Algo 2

Alessandro Zappatore

6 settembre 2024

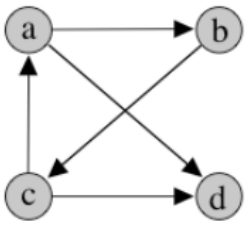
Indice

1 Rappresentazioni

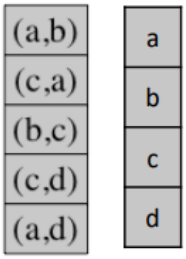
1.1 Lista di archi

Spazio occupato: $O(n+m)$

Operazione	**Tempo di esecuzione**
AggiungiVertice(v)	O(1)
AggiungiArco(x,y)	O(1)
RimuoviVertice(v)	O(m)
RimuoviArco(x,y)	O(m)
Grado(v)	O(m)
ArchiIncidenti(v)	O(m)
SonoIncidenti(x,y)	O(m)



Grafo orientato



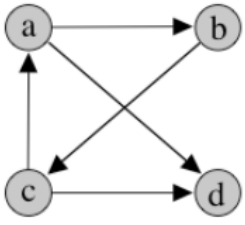
Rappresentazione

1.2 Liste di adiacenza

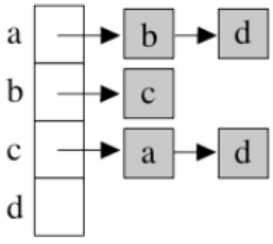
Spazio occupato: $O(n+m)$

Operazione	**Tempo di esecuzione**
AggiungiVertice(v)	O(1)
AggiungiArco(x,y)	O(1)
RimuoviVertice(v)	O(m)
RimuoviArco(e=(x,y))	O($\delta(x)$+$\delta(y)$)
Grado(v)	O($\delta(v)$)
ArchiIncidenti(v)	O($\delta(v)$)
SonoIncidenti(x,y)	O($min\{\delta(x), \delta(y)\}$)

Rappresentazione adatta per **grafi sparsi**



Grafo orientato

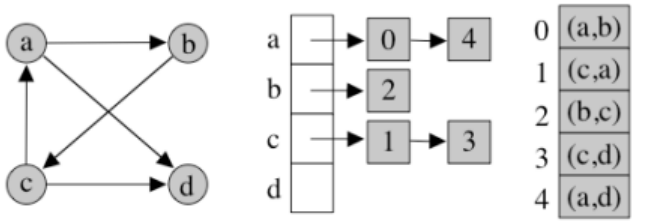


Rappresentazione

1.3 Liste di incidenza

Spazio occupato: $O(n + m)$

Operazione	Tempo di esecuzione
AggiungiVertice(v)	O(1)
AggiungiArco(x,y)	O(1)
RimuoviVertice(v)	O(m)
RimuoviArco(e=(x,y))	O($\delta(x)$+$\delta(y)$)
Grado(v)	O($\delta(v)$)
ArchiIncidenti(v)	O($\delta(v)$)
SonoIncidenti(x,y)	O($min\{\delta(x), \delta(y)\}$)



Grafo orientato Rappresentazione

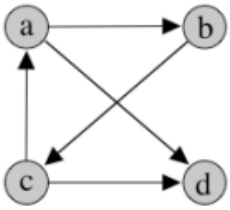
1.4 Matrice di adiacenza

Spazio occupato: $O(n^2)$

Operazione	Tempo di esecuzione
AggiungiVertice(v)	$O(n^2)$
AggiungiArco(x,y)	O(1)
RimuoviVertice(v)	$O(n^2)$
RimuoviArco(e)	O(1)
Grado(v)	O(n)
ArchiIncidenti(v)	O(n)
SonoIncidenti(x,y)	O(1)
ModificaVertice(v)	$O(n^2)$

Rappresentazione adatta per **grafi densi**.

- Calcolare il grado e archi incidenti: basta scorrere la matrice.
- Modificare un vertice: bisogna ricostruire completamente la matrice.
- Rappresenta la presenza di un cammino di lunghezza 1 tra ogni coppia di vertici. Moltiplicando la matrice per sè stessa, il risultato è diverso da 0 solo se esiste un cammino di lunghezza 2.

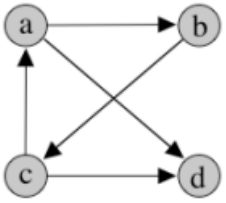


	a	b	c	d
a	0	1	0	1
b	0	0	1	0
c	1	0	0	1
d	0	0	0	0

1.5 Matrice di incidenza

Spazio occupato: O(n*m)

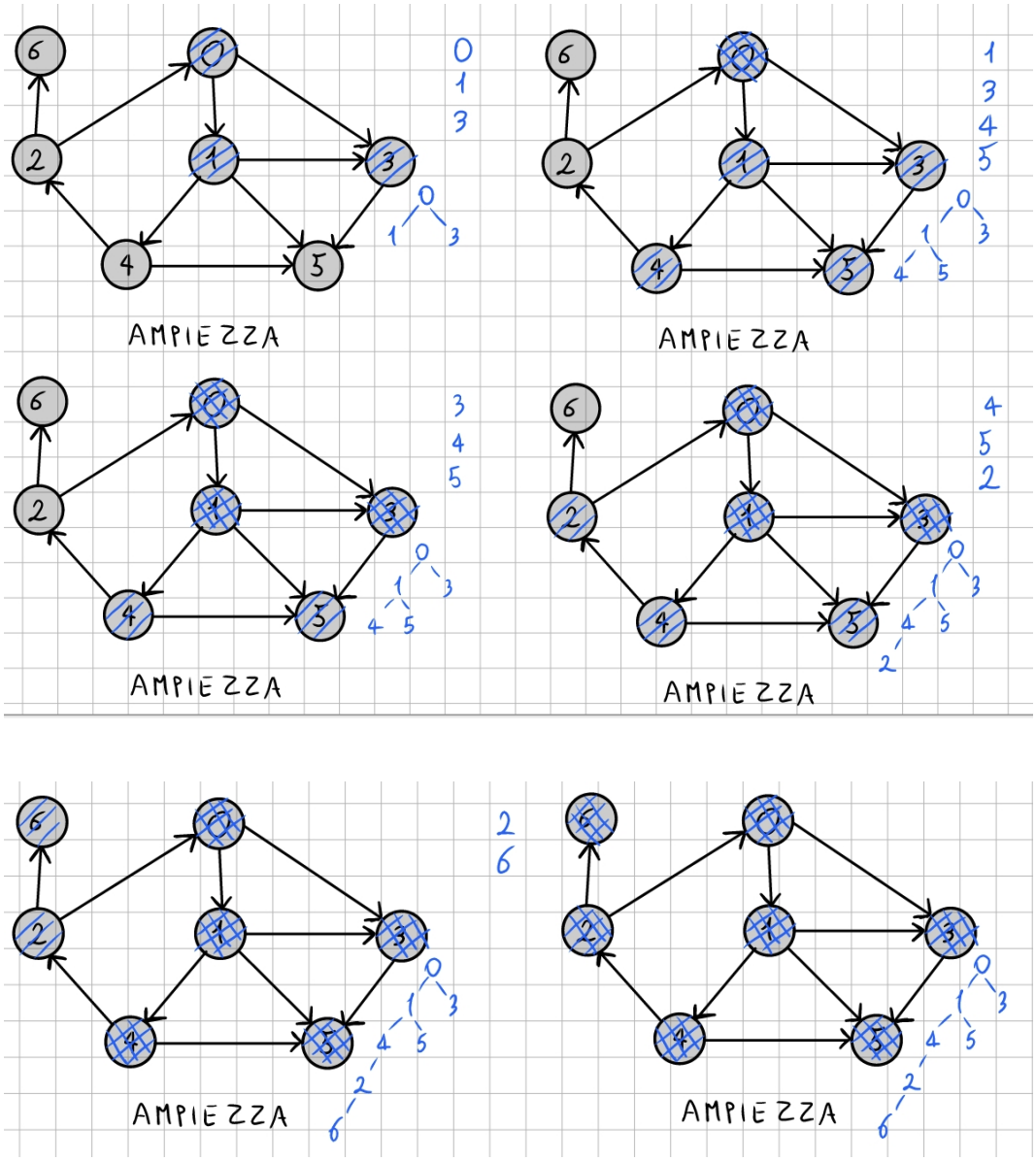
Operazione	**Tempo di esecuzione**
AggiungiVertice(v)	O(nm)
AggiungiArco(x,y)	O(nm)
RimuoviVertice(v)	O(nm)
RimuoviArco(e)	O(n)
Grado(v)	O(m)
ArchiIncidenti(v)	O(m)
SonoIncidenti(x,y)	O(m)



	(a,b)	(c,a)	(b,c)	(c,d)	(a,d)
a	1	-1	0	0	1
b	-1	0	1	0	0
c	0	1	-1	1	0
d	0	0	0	-1	-1

2 Visita in ampiezza (BFS)

- Esploro tutti i vicini del nodo e salvo l'ordine in una coda;
- Al passo successivo rimuovo il primo dalla coda ed esploro tutti i suoi vicini;
- Continuo finché la coda non è vuota.



