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 massito di topo v - discendente di v. Le informazione 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))}
```

Complesstià Temporale: $T(n) = O(n)$

Implementazione in Python usando networkx

```python
def MaxRosso(tree, node, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # cosi la dfs verrà effettuta 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 gia 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)
```

Complesstià Temporale: $T(n) = O(n)$

Implementazione in Python usando networkx

```python
def ContaProf(tree, node, height, ih, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # cosi la dfs verrà effettuta 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 gia stato visitato
        return 0

    visited.add(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 `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 return (SD, k)
```

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

Complesstià Temporale: $T(n) = O(n)$

Implementazione in Python usando networkx

```python
def Bilanciati(tree, node, sum_antenati, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # cosi la dfs verrà effettuta 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 gia stato visitato
        return (0, 0)

    visited.add(node)
    v = tree.nodes[node]
    val = v['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), restiuito 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
```

4. Generazionalmente Profondo

Input: un albero binario T con n nodi, dove ciascun nodo ha `val(v)`> 0.

Output: il numero di nodi generazionalmente profondi di T.

DEF:

- La *profondità* di un nodo v è il numero di archi del cammino dal v alla radice. I nodi che incontrano lungo tale cammino (v) compresi sono detti *antenati* di v.

- Un nodo v è *generazionalmente profondo* se la sua profondità è strettamente maggiore del valore di un suo antenato di valore minimo

L'idea è quella di tenere traccia dall'alto del minimo valore degli antenati e della profondità, mentre dal basso restituire il numero di nodi che soddisfano la condizione.

Pseudocodice

```
GenProf(nodo v, h, min)
    if(v == NULL) then return 0
    if(val(v) < min) then min = val

    left = GenProf(sx(v), h + 1, min)
    right = GenProf(sx(v), h + 1, min)

    if(h > min) then return 1 + left + right
    else return left + right
```

Complesstià Temporale: $T(n) = O(n)$

Implementazione in Python usando networkx

```python
def GenProf(tree, node, h, min_val, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # cosi la dfs verrà effettuta 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 gia stato visitato
        return 0

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

    min_val = min(min_val, val)

    vicini = list(tree.neighbors(node))

    if h > min_val:
        return sum([GenProf(tree, vicino, h + 1, min_val, visited) for vicino in vicini]) + 1
    else:
        return sum([GenProf(tree, vicino, h + 1, min_val, visited) for vicino in vicini])

def CalcGenProf(tree, node):
    root = tree.nodes[node]
    return GenProf(tree, node, 0, root['value'])
```

5. Minima Profondità

Input: un albero binario T con n nodi.

Output: il nodo v di profondità minima la cui profondità è maggiore o uguale al numero dei suoi discendenti.

DEF

- La *profondità* di un nodo v è il numero di archi del cammino da v alla radice.
- Un nodo u è un *discendente* di v se u si trova nel sottoalbero T radicato in v.

L'idea è quella di tenere traccia dall'alto della profondità, e dal basso tenere traccia del numero di discendenti del nodo con profondità minima e la sua profondità.

Pseudocodice

```
MinProf(nodo v, h)
    if(v == NULL) then return (0, NULL, 0)

    (disc_sx, nodo_sx, h_sx) = MinProf(sx(v), h + 1)
    (disc_dx, nodo_dx, h_dx) = MinProf(dx(v), h + 1)

    disc = disc_sx + disc_dx + 1

    if(h >= disc) then return (disc, v, h)
    else if(h_dx > h_sx) then return (disc, nodo_sx, h_sx)
    else return (disc, nodo_dx, h_dx)
```

Complesstià Temporale: $T(n) = O(n)$

Implementazione in Python con networkx

```python
def MinProf(tree, node, h, visited=None):
    # Mediante l'utilizzo di un set, tengo conto dei nodi visitati, principalmente dei padri,
    # cosi la dfs verrà effettuta 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 gia stato visitato
        return (0, 0, h + 1)

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

    vicini = list(tree.neighbors(node))

    result = [MinProf(tree, vicino, h + 1, visited) for vicino in vicini]

    if len(result) == 3:
        sx = result[1]
        dx = result[2]
    else:
        sx = result[0]
        dx = result[1] if len(result) > 1 else (0, 0, h + 1)

    disc = sx[0] + dx[0] + 1
    nodo_sx = sx[1]
    nodo_dx = dx[1]
    h_sx = sx[2]
    h_dx = dx[2]

    if h >= disc:
        return (disc, current_node, h)
    elif h_sx < h_dx:
        return (disc, nodo_sx, h_sx)
    else:
        return (disc, nodo_dx, h_dx)
```

```
def CalcMinNodoProf(tree, node):
    disc, nodo, h =  MinProf(tree, node, 0)
    return nodo
```