

Algoritmi e Strutture Dati

a.a. 2022/23

Compito del 11/9/2023

Cognome: _____

Nome: _____

Matricola: _____

E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

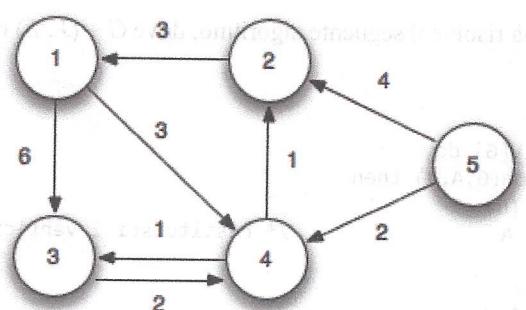
Avvertenza: Si giustifichino tecnicamente tutte le risposte. In caso di discussioni poco formali o approssimative gli esercizi non verranno valutati pienamente.

1. Si disegni l'albero binario di ricerca la cui visita in post-ordine ha come risultato 1, 12, 7, 16, 19, 25, 20, 13. Poi si effettui la cancellazione del nodo 13 e si disegni l'albero risultante.

2. Un algoritmo ricorsivo \mathcal{A} determina un albero di copertura minima in un grafo pesato connesso e ha complessità pari a:

$T(m) = 16T(m/4) + 16m$ dove m rappresenta il numero di archi del grafo. Si stabilisca se \mathcal{A} è asintoticamente più efficiente dell'algoritmo di Kruskal.

3. Si scriva l'algoritmo di Floyd-Warshall e si supponga di volerlo utilizzare sul seguente grafo:



Si produca: (a) la matrice iniziale che dovrà essere data in ingresso all'algoritmo e (b) la matrice generata dall'algoritmo dopo la prima iterazione.

Algoritmi e Strutture Dati

a.a. 2022/23

Compito del 11/9/2023

Cognome: _____

Nome: _____

Matricola: _____

E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

Avvertenza: Si giustifichino tecnicamente tutte le risposte. In caso di discussioni poco formali o approssimative gli esercizi non verranno valutati pienamente.

1. Sia T un albero generale i cui nodi contengono interi e hanno campi: **key**, **left-child** e **right-sib**.
 - a. Si scriva una funzione **efficiente** in C o C++ che verifichi la seguente proprietà: per ogni nodo u , le chiavi dei figli del nodo u devono avere valori non decrescenti considerando i figli di u da sinistra verso destra. Il prototipo della funzione è

```
bool isNonDec(PNodeG r)
```

Restituisce `true` se la proprietà è verificata altrimenti `false`.
 - b. Valutare e giustificare la complessità della funzione
 - c. Specificare il linguaggio di programmazione scelto e la definizione di `PNodeG`.
2. Sia H_1 un vettore di lunghezza $2n$ contenente un max-heap di interi, di dimensione n , secondo lo schema visto a lezione (e nel libro di testo). Sia H_2 un vettore di lunghezza n contenente un max-heap di interi di lunghezza n , secondo lo schema visto a lezione (e nel libro di testo). Si consideri il problema di trasformare il vettore H_1 in un vettore **ordinato** contenente tutti gli elementi degli heap H_1 ed H_2 , senza allocare altri vettori ausiliari.
 - a. Fornire lo pseudocodice di un algoritmo **efficiente** per risolvere il problema proposto utilizzando, tra le procedure viste a lezione, solamente max-heapify.
 - b. Determinarne e giustificare la complessità.
3. Si stabilisca quale problema risolve il seguente algoritmo, dove $G = (V, E)$ rappresenta un grafo non orientato:

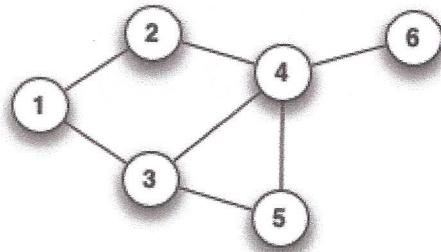
```
MyAlgorithm(G)
1. A = ∅
2. for each u ∈ V[G] do
3.   if MyFunction(G, A, u) then
4.     A = A ∪ {u}
5. return V[G] \ A           /* restituisci i vertici di G non appartenenti ad A */
```

```
MyFunction(G, A, u)
```

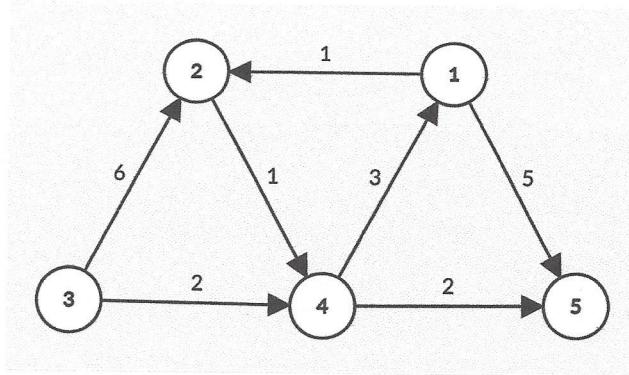
```
1. for each v ∈ A do
2.   if (u, v) ∈ E[G] then
3.     return FALSE
4. return TRUE
```

Si dimostri la correttezza dell'algoritmo e si determini la sua complessità.