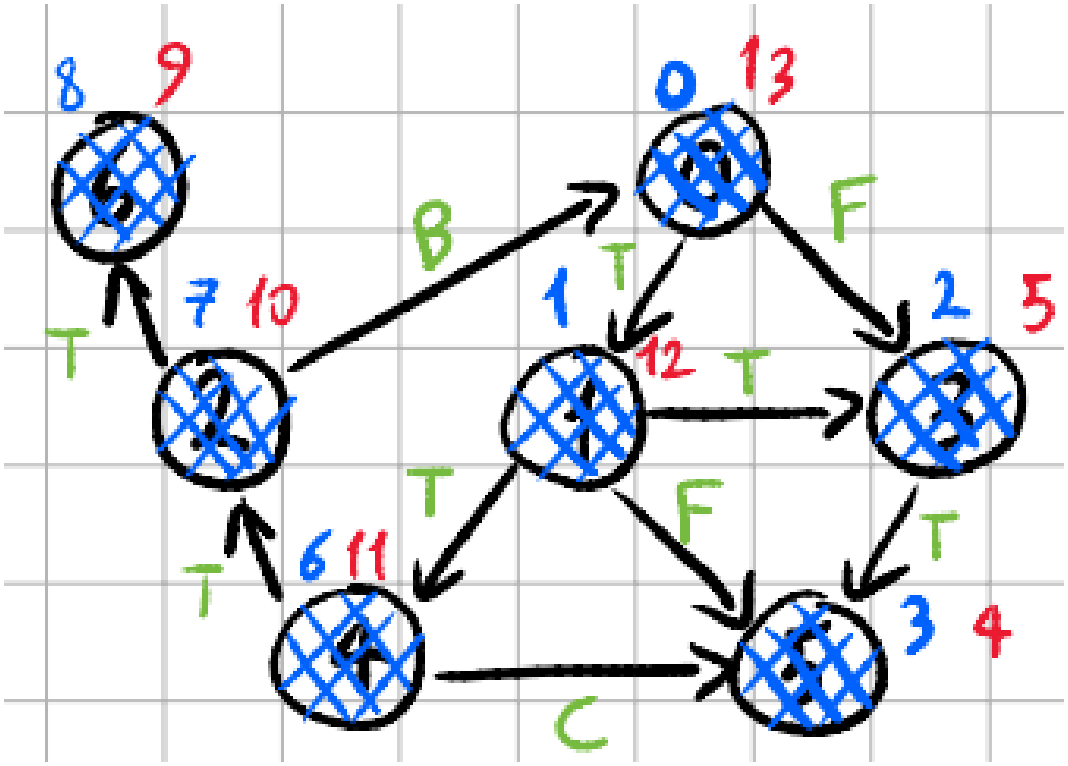
2.1 Caratteristiche BFS

- Nell'albero BFS ogni vertice si trova il più vicino possibile alla radice;
- L'albero di visita rappresenta i cammini minimi;
- Frangia come **coda**.

3 Verifica presenza di cicli

Esegui una visita DFS e contrassegna gli archi:

- **Arco dell'albero (T) :** Arco inserito nello foresta DFS (attraversato per la prima volta durante la visita)
- **Arco all'indietro (B) :** Arco che collega un vertice ad un suo antenato (collega un vertice appena scoperto ad uno già scoperto)
- **Arco in avanti (F) :** Arco che collega un vertice ad un suo discendente (già scoperto da un altro vertice durante la visita)
- **Arco di attraversamento (C) :** Arco che collega due vertici che non sono in relazione



In un grafo è presente un ciclo quando è presente un arco all'indietro.

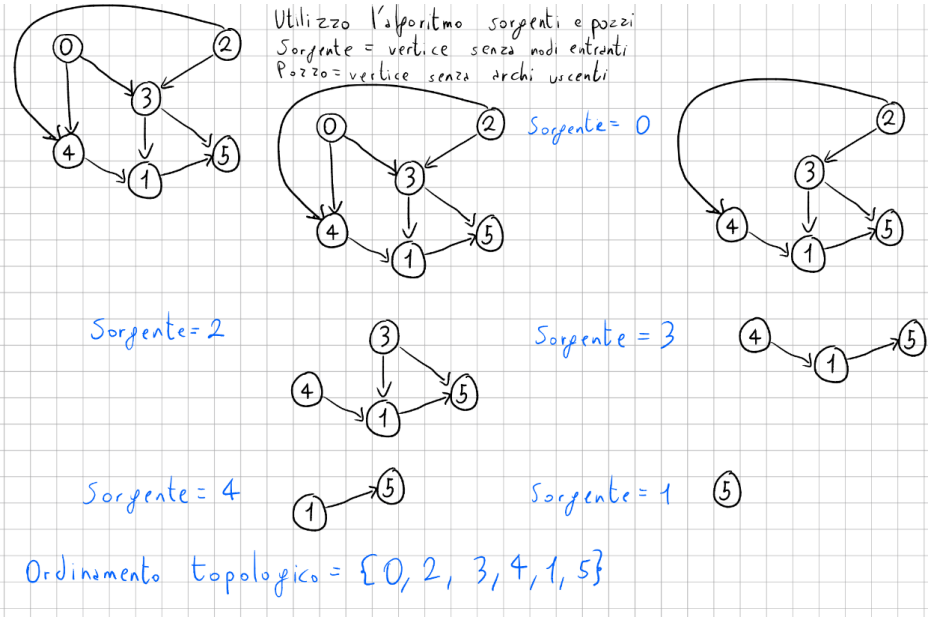
3.1 Caratteristiche DFS

- Un vertice viene chiuso solo quando i suoi discendenti sono stati chiusi;
- Gli intervalli di attivazione di una qualunque coppia di vertici sono o *disgiunti* o *uno contenuto interamente nell'altro*M
- Frangia come **stack**.

4 Ordinamento topologico

4.1 Sorgenti e pozzi

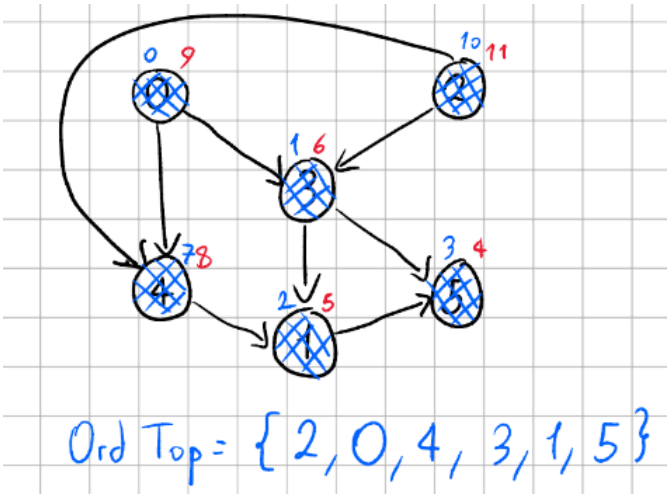
1. Disegnare il grafo orientato;
2. Individuare i nodi che non hanno archi entranti ed eliminarli;
3. Ripetere il ciclo fino a quando non sono finiti i nodi del grafo.



