Indice

Capitolo 1
Programmazione dinamica su stringhe

1.1 Longest Common Subsequence

1.1.1 Definizione del problema

Data una sequenza X, si dice **sottosequenza** di X un qualsiasi sottoinsieme degli elementi di X, disposti secondo l'ordine in cui appaiono. Più formalmente, una sottosequenza di lunghezza n di una sequenza X è definita da un insieme di indici $i_1 < i_2 < ... < i_k$, tale per cui $k \leq n$; la sottosequenza che si ricava da questi indici è $X[i_1] \mid S[i_2] \mid ... \mid S[i_k]$.
Date due sequenze X e Y, una sottosequenza si dice comune ad entrambe le sequenze se appare sia in X che in Y. Il problema della sottosequenza comune più lunga consiste nel trovare, in due sequenze X ed Y, qual'è (se esiste) la sottosequenza comune ad entrambe che sia di lunghezza maggiore, abbreviata con *LCS* (*longest common subsequence*).
Anziché affrontare il problema direttamente, é piú efficiente prima trovare la lunghezza di una possibile LCS di due sequenze e poi, sulla base di questa informazione, ricostruire la LCS "a ritroso". Pertanto, il problema effettivamente in esame sará il problema della lunghezza della sottosequenza comune piú lunga.
Il problema puó essere risolto mediante programmazione dinamica. Date due sequenze X e Y rispettivamente di lunghezza m e n, siano $X_i = X[1 : i]$ e $Y_j = Y[1 : j]$ i prefissi di lunghezza i e j delle rispettive sequenze. La soluzione $S_{i,j}$ per l'i, j-esima istanza del problema corrisponde a trovare la soluzione ottimale per il problema della lunghezza della LCS rispetto alle sottosequenze X_i e Y_j. La soluzione per le intere sequenze X e Y è la soluzione per l'istanza X_m, Y_n.
Per calcolare la soluzione ottimale della generica istanza X_i, Y_j, ovvero $S_{i,j}$, verrá assunto di avere a disposizione tutte le soluzioni parziali nella forma $S_{a,b}$, con $1 \leq a < i$ e $1 \leq b < j$. Pertanto, per il calcolo della soluzione ottimale dell'i, j-esima istanza del problema si hanno a disposizione l'i-esimo carattere della sequenza X, il j-esimo carattere della sequenza Y e le lunghezze delle LCS delle istanze da $i-1, j-1$ a 0, 0, ma non gli elementi delle due sequenze che comportano tali lunghezze. x_i e y_j possono essere come possono non essere elementi della sottosequenza di X_i e Y_j di lunghezza $S_{i,j}$.

1.1.2 Programmazione dinamica: sottostruttura ottima

Proprietà della sottostruttura ottima per il problema Longest Common Subsequence. Siano $X = \langle x_1, x_2, ..., x_m \rangle$ e $Y = \langle y_1, y_2, ..., y_n \rangle$ due sequenze. Sia poi $Z = \langle z_1, z_2, ..., z_k \rangle$ una LCS di X e di Y. Possono presentarsi tre situazioni:

1. Se $x_m = y_n$, allora $x_m = y_n = z_k$ e Z_{k-1} è una LCS di X_{m-1} e Y_{n-1};
2. Se $x_m \neq y_n$ e $x_m \neq z_k$, allora Z è una LCS di X_{m-1} e Y;
3. Se $x_m \neq y_n$ e $y_n \neq z_k$, allora Z è una LCS di X e Y_{n-1};

Dimostrazione. La dimostrazione procede caso per caso:

1. Si supponga per assurdo che $x_m = y_n \neq z_k$. Si accodi allora $x_m = y_n$ a Z, ottenendo la sequenza $Z' = \langle z_1, z_2, ..., z_k, x_m \rangle$. Tale sequenza ha lunghezza $k + 1$ e, essendo $x_m = y_n$, è una LCS per X e Y. Ma allora Z non può essere una LCS per X e Y, perché Z ha lunghezza k mentre Z' ha lunghezza $k + 1$. Questo è in contraddizione con l'ipotesi di partenza secondo cui Z è effettivamente una LCS, pertanto deve aversi $x_m = y_n = z_k$.
   Dato che per ipotesi $x_m = y_n$, il prefisso Z_{k-1} è una sottosequenza comune di X_{m-1} e Y_{n-1} avente lunghezza $k-1$. Si vuole dimostrare che Z_{k-1} è una LCS di X_{m-1} e Y_{n-1}. Si supponga per assurdo che questo non sia vero, e che esista quindi una sequenza W che è LCS di X_{m-1} e Y_{n-1} avente lunghezza maggiore di $k-1$. Allora, accodando $x_m = y_n$ a W si ottiene la sequenza W', che è sottosequenza comune di X e di Y avente lunghezza maggiore di k. Questo è però in contraddizione con il fatto che Z sia una LCS per X e Y, dato che questa è lunga k. Occorre allora assumere che Z_{k-1} è una LCS di X_{m-1} e Y_{n-1}.
2. Si supponga per assurdo che, se $x_m \neq y_n$ e $x_m \neq z_k$, allora Z non sia una LCS di X_{m-1} e Y. Deve allora esistere una sequenza W di lunghezza maggiore di k che è LCS di X_{m-1} e Y. Ma allora W è anche LCS per X e Y, contraddicendo l'ipotesi che lo sia Z. Occorre allora assumere che Z sia una LCS di X_{m-1} e Y.
3. La dimostrazione è simmetrica a quella del punto precedente.

1.1.3 Programmazione dinamica: equazione di ricorrenza

Il caso base dell'equazione di ricorrenza è semplice da determinare; è evidente come due sequenze vuote non abbiano alcun elemento in comune, pertanto la lunghezza della loro LCS é zero. Piú in generale, una qualsiasi sequenza non vuota non ha alcun carattere in comune con la sequenza vuota.

$$S_{i,j} = 0 \text{ se } i = 0 \vee j = 0$$

Per quanto riguarda il passo ricorsivo, possono verificarsi solamente due situazioni, mutualmente esclusive: l'i-esimo carattere di X_i e uguale al j-esimo carattere di Y_j oppure l'i-esimo carattere di X_i è diverso dal j-esimo carattere di Y_j.
Nel primo caso, per definizione, tale carattere deve essere parte della sottosequenza comune più lunga. Pertanto, la soluzione ottimale per la i, j-esima istanza si ottiene semplicemente aumentando di uno la soluzione ottimale dell'istanza precedente, ovvero $S_{i-1,j-1}$.
Nel secondo caso, la soluzione ottimale per la i, j-esima istanza deve necessariamente essere di lunghezza maggiore o uguale (ma non minore) alla lunghezza di tutte le soluzioni per istanze precedenti, e non può, per definizione, contenere l'i-esimo elemento di X (o il j-esimo elemento di Y). In particolare, le uniche soluzioni candidate ad essere le soluzioni per la i, j-esima istanza sono la $i-1, j$-esima e la $i, j-1$-esima, ovvero $S_{i-1,j}$ e $S_{i,j-1}$, perché a loro volta comprendono tutte le istanze a loro precedenti. Essendo egualmente valide, la scelta migliore é quella fra le due avente valore maggiore.

$$S_{i,j} = \begin{cases} S_{i-1,j-1} + 1 & \text{se } x_i = y_j \\ \max\{S_{i-1,j}, S_{i,j-1}\} & \text{se } x_i \neq y_j \end{cases}$$

1.1.4 Programmazione dinamica: implementazione bottom-up

L'algoritmo bottom-up viene costruito a partire dall'equazione di ricorrenza sfruttando una tabella c. In ciascuna cella $c[i, j]$ viene riportato il valore della soluzione ottimale per la i, j-esima istanza del problema, che viene utilizzata per calcolare i valori ottimali per le istanze successive. L'algoritmo riceve in input le due sequenze X e Y e restituisce in output la cella $c[|X|, |Y|]$, la soluzione del problema. Si noti come le celle che hanno 0 come uno dei due indici possano venire riempite immediatamente con 0, come da caso base dell'equazione di ricorrenza.

```
procedure LCS(X, Y)
    for i ← 0 to |X| do
        for j ← 0 to |Y| do
            c[i, j] ← 0


    for i ← 1 to |X| do
        for j ← 1 to |Y| do
            if X[i] = Y[j] then
                c[i, j] ← 1 + c[i - 1, j - 1]
            else
                if (c[i - 1, j] > c[i, j - 1]) then
                    c[i, j] ← c[i - 1, j]
                else
                    c[i, j] ← c[i, j - 1]

    return c[|X|][|Y|]
```

È facile notare come il tempo di esecuzione dell'algoritmo sia $O(nm)$, dove m e n sono le lunghezze di rispettivamente la sequenza X e la sequenza Y. I primi due cicli eseguono una operazione immediata rispettivamente per m e per n volte, dopodiché si presenta un doppio ciclo innestato: il ciclo esterno esegue il ciclo interno m volte, mentre il ciclo interno esegue n volte un blocco di decisione il cui tempo di esecuzione può considerarsi immediato. Si ha allora che, asintoticamente, $O(m) + O(n) + O(mn) = O(mn)$.

