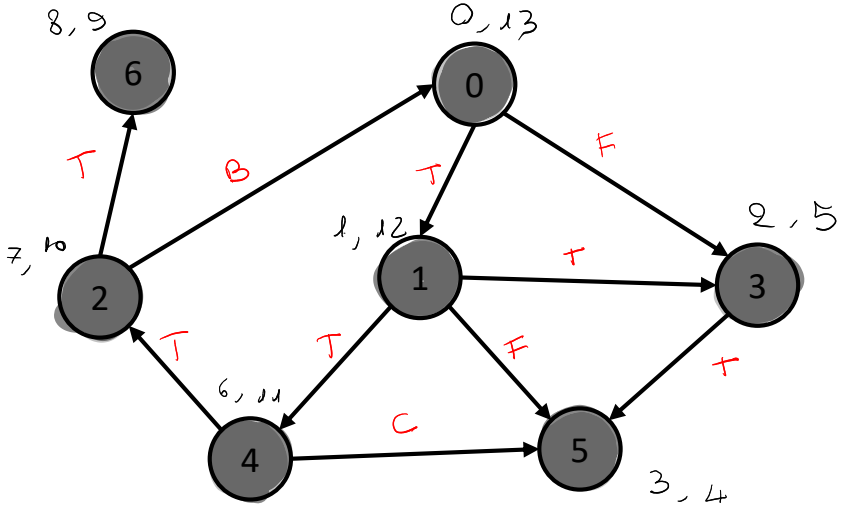
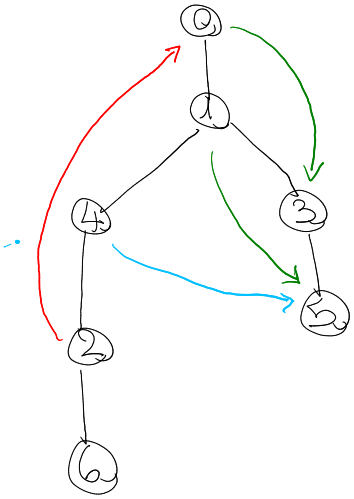
NOTA BENE

Le seguenti slide contengono una serie di esempi di quelli che sono gli esercizi pratici più frequenti all'esame scritto. Esse non sono esaustive. Il compito scritto può contenere anche esercizi più teorici, del genere:

- Dire su quale tecnica algoritmica si basa un determinato algoritmo
- Domande (teoriche) con risposta vero/falso e motivazione
- Modifica di algoritmi noti

Dato il seguente grafo orientato, se ne effettui una visita in profondità di tutti i vertici, considerando 0 come vertice sorgente e con l'ipotesi che i vertici siano memorizzati nelle liste di adiacenza in ordine alfabetico. Per ogni vertice, si indichino il tempo di inizio e fine visita. Etichettare inoltre ogni arco con T (dell'albero), B (all'indietro), F (in avanti) e C (di attraversamento).





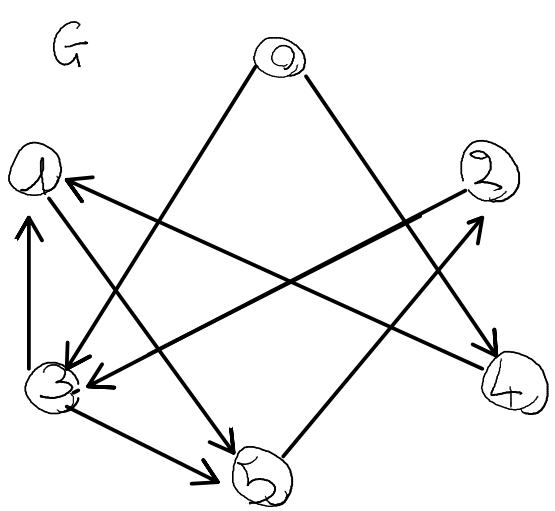
Dire se il grafo è aciclico

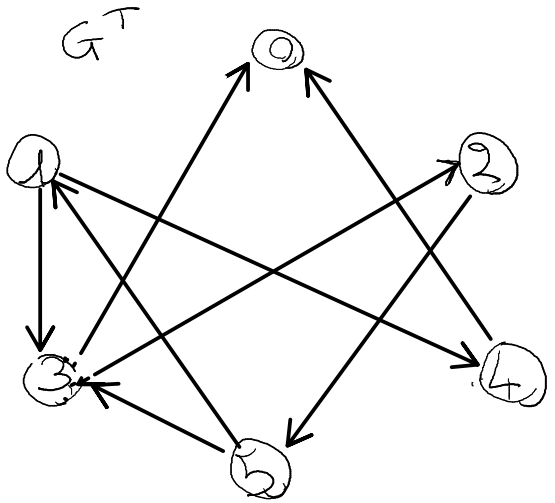
No, in quanto l'arco (2,0) è un arco all'indietro e i grafi aciclici non possono avere archi all'indietro.

Dato il grafo orientato con 6 nodi e i seguenti archi:

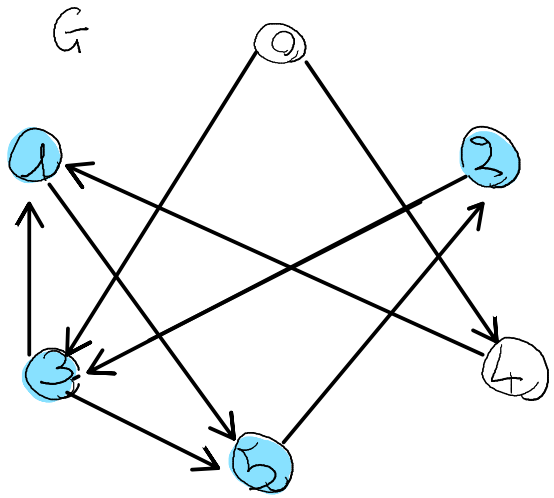
$$\langle 0,3\rangle, \langle 0,4\rangle, \langle 1,5\rangle, \langle 2,3\rangle, \langle 3,1\rangle, \langle 3,5\rangle, \langle 4,1\rangle, \langle 5,2\rangle$$

Utilizzando una qualsiasi tecnica vista, calcolare la componente fortemente connessa contenente il vertice 2. Descrivere il procedimento.

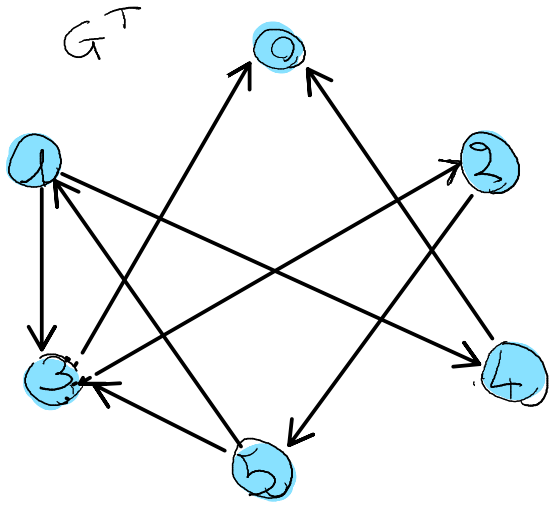




Algoritmo semplice per trovare la cfc contenente il vertice 2:
1. Calcoliamo i discendenti del nodo 2
1. Calcoliamo gli antenati del nodo 2
3. la cfc è data dall'intersezione di questi due insiemi