Complessità: $O(m * n)$

4.2 Basato su DFS

1. Esegui una visita DFS segnando i tempi di apertura e chiusura
2. Ordina i nodi per il tempo di chiusura dal più grande al più piccolo

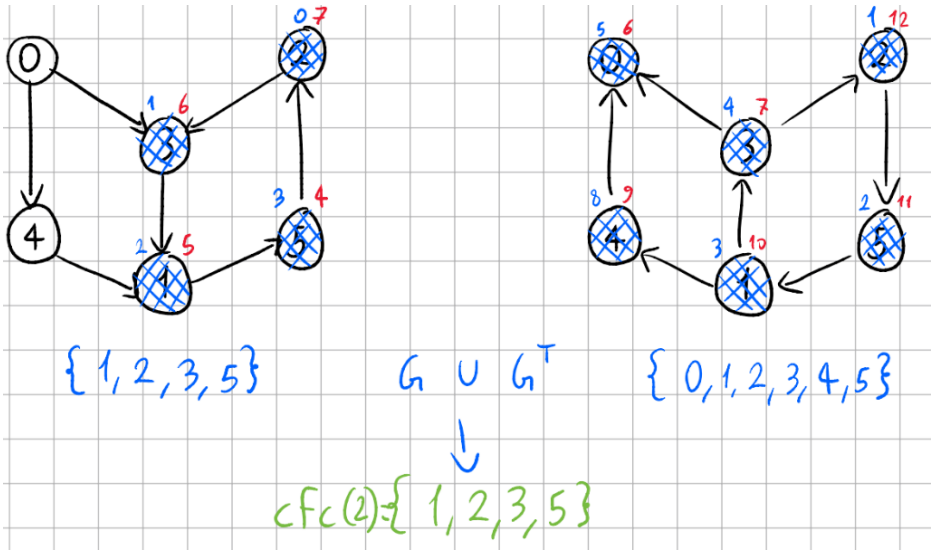


Complessità: $O(m + n)$

5 Componenti fortemente connesse

5.1 Metodo semplice

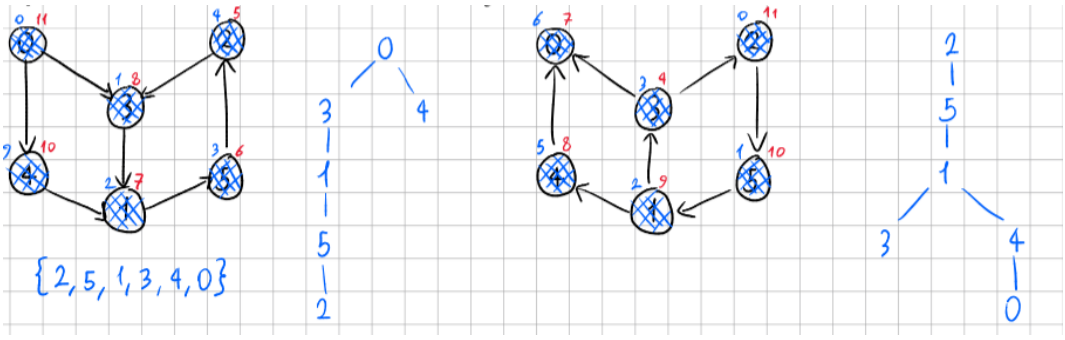
1. Disegnare il grafo originale e il suo grafo trasposto (grafo con la direzione dei vertici invertita);
2. Eseguire una visita DFS a partire dal vertice richiesto in entrambi i grafi;
3. Fare l'intersezione tra i due risultati.



Complessità: $O(n^2 + m)$

5.2 Kosaraju

1. Esegui una visita DFS per tutti i vertici e costruisci una lista di vertici in ordine decrescente dei tempi di fine visita;
2. Costruisci il grafo trasposto;
3. Esegui una visita DFS sul grafo trasposto seguendo l'ordine trovato al passo 1.



In una qualunque DFS di un grafo F orientato, tutti i vertici di una cfc vengono collocati in uno stesso sottoalbero.

Complessità: $O(n + m)$

6 Problema dello zaino frazionario

GREEDY Appetibilità fissa, v_i

1. Calcolo $v_i = \frac{c_i}{v_i}$;
2. Scelgo l'oggetto con v_i maggiore e sottraggo il suo peso al peso totale P (si possono prendere parti frazionarie dello zaino);
3. Continuo fino a che P non è = a 0.

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

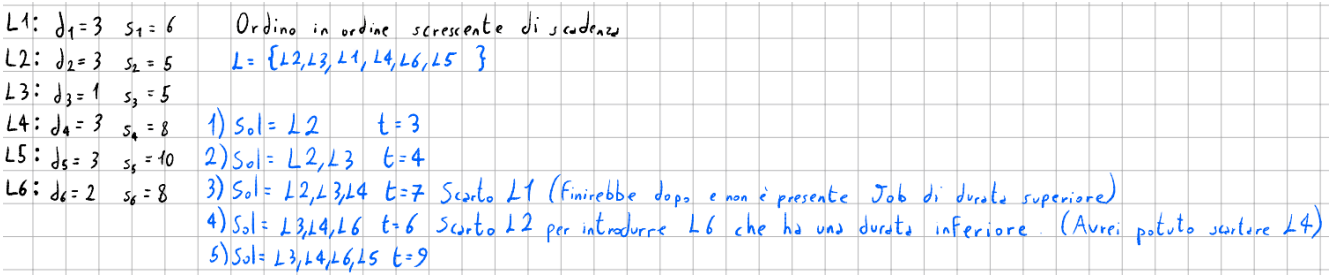
Algoritmo greedy scelgo in ordine di appetibilità v_i

Passo 1: Scelgo $i = 4$ $P = 80 - 10 = 70$ Tutto
Passo 2: Scelgo $i = 1$ $P = 70 - 10 = 60$ //
Passo 3: Scelgo $i = 2$ $P = 60 - 20 = 40$ //
// 4: // $i = 3$ $P = 40 - 30 = 10$ //
// 5: // $i = 6$ $P = 10 - 10 = 0$. Metà

7 Massimo numero di intervalli disgiunti (Moore)

Greedy Appetibilità fissa, istante di fine crescente

1. Ordina l'insieme degli intervalli in una sequenza secondo l'istante di fine;
2. Scandisci la sequenza in ordine e:
   - se A inizia dopo la fine dell'ultimo elemento di Sol, aggiungilo in fondo a Sol;
   - altrimenti non aggiungerlo.