	ε	S	A	T	U	R	D	A	Y
ε	0	0	0	0	0	0	0	0	0
S	0	1	1	1	1	1	1	1	1
U	0	1	1	1	2	2	2	2	2
N	0	1	1	1	2	2	2	2	2
D	0	1	1	1	2	2	3	3	3
A	0	1	2	2	2	2	3	4	4
Y	0	1	2	2	2	2	3	4	5

1.1.5 Programmazione dinamica: ricostruzione di una soluzione

Una volta calcolata la lunghezza della LCS, è possibile individuarne una ripercorrendo la tabella a ritroso. Questo viene fatto modificando la procedura originale affinché restituisca non il valore $c[|X|, |Y|]$, ma bensí l'intera tabella c.
Alla vecchia procedura opportunamente modificata ne viene aggiunta un'altra, `PRINT-LCS`. Questa ha in input la tabella c, le sequenze X e Y e due indici di posizione i e j, mentre in output ha una delle possibili LCS per la sequenza in input. La prima chiamata alla procedura ha $|X|$ come valore per i e $|Y|$ come valore per j, in modo da ottenere la soluzione per l'istanza $|X|, |Y|$.
La procedura ripercorre la tabella c dalla cella (i, j) verso la cella $(0, 0)$: quando questa viene raggiunta, la procedura termina. La tabella viene risalita in tre possibili direzioni:

- In diagonale quando l'i-esimo elemento di X coincide con il j-esimo elemento di Y. Questo perché se tale condizione si verifica, allora $x_i = y_j$ é certamente parte della LCS;
- A sinistra quando l'i-esimo elemento di X differisce dall'j-esimo elemento di Y e la cella $c[i, j]$ contiene lo stesso valore della cella $c[i-1, j]$. Questo perché se tale condizione si verifica, allora né x_i né y_j possono essere parte della LCS, ma la LCS rispetto ai prefissi X_i e Y_j é la medesima di quella rispetto ai prefissi X_{i-1} e Y_j;
- In alto altrimenti.

```
procedure LCS(X, Y)
    for i ← 0 to |X| do
        for j ← 0 to |Y| do
            c[i, j] ← 0

    for i ← 1 to |X| do
        for j ← 1 to |Y| do
            if X[i] = Y[j] then
                c[i, j] ← 1 + c[i - 1, j - 1]
            else
                if c[i - 1, j] > c[i, j - 1] then
                    c[i, j] ← c[i - 1, j]
                else
                    c[i, j] ← c[i, j - 1]
        return c
```

```
procedure PRINT-LCS(c, X, Y, i, j)
    if (i = 0 or j = 0) then
        return

    if (X[i] = Y[j]) then
        PRINT-LCS(c, X, Y, i - 1, j - 1)
        print X[i]
    else
        if (c[i, j] = c[i - 1, j]) then
            PRINT-LCS(c, X, Y, i - 1, j)
        else
            PRINT-LCS(c, X, Y, i, j - 1)
```

Sebbene la procedura sia ricorsiva, il suo tempo di esecuzione è comunque proporzionale alle dimensioni della tabella c, perché la ricorsione è una tail-recursion. In particolare:

- Quando le due sequenze X e Y sono uguali, si avranno $|X| = |Y|$ iterazioni. Questo perché per definizione si ricade sempre nel primo caso della ricorsione, ed i due indici vengono sempre decrementati di uno ad ogni chiamata;
- Quando le due sequenze X e Y non hanno alcun carattere in comune, si avranno $|X|$ iterazioni. Questo perché per definizione la tabella c sará interamente costituita da 0, e si ricadrá quindi sempre nel secondo caso della ricorsione (naturalmente, se i casi due e tre venissero scambiati di posto, si avrebbe tempo di esecuzione $|Y|$). Tecnicamente, sarebbe possibile ottimizzare la procedura `PRINT-LIS` per terminare immediatamente restituendo ε se si ha $c[i, j] = 0$ con $i > 0$ e $j > 0$, perché in tal caso é garantito che l'LCS delle due sequenze sia la sequenza vuota;

- Quando la sequenza X é una sottosequenza di Y, e quindi X é essa stessa la LCS di Y, si avranno $|Y|$ iterazioni. Questo perché ad ogni chiamata si ricadrá sempre o nel primo caso oppure nel secondo, ma in entrambi i casi l'effetto é che l'indice i viene sempre decrementato di uno.

Si noti come, in genere, possano esistere più percorsi all'interno della tabella. Il numero di percorsi dipende dal numero k di biforcazioni. Ogni biforcazione indica una possibile soluzione al problema LCS, eventualmente ridonanti. Il numero di percorsi non può essere superiore a 2^k, ed il numero di possibili soluzioni distinte è non superiore al numero di possibili percorsi.

1.2 Longest Increasing Subsequence

1.2.1 Definizione del problema

Il problema della **Longest Increasing Subsequence** (**LIS**) prevede di trovare la più lunga sottosequenza di una sequenza i cui elementi (nell'ordine in cui si trovano) hanno valore strettamente crescente.

> Sia data la sequenza $X = \langle 14, 2, 4, 2, 7, 0, 13, 21, 11 \rangle$. Una sottosequenza strettamente crescente di X è $\langle 2, 7, 11 \rangle$, ma non è la più lunga; tale sottosequenza è infatti $\langle 2, 4, 7, 13, 21 \rangle$.

Anziché affrontare il problema direttamente, si preferisce prima trovare la lunghezza di una possibile LIS della sequenza e poi, sulla base di questa informazione, ricostruire la LIS "a ritroso". Pertanto, il problema effettivamente in esame sará il problema della lunghezza della sottosequenza strettamente crescente piú lunga.
Il problema puó essere risolto mediante programmazione dinamica. Data una sequenza X avente lunghezza n, sia $X_i = [1 : i]$ il suo prefisso di lunghezza i. La soluzione S_i per l'i-esima istanza del problema corrisponde a trovare la soluzione ottimale per il problema LIS rispetto al prefisso X_i. La soluzione per l'intera sequenza X è la soluzione per l'istanza X_n.
Per calcolare la soluzione ottimale della generica istanza X_i, ovvero S_i, verrá assunto di avere a disposizione tutte le soluzioni parziali $S_{i-1}, S_{i-2}, \ldots, S_1, S_0$. Pertanto, per il calcolo della soluzione ottimale dell'i-esima istanza del problema si hanno a disposizione l'elemento i-esimo della sequenza, ovvero x_i, e le lunghezze delle sottosequenze strettamente crescenti rispetto alle istanze da $i-1$ a 0, ma non gli elementi della sequenza che comportano tale lunghezza. x_i puó essere come puó non essere uno degli elementi della sottosequenza strettamente crescente di X_i di lunghezza S_i.

1.2.2 Definizione del problema ausiliario

Attaccare il problema LIS direttamente tramite programmazione dinamica non è possibile, perché manca un'informazione. Infatti, se non é noto quali siano gli elementi che hanno indotto le lunghezze ottimali parziali $S_{i-1}, \ldots, S_0$ non vi é alcun modo di sapere se l'elemento i-esimo possa essere accodato ad una sottosequenza avente peso S_j con $j < i$ ottenendo di nuovo una sottosequenza strettamente crescente di X_i. Questo perché le lunghezze S_j potrebbero riferirsi a sottosequenze che contengono elementi che vengono dopo di x_i nell'ordine lessicografico.
L'unico caso in cui si avrebbe la certezza di ottenere ancora una sottosequenza strettamente crescente di X_i si ha quando é garantito che la sottosequenza soluzione ottimale per l'i-esima istanza contiene anche x_i. É pertanto necessario introdurre un problema ausiliario che includa l'informazione mancante.
Una versione più semplice del problema LIS, chiamato **LIS vincolato** (**LISV**), prevede di trovare la più lunga sottosequenza di una sequenza i cui elementi (nell'ordine in cui si trovano) hanno valore strettamente crescente e dove l'ultimo elemento di tale sottosequenza coincide con l'ultimo elemento della sequenza originaria. Anche in questo caso, si preferisce considerare non l'effettiva soluzione di ciascuna istanza del problema, bensí il suo peso complessivo, e poi ricostruire "a ritroso" la soluzione.
Data una sequenza X avente lunghezza n, sia $X_i = [1 : i]$ il suo prefisso di lunghezza i. La soluzione S_i per l'i-esima istanza del problema corrisponde a trovare la soluzione ottimale per il problema LIS vincolato rispetto al prefisso X_i. La soluzione per l'intera sequenza X è la soluzione per l'istanza X_n.
Per calcolare la soluzione ottimale della generica istanza X_i, ovvero S_i, verrá assunto di avere a disposizione tutte le soluzioni parziali $S_{i-1}, S_{i-2}, \ldots, S_1, S_0$. Pertanto, per il calcolo della soluzione ottimale dell'i-esima istanza del problema si hanno a disposizione l'elemento i-esimo della sequenza, ovvero x_i, e le lunghezze delle sottosequenze strettamente crescenti rispetto alle istanze da $i-1$ a 0, ma non gli elementi della sequenza che comportano tale lunghezza. A differenza del problema originale, x_i é per forza uno degli elementi (in particolare, l'ultimo) della sottosequenza strettamente crescente di X_i di lunghezza S_i.