Si simuli infine la sua esecuzione sul seguente grafo e si mostri (con alcuni esempi) che il risultato finale può dipendere dall'ordine in cui vengono estratti i vertici al passo 2 di `MyAlgorithm`. In particolare, si determini l'ordine di estrazione che consente all'algoritmo di restituire in uscita il massimo numero di vertici.



4. Si scriva l'algoritmo di Dijkstra, si derivi la sua complessità, si dimostri la sua correttezza e si simuli la sua esecuzione sul seguente grafo utilizzando il vertice 1 come sorgente:



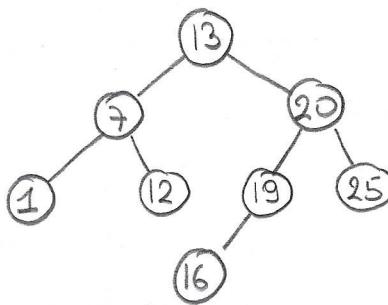
In particolare:

- a) si indichi l'ordine con cui vengono estratti i vertici
- b) si riempia la tabella seguente con i valori dei vettori d e π , iterazione per iterazione:

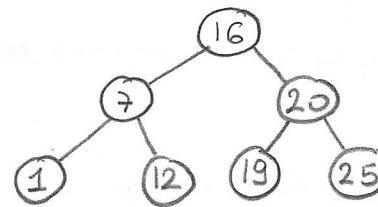
	vertice 1		vertice 2		vertice 3		vertice 4		vertice 5	
	$d[1]$	$\pi[1]$	$d[2]$	$\pi[2]$	$d[3]$	$\pi[3]$	$d[4]$	$\pi[4]$	$d[5]$	$\pi[5]$
1	dopo inizializzazione	0	NIL	∞	NIL	∞	NIL	∞	NIL	∞
2	iterazione 1	0	NIL	1	1	∞	NIL	∞	NIL	5
3	iterazione 2	0	NIL	1	1	∞	NIL	2	2	5
4	iterazione 3	0	NIL	1	1	00	NIL	2	2	4
5	iterazione 4	0	NIL	1	1	00	NIL	2	2	4
3	iterazione 5	0	NIL	1	1	00	NIL	2	2	4

PART I

1)



DOPO CANCELLAZIONE 13



2) $T(m) = 16 T\left(\frac{m}{4}\right) + 16m$

$\log_4 16 = 2 \Rightarrow g(m) = m^2 \quad f(m) = 16m = m$

$f(m) \leq g(m)$

1° CASO TEOREMA MASTER

$f(m) = O(m^{d-\varepsilon}) \quad \exists \varepsilon > 0$

$m = m^{2-\varepsilon} \rightarrow \varepsilon = 1 \vee$

$T(m) = \Theta(m^2)$

$T(KRUSKAL) = \Theta(m \log m) \text{ PIÙ EFFICIENTE DI A}$

3) FLOYD-WARSHALL (W)

$m \leftarrow \text{rows}(W)$

$D^{(0)} \leftarrow W$

for $k=1$ TO m for $i=1$ TO m for $j=1$ TO m

$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$$

return $D^{(m)}$

$$D^{(0)} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & \infty & 6 & 3 & \infty \\ 2 & 0 & 0 & 0 & 0 \\ 3 & \infty & 0 & 0 & 2 & \infty \\ 4 & \infty & 1 & 1 & 0 & \infty \\ 5 & \infty & 4 & \infty & 2 & 0 \end{pmatrix}$$

$$D^{(6)} = \begin{pmatrix} 0 & \infty & 6 & 3 & \infty \\ 3 & 0 & 9 & 6 & \infty \\ 0 & \infty & 0 & 2 & \infty \\ 0 & 1 & 1 & 0 & \infty \\ \infty & 4 & \infty & 2 & 0 \end{pmatrix}$$

PARTE II

3) L'algoritmo MyAlgorithm risolve il problema di trovare un insieme indipendente massimale in un grafo non orientato $G = (V, E)$. Un insieme indipendente è un sottoinsieme di vertici A in cui nessuna coppia di vertici è adiacente. L'algoritmo restituisce il complemento di A .