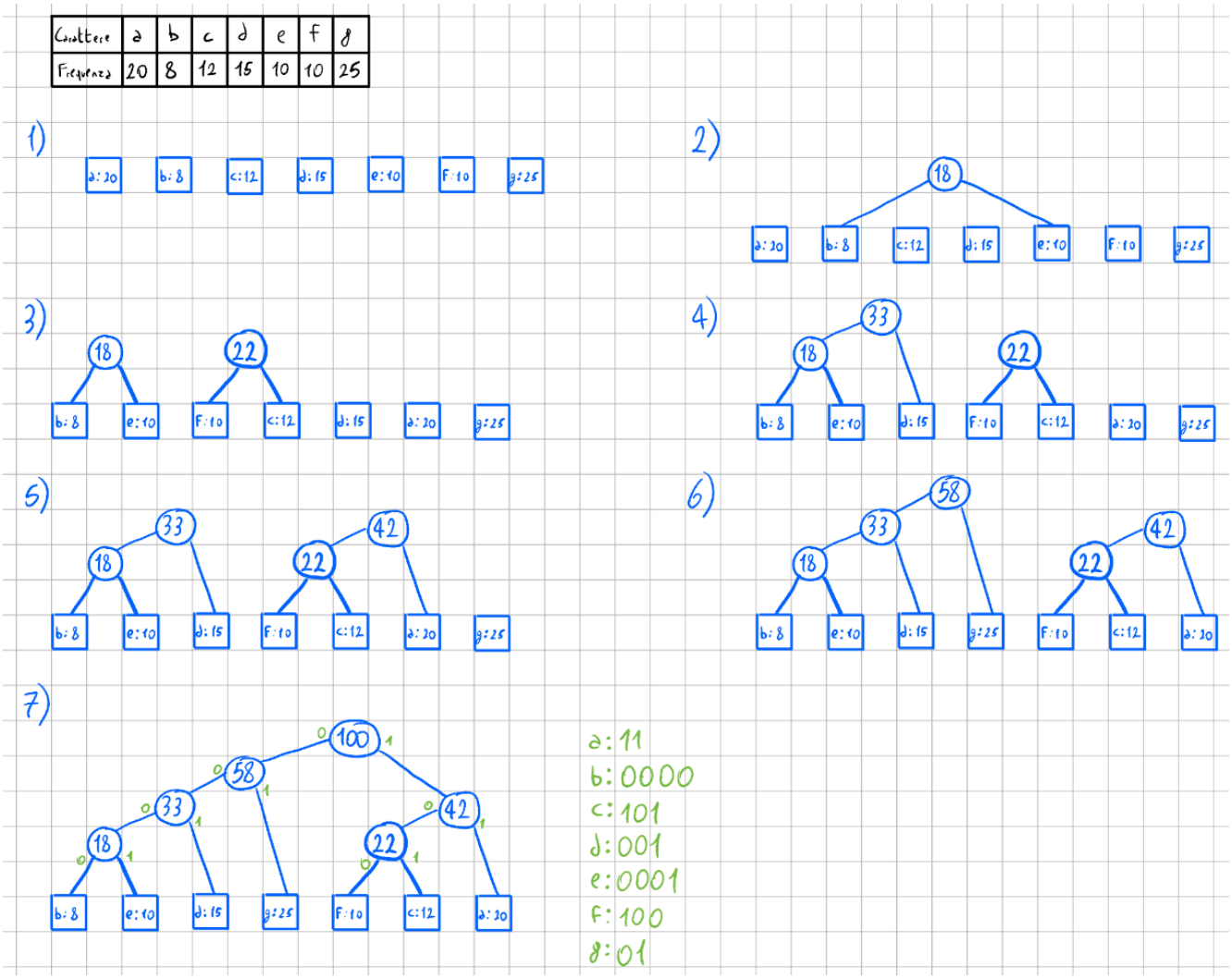


Complessità: $O(nlogn)$

8 Codifica a lunghezza variabile (Huffman)

Greedy Appetibilità modificabile, frequenze minime

1. Disegnare le foglie (rappresentate come la lettera e la sua frequenza);
2. Unire progressivamente le foglie con frequenza minore tra di loro;
3. Seguire l'albero e scrivere le codifiche per ogni lettera.

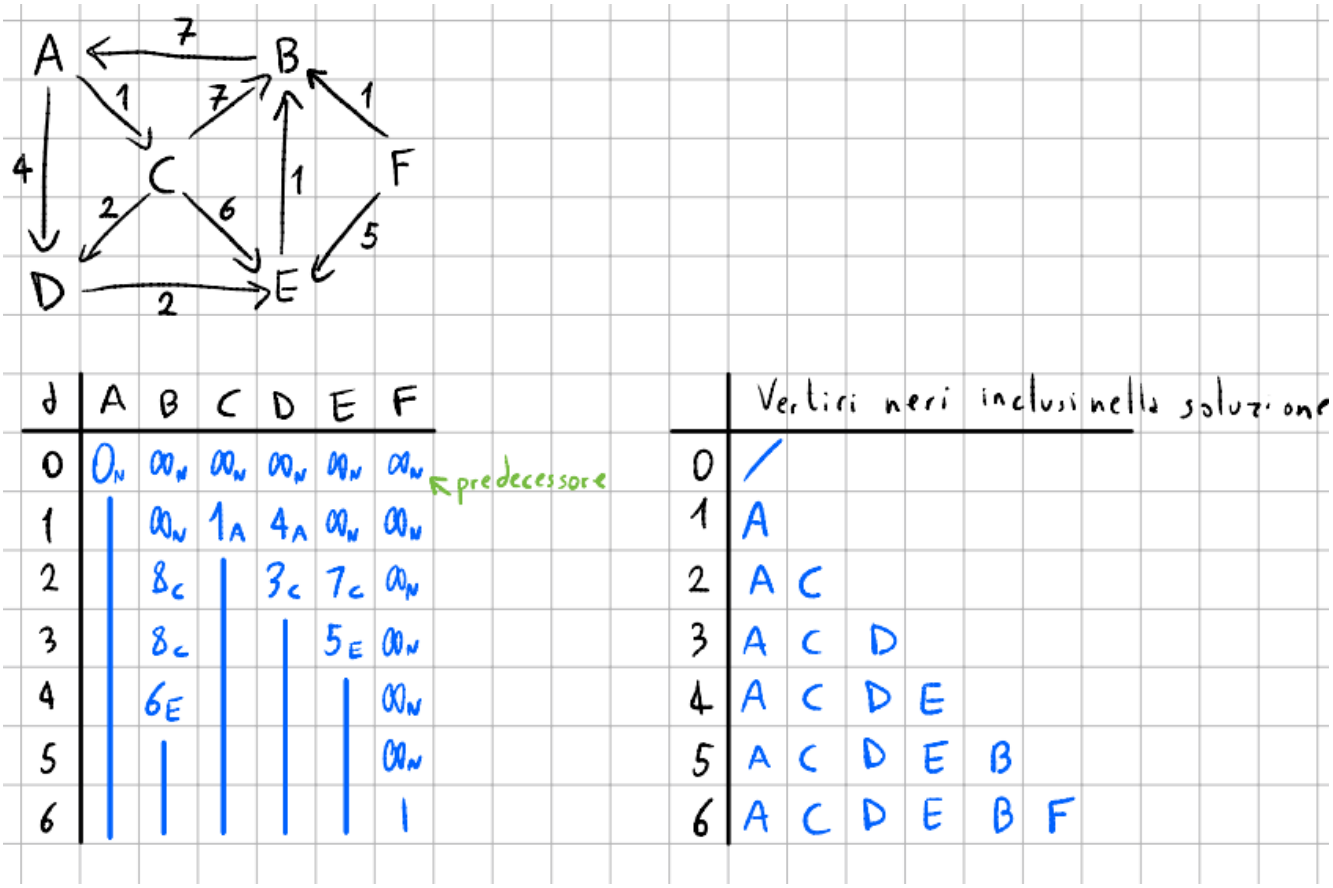


Complessità: $O(logn)$

9 Dijkstra

Greedy Appetibilità modificabile, stima della distanza tra s e u.

1. Inizializzo la distanza di ogni nodo ad infinito;
2. Ad ogni passaggio trovo il nodo con distanza dal vertice di partenza inferiore;
3. Se alla fine rimane un vertice scoperto lo aggiungo in coda.



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

	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

È possibile non distinguere nodi bianchi e grigi ed inserire tutti nella coda fin dall'inizio, assegnando loro distanza infinita da s. I nodi neri sono detti definitivi e quelli bianchi e grigi non definitivi. Non è necessario distinguere i nodi neri, poiché non saranno in coda.

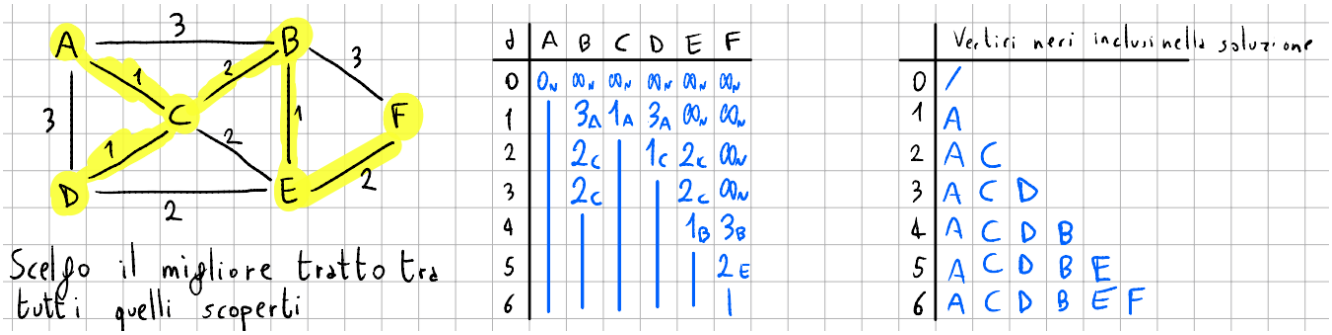
Complessità:

- con coda di priorità realizzata come sequenza non ordinata: $O(n^2 + m)$
- con coda di priorità realizzata come sequenza ordinata: $O(n + n * m)$
- versione di Johnson (coda di priorità implementata come **heap**): $O((m + n)logn)$

10 Prim

Greedy Appetibilità modificabile, stima della distanza del nodo dall'albero di visita.