Sia data la sequenza $X = \langle 14, 1, 4, 6, 13, 15, 0, 13, 29, 8 \rangle$. La più lunga sottosequenza strettamente crescente di X si rivela essere $\langle 1, 4, 6, 13, 15, 29 \rangle$. D'altro canto, la più lunga sottosequenza strettamente crescente di X vincolata si rivela invece essere $\langle 1, 4, 6, 8 \rangle$.

1.2.3 Programmazione dinamica: sottostruttura ottima

Proprietà della sottostruttura ottima per il problema Longest Increasing Subsequence (Vincolato). Sia X una sequenza di m numeri interi e sia X_i un suo prefisso di lunghezza i, con $1 \leq i \leq m$. Sia Z^i una LISV di X_i, che termina con x_i. Sia infine W_i l'insieme di tutte le possibili sottosequenze crescenti (non necessariamente le più lunghe) di X_j che finiscono con x_j e a cui è possibile concatenare x_i ottenendo ancora una sequenza crescente, ovvero:

$$W_i = \bigcup_{1 \leq j < i;\ x_j < x_i} \{W \text{ sottosequenza crescente di } X_j \text{ che termina con } x_j\}$$

Allora la LISV Z^i è data dalla concatenazione fra l'elemento di W_i avente la maggior cardinalità con il carattere x_i, ovvero $Z^i = Z^* \mid x_i$ con $Z^* \in W_i$ e $|Z^*| = \max_{W \in W_i}\{|W|\}$.

Dimostrazione. Si supponga per assurdo che $Z^i = Z^* \mid x_i$ non sia la soluzione ottimale per l'i-esima istanza del problema. Questa deve allora essere $Z^i = Z' \mid x_i$, dove Z' è una sottosequenza crescente avente cardinalità maggiore di Z^*.
Sia z' l'elemento in coda a Z'. Essendo Z^i costruito accodando x_i a Z' ottenendo ancora una sequenza crescente, deve aversi $z' < x_i$. Sia poi $h < i$ il più grande indice tale che $x_h = z'$. Di conseguenza, per come è stato definito W_i, si ottiene che $Z' \in W_i$. Infatti, Z' è una sottosequenza crescente di X_h, la quale termina con $x_h < x_i$. Ciò porta però ad una contraddizione: infatti si ha $|Z^*| = \max_{W \in W_i}\{|W|\}$, ma al contempo Z' è membro di W_i ed è stato dimostrato che $|Z'| > |Z^*|$. Occorre allora assumere che $Z^i = Z^* \mid x_i$ sia effettivamente la soluzione ottimale alla i-esima istanza del problema.

1.2.4 Programmazione dinamica: equazione di ricorrenza

Il caso base dell'equazione di ricorrenza è immediato: se la sequenza è la sequenza vuota, allora la sottosequenza strettamente crescente più lunga vincolata è essa stessa la sequenza vuota, che ha lunghezza 0. Se la sequenza é composta da un solo elemento, allora la sottosequenza strettamente crescente piú lunga vincolata é la sequenza stessa, che ha lunghezza 1. Infatti, per definizione é sia strettamente crescente, dato che é composta da un solo elemento, ed é vincolata, dato che di fatto contiene l'ultimo (e unico) carattere della sequenza.

$$S_0 = 0 \qquad\qquad S_1 = 1$$

Per quanto riguarda il passo ricorsivo, si ha che una generica soluzione per la i-esima istanza del problema deve essere costruita accodando x_i ad sottosequenza strettamente crescente tale che tutti gli elementi di tale sottosequenza vengono prima, in ordine lessicografico, di x_i. Questo perché l'aggiunta di x_i in coda ad una tale sottosequenza restituisce ancora una sottosequenza strettamente crescente. La soluzione ottimale S_i è data dalla lunghezza della sottosequenza con queste caratteristiche più lunga possibile aumentata di uno.
La sottosequenza strettamente crescente più lunga possibile con tutti gli elementi che vengono prima, in ordine lessicografico, di x_i sarà a sua volta soluzione di una certa istanza h del problema, con $h < i$. La soluzione S_i é allora data dalla soluzione S_h a cui viene sommato 1. Si noti come S_h possa anche non esistere, ed in particolare questo si verifica se x_i viene dopo tutti gli altri elementi di X_i nell'ordine lessicografico. In questo caso, la soluzione per la i-esima istanza del problema è 1, perché la sottosequenza crescente piú lunga possibile vincolata é costituita dal solo x_i.

$$S_i = \begin{cases} \max\{S_h \ t.c. \ 1 \leq h < i \wedge x_h < x_i\} + 1 & \text{se } \exists S_h \\ 1 & \text{se } \nexists S_h \end{cases}$$

1.2.5 Programmazione dinamica: implementazione bottom-up

L'algoritmo bottom-up viene costruito a partire dall'equazione di ricorrenza sfruttando un vettore c. In ciascuna cella $c[i]$ viene riportato il valore della soluzione ottimale per la i-esima istanza del problema, che viene utilizzata per calcolare i valori ottimali per le istanze successive. L'algoritmo riceve in input la sequenza X e restituisce in output il vettore c.
Si noti come tutte le celle del vettore c, ad eccezione della prima, possono essere inizializzate subito ad 1, perché questo è il minimo valore che in una cella può essere presente. La prima cella corrisponde invece al caso base in cui la sequenza in esame é la sequenza vuota.

```
procedure BOUNDED-LIS(X)
    c[0] ← 0
    for i ← 1 to |X| do
        c[i] ← 1

    for i ← 2 to |X| do
        t ← 0
        for j ← 1 to i - 1 do
            if (X[j] < X[i]) and (c[j] ≥ t) then
                t ← c[j]
        c[i] ← t + 1

    return c
```

Il problema LIS non vincolato può essere ridotto al problema LIS vincolato. Infatti, la soluzione del problema LIS non vincolato non è altro che la maggior soluzione parziale del problema LIS vincolato.

```
procedure LIS(X)
    c[0] ← 0
    for i ← 1 to |X| do
        c[i] ← 1
    best ← 0

    for i ← 2 to |X| do
        t ← 0
        for j ← 1 to i - 1 do
            if (X[j] < X[i]) and (c[j] ≥ t) then
                t ← c[j]
        c[i] ← t + 1
        if (c[i] > best) then
            best ← c[i]

    return best
```

È facile notare come il tempo di esecuzione dell'algoritmo sia $O(n^2)$, dove n è la lunghezza della sequenza X. Il primo ciclo esegue una operazione immediata per n volte, dopodiché si presenta un doppio ciclo innestato: il ciclo esterno esegue il ciclo interno n volte, mentre il ciclo interno esegue al più n volte (dovendo potenzialmente ripercorrere l'intera sequenza a ritroso) un blocco di decisione il cui tempo di esecuzione può considerarsi immediato. Si ha allora che, asintoticamente, $O(n) + O(n^2) = O(n^2)$.

