

Algoritmi e Strutture Dati

a.a. 2021/22

Compito del 12/9/2022

Cognome: _____

Nome: _____

Matricola: _____

E-mail: _____

Parte I

(30 minuti; ogni esercizio vale 2 punti)

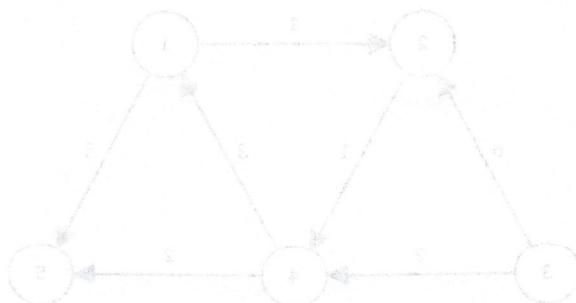
Avvertenza: Si giustifichino tecnicamente tutte le risposte. In caso di discussioni poco formali o approssimative gli esercizi non verranno valutati pienamente.

1. Sia H un **min-heap** contenente n interi distinti ed implementato con un vettore (come visto durante il corso). Rispondere alle seguenti domande, **motivando le risposte**.

- Assumendo che i nodi interni di H siano già ordinati, è possibile ordinare tutti gli elementi di H con complessità strettamente inferiore a $O(n \log n)$?
- Sia k costante rispetto ad n . Quanto costa determinare la k -esima chiave più piccola di H ?
- Quanto costa eliminare la chiave $H[i]$?

2. Il Prof. Cook sostiene che, sebbene (come visto a lezione) $\text{3-SAT-FNC} \leq_p \text{CLIQUE}$, non sia vero il contrario, ovvero che CLIQUE non è riducibile polinomialmente a 3-SAT-FNC . L'affermazione è corretta? (Spiegare.)

3. Dato un grafo orientato G con n vertici ed m archi, quante sono *esattamente* le operazioni di rilassamento (“relax”) realizzate dall’algoritmo di Bellman-Ford su G ? Potremmo affermare qualcosa riguardo alla correttezza dell’algoritmo nel caso si facesse qualche “relax” in più? (Spiegare.)



Algoritmi e Strutture Dati

a.a. 2021/22

Compito del 12/9/2022

Cognome: _____

Nome: _____

Matricola: _____

E-mail: _____

Parte II

(2.5 ore; ogni esercizio vale 6 punti)

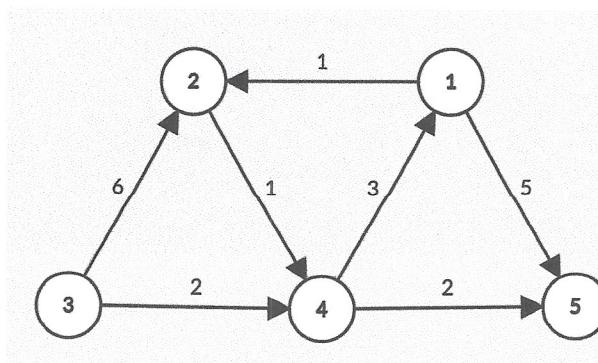
Avvertenza: Si giustifichino tecnicamente tutte le risposte. In caso di discussioni poco formali o approssimative gli esercizi non verranno valutati pienamente.

1. Sia T un albero generale i cui nodi hanno campi: **key**, **left-child** e **right-sib**. Scrivere una funzione **efficiente** in C o C++ che calcoli il numero di foglie di T e analizzarne la **complessità**.
Specificare quale linguaggio è stato utilizzato.

2. Sia A un array di n numeri naturali. Si consideri il problema di stampare in ordine **crescente** tutti i numeri che compaiono in A almeno $\lceil n/k \rceil$ volte, dove $k > 0$ è una costante.
Si scriva una procedura **efficiente** che, dati A , n e k , risolva il problema proposto.
Valutare e giustificare la complessità dell'algoritmo proposto.

Si devono scrivere le eventuali funzioni/procedure ausiliarie utilizzate.

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

4. Sia $G = (V, E)$ un grafo non orientato e connesso con funzione peso $w : E \rightarrow \mathbb{R}$ e sia S un sottoinsieme di vertici di G . Si supponga che il taglio $(S, V \setminus S)$ sia attraversato da un unico arco leggero (u, v) (ovvero: $w(u, v) < w(x, y)$ per ogni altro arco (x, y) che attraversa il taglio).

Stabilire se le seguenti affermazioni sono vere o false:

- a. G contiene un unico albero di copertura minimo
- b. Tutti gli alberi di copertura minimi di G contengono l'arco (u, v)
- c. Esiste un albero di copertura minimo di G che contiene l'arco (u, v)

Nel primo caso si offre una dimostrazione, nel secondo un controesempio.

