

Esercitazione 14 novembre 2023

1. Max Rosso

Input: un albero binario T con n nodi, ogni nodo ha: - $val(v) > 0$ - $col(v) \in \{R, N\}$

Output: valore del cammino rosso di tipo radice-nodo di valore massimo

DEF: il valore di un cammino è la somma di valori dei nodi del cammino **DEF:** un cammino è rosso se tutti i suoi nodi sono di colore rosso

L'algoritmo **MaxRosso**(v) restituisce il valore del cammino rosso di valore massimo di tipo v - discendente di v . Le informazioni che vengono *dal basso*, calcolate rispetto al sottoalbero con radice v , possono essere usate per *passare informazioni* al padre di v .

Pseudocodice

```
MaxRosso(nodo v)
    if(v == NULL) then return 0
    if(col(v) == 'N') then return 0
    return 1 + max{MaxRosso(sx(v)), MaxRosso(dx(v))}
```

Complessità Temporale: $T(n) = O(n)$

Implementazione in Python usando networkx

```
def MaxRosso(tree, node, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # così la dfs verrà effettuata solo sui figli sinistri e destri

    if visited is None:
        visited = set()

    if node is None or node in visited: # se il nodo attuale è None oppure è già stato visitato ritorna 0
        return 0

    visited.add(node)
    v = tree.nodes[node]
    col = v['colore']

    if col == 'N':
        return 0

    vicini = list(tree.neighbors(node))

    # Ricorsivamente calcola il valore massimo del cammino rosso per ciascun vicino (figlio)
    max_vicini = [MaxRosso(tree, vicino, visited) for vicino in vicini]

    return val + max(max_vicini)
```

2. Conta Profondità

Input: un albero binario T con n nodi e un intero $h \geq 0$.

Output: numero di nodi di T con profondità almeno h

DEF: la profondità di un nodo è la distanza (numero di archi) dalla radice

L'algoritmo **ContaProf**(v , h , i) restituisce il numero di nodo nel sottoalbero radicato in v che hanno profondità $\geq h$ (**info dal basso**), assumendo che v ha profondità i (**info dall'alto**). Quindi dall'alto, mediante i tengo conto della profondità del nodo e dal basso restituisco il numero di nodi che hanno profondità almeno h .

Pseudocodice

```
ContaProf(nodo v, h, i)
    if(v == NULL) then return 0
    if(i >= h) then return 1 + ContaProf(sx(v), h, i + 1) + ContaProf(dx(v), h, i + 1)
    else return ContaProf(sx(v), h, i + 1) + ContaProf(dx(v), h, i + 1)
```

Complessità Temporale: $T(n) = O(n)$

Implementazione in Python usando networkx

```
def ContaProf(tree, node, height, ih, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # così la dfs verrà effettuata solo sui figli sinistri e destri

    if visited is None:
        visited = set()

    if node is None or node in visited: ## se il nodo attuale è None oppure è già stato visitato
        return 0

    visited.add(node)
    v = tree.nodes[node]

    vicini = list(tree.neighbors(node))

    # Ricorsivamente conta i nodi che hanno profondità almeno h per ciascun vicino (figlio)
    if ih >= height:
        return sum([ContaProf(tree, vicino, height, ih + 1, visited) for vicino in vicini]) + 1
    else:
        return sum([ContaProf(tree, vicino, height, ih + 1, visited) for vicino in vicini])
```

3. Bilanciati

Input: un albero binario T di n nodi, ogni nodo v ha un valore $\text{val}(v) > 0$

Output: numero di nodi che soddisfano: somma dei valori degli antenati del nodo = somma dei valori dei discendenti del nodo (Δ)

L'algoritmo Bilanciati(v , SA) restituisce (SD, k), dove - SD: somma il valore dei discendenti di v (v incluso) (**info dal basso**). - k: numero nodi nel sottoalbero radicato in v che soddisfano Δ (**info dal basso**). - SA: somma il valore dei antenati di v (v incluso) (**info dall'alto**).

Pseudocodice

```
Bilanciati(nodo v, SA)
    if(v == NULL) then return 0
    SA = SA + val(v)
    (SD_sx, k_sx) = Bilanciati(sx(v), SA)
    (SD_dx, k_dx) = Bilanciati(dx(v), SA)
    SD = SD_sx + SD_dx
    k = k_sx + k_dx
    if(SA == SD) then return (SD, 1 + k)
    else return (SD, k)
```

```
CalcBilanciati(nodo r)
    (SD, k) = Bilanciati(r, 0)
    return k
```

Complessità Temporale: $T(n) = O(n)$

Implementazione in Python usando networkx

```
def Bilanciati(tree, node, sum_antenati, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # così la dfs verrà effettuata solo sui figli sinistri e destri

    if visited is None:
        visited = set()

    if node is None or node in visited: ## se il nodo attuale è None oppure è già stato visitato
        return (0, 0)

    visited.add(node)
    v = tree.nodes[node]
    val = node['value']

    sum_antenati += val

    vicini = list(tree.neighbors(node))

    # Ricorsivamente calcolo la somma dei discendenti e il numero di nodi che soddisfano delta,
    # per ciascun vicino (figlio)
    result = [Bilanciati(tree, vicino, sum_antenati, visited) for vicino in vicini]

    if len(result) == 3: # nella posizione 0, se result è lungo 3, trovo (0, 0), restituito dal caso base
        sx = result[1]
        dx = result[2]
    else:
        sx = result[0]
        dx = result[1] if len(result) > 1 else (0, 0)

    sum_disc = sx[0] + dx[0] + val # sx[0] = SD_sx - dx[0] = SD_dx

    k = sx[1] + dx[1] # sx[1] = k_sx - dx[1] = k_dx
    if sum_antenati == sum_disc:
        return (sum_disc, k + 1)
    else:
        return (sum_disc, k)

def CalcBilanciati(tree, _node):
    sum_disc, k = Bilanciati(tree, node, 0)
    return k
```