1.2.6 Programmazione dinamica: ricostruzione della soluzione

Una volta calcolata la lunghezza di una LISV, è possibile individuarla introducendo un vettore di appoggio h e ripercorrendolo a ritroso. Nella i-esima cella di tale vettore viene riportato l'indice della cella di c in cui si trova l'elemento predecessore dell'elemento nella i-esima cella di c. Se l'elemento predecessore non esiste, nella i-esima cella di h viene riportato `0`. La nuova procedura restituisce non il vettore c, bensí il vettore h.
Alla vecchia procedura opportunamente modificata ne viene aggiunta un'altra, `PRINT-BOUNDED-LIS`. Questa ha in input il vettore h, la sequenza X ed un indice di posizione i, mentre in output ha una delle possibili LISV per la sequenza in input.
La procedura restituisce la soluzione per la i-esima istanza del problema ripercorrendo il vettore h dalla cella i verso la cella `0`; ogni volta che si incontra un elemento non nullo di h, l'elemento della sequenza in tale posizione è un elemento della soluzione.

```
procedure BOUNDED-LIS(X)
    c[0] ← 0
    h[0] ← 0
    for i ← 1 to |X| do
        c[i] ← 1
        h[i] ← 0

    for i ← 2 to |X| do
        t ← 0
        for j ← 1 to i - 1 do
            if (X[j] < X[i]) and (c[j] ≥ t) then
                t ← c[j]
                h[i] ← j
        c[i] ← t + 1

    return h
```

```
procedure PRINT-BOUNDED-LIS(h, X, i)
    if h[i] ≠ 0 then
        PRINT-BOUNDED-LIS(h, X, h[i])
    print X[i]
```

Ricordando come la soluzione del problema originale sia la maggior soluzione parziale del problema vincolato, é possibile modificare la procedura `BOUNDED-LIS` per restituire tale soluzione oltre al vettore h. Le procedure `PRINT-BOUNDED-LIS` e `PRINT-LIS` sono identiche: l'unica differenza é che la prima chiamata a `PRINT-LIS` deve avere `best` come terzo parametro, di modo da ricavare la soluzione per l'istanza corretta.

```
procedure LIS(X)
    c[0] ← 0
    h[0] ← 0
    for i ← 1 to |X| do
        c[i] ← 1
        h[i] ← 0
    best ← 0

    for i ← 2 to |X| do
        t ← 0
        for j ← 1 to i - 1 do
            if (X[j] < X[i]) and (c[j] ≥ t) then
                t ← c[j]
                h[i] ← j
        c[i] ← t + 1
        if (c[i] > best) then
            best ← c[i]

    return h, best
```

```
procedure PRINT-LIS(h, X, i)
    if h[i] ≠ 0 then
        PRINT-LIS(h, X, h[i])
    print X[i]
```

Sebbene la procedura sia ricorsiva, il suo tempo di esecuzione è comunque proporzionale alle dimensioni del vettore h, perché la ricorsione è una tail-recursion. In particolare:

- Quando la sequenza X é giá di per sé strettamente crescente, la procedura viene chiamata esattamente $|h|$ volte. Questo perché ogni elemento i-esimo di X con $i > 0$ ha per predecessore nel vettore h l'elemento $i-1$-esimo;
- Quando la sequenza X é decrescente, la procedura viene chiamata una sola volta. Questo perché ogni elemento di h é nullo e la procedura entra immediatamente nel caso base.

$\langle 14, 2, 4, 2, 7, 0, 13, 21, 20 \rangle$

X_0	X_1	X_2	X_3	X_4	X_5	X_6	X_7	X_8	X_9
0	1	1	2	1	3	1	4	5	5
0	0	0	2	0	3	0	5	7	7

1.3 Longest Increasing Common Subsequence

1.3.1 Definizione del problema

Il problema della **Longest Increasing Common Subsequence** (**LICS**) prevede di trovare la più lunga sottosequenza comune a due sequenze i cui elementi (nell'ordine in cui si trovano) hanno valore strettamente crescente.

> Siano date le sequenze $X = \langle 14, 2, 4, 2, 7, 0, 13, 21, 11 \rangle$ e $Y = \langle 13, 2, 6, 5, 4, 11, 0 \rangle$. Una sottosequenza strettamente crescente comune ad X e a Y di lunghezza massima è $\langle 2, 4, 11 \rangle$.

Anziché affrontare il problema direttamente, si preferisce prima trovare la lunghezza di una possibile LICS delle due sequenze e poi, sulla base di questa informazione, ricostruire la LICS "a ritroso". Pertanto, il problema effettivamente in esame sará il problema della lunghezza della sottosequenza strettamente crescente comune piú lunga.
Il problema puó essere risolto mediante programmazione dinamica. Date due sequenze X e Y di lunghezza rispettivamente n e m, siano $X_i = X[1 : i]$ e $Y_j = Y[1 : j]$ i prefissi di lunghezza i e j rispettivamente della sequenza X e della sequenza Y. La soluzione $S_{i,j}$ per l'i, j-esima istanza del problema corrisponde a trovare la soluzione ottimale per il problema LICS rispetto alle sottosequenze X_i e Y_j. La soluzione per le intere sequenze X e Y è la soluzione per le istanze X_n e Y_m.
Per calcolare la soluzione ottimale della generica istanza X_i, Y_j ovvero $S_{i,j}$, verrá assunto di avere a disposizione tutte le soluzioni parziali nella forma $S_{a,b}$, con $1 \leq a < i$ e $1 \leq b < j$. Pertanto, per il calcolo della soluzione ottimale della i, j-esima istanza del problema si hanno a disposizione l'elemento i-esimo della sequenza X, ovvero x_i, l'elemento j-esimo della sequenza Y, ovvero y_j, e le lunghezze delle sottosequenze strettamente crescenti rispetto alle istanze da $i-1, j-1$ a 0, 0, ma non gli elementi della sequenza che comportano tale lunghezza. x_i e y_j possono essere come possono non essere elementi della sottosequenza di X_i e Y_j strettamente crescente di lunghezza $S_{i,j}$.

1.3.2 Definizione del problema ausiliario

Attaccare il problema LICS direttamente tramite programmazione dinamica non è possibile, perché manca un'informazione. Infatti, se non é noto quali siano gli elementi che hanno indotto le lunghezze ottimali parziali da $S_{i-1,j-1}$ a $S_{0,0}$ non vi é alcun modo di sapere se l'elemento i-esimo edi X o l'elemento j-esimo di Y possano essere accodati ad una sottosequenza avente peso $S_{a,b}$ con $1 \leq a < i$ e $1 \leq b < j$ ottenendo di nuovo una sottosequenza strettamente crescente comune di X_i e Y_j. Questo perché le lunghezze $S_{i,j}$ potrebbero riferirsi a sottosequenze che contengono elementi che vengono dopo di x_i e/o di y_j nell'ordine lessicografico.
L'unico caso in cui si avrebbe la certezza di ottenere ancora una sottosequenza strettamente crescente di X_i e di Y_j si ha quando é garantito che la sottosequenza soluzione ottimale per l'i, j-esima istanza contiene anche $x_i = y_j$. É pertanto necessario introdurre un problema ausiliario che includa l'informazione mancante.
Una versione più semplice del problema LICS, chiamato **LICS vincolato** (**LICSV**), prevede di trovare la più lunga sottosequenza comune di due sequenze i cui elementi (nell'ordine in cui si trovano) hanno valore strettamente crescente e dove l'ultimo elemento di tale sottosequenza coincide con l'ultimo elemento di entrambe le sequenze originarie. Anche in questo caso, si preferisce considerare non l'effettiva soluzione di ciascuna istanza del problema, bensí il suo peso complessivo, e poi ricostruire "a ritroso" la soluzione.
Date due sequenze X e Y di lunghezza rispettivamente n e m, siano $X_i = X[1 : i]$ e $Y_j = Y[1 : j]$ i prefissi di lunghezza i e j rispettivamente della sequenza X e della sequenza Y. La soluzione $S_{i,j}$ per l'i, j-esima istanza del problema corrisponde a trovare la soluzione ottimale per il problema LICS vincolato rispetto alle sottosequenze X_i e Y_j. La soluzione per le intere sequenze X e Y è la soluzione per le istanze X_n e Y_m.
Per calcolare la soluzione ottimale della generica istanza X_i, Y_j ovvero $S_{i,j}$, verrá assunto di avere a disposizione tutte le soluzioni parziali nella forma $S_{a,b}$, con $1 \leq a < i$ e $1 \leq b < j$. Pertanto, per il calcolo della soluzione ottimale della i, j-esima istanza del problema si hanno a disposizione l'elemento i-esimo della sequenza X, ovvero x_i, l'elemento j-esimo della sequenza Y, ovvero y_j, e le lunghezze delle sottosequenze strettamente crescenti rispetto alle istanze da $i-1, j-1$ a 0, 0, ma non gli elementi della sequenza che comportano tale lunghezza. A differenza del problema originale, x_i e y_j devono non solo essere uguali, ma devono anche essere per forza un elemento (in particolare, l'ultimo) della sottosequenza strettamente crescente comune di X_i e di Y_j di lunghezza S_i.