PART

- 1) a) No, non è possibile ottenere una complessità strettamente minore a $O(n \log n)$ anche se i nodi interni sono già ordinati.
- Un min-heap è una struttura dati che garantisce solo che ogni nodo padre è minore o uguale di suoi figli, non garantisce un ordinamento completo.
 - Anche se i nodi interni sono ordinati, le foglie potrebbero non esserlo e potrebbero essere sparse in modo arbitrario nell'ultimo livello.
 - Per ottenere un ordinamento completo, è comunque necessario estrarre tutti gli elementi uno per uno (operazione $O(\log n)$ ciascuna) ottenendo $O(n \log n)$ complessivo.
 - Il limite inferiore per l'ordinamento basato su confronti è $\Omega(n \log n)$, quindi non è possibile fare meglio.
- b) $O(k \log n)$
- In un min-heap, la prima chiave più piccola è la radice $O(1)$.
 - Per trovare la seconda, terza, ..., k -esima più piccola possiamo:
 - Estrarre k volte la radice (ogni estrazione costa $O(\log n)$): $O(k \log n)$
 - Oppure usare un heap ausiliario che mantiene i candidati: $O(k \log k)$
 - Poiché k è costante, entrambi i metodi diventano $O(1)$, ma la complessità asintotica per k variabile è come sopra.
- c) $O(\log n)$
- Per eliminare un elemento in posizione i in un heap:
 - Sostituirlo con l'ultimo elemento dell'heap
 - Ridurre la dimensione dell'heap di 1
 - Eseguire heapify sulla posizione i
 - Heapify ha complessità $O(\log n)$ nel caso peggiore (altezza dell'heap)
 - L'accesso all'elemento $H[i]$ è $O(1)$, quindi la complessità totale è dominata dall'operazione di heapify.

2) Questa affermazione è FALSA, perché:

- Entrambi i problemi sono NP-completi
- Per definizione, ogni problema in NP è riducibile in tempo polinomiale a un problema NP-completo, in particolare 3-SAT-FNC
- Quindi CLIQUE \leq_p 3-SAT-FNC è vero

3) L'algoritmo consiste in $m-1$ fasi (dove m è il numero di vertici)

- In ogni fase, tutti gli m archi vengono rilassati
- Quindi: operazioni totali = $(m-1) \times m$

Per un grafo orientato G con m vertici e m archi l'algoritmo di Bellman-Ford esegue:

$(m-1) \times m$ operazioni di relax nel caso standard

ESECUZIONE DI RILASSAMENTI AGGIUNTIVI

Dopo la $(m-1)$ -esima Fase:

- Se eseguiamo un'ulteriore Fase (Fase m) e qualche rilassamento ha ancora successo:
 - Questo indica la presenza di un ciclo negativo raggiungibile dalla sorgente
 - È proprio così che Bellman-Ford rileva i cicli negativi

Durante le prime $m-1$ fasi:

- Eseguire rilassamenti extra (oltre quelli standard) non influisce sulla correttezza:
 - I rilassamenti aggiuntivi saranno semplicemente INEFFICACCI (non modificheranno le stime di distanza)
 - L'algoritmo rimane corretto purché si rispetti l'ordine delle $m-1$ fasi complete

Possiamo affermare che:

• L'algoritmo RESTA CORRETTO anche con rilassamenti extra, purché:

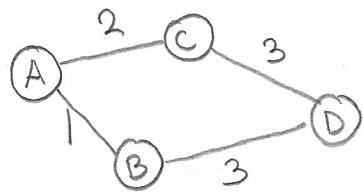
- Si completino tutte le $m-1$ fasi standard
- I rilassamenti aggiuntivi non alterino l'ordine fondamentale dell'algoritmo

• I rilassamenti in più possono:

- Non avere effetto (se le distanze sono già ottime)
- Anticipare la convergenza (ma la complessità asintotica rimane $O(m \cdot m)$)
- Rilevare precocemente cicli negativi in alcuni casi.

4) a) FALSO

L'unicità dell'arco leggero su un singolo taglio non garantisce l'unicità del MST globale.



- Taglio $S = \{A\}$, $V \setminus S = \{B, C, D\}$: Unico arco leggero (A, B) con peso 1
- Taglio $S = \{A, B\}$, $V \setminus S = \{C, D\}$:
 - Archi: (A, C) peso 2 e (B, D) peso 3
 - (A, C) è l'unico arco leggero
- MST possibili:
 - 1) $\{(A, B), (A, C), (C, D)\}$
 - 2) $\{(A, B), (B, D), (A, C)\}$

Nonostante ci siano tagli con unici archi leggeri, il grafo ha più MST.

b) VERO

Per il teorema fondamentale degli MST:

- 1) Dato un taglio $(S, V \setminus S)$ in un grafo connesso
- 2) se esiste un unico arco leggero (u, v) che attraversa il taglio
- 3) Allora TUTTI gli MST devono contenere (u, v)

Perché:

- Se un MST non contiene (u, v) dovrebbe contenere un altro arco (x, y) che attraversa il taglio
- Ma $w(x, y) > w(u, v)$, quindi potremmo sostituire (x, y) con (u, v) ottenendo un albero di peso minore
- Questo contraddice la minimalità dell'MST.

c) VERO

Questo è un caso particolare dell'affermazione b, ma più debole.