$\{1, 2, 3, 5\}$



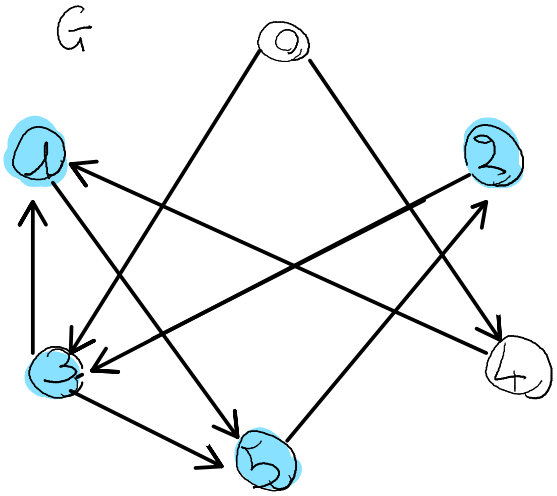
$\{0, 1, 2, 3, 4, 5\}$

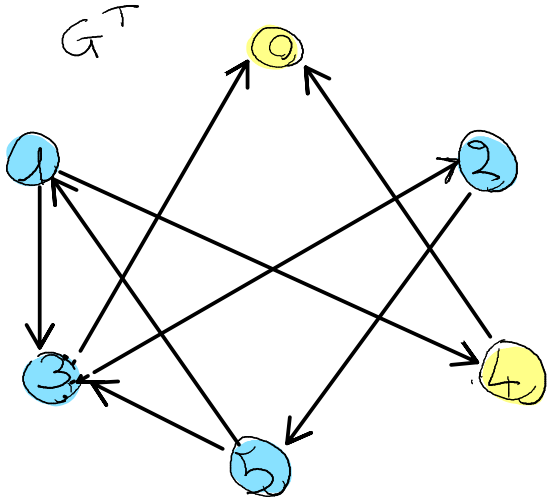
$\cap$

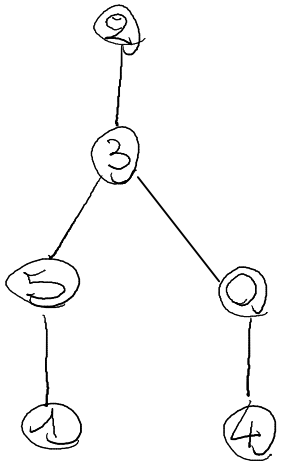
$\Downarrow$

$\{1, 2, 3, 5\}$

Algoritmo di Kosaraju
1. Visita G con una DFS e costruisci una lista di vertici in ordine decrescente dei tempi di fine visita
2. Costruisci GT
3. Visita GT con una DFS considerando i vertici nell'ordine trovato al passo 1



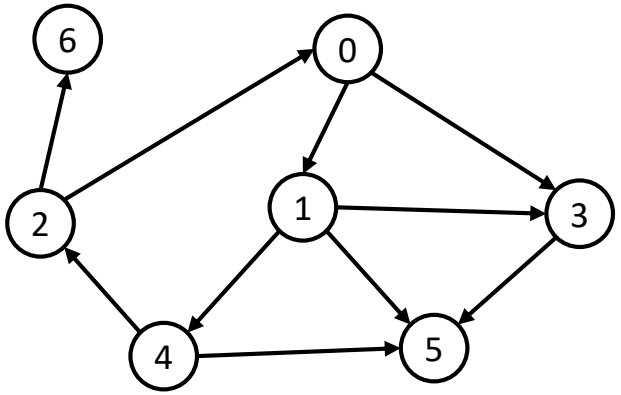




Scrivere (in pseudocodice) un algoritmo che, dato un grafo non pesato orientato G ed un vertice t di G, restituisca un vettore contenente in posizione i-esima, con i = 0..n-1:

- V (di Vicino) se il vertice i è a una distanza compresa tra 0 e 1 da t
- M (di Media distanza) se il vertice i è a una distanza compresa tra 2 e 3 da t
- L (di Lontano) se il vertice i è a una distanza di 4 o più da t

Ad esempio, dato il seguente grafo, e considerando t = 0,

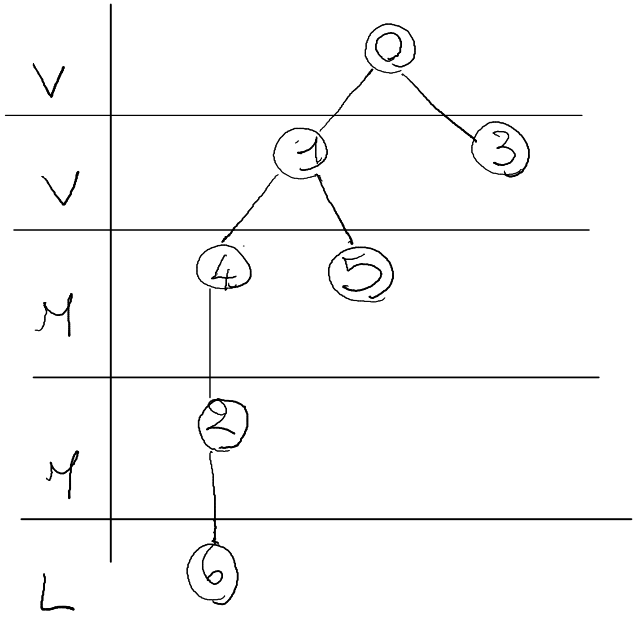


l'algoritmo deve restituire

V	V	M	V	M	M	L

Dato che dobbiamo tenere conto delle distanze dalla sorgente per livello, possiamo usare una BFS per visitare il grafo e costruire un albero di visita.
All'atto pratico, lo pseudocodice sarebbe:

```
D ← coda vuota
L ← vettore dei livelli
l ← contatore dei livelli
color[s] ← grigio
enqueue(D,s)
L[s] ← V
while NotEmpty(D) do
    u ← head(D)
    {visita u}
    l++
    for ogni v adj ad u then
        if color[v] = bianco
            color[v] ← grigio
            predecessore[v] ← u
            enqueue(D,v)
            if l == 1 L[v] ← V
            else if 2 ≤ v ≤ 3 L[v] ← M
            else L[v] ← L
    color[u] ← nero
    dequeue(D,u)
```

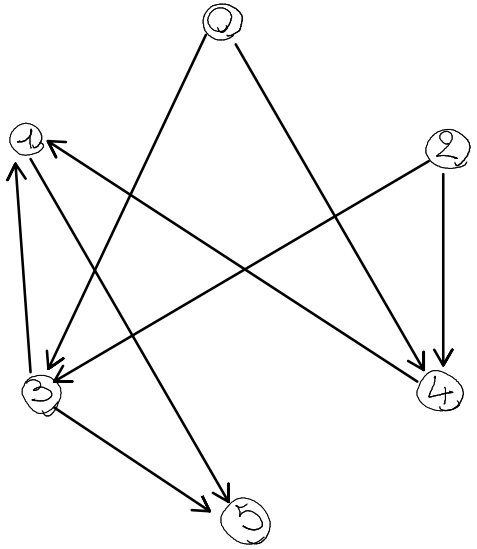


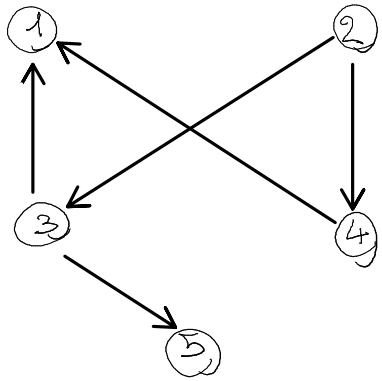
Dato il grafo orientato con 6 nodi e i seguenti archi:

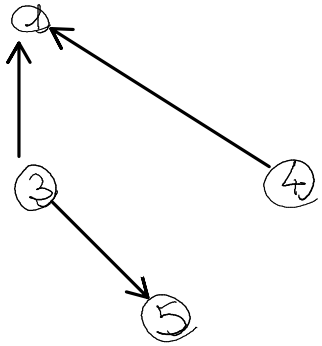
$$\langle 0,3\rangle, \langle 0,4\rangle, \langle 1,5\rangle, \langle 2,3\rangle, \langle 2,4\rangle, \langle 3,1\rangle, \langle 3,5\rangle, \langle 4,1\rangle$$

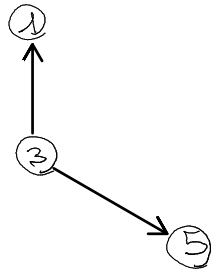
Utilizzando una qualsiasi tecnica vista, calcolarne un ordinamento topologico. Descrivere il procedimento.

```
Algoritmo astratto (sorgenti e pozzi):
G' ← fai una copia di G
ord ← lista vuota di vertici
while esiste u senza archi entranti in G'
    appendi u come ultimo elemento di ord
    rimuovi da G' u e tutti i suoi archi uscenti
    if G' non è vuoto then errore il grafo non è aciclico
    else return ord
```