> Siano date le sequenze $X = \langle 14, 2, 4, 6, 13, 15, 0 \rangle$ e $Y = \langle 13, 2, 6, 5, 4, 11, 0 \rangle$. Una sottosequenza strettamente crescente di massima lunghezza comune ad X e ad Y è $\langle 2, 4 \rangle$. D'altro canto, la sottosequenza strettamente crescente di massima lunghezza comune ad X e ad Y vincolata è $\langle 0 \rangle$.

1.3.3 Programmazione dinamica: sottostruttura ottima

Proprietà della sottostruttura ottima per il problema Longest Increasing Common Subsequence (Vincolato). Sia X una sequenza di m numeri interi e sia X_i un suo prefisso di lunghezza i, con $1 \leq i \leq m$. Sia Y una sequenza di n numeri interi e sia Y_j un suo prefisso di lunghezza j, con $1 \leq j \leq n$.
Sia $Z^{i,j}$ una LICSV di X_i e di Y_j tale che termini con $x_i = y_j$. Sia infine $W_{i,j}$ l'insieme di tutte le possibili sottosequenze crescenti comuni (non necessariamente le più lunghe) di X_h e di Y_k che finiscono con $x_h = y_k$ e a cui è possibile concatenare x_i (o y_j) ottenendo ancora una sequenza crescente comune, ovvero:

$$W_{i,j} = \bigcup_{1 \leq h < i;\ i \leq k < j;\ x_h = y_k < x_i = y_j} \{W \text{ sottosequenza comune crescente di } X_h \text{ e di } Y_k \text{ che termina con } x_h = y_k\}$$

Allora la LICSV $Z^{i,j}$ è data dalla concatenazione fra l'elemento di $W_{i,j}$ avente la maggior cardinalità con il carattere x_i o con il carattere y_j, ovvero $Z^{i,j} = Z^* \mid x_i = Z^* \mid y_j$ con $Z^* \in W_{i,j}$ e $|Z^*| = \max_{W \in W_{i,j}}\{|W|\}$.

Dimostrazione. Si supponga per assurdo che $Z^{i,j} = Z^* \mid x_i = Z^* \mid y_j$ non sia la soluzione ottimale per l'i,j-esima istanza del problema. Questa deve allora essere $Z^{i,j} = Z' \mid x_i = Z' \mid y_j$, dove Z' è una sottosequenza crescente comune avente cardinalità maggiore di Z^*.
Sia z' l'elemento in coda a Z'. Essendo $Z^{i,j}$ costruito accodando $x_i = y_j$ a Z' ottenendo ancora una sequenza crescente, deve aversi $z' < x_i = y_j$. Siano poi $r < i$ e $s < j$ la più grande coppia di indici tali per cui $x_r = y_s = z'$. Di conseguenza, per come è stato definito W_i, si ottiene che $Z' \in W_{i,j}$. Infatti, Z' è una sottosequenza crescente comune di X_r e di Y_s, la quale termina con $x_r < x_i = y_j$. Ciò porta però ad una contraddizione: infatti si ha $|Z^*| = \max_{W \in W_{i,j}}\{|W|\}$, ma al contempo Z' è membro di $W_{i,j}$ ed è stato dimostrato che $|Z'| > |Z^*|$. Occorre allora assumere che $Z^{i,j} = Z^* \mid x_i = Z^* \mid y_j$ sia effettivamente la soluzione ottimale alla i-esima istanza del problema.

1.3.4 Programmazione dinamica: equazione di ricorrenza

Certamente, la soluzione per tutte le coppie di indici i, j con $i = 0$ oppure $j = 0$ si rivela immediatamente essere la sequenza vuota, perché questa è l'unica sottosequenza che una sequenza generica può avere in comune con la sequenza vuota stessa.
Tuttavia, anche nel caso in cui $x_i \neq y_j$, ovvero quando le due sottosequenze X_i e Y_j non terminano con lo stesso carattere, la sottosequenza crescente comune più lunga è la sequenza vuota. Questo perché il problema chiede di trovare una sottosequenza che termini con il carattere (uguale) delle due sequenze, e se tale carattere è distinto l'unica soluzione accettabile non può che essere la sequenza vuota.
Dato che, per definizione, l'ultimo elemento della sequenza vuota (non esistendo) non può coincidere con l'ultimo carattere dell'altra sequenza, il caso $i = 0 \vee j = 0$ rientra di fatto nel caso $x_i \neq y_j$, pertanto è sufficiente considerare solamente quest'ultimo come caso base.

$$S_{i,j} = 0 \text{ se } x_i \neq y_j$$

Per quanto riguarda il passo ricorsivo, si ha che una generica soluzione per la i,j-esima istanza del problema deve essere costruita accodando $x_i = y_j$ ad sottosequenza strettamente crescente comune tale che tutti gli elementi di tale sottosequenza vengono prima, in ordine lessicografico, di $x_i = y_j$. Questo perché l'aggiunta di $x_i = y_j$ in coda ad una tale sottosequenza restituisce ancora una sottosequenza strettamente crescente comune. La soluzione ottimale $S_{i,j}$ è data dalla lunghezza della sottosequenza con queste caratteristiche più lunga possibile aumentata di uno.
La sottosequenza strettamente crescente comune più lunga possibile con tutti gli elementi che vengono prima, in ordine lessicografico, di $x_i = y_j$ sarà a sua volta soluzione di una certa istanza h, k del problema, con $h < i$ e $k < j$. La soluzione $S_{i,j}$ é allora data dalla soluzione $S_{h,k}$ a cui viene sommato 1. Si noti come $S_{h,k}$ possa anche non esistere, ed in particolare questo si verifica se $x_i = y_j$ viene dopo tutti gli altri elementi di X_i e di Y_j nell'ordine lessicografico. In questo caso, la soluzione per la i,j-esima istanza del problema è 1, perché la sottosequenza crescente piú lunga possibile vincolata é costituita dal solo $x_i = y_j$.

$$S_{i,j} = \begin{cases} \max\{S_{h,k} \mid 1 \leq h < i, 1 \leq k < j, x_h < x_i\} + 1 & \text{se } \exists S_{h,k} \\ 0 & \text{se } \nexists S_{h,k} \end{cases}$$

1.3.5 Programmazione dinamica: implementazione bottom-up

L'algoritmo bottom-up viene costruito a partire dall'equazione di ricorrenza sfruttando una tabella c. In ciascuna cella $c[i, j]$ viene riportato il valore della soluzione ottimale per la i, j-esima istanza del problema, che viene utilizzata per calcolare i valori ottimali per le istanze successive. L'algoritmo riceve in input le due sequenze X e Y e restituisce in output la tabella c.

```
procedure BOUNDED-LICS(X, Y)
    for i ← 1 to m do
        for j ← 1 to n do
            if X[i] ≠ Y[j] then
                c[i, j] ← 0
            else
                t ← 0
                for a ← 1 to i - 1 do
                    for b ← 1 to j - 1 do
                        if (X[a] < X[i]) and (c[a, b] ≥ t) then
                            t ← c[a, b]
                c[i, j] ← t + 1

    return c
```

Il problema LIS non vincolato può essere ridotto al problema LIS vincolato. Infatti, la soluzione del problema LIS non vincolato non è altro che la maggior soluzione parziale del problema LIS vincolato.

```
procedure LICS(X, Y)
    best ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if X[i] ≠ Y[j] then
                c[i, j] ← 0
            else
                t ← 0
                for a ← 1 to i - 1 do
                    for b ← 1 to j - 1 do
                        if (X[a] < X[i]) and (c[a, b] ≥ t) then
                            t ← c[a, b]
                c[i, j] ← t + 1
            if (c[i, j] > best) then
                best ← c[i, j]

    return best
```

Il tempo di esecuzione dell'algoritmo è $O(m^2n^2)$, dove m è la lunghezza della sequenza X e n è la lunghezza della sequenza Y. Si noti infatti come l'algoritmo sia costituito da due cicli innestati, dove il primo esegue il secondo m volte ed il secondo esegue potenzialmente n volte a sua volta una coppia di cicli innestati, dove il primo esegue il secondo al più m volte ed il secondo esegue al più n volte un blocco di istruzioni avente tempo di esecuzione immediato. Si ha quindi $O(m \cdot n \cdot (m \cdot n)) = O(m^2n^2)$.