L'esistenza è garantita dall'algorithmo di Prim o Kruskal:

- Prim: partendo da S , sceglieni (u, v) quando espande S
- Kruskal: (u, v) sarà il primo arco del taglio considerato

Più in generale, poiché b è vera e gli MST esistono sempre in grafi connessi, ne esiste almeno uno che contiene (u, v)

3) Dijkstra (G, w, s)

INIT-SS (G, s)

$Q \leftarrow V[G]$

$S \leftarrow \emptyset$

while $Q \neq \emptyset$

$u \leftarrow \text{EXTRACT MIN } (Q)$

$S \leftarrow S \cup \{u\}$

for each $v \in \text{Adj}[u]$

RELAX $(u, v, w(u, v))$

return (d, G_{π})

INIT-SS (G, s)

for each $u \in V[G]$

$d[u] = +\infty$

$\pi[u] = \text{NIL}$

$d[s] = 0$

RELAX $(u, v, w(u, v))$

if $d[v] > d[u] + w(u, v)$ then

$d[v] = d[u] + w(u, v)$

$\pi[v] = u$

T(DIJKSTRA) :

• HEAP $O(m \log n)$

- GRAFO SPARSO : $m \approx n = O(n \log n)$

- GRAFO DENSO : $m \approx n^2 = O(n^2 \log n)$

• ARRAY $O(n^2)$

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'algorithmo avremo:

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

2) G_{π} 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 $d[u] = S(s, u)$

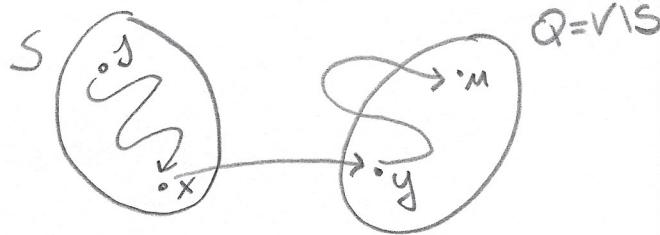
La dimostrazione avviene per assurdo e si avvale di 3 osservazioni:

DIMOSTRAZIONE

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

OSSERVAZIONI

- 1) u non può essere la sorgente: dopo la $\text{INIT-ss}(G, s)$, $d[s] = 0 = \delta(s, s)$, 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 irraggiungibile dalla sorgente: se così fosse, $\delta(s, u) = +\infty = d[u]$, che sarebbe controlo e non violerebbe la 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 no ($u \in Q = V \setminus S$). 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 $d[x] = S(s, x)$: infatti, s 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 $d[y] = S(s, x) + w(x, y) = S(s, y)$

7) Dal momento che stiamo per estrarre il nodo u , siccome l'algoritmo di Dijkstra estrae il vertice avente campo d più piccolo, dovrà valere che $d[u] \leq d[y]$

8) Non può accadere che $S(s, y) > S(s, u)$: in virtù del fatto che ci troviamo in un cammino minimo e che i pesi sono tutti maggiori o uguali a zero, allora $S(s, y) \leq S(s, u)$

9) Per la proprietà del limite inferiore, vale sicuramente che $S(s, u) \leq d[u]$

Ricostuiamo l'assurdo sulla base delle precedenti osservazioni:

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

Avvalendoci delle osservazioni, siamo riusciti a "inchiodare" il valore di $d[u]$ tra $\delta(s, u)$ e $\delta(s, y)$:

$$\delta(s, u) \leq d[u] \leq \delta(s, u) \Rightarrow d[u] = \delta(s, u)$$

che è assurdo in quanto andiamo ad invalidare l'ipotesi che $d[u] \neq \delta(s, u)$

1) struct Node {

```
int key;
Node* left-child; // Puntatore al primo figlio
Node* right-sib; // Puntatore al fratello destro
};
```

```
typedef Node* PNode;
```

```
int count-leaves (PNode root) {
```

```
if (root == nullptr)
    return 0;
```

```
// Se non ha figli è una foglia
if (root->left-child == nullptr)
    return 1;
```

```
int leaves = 0;
```

```
PNode child = root->left-child;
```

```
// Scorsi tutti i figli
```

```
while (child != nullptr) {
    leaves += count-leaves (child);
    child = child->right-sib;
}
```

```
return leaves;
```

```
}
```

La complessità è $O(m)$, dove m è il numero di nodi nell'albero. Ogni nodo viene visitato esattamente una volta.

```
2) std::vector<int> findFrequent(std::vector<int>& A, int m, int k) {  
    // Trova il valore massimo dell'array  
    int max_val = std::max_element(A.begin(), A.end());  
    // Crea un vettore delle occorrenze  
    std::vector<int> count(max_val + 1, 0);  
    int threshold = (m + k - 1) / k; // Calcolo  $\lceil m/k \rceil$   
    // Conta le occorrenze  
    for (int num : A) {  
        count[num]++;  
    }  
    // Raccogli gli elementi frequenti  
    std::vector<int> result;  
    for (int i = 0; i <= max_val; i++) {  
        if (count[i] >= threshold)  
            result.push_back(i);  
    }  
    // Non serve ordinare perché i valori sono già in ordine crescente  
    return result;  
}
```

$$T(m) = O(m + \max\{v\})$$