1. Inizializzo la distanza di ogni nodo ad infinito;
2. Ad ogni passaggio controllo che la distanza sia minore dal passaggio precedente:
   - Se la distanza è minore utilizzo la nuova distanza minore;
   - Altrimenti utilizzo la distanza scoperta precedentemente.
3. Continuo fino ad aver scoperto tutti i nodi del grafo



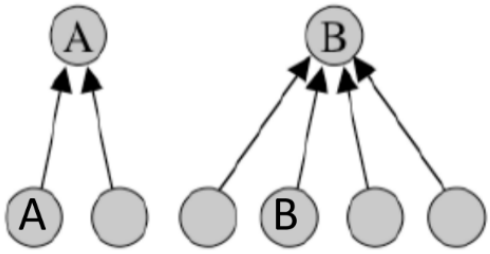
Non è possibile eliminare la gestione dei nodi neri in quanto bisogna controllare che tra gli adiacenti del nodo u estratto dalla coda non vi sia il genitore di u.

Complessità: stessa di quella dell'algoritmo di Johnson, ma il grafo è connesso ($m \geq n-1$), quindi $O(mlogn)$

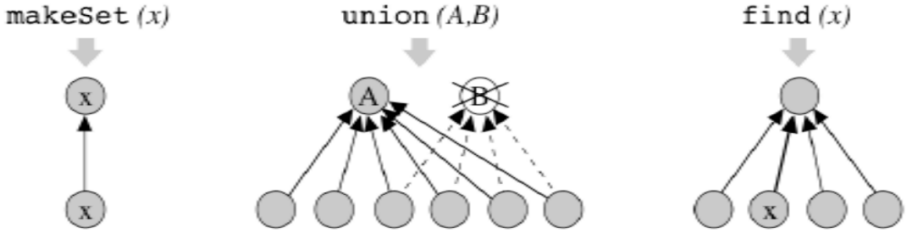
11 Union find

11.1 Quick Find

Alberi con 2 livelli (Il rappresentate è contenuto sia nella che in una foglia).

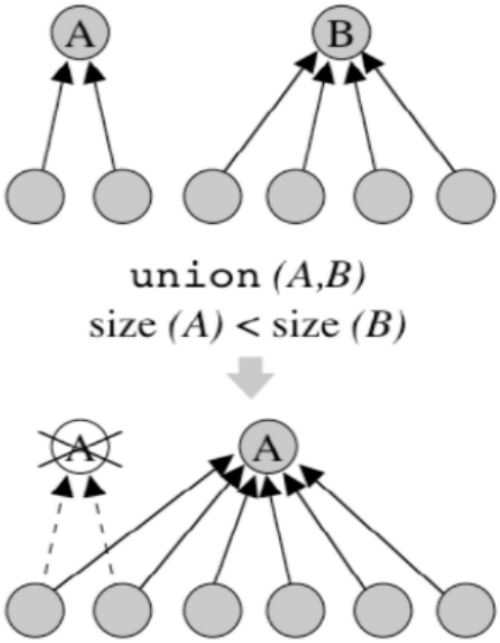


11.1.1 Operazioni Quick Find



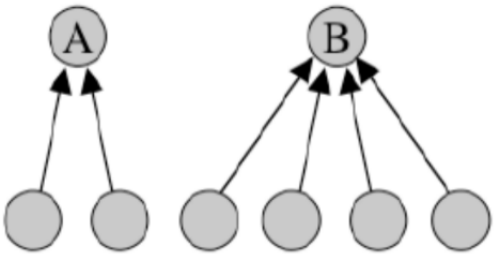
11.1.2 Quick Find bilanciamento

Salviamo una variabile **size** che aumenta di 1 ad ogni aggiunta di nodi

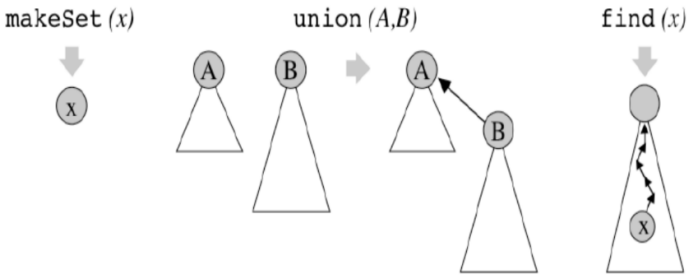


11.2 Quick Union

Alberi con più di 2 livelli (Il rappresentate sta solo nella radice).

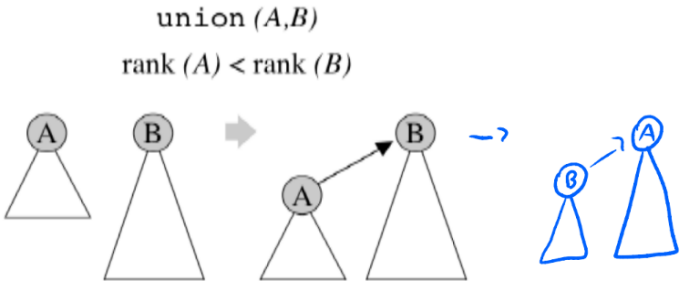


11.2.1 Operazioni Quick Union



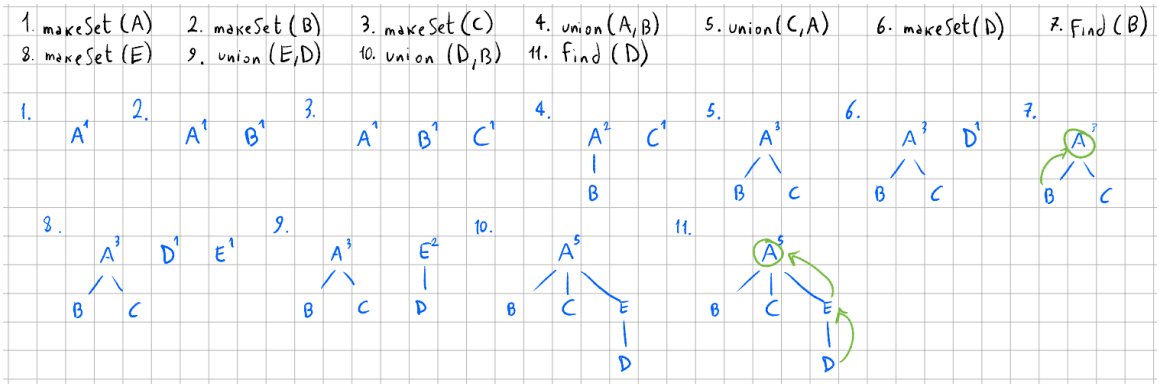
11.2.2 Union by Rank

Rendiamo la radice dell'albero più basso figlia della radice dell'albero più alto.



11.2.3 Union by Size

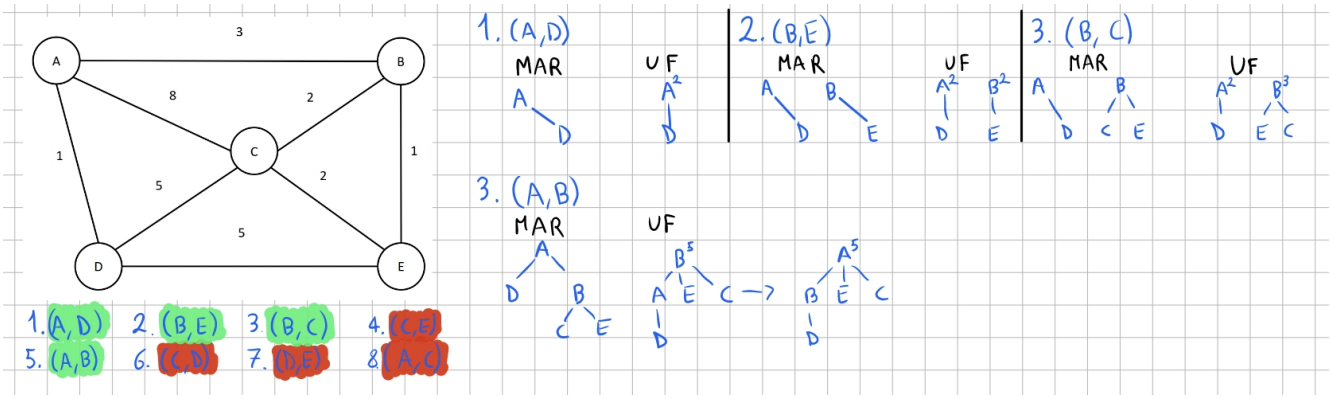
Rendiamo la radice dell'albero con meno nodi figlia della radice dell'albero con più nodi.