1.3.6 Programmazione dinamica: ricostruzione di una soluzione

Una volta calcolata la lunghezza di una LICSV, è possibile individuarla introducendo una tabella di appoggio h e ripercorrendola a ritroso. Nella i, j-esima cella di tale tabella viene riportato la coppia di indici della cella di c in cui si trova l'elemento predecessore dell'elemento nella i, j-esima cella di c. Se l'elemento predecessore non esiste, nella i, j-esima cella di h viene riportato $(0, 0)$. La nuova procedura restituisce non la tabella c, bensí la tabella h.
Alla vecchia procedura opportunamente modificata ne viene aggiunta un'altra, `PRINT-BOUNDED-LICS`. Questa ha in input la tabella h, le due sequenze X e Y e due indici di posizione i e j, mentre in output ha una delle possibili LICSV per le sequenze in input.
La procedura restituisce la soluzione per la i, j-esima istanza del problema ripercorrendo la tabella h dalla cella (i, j) verso la cella $(0, 0)$; ogni volta che si incontra un elemento non nullo di h, l'elemento della sequenza in tale posizione è un elemento della soluzione.

```
procedure BOUNDED-LICS(X, Y)
    for i ← 1 to m do
        for j ← 1 to n do
            if X[i] ≠ Y[j] then
                c[i, j] ← 0
                h[i, j] ← 0
            else
                t ← 0
                for a ← 1 to i - 1 do
                    for b ← 1 to j - 1 do
                        if (X[a] < X[i]) and (c[a, b] ≥ t) then
                            t ← c[a, b]
                            h[i, j] ← (a, b)
                c[i, j] ← t + 1

    return h
```

```
procedure PRINT-BOUNDED-LICS(h, X, [i, j])
    if h[i, j] ≠ (0, 0) then
        PRINT-BOUNDED-LICS(h, X, h[i, j])
    print X[i]
```

Ricordando come la soluzione del problema originale sia la maggior soluzione parziale del problema vincolato, é possibile modificare la procedura `BOUNDED-LICS` per restituire tale soluzione oltre al vettore *h*. Le procedure `PRINT-BOUNDED-LICS` e `PRINT-LICS` sono identiche: l'unica differenza é che la prima chiamata a `PRINT-LICS` deve avere `best` come terzo parametro, di modo da ricavare la soluzione per l'istanza corretta.

```
procedure LICS(X, Y)
    best ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if X[i] ≠ Y[j] then
                c[i, j] ← 0
                h[i, j] ← 0
            else
                t ← 0
                for a ← 1 to i - 1 do
                    for b ← 1 to j - 1 do
                        if (X[a] < X[i]) and (c[a, b] ≥ t) then
                            t ← c[a, b]
                            h[i, j] ← (a, b)
                c[i, j] ← t + 1
            if (c[i, j] > best) then
                best ← c[i, j]

    return h, best
```

```
procedure PRINT-LICS(h, X, [i, j])
    if h[i, j] ≠ (0, 0) then
        PRINT-LICS(h, X, h[i, j])
    print X[i]
```

Sebbene la procedura sia ricorsiva, il suo tempo di esecuzione è comunque proporzionale alle dimensioni del vettore *h*, perché la ricorsione è una tail-recursion.

Capitolo 2
Programmazione dinamica su insiemi

2.1 Weighted Interval Scheduling

2.1.1 Definizione del problema

Sia dato un insieme X costituito da "attività". Ciascuna attività $i \in X$ è definita a partire da una tripla (s_i, f_i, v_i), dove i tre valori indicano rispettivamente il tempo di inizio dell'attività, il tempo di fine ed il suo peso. Due attività i e j distinte si dicono *compatibili* se i termina prima che j inizi, ovvero se $f_i \leq s_j$.

$$i \in X = (s_i, f_i, v_i)$$

$$\text{Comp}(i,j) = \begin{cases} T & \text{se } f_i \leq s_j \\ F & \text{se } f_i > s_j \end{cases}$$

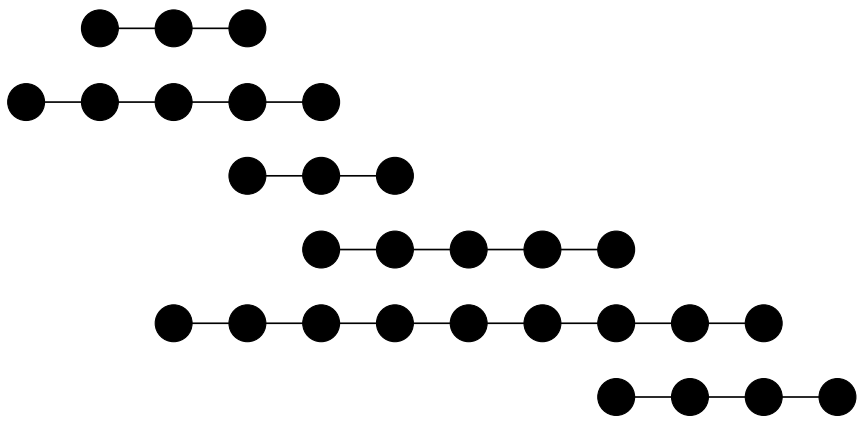
La nozioni di peso e di compatibilità vengono poi estese dai singoli elementi di X a sottoinsiemi A di X. Il peso di un insieme $A \subseteq X$ è dato dalla somma dei pesi di tutte le attività di cui è costituito (per convenzione, se A è l'insieme vuoto, il suo peso è 0). Similmente, un insieme $A \subseteq X$ si dice compatibile se contiene solo ed esclusivamente elementi di X tutti compatibili fra di loro o, equivalentemente, non contiene nemmeno una coppia di elementi di X che non sono fra loro compatibili.

$$V(A) = \begin{cases} \sum_{i \in A} v_i & \text{se } A \neq \emptyset \\ 0 & \text{se } A = \emptyset \end{cases}$$

$$\text{Comp}(A) = \begin{cases} T & \text{se } \nexists i,j \in A \text{ t.c. } f_i > s_j \\ F & \text{se } \exists i,j \in A \text{ t.c. } f_i > s_j \end{cases}$$

Il problema **Weighted Interval Scheduling** richiede di individuare qual'è, dato un insieme di attività, il sottoinsieme di attività fra loro compatibili che ha peso maggiore.

Attività	Tempo iniziale	Tempo finale	Peso
1	1	3	10
2	2	0	4
3	8	3	5
4	1	4	8
5	1	2	10
6	3	8	11



Il sottoinsieme $P = \{2, 4, 6\}$ è un insieme compatibile, ma non è il sottoinsieme con peso maggiore. Si osserva infatti che il sottoinsieme con queste caratteristiche è $Q = \{1, 3, 6\}$, che ha peso pari a 21.

Il problema é risolvibile mediante programmazione dinamica. Dato un insieme di attività X avente cardinalità n, ordinate per tempo di fine, sia X_i l'insieme costituito dalle prime i attività di X. La soluzione S_i per la i-esima istanza del problema corrisponde a trovare la soluzione ottimale per il problema WIS rispetto al sottoinsieme X_i. La soluzione per l'intero insieme X è la soluzione per l'istanza X_n. Il peso totale della soluzione ottimale per la i-esima istanza del problema é dato dalla funzione Opt(i).
Per calcolare la soluzione ottimale della generica istanza X_i, ovvero S_i, verrá assunto di avere a disposizione tutte le soluzioni parziali $S_{i-1}, S_{i-2}, \ldots, S_1, S_0$. Pertanto, per il calcolo della soluzione ottimale dell'i-esima istanza del problema si hanno a disposizione le informazioni relative all'i-esima attivitá (tempo di inizio, tempo di fine, peso) ed il peso totale delle soluzioni $i-1$ a 0. i puó essere come puó non essere uno degli elementi della soluzione ottimale per l'istanza i-esima.

