pesoComponente(T,c[j],c,w)
  if f(W) < Fstar:
    Fstar = f(W)
    cmin1 = [c[i] for i in range(p)]
    Wmin1 = [W[i] for i in range(p)]

print("Valutate", cont, "partizioni")
print(" f*:   ", Fstar)
print(" tagli:", cmin)
print(" pesi: ", Wmin)
treePlot(T, 1)
```