CORRETTEZZA

L'algoritmo garantisce che A sia indipendente perché ogni vertice aggiunto a A non è adiacente a nessun vertice già presente in A . L'algoritmo è greedy e aggiunge vertici a A finché possibile, garantendo che A sia massimale.

COMPLESSITÀ

MyFunction:

- Sono tutti i vertici in A e verifica l'adiacenza con \cup
- Complessità: $O(|A|)$, che nel caso peggiore è $O(n)$, con $n = |V|$

MyAlgorithm:

- Per ogni vertice $v \in V$, chiama MyFunction
- Complessità totale: $O(n^2)$ nel caso peggiore (se A diventa grande quanto V)

ESECUZIONE SUL GRAFO

Ordine estrazione $[1, 2, 3, 4, 5, 6]$

$$A = \emptyset$$

Aggiungo 1 poiché A è vuoto e non ci sono confronti

$$A = \{1\}$$

Verifico 2, è adiacente (sono $\{1, 2\}$) \rightarrow NON AGGIUNTO

Verifico 3, è adiacente (sono $\{1, 3\}$) \rightarrow NON AGGIUNTO

Verifico 4, non adiacente a vertici in A \rightarrow AGGIUNGO 4 AD A ($A = \{1, 4\}$)

Verifico 5, è adiacente (sono $\{4, 5\}$) \rightarrow NON AGGIUNTO

Verifico 6, è adiacente (sono $\{4, 6\}$) \rightarrow NON AGGIUNTO

$$\text{Risultato } V \setminus A = \{2, 3, 5, 6\}$$

ESECUZIONE SUL GRAFO CON ALTRO ORDINE

Ordine estrazione $[3, 6, 1, 2, 4, 5]$ $A = \emptyset$

Aggiungo 3 poiché A è vuoto e non ci sono confronti

Verifico 6, non è adiacente \rightarrow AGGIUNGO AD A ($A = \{3, 6\}$)Verifico 1, è adiacente \rightarrow NON AGGIUNGOVerifico 2, non è adiacente \rightarrow AGGIUNGO AD A ($A = \{3, 6, 2\}$)Verifico 4, è adiacente \rightarrow NON AGGIUNGOVerifico 5, è adiacente \rightarrow NON AGGIUNGORisultato $V \setminus A = \{1, 4, 5\}$ Se voglio massimizzare l'estrazione dei nodi da V allora si può usare la sequenza $[6, 1, 2, 5, 3, 4]$ che dà come risultato: $A = \{1, 2, 5, 6\} \quad V \setminus A = \{3, 4\}$ Se invece voglio ottimizzare la quantità di nodi che rimangono in V allora va bene la sequenza iniziale $[1, 2, 3, 4, 5, 6]$ perché al minimo devo avere due nodi in A poiché non esiste in G un nodo che abbia archi con TUTTI i vertici.4) Dijkstra (G, s)INIT-SS (G, s) $Q \leftarrow V[G]$ $S \leftarrow \emptyset$ while $Q \neq \emptyset$ do $u \leftarrow \text{EXTRACT-MIN}(Q)$ $S \leftarrow S \cup \{u\}$ For each $v \in \text{Adj}[u]$ RELAX ($u, v, w(u, v)$)return d, π INIT-SS (G, s)For each $v \in V[G]$ $d[v] = \infty$ $\pi[v] = \text{NIL}$ $d[s] = 0$ RELAX ($u, v, w(u, v)$)If $d[v] > d[u] + w(u, v)$ $d[v] = d[u] + w(u, v)$ $\pi[v] = u$

T (DIJKSTRA):

- ARRAY = $O(n^2)$ - HEAP BINARIO = $O(m \log n)$

CORRETTEZZA

Sia $G(V, E)$ un grafo orientato pesato sugli archi, cioè con $w: E \rightarrow \mathbb{R}$ tale che $\forall (u, v) \in E: w(u, v) \geq 0$

Allora alla fine dell'algoritmo di Dijkstra, si ha:

$$1) \forall v \in V: d[v] = \delta(s, v)$$

2) G_p è un albero di cammini minimi

Dimostreremo la correttezza del primo punto attraverso la dimostrazione di un'affermazione più forte ma equivalente, cioè che per ogni $u \in V$, al momento dell'estrazione di u , risulta $d[u] = \delta(s, u)$. La dimostrazione avviene per assurdo e si avvale di 9 osservazioni.

DIMOSTRAZIONE

Supponiamo per assurdo che esista un vertice $u \in V$ tale che al momento della sua estrazione $d[u] \neq \delta(s, u)$, e che u sia il primo vertice per cui questo accade.