12 Minimo albero ricoprente (Kruskal)

Greedy Appetibilità fissa, peso degli archi in ordine crescente.

1. Metto in ordine di peso gli archi, dal più piccolo al più grande;
2. Per ogni arco creo una Union Find:
   - Se i due nodi fanno parte della stessa UF scarto l'arco;
   - Altrimenti faccio la makeset oppure la union se esistono già (attenzione all'ordine delle union).
3. Creo il mar seguendo i nodi contenuti nella Union Find.



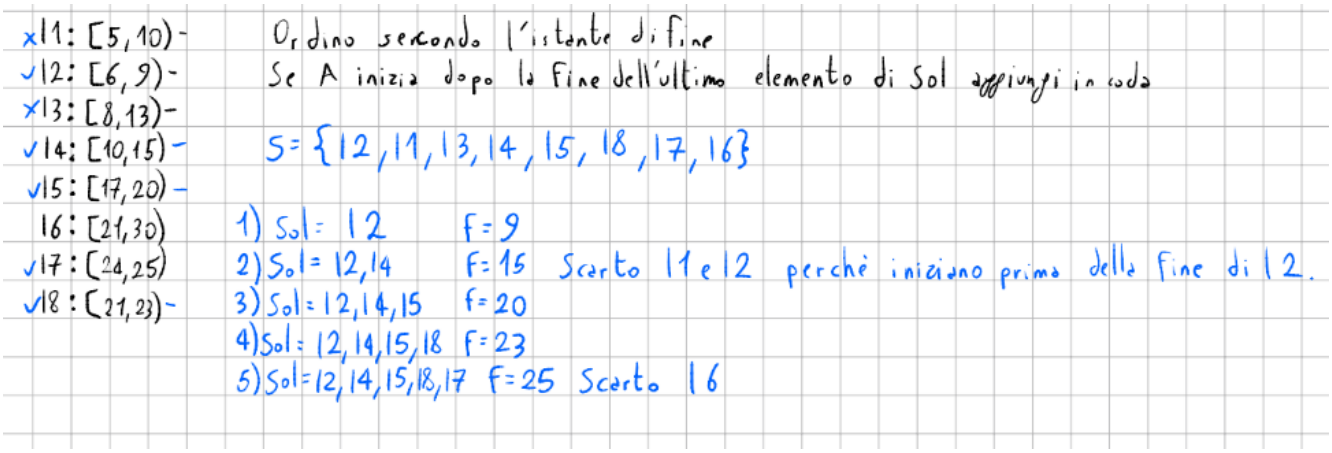
Complessità: $O(mlogn)$

13 Massimo Sottoinsieme Indipendente MSI

Programmazione dinamica Vettore.

Ad ogni nodo v_i è associato un peso non negativo w_i. Si determini il sottoinsieme indipendente di nodi del grafo che ha massimo peso.

MSI(n,A)
A[0] $\leftarrow \{\}$
A[1] $\leftarrow \{V_1\}$
for $i = 2, \ldots, n$ **do**
  $A[i] \leftarrow maxPeso(A[i-1], A[i-2] + V_i)$
end for
return $A[n]$



14 Longest Common Subsequence LCS

Programmazione dinamica Matrice [m+1][n+1]. LCS[i,j] contiene la più lunga sotto sequenza comune dei due segmenti iniziali di lunghezza rispettive i e j.

14.1 Matrice LCS

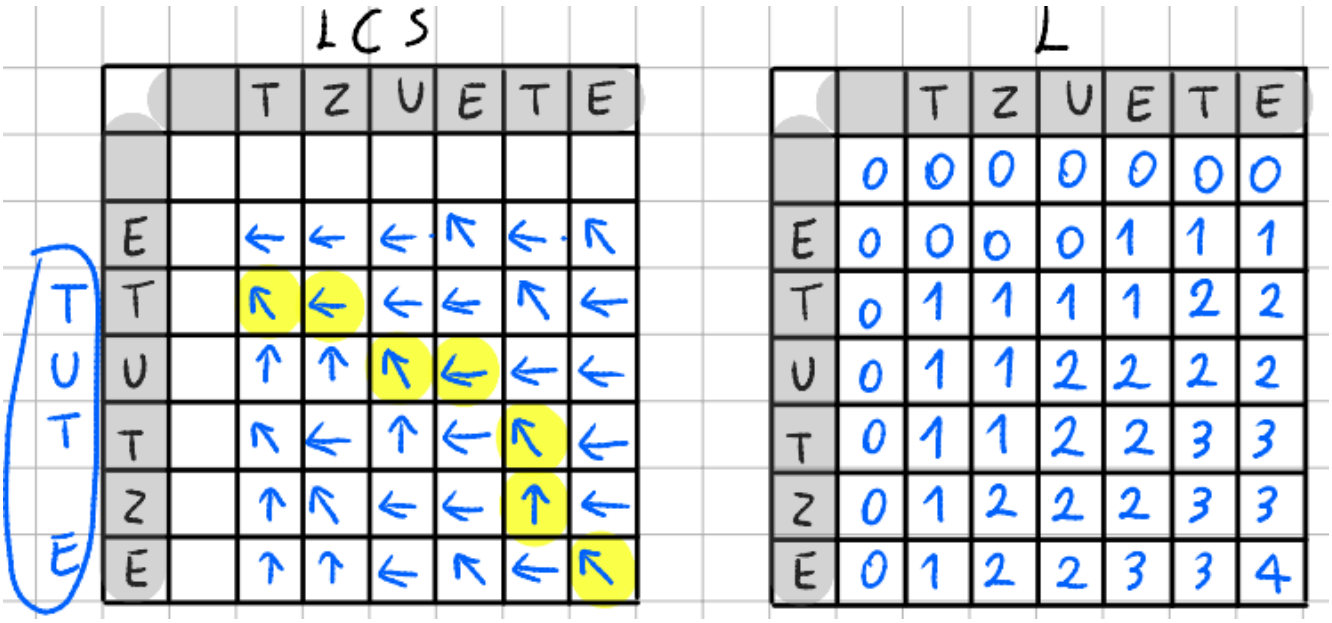
- $\leftarrow$ **:** Il numero presente nella matrice L è maggiore a sinistra;
- $\uparrow$ **:** Il numero presente nella matrice L è maggiore a destra;
- $\nwarrow$ **:** La lettera è uguale.

14.2 Matrice L

- Se le lettere sono diverse ricopio il massimo tra sinistra e in alto;
- Se la lettera è uguale scrivo il numero + 1 rispetto a quello della diagonale in alto a sinistra.

14.3 Fine

- Parto dal punto in basso a destra della matrice LCS e seguo il verso delle frecce;
- Quando trovo frecce $\nwarrow$ le aggiungo alla sotto sequenza più lunga.



Complessità: costruzione e popolamento matrice $O(mn)$, ricostruzione della soluzione $O(m+n)$ costo totale $O(mn)$

15 Zaino 0-1

Programmazione dinamica Matrice [n+1][P+1], soluzione contenuta in V[n, P].