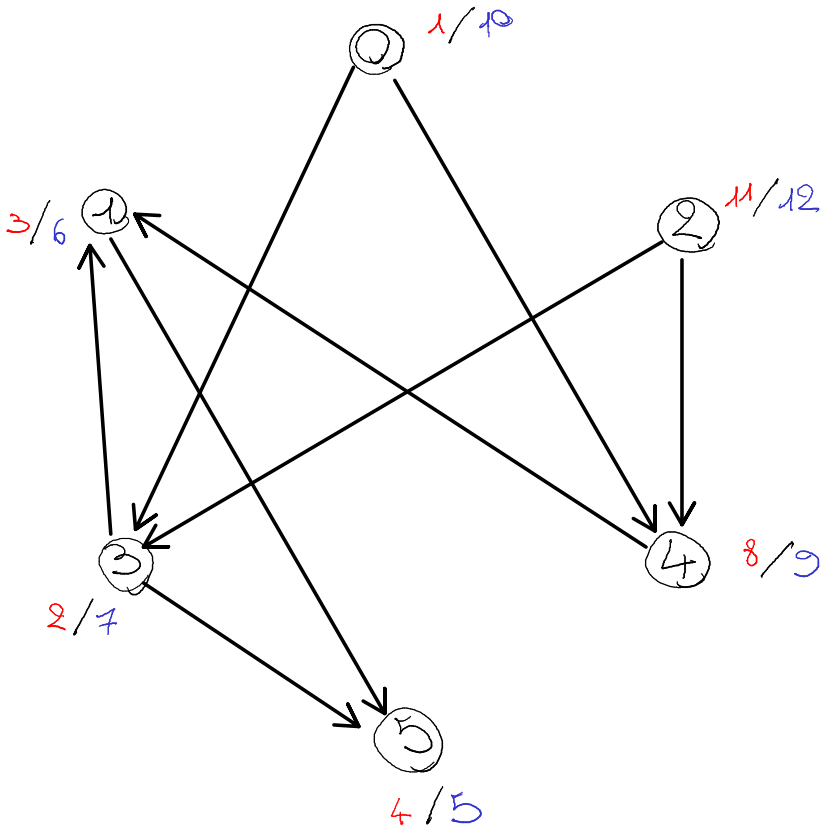


ORD: {0, 2, 4, 3, 1, 5} OPPURE ORD: {2, 0, 4, 3, 1, 5}

Algoritmo di ordinamento basato di DFS:

```
INIZIALIZZA (G)
ord ← un vettore di lunghezza n
t ← n-1
for ogni u ∈ V do
    if color [u] = white
    then DFS-TOPOLOGICAL (G,u,ord,t)
return ord

DFS-TOPOLOGICAL (G, u, ord, t)
color [u] ← gray
d[u] ← time ← time+1
for ogni v adiacente ad u
    if color[v] = white
        π[v] ← u
        DFS-TOPOLOGICAL (G,v,ord,t)
color[u] ← black
f[u] ← time ← time+1
ord[t] ← u
t--
```



ORD: { 2, 0, 4, 3, 1, 5 }

Si consideri la seguente tabella che associa ad ogni oggetto i un peso p_i ed un costo c_i. Dato uno zaino di capienza $P = 80$, si trovi una soluzione ottima per il problema dello zaino frazionario.

i	1	2	3	4	5	6
p_i	10	20	30	10	10	20
c_i	60	100	120	70	10	60
v_i	6	5	4	7	1	3

Algoritmo greedy per lo zaino frazionario:
1. Scegli la variabile xi con vi maggiore
2. xi = min(1,P/pi)
3. Elimina xi dal problema e P=P−xi*pi
4. Se il problema non contiene più nessuna variabile o P=0,
   restituisci gli elementi selezionati, altrimenti torna al punto 1

			x_i
1)	i = 4	P = 70	1
2)	i = 1	P = 60	1
3)	i = 2	P = 40	1
4)	i = 3	P = 10	1
5)	i = 6	P = 0	1/2

Dato l'alfabeto composto dai caratteri **a, b, c, d, e, f, g** e la seguente tabella delle frequenze, si calcoli una codifica binaria a lunghezza variabile dell'alfabeto secondo l'algoritmo di Huffman. (Si mostri come la struttura mantenuta dall'algoritmo cambia ad ogni iterazione)

Carattere	a	b	c	d	e	f	g
Frequenza	0.20	0.08	0.12	0.15	0.10	0.10	0.25

Algoritmo di Huffman:

1. Per ciascun carattere crea un albero formato solo da una foglia contenente il carattere e la frequenza del carattere
2. Fondi i due alberi che hanno frequenze mimine e costruisci un nuovo albero che ha come frequenza la somma delle frequenza degli alberi fusi
3. Ripeti la fusione finchè non si ottiene un unico albero

1) PQ = { B 0.08, E 0.10, F 0.10, C 0.12, D 0.15, A 0.20, G 0.25 }

2) PQ = { 0.10, 0.12, 0.15, 0.18, 0.20, 0.25 }

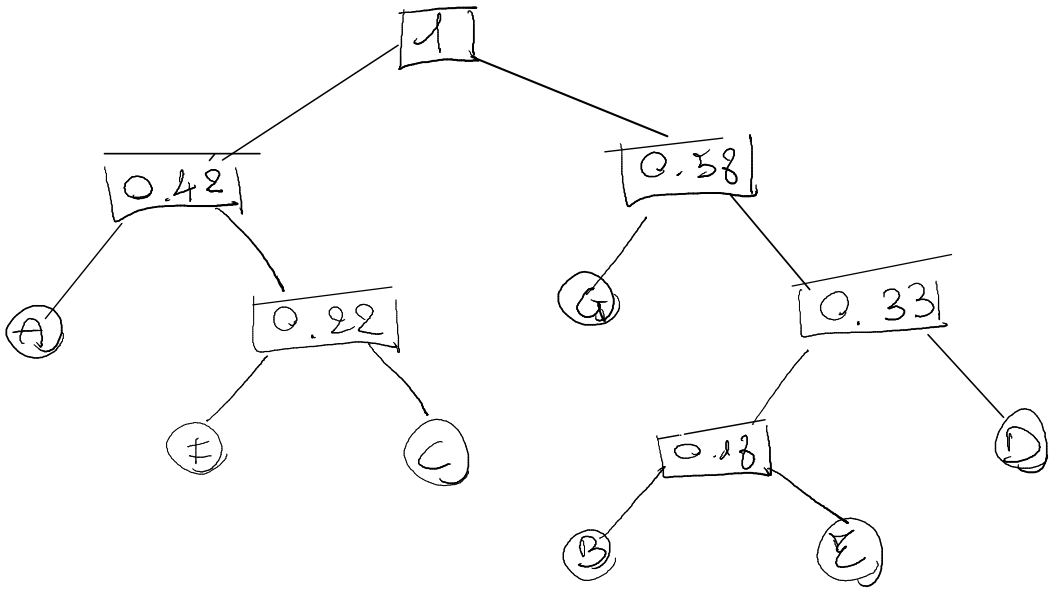
3) PQ = { 0.15, 0.18, 0.20, 0.22, 0.25 }

4) PQ = { 0.20, 0.22, 0.25, 0.33 }

5) PQ = { 0.25, 0.33, 0.42 }

6) PQ = { 0.58, 0.42 }

7) PQ = { 1 }



1. Si applichi l'algoritmo di Moore al seguente insieme di lavori, dove dx è la durata del lavoro Lx e sx è la scadenza del lavoro Lx.

L1: d1: 3 s1: 6 |

L2: d2: 3 s2: 5 |

L3: d3: 1 s3: 5 |

L4: d4: 3 s4: 8 |

L5: d5: 3 s5: 10 |

L6: d6: 2 s6: 8

```
Algoritmo di Moore:
1. ordina la sequenza dei lavori per ordine crescente di istante
   di scadenza
2. inizializza la sequenza soluzione Sol dei job schedulati come
   sequenza vuota e inizializza il tempo t a 0
3. for i = 1 to n
        aggiungi Li a Sol
        t = t + durata di Li
        if t > scadenza di Li
            togli da Sol il lavoro Lmax di durata massima
            t = t - durata Lmax
```

$L = \{ L3, L2, L1, L4, L6, L5 \}$ (durate: 1, 3, 3, 3, 2, 3)

1) $Sol = \{ L3 \}$ $t = 1$

2) $Sol = \{ L3, L2 \}$ $t = 4$

3) $Sol = \{ L3, L2, L1 \}$ $t = 7$ tolgo L1 $t = 4$

4) $Sol = \{ L3, L2, L4 \}$ $t = 7$

5) $Sol = \{ L3, L2, L4, L6 \}$ $t = 9$ tolgo L4 $t = 6$

6) $Sol = \{ L3, L2, L6, L5 \}$ $t = 9$

Dati i seguenti intervalli, con tempi di inizio e fine, trovarne un sottoinsieme costituito da intervalli tutti disgiunti e tale che il numero di intervalli sia il massimo.