OSSERVAZIONI

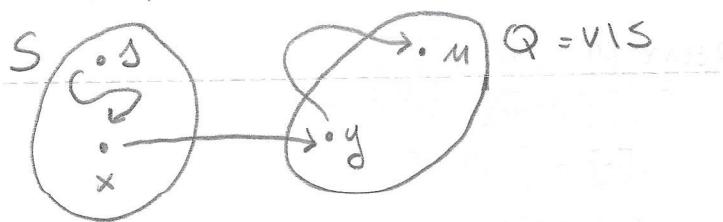
1) u non può essere la sorgente: dopo l'INIT-ss, $d[s] = 0 = \delta(s, s)$, e $\delta(s, s)$ non può valere $-\infty$ perché per ipotesi non ci sono archi con pesi negativi, quindi neanche cicli negativi

2) Al momento dell'estrazione di u , $S \neq \emptyset$, perché in S ci sarà almeno la sorgente s

3) u non è raggiungibile dalla sorgente: se così fosse, $\delta(s, u) = +\infty = d[u]$, che sarebbe contro a e non violerebbe le proprietà che abbiamo voluto violare per assurdo; quindi $\delta(s, u) \neq +\infty$

4) Ci poniamo nell'istante in cui s è già stato estratto ($s \in S$) ma $u \in Q = V \setminus S$

Per il punto precedente, esiste un cammino minimo p tra s e u : sia (x, y) un arco appartenente a p che attraversa il taglio, cioè tale che $x \in S$ e $y \in Q$



5) Per ipotesi, vale che $d[x] = \delta(s, x)$: infatti, u è il primo vertice per cui questa proprietà non vale

6) Poiché siamo su un cammino minimo, possiamo applicare la proprietà della convergenza: al momento dell'estrazione di x , applichiamo la RELAX su y e otteniamo che $d[y] = \delta(s, x) + w(x, y) = \delta(s, y)$

- 7) Dal momento che stiamo per estrarre il nodo u , siccome l'algoritmo di Dijkstra esce il vertice avente campo d più piccolo, dovrà valere che $d[u] \leq d[y]$
- 8) Non può accadere che $\delta(s,y) > \delta(s,u)$: in virtù del fatto che ci troviamo in un cammino minimo e che i pesi sono tutti maggiori o uguali a zero, allora $\delta(s,y) \leq \delta(s,u)$
- 9) Per la proprietà del limite inferiore, vale sicuramente $\delta(s,u) \leq d[u]$

Ricostuiamo l'assurdo sulla base delle precedenti osservazioni:

$$\begin{aligned} \delta(s,u) &\leq d[u] \quad \text{PER OSS. 9} \\ &\leq d[y] \quad \text{PER OSS. 7} \\ &= \delta(s,y) \quad \text{PER OSS. 6} \\ &\leq \delta(s,u) \quad \text{PER OSS. 8} \end{aligned}$$

A valendosi delle osservazioni, siamo riusciti a "rinchiudere" il valore $d[u]$ tra $\delta(s,u)$ e $\delta(s,u)$: $\delta(s,u) \leq d[u] \leq \delta(s,u) \Leftrightarrow d[u] = \delta(s,u)$, che è assurdo in quanto andiamo ad invalidare l'ipotesi che $d[u] \neq \delta(s,u)$.

```
1) struct NodeG {
    int key;
    NodeG* leftChild;
    NodeG* rightSib;
};

typedef NodeG* PNodeG;
```

$$T(n) = O(n)$$

n = numero nodi dell'albero

Ogni nodo viene visitato una sola volta. Per ogni nodo si scambiano i suoi figli per verificare l'ordine e si effettua una chiamata ricorsiva su ciascun figlio.

```
bool isNonDec(PNodeG r) {
    if (r == nullptr)
        return true;
    // Controlla l'ordine dei figli del nodo corrente
    PNodeG child = r->leftChild;
    while (child != nullptr && child->rightSib != nullptr) {
        if (child->key > child->rightSib->key)
            return false; // Ordine non rispettato
        child = child->rightSib;
    }
    // Controlla ricorsivamente tutti i sottosalberi
    child = r->leftChild;
    while (child != nullptr) {
        if (!isNonDec(child))
            return false;
        child = child->rightSib;
    }
    return true;
}
```

2)

// Copia gli elementi di H_2 nella seconda metà di H_1

for $i = 0$ TO $m-1$

$$H_1[m+i] = H_2[i]$$

// costruisce un max-heap sull'intero vettore H_1

buildMaxHeap(H_1);

// Utilizzo l'Heapsort per avere l'array ordinato in modo crescente

for $i = 2m-1$ DOWN TO 1:

swap($H_1[0], H_1[i]$)

max-heapify($H_1, 0$)

COMPLESSITÀ

Copia H_2 in H_1 : $O(m)$

Costruzione max heap su H_1 : $O(m)$

Heapsort: $O(m \log m)$

Costo totale: $O(m \log m)$