- Controllo se: $V[i, w] = max\{v[i-1, w], V[i-1, w-w[i]] + p[i]\}$

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

Soluzione:

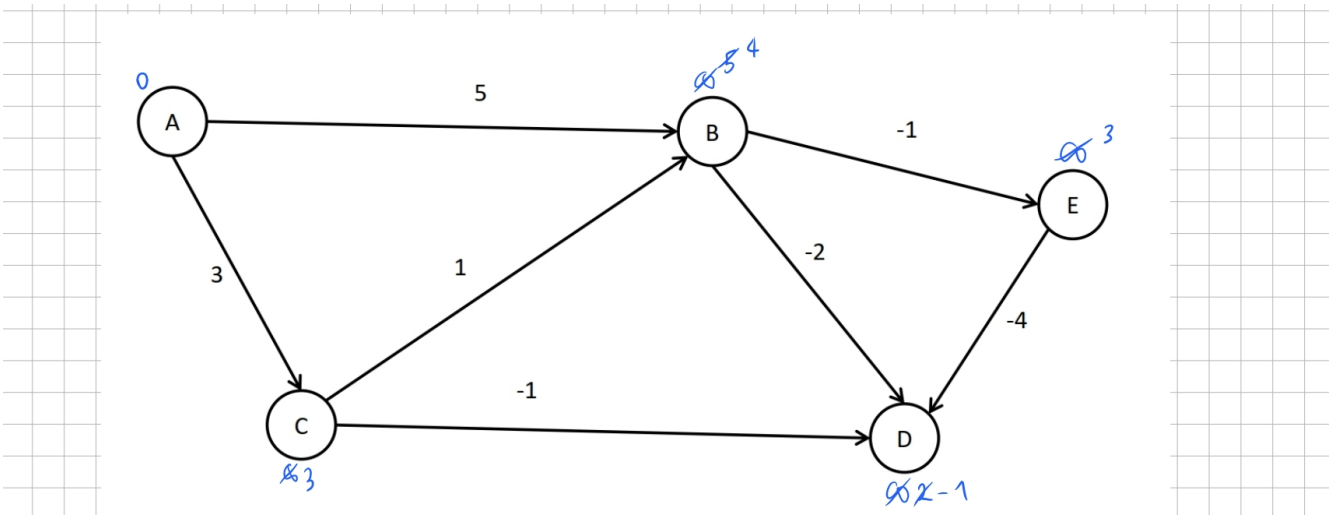
Matrice V											
	0	1	2	3	4	5	6	7	8	9	10
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	12,7	12,7	12,7	12,7	12,7	12,7	12,7	12,7	12,7
2	0	0	12,7	12,7	12,7	12,7	12,7	12,7	12,7	19,1	19,1
3	0	0	12,7	12,7	12,7	12,7	12,7	12,7	14,4	19,1	19,1
4	0	0	12,7	12,7	12,7	12,7	13	13	14,4	19,1	19,1

16 Cammini minimi (Bellman Ford)

Programmazione dinamica

16.1 Algoritmo normale

1. Scrivo tutti gli archi presenti nel grafo;
2. Per ogni arco controllo se $d[u] + c(u, v) < d[v]$, se vero sostituisco $d[v]$
3. Ciclo n - 1 volte, se alla n esima volta cambia nuovamente vuol dire che sono in presenza di un ciclo negativo.



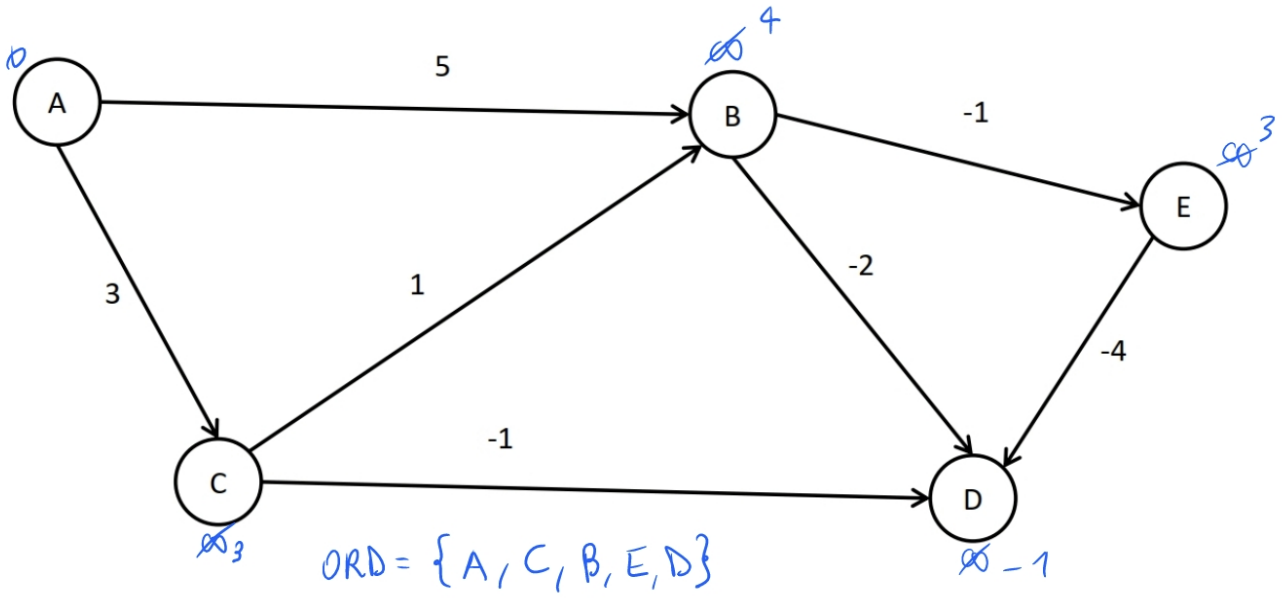
d	init	(A,B) 5	(A,C) 3	(C,D) -1	(C,B) 1	(E,D) -4	(B,E) -1	(B,D) -2
A	0	0_N	0_N	0_N	0_N	0_N	0_N	0_N
B	∞	5_A	5_A	5_A	4_C	4_C	4_C	4_C
C	∞	∞_N	3_A	3_A	3_A	3_A	3_A	3_A
D	∞	∞_N	∞_N	2_C	2_C	2_C	2_C	2_C
E	∞	∞_N	∞_N	∞_N	∞_N	∞_N	3_B	3_B

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

Complessità: $O(n * m)$

16.2 Ottimizzazione per i DAG

1. Calcolo un ordinamento topologico per il grafo;
2. Rilasso gli archi una sola volta seguendo l'ordine topologico.



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

Complessità: $O(n+m)$

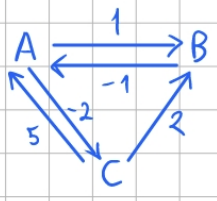
17 Distanze e cammini minimi k-vincolati (Floyd Warshall)

- Per ogni nodo creo una matrice e controllo se il peso dell'arco è maggiore del peso dell'arco sommandolo al peso dell'arco passando dal nodo k. $A^k[i,j] = min\left\{A^k - 1[i,j], A^k - 1[i,k] + A^k - 1[k,j]\right\}$

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

$D^0[A,A]\ 0 = D^0[A,A]^0 + D^0[A,A]^0$
$D^0[A,B]\ 1 = D^0[A,A]^0 + D^0[A,B]^1$
$D^0[A,C]\ -2 = D^0[A,A]^0 + D^0[A,C]^{-2}$
$D^0[B,A]\ -1 = D^0[B,A]^{-1} + D^0[A,A]^0$
$D^0[B,B]\ 0 = D^0[B,A]^{-1} + D^0[A,B]^1$
$D^0[B,C]\ \infty > D^0[B,A]^{-1} + D^0[A,C]^{-2}$
$D^0[C,A]\ 5 = D^0[C,A]^5 + D^0[A,A]^0$
$D^0[C,B]\ 2 < D^0[C,A]^5 + D^0[A,B]^1$
$D^0[C,C]\ 0 < D^0[C,A]^5 + D^0[A,C]^{-2}$



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

$D^A[A,A]\ 0 = D^A[A,B]^1 + D^A[B,A]^{-1}$
$D^A[A,B]\ 1 > D^A[A,B]^1 + D^A[B,A]^{-1}$
$D^A[A,C]\ -2 = D^A[A,B]^1 + D^A[B,C]^{-3}$
$D^A[B,A]\ -1 = D^A[B,B]^0 + D^A[B,A]^{-1}$
$D^A[B,B]\ 0 = D^A[B,B]^0 + D^A[B,B]^0$
$D^A[B,C]\ -3 = D^A[B,B]^0 + D^A[B,C]^{-3}$
$D^A[C,A]\ 5 > D^A[C,B]^2 + D^A[B,A]^{-1}$
$D^A[C,B]\ 2 = D^A[C,B]^2 + D^A[B,B]^0$
$D^A[C,C]\ 0 > D^A[C,B]^2 + D^A[B,C]^{-3}$

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

$D^B[A,A]\ 0 > D^B[A,C]^{-2} + D^B[C,A]^1$
$D^B[A,B]\ 0 = D^B[A,C]^{-2} + D^B[C,B]^2$
$D^B[A,C]\ -2 > D^B[A,C]^{-2} + D^B[C,C]^{-1}$
$D^B[B,A]\ -1 > D^B[B,C]^{-3} + D^B[C,A]^1$
$D^B[B,B]\ 0 > D^B[B,C]^{-3} + D^B[C,B]^2$
$D^B[B,C]\ -3 > D^B[B,C]^{-3} + D^B[C,C]^{-1}$
$D^B[C,A]\ 1 > D^B[C,C]^{-1} + D^B[C,A]^1$
$D^B[C,B]\ 2 > D^B[C,C]^{-1} + D^B[C,B]^2$
$D^B[C,C]\ -1 > D^B[C,C]^{-1} + D^B[C,C]^{-1}$

In questo caso vi è un ciclo negativo, quindi **non** esistono cammini minimi.