I1: [5,10)

I2: [6,9)

I3: [8,13)

I4: [10,15)

I5: [17,20)

I6: [21,30)

I7: [24,25)

I8: [21,23)

```
Algoritmo intervalli disgiunti:
1. ordina l'insieme degli intervalli in una sequenza S ordinata
   secondo l'istante finale
2. inizializza la soluzione Sol come sequenza vuota
3. scandisci S in ordine, e per ogni suo elemento A:
   - se A inizia dopo la fine dell'ultimo elemento di Sol,
     aggiungilo al fondo di Sol
   - altrimenti non aggiungerlo
```

$S = \{I2, I1, I3, I4, I5, I8, I7, I6\}$

1) $Sol = \{I2\}$ $f = 9$ scarto I1, I3

2) $Sol = \{I2, I4\}$ $f = 15$

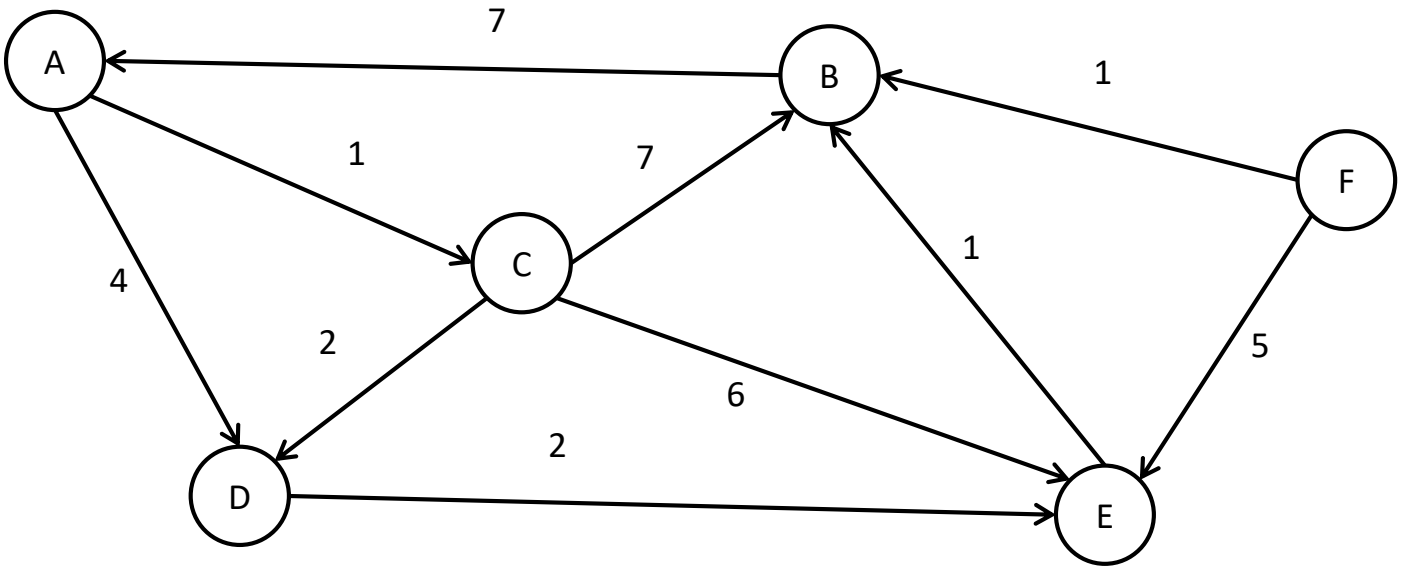
3) $Sol = \{I2, I4, I5\}$ $f = 20$

4) $Sol = \{I2, I4, I5, I8\}$ $f = 23$

5) $Sol = \{I2, I4, I5, I8, I7\}$ $f = 25$ scarto I6

Si applichi l'algoritmo di **Dijkstra** al seguente grafo, con vertice di partenza A e considerando le liste di adiacenza ordinate in ordine alfabetico. In particolare, per ogni ciclo dell'algoritmo (0 indica la condizione prima di entrare nel ciclo)

a. compilate la tabella d delle distanze stimate dei vertici da A
b. compilate la tabella dei vertici inclusi nella soluzione (per cui d[v] = δ(A, v))
c. disegnate (sul foglio protocollo) l'albero dei predecessori mantenuto dall'algoritmo (o equivalentemente, compilate una matrice π).



d	A	B	C	D	E	F
0	0_N	∞_N	∞_N	∞_N	∞_N	∞_N
1	0_N	∞_N	1_A	4_A	∞_N	∞_N
2	0_N	8_C	1_A	3_C	7_C	∞_N
3	0_N	8_C	1_A	3_C	5_D	∞_N
4	0_N	6_C	1_A	3_C	5_D	∞_N
5	0_N	6_C	1_A	3_C	5_D	∞_N
6	0_N	6_C	1_A	3_C	5_D	∞_N

	Vertici (neri) inclusi nella soluzione
0	
1	A
2	A, C
3	A, C, D
4	A, C, D, E
5	A, C, D, E, B
6	A, C, D, E, B, F

Si consideri una struttura **Union Find** di tipo Quick Union con ottimizzazione by-size e le seguenti operazioni. Si mostri la struttura (con eventuali variabili vicine ai nodi) dopo ogni operazione e gli eventuali output delle operazioni:

1. makeSet(A)
2. makeSet(B)
3. makeSet(C)
4. union(A,B)
5. union(C,A)
6. makeSet(D)
7. find(B)
8. makeSet(E)
9. union(E,D)
10. union(D,B)
11. find(D)