2.1.2 Programmazione dinamica: sottostruttura ottima

Proprietà della sottostruttura ottima per il problema Weighted Interval Scheduling. Sia data una istanza $X_i = \{1, 2, ..., i\}$ del problema Weighted Interval Scheduling, con $i \geq 1$. Assunto di avere a disposizione tutte le soluzioni per le istanze precedenti ad X_i, ovvero $S_1, S_2, ..., S_{i-1}$, si ha:

$$S_i = \begin{cases} S_{p(i)} \cup \{i\} & \text{se } i \in S_i \\ S_{i-1} & \text{se } i \notin S_i \end{cases}$$

Dimostrazione. L'i-esimo elemento dell'istanza X_i può oppure non può fare parte della i-esima soluzione. Si distinguono allora due casi:

- $i \notin S_i$. Si supponga per assurdo che S_{i-1} non sia la soluzione ottimale del problema per l'istanza X_i. Se così è, allora la soluzione ottimale per tale istanza (quale che sia) deve necessariamente avere un valore maggiore di quello della soluzione ottimale S_{i-1}, ovvero deve valere $\text{Opt}(i) > \text{Opt}(i-1)$.
  Se, come da ipotesi, i non appartiene ad S_i, deve aversi $S_i \subseteq X_{i-1}$. Ma allora S_i è una potenziale soluzione per la $i-1$-esima istanza del problema. Ricordando però che $\text{Opt}(i) > \text{Opt}(i-1)$, questo equivale a dire che S_i è una soluzione migliore per la $i-1$-esima istanza del problema di quanto lo sia S_{i-1}, e questo è in contraddizione con l'ipotesi che la soluzione ottimale per l'istanza X_{i-1} sia S_{i-1}. Occorre allora assumere che S_{i-1} sia l'effettiva soluzione ottimale per la i-esima istanza del problema quando $i \notin S_i$.
- $i \in S_i$. Si supponga per assurdo che $S_{p(i)} \cup \{i\}$ non sia la soluzione ottimale al problema per l'istanza X_i. Se così è, allora la soluzione ottimale per tale istanza (quale che sia) deve avere un valore maggiore di quello della soluzione ottimale $S_{p(i)} \cup \{i\}$, ovvero deve valere $\text{Opt}(i) > \text{Opt}(p(i)) + v_i$.
  Dato che, per ipotesi, i appartiene ad S_i, è possibile scomporre tale soluzione in $S_h \cup \{i\}$, con $S_h \neq S_{p(i)}$, così come è possibile scomporre $\text{Opt}(i)$ in $\text{Opt}(h) + v_i$. Essendo però $\text{Opt}(i) > \text{Opt}(p(i)) + v_i$, deve valere $\text{Opt}(h) + v_i > \text{Opt}(p(i)) + v_i$, cioè $\text{Opt}(h) > \text{Opt}(p(i))$.
  Dovendo essere $S_h \cup \{x_i\}$ un insieme costituito da elementi mutualmente compatibili, deve necessariamente valere $S_h \subseteq X_{p(i)}$. Ma allora S_h è una potenziale soluzione per la $p(i)$-esima istanza del problema. Ricordando però che $\text{Opt}(h) > \text{Opt}(p(i))$, questo equivale a dire che S_h è una soluzione migliore per la $p(i)$-esima istanza del problema di quanto lo sia $S_{p(i)}$, e questo è in contraddizione con l'ipotesi che la soluzione ottimale per l'istanza $X_{p(i)}$ sia $S_{p(i)}$. Occorre allora assumere che $S_{p(i)} \cup \{i\}$ sia l'effettiva soluzione ottimale per la i-esima istanza del problema quando $i \in S_i$.

2.1.3 Programmazione dinamica: equazione di ricorrenza

Il caso base dell'equazione di ricorrenza è semplice da determinare; l'istanza X_0, che corrisponde all'insieme vuoto, ha per soluzione l'insieme vuoto stesso, che per definizione ha associato il valore 0.

$$S_0 = \emptyset \qquad \text{Opt}(0) = 0$$

Per quanto riguarda il passo ricorsivo, possono verificarsi solamente due situazioni, mutualmente esclusive: l'i-esima attività fa parte della soluzione ottimale S_i oppure non ne fa parte.
Nel primo caso, è evidente come S_i e S_{i-1} debbano essere lo stesso insieme. Questo perché, essendo sia S_i sia S_{i-1} soluzioni ottimali, se i non è uno dei componenti della soluzione ottimale dell'istanza X_i, allora la soluzione ottimale di tale istanza dovrà essere la stessa dell'insieme X_i a cui viene tolto l'elemento i, ovvero X_{i-1}, la cui soluzione è proprio S_{i-1}.
Nel secondo caso, la soluzione ottimale dell'istanza X_i deve essere un insieme che contiene i e tutte le attività precedenti ad i che sono con questa compatibili. Dato che l'insieme S_i è soprainsieme di tutti gli insiemi S_j con $j < i$, per individuare il sottoinsieme di S_i che contiene tutte le attività compatibili con i è sufficiente cercare quello di indice maggiore, perché per definizione conterrà anche tutti i sottoinsiemi di elementi compatibili con i a loro volta più piccoli. L'indice con queste caratteristiche viene indicato con $p(i)$.

$$p(i) = \max\{j \mid j < i \wedge \text{Comp}(i, j) = T\}$$

$$S_i = \begin{cases} S_{p(i)} \cup \{i\} & \text{se } i \in S_i \\ S_{i-1} & \text{se } i \notin S_i \end{cases} \qquad \text{Opt}(i) = \begin{cases} \text{Opt}(p(i)) + v_i & \text{se } i \in S_i \\ \text{Opt}(i-1) & \text{se } i \notin S_i \end{cases}$$

Naturalmente non è possibile sapere a priori se i fa oppure non fa parte di S_i, tuttavia è possibile scegliere se includere oppure non includere i nella i-esima soluzione ottimale in base a quale delle due scelte rende maggiore il valore di Opt(i):

$$S_i = \begin{cases} S_{p(i)} \cup \{i\} & \text{se Opt}(p(i)) + v_i \geq \text{Opt}(i-1) \\ S_{i-1} & \text{altrimenti} \end{cases} \qquad \text{Opt}(i) = \max\{\text{Opt}(p(i)) + v_i; \text{Opt}(i-1)\}$$

2.1.4 Programmazione dinamica: implementazione bottom-up

L'algoritmo bottom-up viene costruito a partire dall'equazione di ricorrenza sfruttando due vettori, c e d. In ciascuna cella $c[i]$ viene riportato il valore della soluzione ottimale per la i-esima istanza del problema, mentre in ciascuna cella $d[i]$ viene riportata la soluzione stessa. I valori di $c[i]$ vengono utilizzati per calcolare i valori ottimali per le istanze successive. L'algoritmo riceve in input l'istanza X e restituisce in output la coppia $c[|X|], d[|X|]$, la soluzione del problema.
Si noti come la prima cella del vettore c possa venire riempita immediatamente con 0, come da caso base dell'equazione di ricorrenza. Per lo stesso motivo, la prima cella del vettore d puó venire riempita immediatamente con l'insieme vuoto.

```
procedure WIS(X)
    c[0] ← 0
    d[0] ← ∅

    for i ← 1 to |X| do
        if (c[i - 1] ≥ c[p[i]] + vᵢ then
            c[i] ← c[i - 1]
            d[i] ← d[i - 1]
        else
            c[i] ← c[p[i]] + vᵢ
            d[i] ← d[p[i]] ∪ {X[i]}

return c[|X|], d[|X|]
```

È facile notare come il tempo di esecuzione dell'algoritmo sia $O(n^2)$, dove n è la lunghezza del vettore X. Questo perché è presente un ciclo che per l'attività in esame deve cercare a ritroso l'attivitá con questa compatibile di peso massimo, e tale ricerca avviene esattamente n volte.

2.2 Hateville

2.2.1 Definizione del problema

Si consideri una via di una città, composta da un certo numero di case. Per la costruzione di un ospedale si richiede agli abitanti della via una donazione. Ciascun abitante è disposto a donare una certa somma di denaro, ma a condizione che entrambi i suoi vicini (se esistono) si rifiutino di partecipare.
Formalmente, sia $X = \{1, 2, \ldots, n\}$ un insieme di n abitanti: a ciascuno è associato un valore d_i, ovvero la quantitá di denaro che questi é disposto a donare. Per qualsiasi insieme $A \subseteq X$ è possibile definire un valore $D(A)$ come la somma dei valori di tutti gli abitanti che costituiscono A. Due abitanti i e j, con $i > j$, sono *compatibili* se $j \neq i-1$ e $j \neq i+1$. La nozione di compatibilità viene estesa ad un qualsiasi insieme $A \subseteq X$ assegnandovi un valore booleano Comp(A), che ha valore di verità T se tutti gli abitanti che compongo A sono fra loro compatibili, e F altrimenti.

$$D(A) = \sum_{i \in A} d_i \qquad \text{Comp}(A) = \begin{cases} T & \text{se } i-1 \notin A \wedge i+1 \notin A \forall i \in A \\ F & \text{altrimenti} \end{cases}$$

Il problema **Hateville** richiede di trovare l'insieme di abitanti tale per cui nessuno di questi è vicino di un altro ed al contempo permette di ricavare la massima quantità di denaro dalle donazioni. Formalmente, la soluzione del problema è data dall'insieme $A^* \subset X$ formato esclusivamente da abitanti fra loro compatibili che ha il massimo valore fra tutti gli insiemi con questa caratteristica:

$$\text{Comp}(A^*) = T \wedge D(A^*) = \max_{A \subset X \text{ t.c. Comp}(A)=T}\{D(A)\}$$

Il problema può essere risolto mediante programmazione dinamica. Si consideri $X_i = \{1, 2, ..., i\} \subseteq X$, l'insieme costituito dai primi i abitanti di X, che corrisponde alla i-esima istanza del problema. A tale insieme è associato l'insieme $S_i = \{1, 2, ..., i\} \subseteq X_i$, la soluzione ottimale per la i-esima istanza del problema. Il valore della soluzione S_i viene indicato con $\text{Opt}(i)$. La soluzione finale del problema è S_n, in quanto associata all'istanza X_n, che coincide con l'intero insieme X.
Per calcolare la soluzione ottimale della generica istanza X_i, ovvero S_i, verrá assunto di avere a disposizione tutte le soluzioni parziali $S_{i-1}, S_{i-2}, ..., S_1, S_0$. Pertanto, per il calcolo della soluzione ottimale dell'i-esima istanza del problema si hanno a disposizione l'abitante i-esimo ed i sottoinsiemi di valore massimo rispetto alle istanze da $i-1$ a 0. i puó essere come puó non essere uno degli abitanti del sottoinsieme S_i.