18 Costruzione di algoritmi (programmazione dinamica)

1. Descrivere la struttura dati necessaria per la memoizzazione (Numero di parametri);
2. Definire i casi base e le loro soluzioni (quelli che non presentano ricorsione);
3. Scrivere l'algoritmo che inizializzi la struttura di memoizzazione seguendo i casi base, e successivamente la popoli in maniera bottom up. Alla fine, restituisce il valore che corrisponde alla soluzione;
4. Capire dove si trova la soluzione nella struttura di memoizzazione.

Esempio:

Una stringa s si dice palindroma se è uguale letta da sinistra verso destra o viceversa. Considerata una stringa s, indicizzata come un array, e due indici x e y, la funzione Pal(x,y,s) restituisce il minimo numero di caratteri necessari da aggiungere per rendere s paindroma, (es, con s = "casa", Pal(0,3,"casa") = 1 perché "casaC" è palindroma).

$$Pal(x,y,s) = \begin{cases} 0 & se\ x = y \vee x > y \\ Pal(x+1,y-1,s) & se\ s[x] = s[y] \\ 1 + \min(Pal(x+1,y,s), Pal(x,y-1,s)) & altrimenti \end{cases}$$

Si scriva un algoritmo di programmazione dinamica che, per una qualsiasi stringa s, dati x = 0 (indice del primo carattere) e y = s.length-1 (indice dell'ultimo carattere), calcoli Pal(x,y,s) per s. In particolare:

1. Si descriva la struttura dati necessaria per la memoizzazione;
2. Si definiscano i casi base, e le loro soluzioni;
3. Si dica dove, nella struttura di memoizzazione, sarà presente la soluzione;
4. Si scriva in pseudo codice un algoritmo di programmazione dinamica che risolve il problema.

Soluzione:

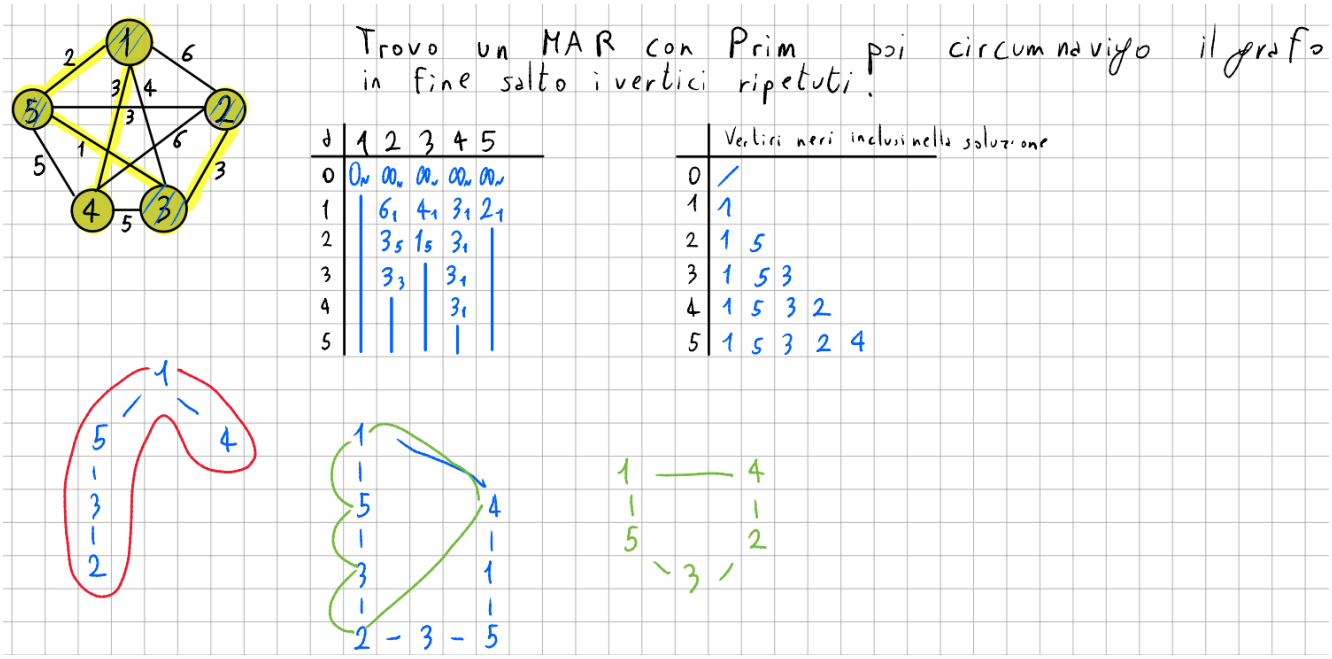
1. La struttura necessaria per la memoizzazione è una matrice di dimensione (s+1)x(s+1), dato che sono presente due parametri (x e y);
2. I casi base sono quelli che non presentano ricorsione: sono quindi x=y o x¿y. La loro soluzione è 0(se la stringa è di lunghezza 0 o 1), altrimenti la parola ricercata, in quanto l'algoritmo conclude le sue chiamate ricorsive;
3. La soluzione sarà presente nella prima cella in alto a destra.

```
PAL(x,y,s)
n = s.length
p[ ] ← nuova matrice (s+1)x(s+1)
for i = 0, . . . , n do
    p[i,0] = 0
end for
for j = 0, . . . , n do
    p[0,j] = 0
end for
```

```
for i = 1, ..., n do
    for j = n, ..., 1 do
        if s[i − 1] == s[j − 1] then
            p[i,j] ← p(i+1, j-1, s)
        else
            p[i,j] ← 1+min(p(i+1, j, s), p(i, j-1, s))
        end if
    end for
end for
return p[x,y]
```

19 Ciclo Hamiltoniano

1. Scegli un vertice casuale del grafo;
2. Esegui l'algoritmo di Prim per trovare un MAR;
3. Percorri il MAR segnando i nodi attraversati;
4. Ripercorri il grafo ottenuto saltando i nodi duplicati.



20 Algoritmi di approssimazione

20.1 Copertura dei vertici

Decidere se una certa copertura è ottima è un problema **NP-completo**. APPROX-COVER(G):

C $\leftarrow$ insieme vuoto
$E' \leftarrow E$
while $E' \neq \emptyset$ **do**
    sia (u, v) un arco in E'
    $C \leftarrow C \cup \{u, v\}$
    rimuovi da E' ogni arco incidente in u o v
end while
return C

Fattore di approssimazione: $2 - approssimato$.

20.2 Problema del commesso viaggiatore

Trovare un ciclo Hamiltoniano di costo minimo. Capire se una soluzione è ottima è esso stesso **NP-completo**. APPROX-TSP(G,W,r):

scegli un vertice r casualmente
$A \leftarrow PRIM(G, W, r)$
$ord \leftarrow DFS(A, r)$
return ord

Fattore di approssimazione: $2 - approssimato$.

20.3 Ricerca locale

Tecnica generale in un problema di ottimizzazione:

1. si parte da una soluzione ammissibile x
2. si calcolano tutte le soluzioni in un suo vicinato N(x)
3. se esiste una soluzione y appartenente a N(x) migliore di x, allora $x \leftarrow y$ e si riparte dal punto 2
4. altrimenti, x è un minimo (o massimo) locale e ci possiamo fermare

Ricerca locale per TSP Tecnica dei k-scambi:

1. si genera un ciclo Hamiltoniano casuale
2. se ne cancellano k archi non adiacenti e se ne aggiunto altri k in modo da ricreare il ciclo
3. se così facendo si crea un ciclo di peso minore, si ricomincia l'iterazione dal punto 2
4. altrimenti, si è trovata una soluzione minima localmente