1) A

2) A B

3) A B C

4) A C B

5) A B C

6) A D B C

7) A

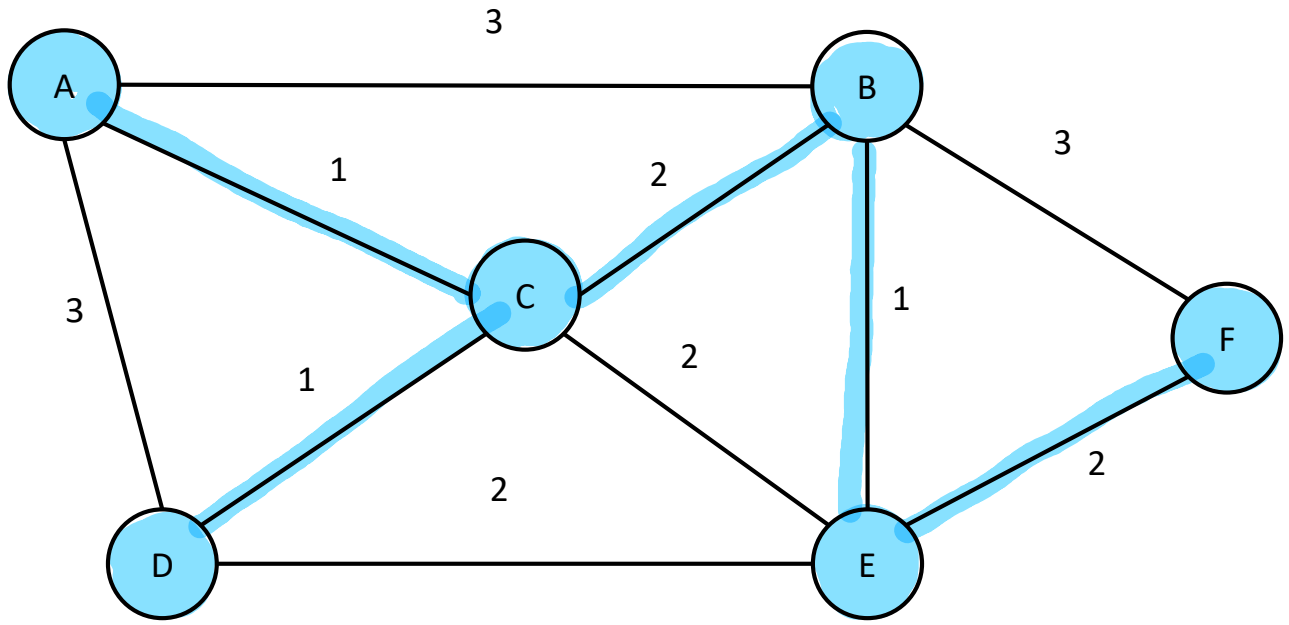
8) A D E B C

9) A D E B C

10) A B C D E

11) A

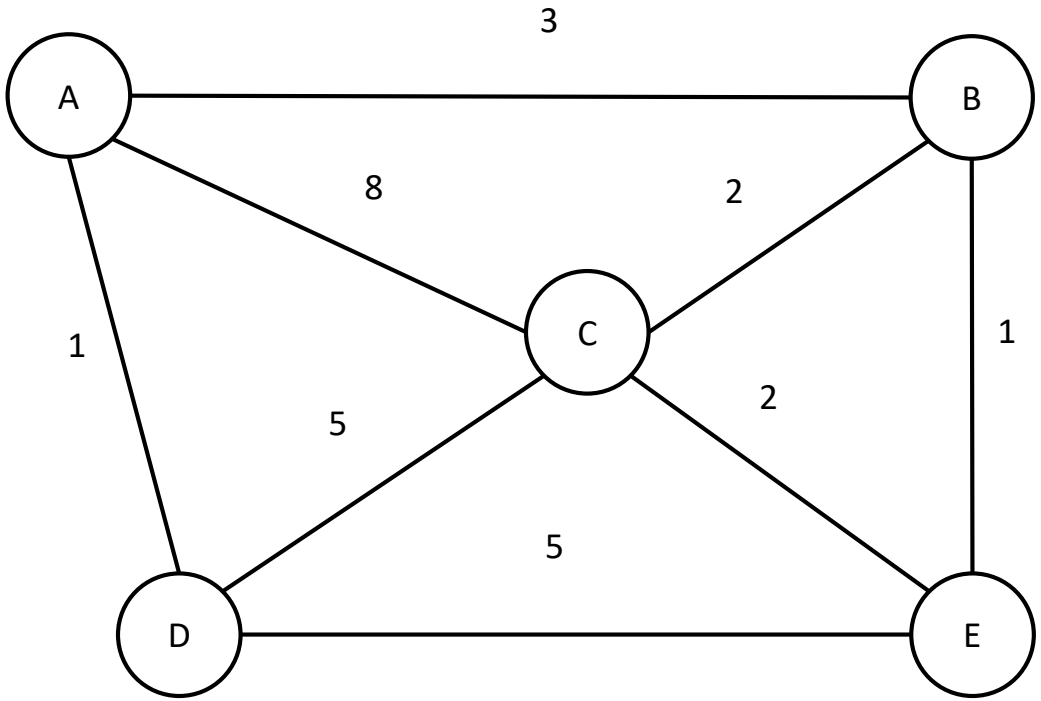
Si applichi l'algoritmo di **Prim** al seguente grafo, con vertice di partenza A e e considerando le liste di adiacenza ordinate in ordine alfabetico. Dopo ogni iterazione del ciclo (la riga 0 corrisponde alla situazione iniziale, prima di entrare nel ciclo) si compili la tabella delle distanze d e quella dei vertici ("definitivi") inclusi nella soluzione



d	A	B	C	D	E	F
0	0_N	∞_N	∞_N	∞_N	∞_N	∞_N
1	0_N	3_A	1_A	3_A	∞_N	∞_N
2	0_N	2_C	1_A	1_C	2_C	∞_N
3	0	2_C	1_A	1_C	2_C	∞
4	0	2	1	1	2	∞
5	0	2	1	1	1_B	3_B
6	0_N	2_C	1_A	1_C	1_B	2_E

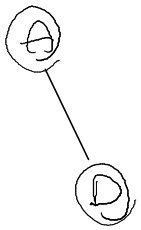
	Vertici (neri) inclusi nella soluzione
0	/
1	A
2	A, C
3	A, C, D
4	A, C, D, B
5	A, C, D, B, E
6	A, C, D, B, E, F

Si applichi l'algoritmo di **Kruskal** al seguente grafo. Si mostri come la foresta e la union find mantenute dall'algoritmo cambiano ad ogni iterazione (non è necessario rappresentare le union find come alberi, basta una rappresentazione grafica/insiemistica).

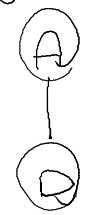


ARCHI = $(A,D)^1$ $(B,E)^1$ $(B,C)^2$ $(C,E)^2$ $(A,B)^3$ $(C,D)^5$ $(D,E)^5$ $(A,C)^8$

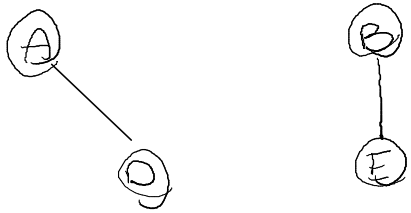
(A,D) MAR



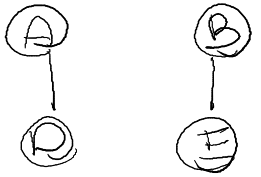
UF



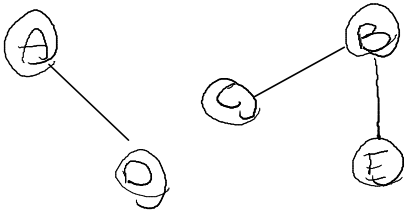
(B,E) MAR



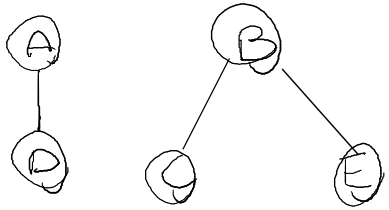
UF



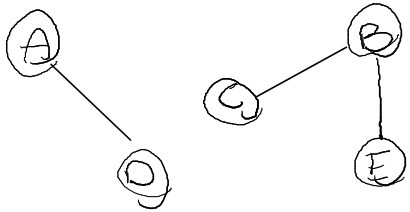
(B,C) MAR



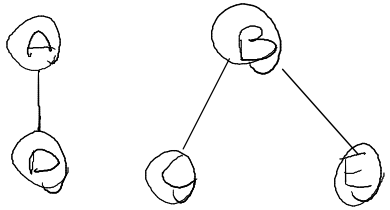
UF



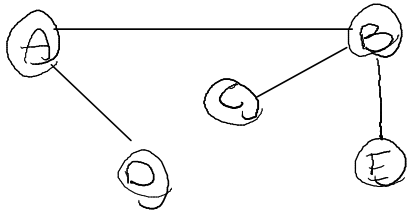
(C,E) MAR CICLO



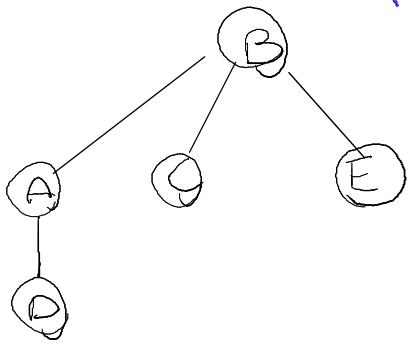
UF



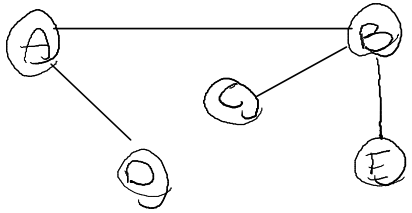
(A,B) MAR



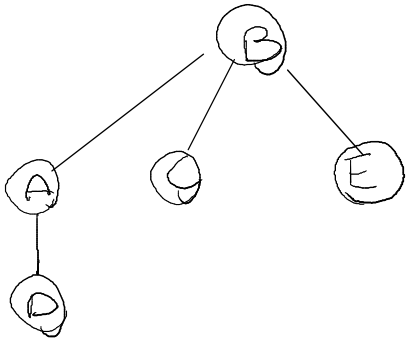
UF



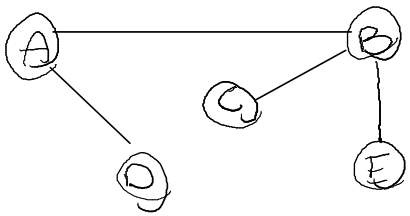
(C,D) MAR CICLO



UF

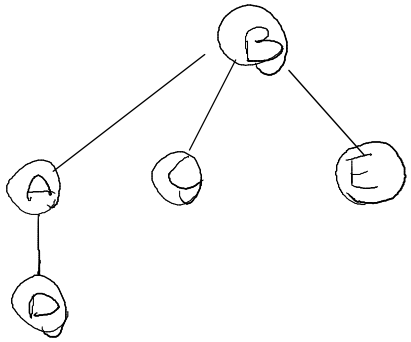


(D, E) MAR

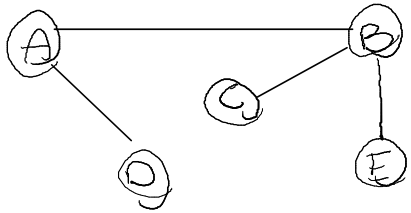


CICLO

UF

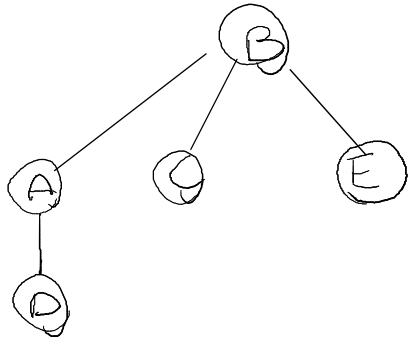


(A, C) MAR

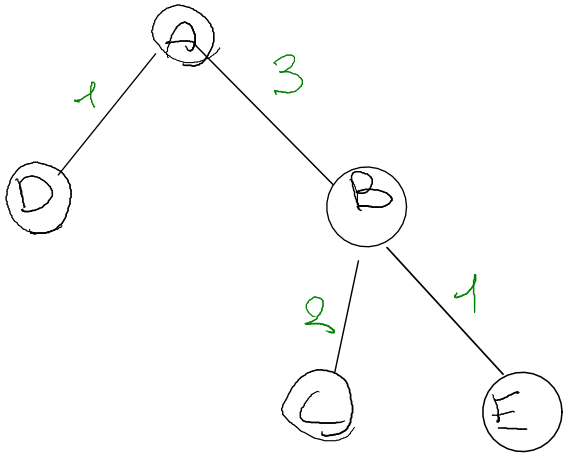


CICLO

UF



MAR



PESO MAR : 7

Utilizzando l'algoritmo visto a lezione, trovare la più lunga sottosequenza comune (**LCS**) tra le stringhe "ETUTZE" e "TZUETE".

Per la matrice LCS, utilizzare l'ottimizzazione delle frecce vista a lezione.

matrice LCS

		T	Z	U	E	T	E
E		↑	↑	↑	↖	←	↖
T		↖	←	←	↑	↖	←
U		↑	←	↖	←	←	←
T		↖	←	↑	←	↖	←
Z		↑	↖	←	←	↑	←
E		↑	↑	←	↖	←	↖

"TUTE"

matrice L

		T	Z	U	E	T	E
	0	0	0	0	0	0	0
E	0	0	0	0	1	1	1
T	0	1	1	1	1	2	2
U	0	1	1	2	2	2	2
T	0	1	1	2	2	3	3
Z	0	1	2	2	2	3	3
E	0	1	2	2	3	3	4

Utilizzando l'algoritmo visto a lezione, trovare la più lunga sottosequenza comune (**LCS**) tra le stringhe "AGCCGGATCGAGT" e "TCAGTACGTTA".

Per la matrice LCS, utilizzare l'ottimizzazione delle frecce vista a lezione.

matrice LCS

A G A C G T

		A	G	C	C	G	G	A	T	C	G	A	G	T
T		↑	↑	↑	↑	↑	↑	↑	↖	←	←	↑	←	↖
C		↑	↑	↖	↖	←	←	←	←	↖	←	←	←	←
A		↖	←	←	←	←	←	↖	←	↑	←	↖	←	←
G		↑	↖	←	←	←	←	←	←	←	↖	↑	↖	←
T		↑	↑	←	←	←	←	←	↖	←	←	←	↑	↖
A		↖	←	←	←	←	←	↖	←	←	←	↖	←	↑
C		↑	←	↖	↖	←	←	←	←	↖	←	←	←	↑
G		↑	↖	↑	←	↖	↖	←	←	←	↖	←	↖	←
T		↑	↑	←	←	↑	←	←	↖	←	←	←	↑	↖
T		↑	↑	←	←	↑	←	←	↖	←	←	←	←	↖
A		↖	←	←	←	↑	←	↖	←	←	←	↖	←	↑

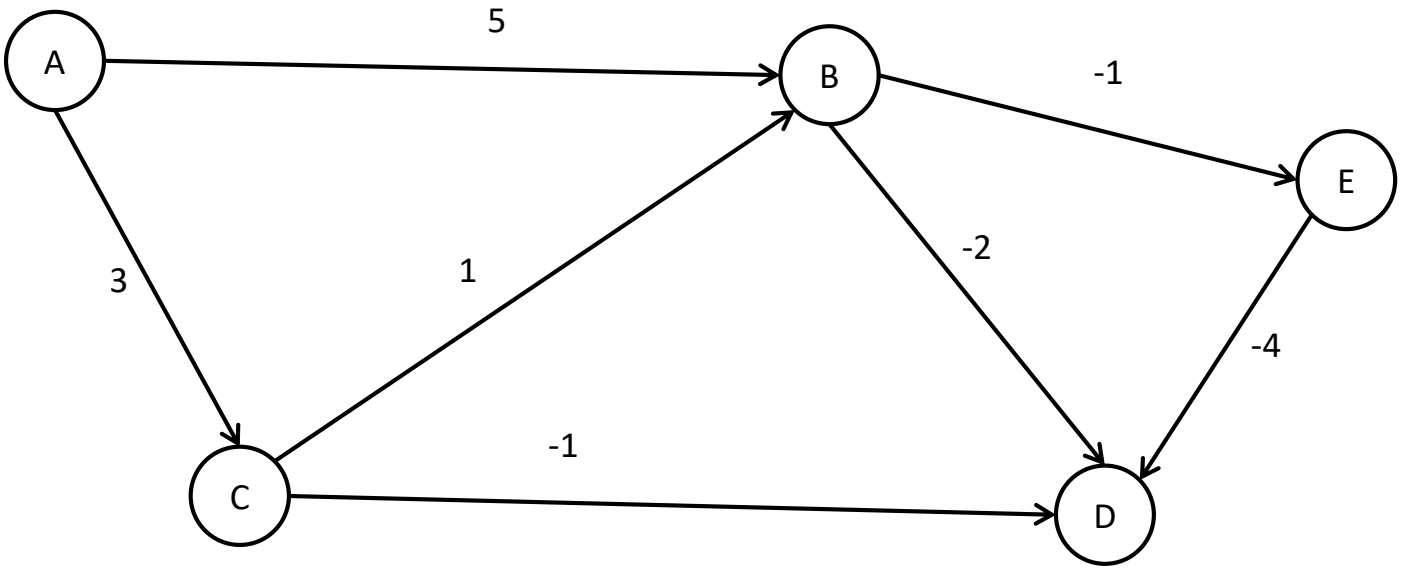
matrice L

		A	G	C	C	G	G	A	T	C	G	A	G	T
T		0	0	0	0	0	0	0	1	1	1	1	1	1
C		0	0	1	1	1	1	1	1	2	2	2	2	2
A		1	1	1	1	1	1	1	1	2	2	3	3	3
G		1	2	2	2	2	2	2	2	2	2	3	4	4
T		1	2	2	2	2	2	2	3	3	3	3	4	5
A		2	2	2	2	2	2	3	3	3	3	4	4	5
C		2	2	3	3	3	3	3	3	4	4	4	4	5
G		2	3	3	3	3	3	3	3	4	5	5	5	5
T		2	3	3	3	3	3	3	4	4	5	5	5	6
T		2	3	3	3	3	3	3	4	4	5	5	5	6
A		3	3	3	3	3	3	4	4	4	5	5	6	6

Si applichi l'algoritmo di **Bellman-Ford** al seguente grafo, considerando gli archi nel seguente ordine:

(A,B) (A,C) (C,D) (C,B) (E,D) (B,E) (B,D)

Si utilizzi la tabella d, se ne compili una per ogni ciclo dell'algoritmo.

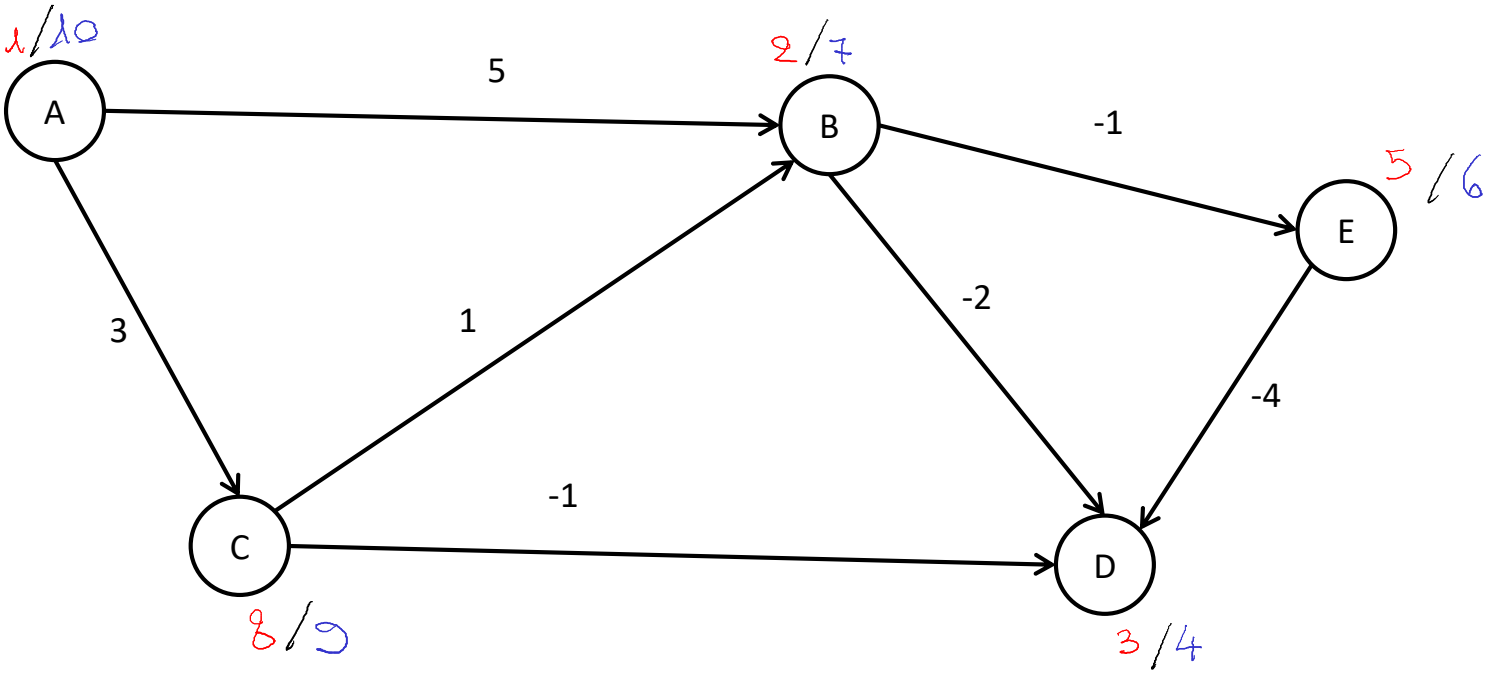


d	init	(A,B) 5	(A,C) 3	(C,D) -1	(C,B) 1	(E,D) -4	(B,E) -1	(B,D) -2
A	0	0_N	0_N	0_N	0_N	0_N	0_N	0_N
B	∞	5_A	5_A	5_A	4_C	4_C	4_C	4_C
C	∞	∞_N	3_A	3_A	3_A	3_A	3_A	3_A
D	∞	∞_N	∞_N	2_C	2_C	2_C	2_C	2_C
E	∞	∞_N	∞_N	∞_N	∞_N	∞_N	3_B	3_B

d	init	(A,B) 5	(A,C) 3	(C,D) -1	(C,B) 1	(E,D) -4	(B,E) -1	(B,D) -2
A	0	0_N	0_N	0_N	0_N	0_N	0_N	0_N
B	4_C	4_C	4_C	4_C	4_C	4_C	4_C	4_C
C	3_A	3_A	3_A	3_A	3_A	3_A	3_A	3_A
D	2_C	2_C	2_C	2_C	2_C	-1_E	-1_E	-1_E
E	3_B	3_B	3_B	3_B	3_B	3_B	3_B	3_B

Si applichi l'algoritmo di **Bellman-Ford** al seguente grafo, utilizzando **l'ottimizzazione per DAG**

```
INIZIALIZZA(G)
d[s] ← 0
ord ← TOPOLOGICAL SORT(G)
for i = 0 … n-1
    for ogni (ord[i],v) then
        if d[v] > d[ord[i]] + W(ord[i],v) then
            predecessore[v] ← ord[i]
            d[v] ← d[ord[i]] + W(ord[i],v)
```



$ORD = \{A, C, B, E, D\}$

$ARCHI = (A,B)\ (A,C)\ (C,B)\ (C,D)\ (B,D)\ (B,E)\ (E,D)$

(A,B)

v	A	B	C	D	E
d(v)	0	5	∞	∞	∞
v	A	B	C	D	E
π(v)	NULL	A	NULL	NULL	NULL

(A,C)

v	A	B	C	D	E
d(v)	0	5	3	∞	∞
v	A	B	C	D	E
π(v)	NULL	A	A	NULL	NULL

(C,B)

v	A	B	C	D	E
d(v)	0	4	3	∞	∞
v	A	B	C	D	E
π(v)	NULL	C	A	NULL	NULL

(C,D)

v	A	B	C	D	E
d(v)	0	4	3	2	∞
v	A	B	C	D	E
π(v)	NULL	C	A	C	NULL

(B,D)

v	A	B	C	D	E
d(v)	0	4	3	2	∞
v	A	B	C	D	E
π(v)	NULL	C	A	C	NULL

(B,E)

v	A	B	C	D	E
d(v)	0	4	3	2	3
v	A	B	C	D	E
π(v)	NULL	C	A	C	B

(E,D)

v	A	B	C	D	E
d(v)	0	4	3	-1	3
v	A	B	C	D	E
π(v)	NULL	C	A	E	B

Dato il grafo rappresentato con la seguente matrice di adiacenza, trovare i cammini minimi (ed i loro pesi) tra tutte le coppie di vertici, applicando l'algoritmo di Floyd-Warshall.
Si mostrino le matrici D (dei pesi) e P (dei predecessori) dopo ogni ciclo esterno dell'algoritmo (0 è la situazione iniziale, prima di entrare nel ciclo).

Se I cammini minimi non esistono, si dica il perché.

D^0	A	B	C
A	0	1	-2
B	-1	0	∞
C	5	2	0

D^A	A	B	C
A	0	1	-2
B	-1	0	-3
C	5	2	0

D[A,A] > D[A,A] + D[A,A]
D[A,B] > D[A,A] + D[A,B]
D[A,C] > D[A,A] + D[A,C]
D[B,A] > D[B,A] + D[A,A]
D[B,B] > D[B,A] + D[A,B]
D[B,C] > D[B,A] + D[A,C]
D[C,A] > D[C,A] + D[A,A]
D[C,B] > D[C,A] + D[A,B]
D[C,C] > D[C,A] + D[A,C]

D^B	A	B	C
A	0	1	-2
B	-1	0	-3
C	-1	2	-1

$D[A,A] > D[A,B] + D[B,A]$

$D[A,B] > D[A,B] + D[B,B]$

$D[A,C] > D[A^{1},B] + D[B^{-3},C]$

$D[B^{-1},A] > D[B,B^{0}] + D[B^{-1},A]$

$D[B^{0},B] > D[B,B^{0}] + D[B^{0},B]$

$D[B^{-3},C] > D[B^{0},B] + D[B,C^{-3}]$

$D[C^{5},A] > D[C^{2},B] + D[B^{-1},A]$

$D[C^{2},B] > D[C^{2},B] + D[B^{9},B]$

$D[C^{0},C] > D[C^{2},B] + D[B,C^{-3}]$

Non esistono cammini minimi in quanto vi sono cicli negativi.

D	A	B	C
A	-3	0	-3
B	-2	-1	-4
C	-1	-1	-1

$D[A^{0},A] > D[A,C^{-2}] + D[C^{-1},A]$

$D[A^{1},B] > D[A^{-2},C] + D[C^{2},B]$

$D[A^{-2},C] > D[A^{-2},C] + D[C^{-1},C]$

$D[B^{-1},A] > D[B^{-3},C] + D[C^{-1},A]$

$D[B^{0},B] > D[B^{-3},C] + D[C^{2},B]$

$D[B^{-3},C] > D[B^{-3},C] + D[C^{-1},C]$

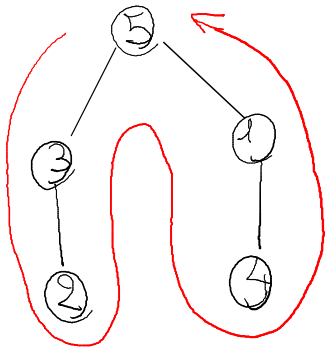
$D[C^{-1},A] > D[C^{-1},C] + D[C^{-1},A]$

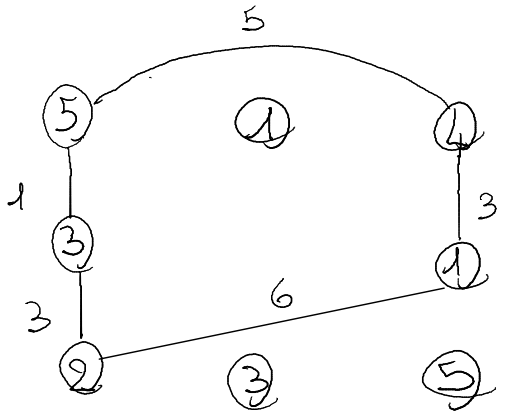
$D[C^{2},B] > D[C^{-1},C] + D[C^{0},B]$

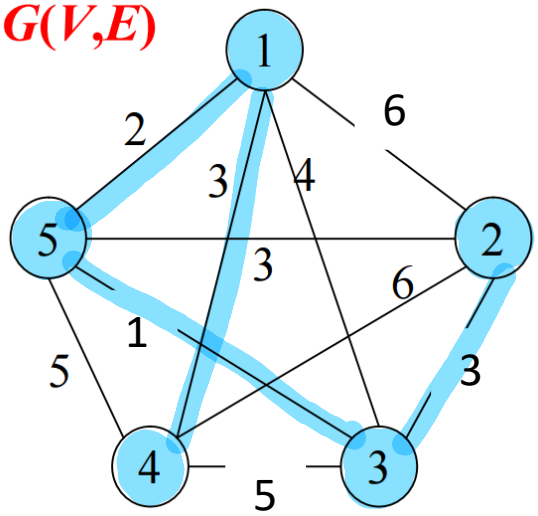
$D[C^{-1},C] > D[C^{-1},C] + D[C^{-1},C]$

Utilizzando l'algoritmo approssimato visto a lezione, si trovi un ciclo Hamiltoniano di peso al più 2 volte il peso del cammino Hamiltoniano di peso minimo.

```
Algoritmo:
scegli un vertice r casualmente
A ← PRIM(G,W,r)
ord ← DFS(A,r)
return ord
```







Altro possibile esercizo

Dati un grafo ed un ciclo Hamiltoniano contenuto in esso, generare il vicinato con la tecnica dei k-scambi con k=2

Costruzione di algoritmi

Un ladro entra in un magazzino e trova n oggetti. L'i-esimo oggetto ha un valore di $\boldsymbol{v_i}$ euro e pesa $\boldsymbol{p_i}$ chilogrammi (i pesi sono numeri **interi positivi**).

Gli oggetti NON sono frazionabili. Quindi il ladro può o prendere l'intero oggetto i, o non prenderlo.

Il ladro ha solo uno zaino, che può contenere oggetti per un massimo di $\boldsymbol{P}$ chilogrammi.

Scrivere un algoritmo di programmazione dinamica che restituisca il massimo valore che il ladro può prendere, sapendo che tale valore è dato dall'equazione ricorsiva

$$V(i,j) = \begin{cases} V(i-1,j) & se\ j < p_i \\ \max(V(i-1,j), V(i-1,j-p_i) + v_i) & altrimenti \end{cases}$$

Con $V(i,j)$ che è la soluzione ottima del sottoproblema limitato agli oggetti $1 \dots i$ e con zaino di capienza massima j.

Costruzione di algoritmi – II

In particolare,

1. Si descriva la struttura dati necessaria per la memoizzazione
2. Si definiscano i casi base, e le loro soluzioni
3. Si scriva in pseudocodice un algoritmo di programmazione dinamica che risolva il problema

Costruzione di algoritmi – SOLUZIONE

Struttura di memoizzazione.

$V(i, j)$ ha due parametri:

- i è l'ultimo oggetto che consideriamo

- j è la capienza

Visto che ci sono 2 parametri, possiamo usare una matrice V[]. Di quali dimensioni?

Il problema richiede di trovare la soluzione con n oggetti e P di capienza massima. Quindi la soluzione sarà contenuta in V[n,P].

Ci servono però anche i casi base. In particolare, ci serviranno i V[i,j] tali che i = 0 (nessun oggetto considerato) e/o j = 0 (peso massimo 0).

Quindi la matrice sarà grande $(n + 1) \times (P + 1)$.

Costruzione di algoritmi – SOLUZIONE

Valori casi base.

i = 0 (nessun oggetto considerato) – dato che non abbiamo considerato nessun oggetto, V[0,j] = 0 per ogni $0 \leq j \leq P$.

j = 0 (peso massimo 0) – dato che non possiamo prendere nessun oggetto, il valore massimo raggiungibile sarà 0. Quindi V[i,0] = 0 per ogni $0 \leq i \leq n$.

Costruzione di algoritmi – SOLUZIONE

Algoritmo.

```
Zaino(n,P,v[],p[]) // v[] e p[] sono i vettori dei valori e dei pesi
V[] <- nuova matrice (n+1) x (P+1)
%inizializzazione
for i=0..n do
   V[i,0] = 0
for j=0..P do
   V[0,j] = 0
%riempimento matrice
for i=1..n
   for j=1..P do
      if(j<p[i]) then
         V[i,j] = V[i-1,j]
      else
         V[i,j] = max(V[i-1,j], V[i-1,j-p[i]]+v[i])
%soluzione
return V[n,P]
```

Si consideri la seguente tabella che associa ad ogni oggetto i un peso p_i ed un valore v_i. Dato uno zaino di capienza $P = 10$, si trovi una soluzione ottima per il problema dello zaino 0-1.

i	1	2	3	4
p_i	2	7	6	4
v_i	12.7	6.4	1.7	0.3

Soluzione:

Matrice V											
	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	12,7	12,7	12,7	12,7	12,7	12,7	12,7	12,7	12,7
2	0	0	12,7	12,7	12,7	12,7	12,7	12,7	12,7	19,1	19,1
3	0	0	12,7	12,7	12,7	12,7	12,7	12,7	14,4	19,1	19,1
4	0	0	12,7	12,7	12,7	12,7	13	13	14,4	19,1	19,1

Extra – è possible anche sapere quali oggetti appartengono alla soluzione dello zaino 0-1?

Sì, si deve utilizzare una matrice ausiliaria K (delle stesse dimensioni di V), che conterrà 1 se l'oggetto i-esimo fa parte della soluzione ottima che ha valore complessivo V[i, j]

```
Zaino(n,P,v[],p[]) // v[] e p[] sono i vettori dei valori e dei pesi
V[] <- nuova matrice (n+1) x (P+1)
K[] <- nuova matrice (n+1) x (P+1)
%inizializzazione
for i=0..n do
    V[i,0] = 0
    K[i,0] = 0
for j=0..P do
    V[0,j] = 0
    K[0,j] = 0
%riempimento matrice
for i=1..n
    for j=1..P do
        V[i,j] = V[i-1,j]
        K[i,j] =  0
        if V[i,j] < V[i-1,j-p[i]]+v[i] then
            V[i,j] = V[i-1,j-p[i]]+v[i]
            K[i,j] = 1
%soluzione
return V[n,P]
```

Extra – è possible anche sapere quali oggetti appartengono alla soluzione dello zaino 0-1?

Per sapere quali oggetti appartengono alla soluzione, visito K partendo dall'ultima cella (in fondo a destra)

```
d = P
i = n
while( i>0 ) do
   if K[i,d] = 1 then
      stampa "Seleziono oggetto" i
      d = d-p[i]
   i = i-1
```

	Matrice K (in verde le celle visitate)										
	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	1	1	1	1	1	1	1	1	1
2	0	0	0	0	0	0	0	0	0	1	1
3	0	0	0	0	0	0	0	0	1	0	0
4	0	0	0	0	0	0	1	1	0	0	0