2.2.2 Programmazione dinamica: sottostruttura ottima

Proprietà della sottostruttura ottima per il problema Hateville. Sia data una istanza $X_i = \{1, 2, ..., i\}$ del problema Hateville, con $i \geq 2$. Assunto di avere a disposizione tutte le soluzioni per le istanze precedenti ad X_i, ovvero $S_1, S_2, ..., S_{i-1}$, si ha:

$$S_i = \begin{cases} S_{i-2} \cup \{i\} & \text{se } i \in S_i \\ S_{i-1} & \text{se } i \notin S_i \end{cases}$$

Dimostrazione. L'i-esimo elemento dell'istanza X_i può oppure non può fare parte della i-esima soluzione. Si distinguono allora due casi:

- $i \notin S_i$. Si supponga per assurdo che S_{i-1} non sia la soluzione ottimale del problema per l'istanza X_i. Se così è, allora la soluzione ottimale per tale istanza (quale che sia) deve necessariamente avere un valore maggiore di quello della soluzione ottimale S_{i-1}, ovvero deve valere $\text{Opt}(i) > \text{Opt}(i-1)$.
  Se, come da ipotesi, i non appartiene ad S_i, deve aversi $S_i \subseteq X_{i-1}$. Ma allora S_i è una potenziale soluzione per la $i-1$-esima istanza del problema. Ricordando però che $\text{Opt}(i) > \text{Opt}(i-1)$, questo equivale a dire che S_i è una soluzione migliore per la $i-1$-esima istanza del problema di quanto lo sia S_{i-1}, e questo è in contraddizione con l'ipotesi che la soluzione ottimale per l'istanza X_{i-1} sia S_{i-1}. Occorre allora assumere che S_{i-1} sia l'effettiva soluzione ottimale per la i-esima istanza del problema quando $i \notin S_i$.
- $i \in S_i$. Si supponga per assurdo che $S_{i-2} \cup \{i\}$ non sia la soluzione ottimale al problema per l'istanza X_i. Se così è, allora la soluzione ottimale per tale istanza (quale che sia) deve avere un valore maggiore di quello della soluzione ottimale $S_{i-2} \cup i$, ovvero deve valere $\text{Opt}(i) > \text{Opt}(i-2) + d_i$.
  Dato che, per ipotesi, i appartiene ad S_i, è possibile scomporre tale soluzione in $S_h \cup \{i\}$, con $S_h \neq S_{i-2}$, così come è possibile scomporre $\text{Opt}(i)$ in $\text{Opt}(h) + d_i$. Essendo però $\text{Opt}(i) > \text{Opt}(i-2) + d_i$, deve valere $\text{Opt}(h) + d_i > \text{Opt}(i-2) + d_i$, cioè $\text{Opt}(h) > \text{Opt}(i-2)$.
  Dovendo essere $S_h \cup \{i\}$ un insieme costituito da elementi mutualmente compatibili, la soluzione S_h non può contenere $i-1$ (così come non può contenere i), pertanto si ha $S_h \subseteq X_{i-2}$. Ma allora S_h è una potenziale soluzione per la $i-2$-esima istanza del problema. Ricordando però che $\text{Opt}(h) > \text{Opt}(i-2)$, questo equivale a dire che S_h è una soluzione migliore per la $i-2$-esima istanza del problema di quanto lo sia S_{i-2}, e questo è in contraddizione con l'ipotesi che la soluzione ottimale per l'istanza X_{i-2} sia S_{i-2}. Occorre allora assumere che $S_{i-2} \cup \{i\}$ sia l'effettiva soluzione ottimale per la i-esima istanza del problema quando $i \notin S_i$.

2.2.3 Programmazione dinamica: equazione di ricorrenza

Il caso base dell'equazione di ricorrenza è immediato: si consideri l'istanza X_0 del problema, che non contiene alcuna entità. Tale insieme è certamente composto esclusivamente da entità fra loro compatibili, non contenendone alcuno. Inoltre, per lo stesso motivo, il valore di tale insieme è nullo.
Similmente, si consideri l'istanza X_1, che contiene solamente il primo elemento: tale insieme é certamente composto esclusivamente da entità fra loro compatibili, dato che ne contiene solamente una. Inoltre, per lo stesso motivo, il valore di tale insieme coincide con il valore del solo primo elemento.

$$\begin{cases} S_0 = X_0 = \emptyset \\ S_1 = X_1 = \{1\} \end{cases} \qquad \begin{cases} \text{Opt}(0) = D(S_0) = 0 \\ \text{Opt}(1) = D(S_1) = d_1 \end{cases}$$

Per quanto riguarda la relazione di ricorrenza, possono presentarsi due casistiche: i fa parte oppure non fa parte di S_i. Se $i \notin S_i$, allora è possibile assumere che la soluzione ottimale per la i-esima istanza sia la medesima della $i-1$-istanza.
Se invece $i \in S_i$, allora la soluzione ottimale $S_{i-1} \subseteq \{1, 2, ..., i-1\}$ potrebbe dover venire esclusa, perché i e $i-1$ sono elementi certamente incompatibili e S_{i-1} potrebbe contenere $i-1$. D'altro canto, i e $i-2$ sono certamente compatibili, pertanto S_{i-2} è una possibile soluzione per la i-esima istanza, dato che non può contenere $i-1$. Tuttavia, proprio per il fatto che i e $i-2$ sono compatibili, anche $S_{i-2} \cup \{i\}$ è una soluzione, ed è certamente una soluzione migliore di S_{i-2} e di tutte le precedenti.

$$S_i = \begin{cases} S_{i-2} \cup \{i\} & \text{se } i \in S_i \\ S_{i-1} & \text{se } i \notin S_i \end{cases} \qquad \text{Opt}(i) = \begin{cases} \text{Opt}(i-2) + d_i & \text{se } i \in S_i \\ \text{Opt}(i-1) & \text{se } i \notin S_i \end{cases}$$

Naturalmente non è possibile sapere a priori se i fa oppure non fa parte di S_i, tuttavia è possibile scegliere se includere oppure non includere i nella i-esima soluzione ottimale in base a quale delle due scelte rende maggiore il valore di $\text{Opt}(i)$:

$$S_i = \begin{cases} S_{i-1} & \text{se } \text{Opt}(i-1) \geq \text{Opt}(i-2) + d_i \\ S_{i-2} \cup \{i\} & \text{altrimenti} \end{cases} \qquad \text{Opt}(i) = \max\{\text{Opt}(i-1), \text{Opt}(i-2) + d_i\}$$

2.2.4 Programmazione dinamica: implementazione bottom-up

L'algoritmo bottom-up viene costruito a partire dall'equazione di ricorrenza sfruttando due vettori, c e d. In ciascuna cella $c[i]$ viene riportato il valore della soluzione ottimale per la i-esima istanza del problema, mentre in ciascuna cella $d[i]$ viene riportata la soluzione stessa. I valori di $c[i]$ vengono utilizzati per calcolare i valori ottimali per le istanze successive. L'algoritmo riceve in input l'istanza A e restituisce in output la coppia $c[|A|], d[|A|]$, la soluzione del problema.
Si noti come la prima cella del vettore c possa venire riempita immediatamente con 0, mentre la seconda con il valore del primo elemento di A, come da caso base dell'equazione di ricorrenza. Per lo stesso motivo, la prima cella del vettore d puó venire riempita immediatamente con l'insieme vuoto, mentre la seconda con il valore del solo primo elemento di A.

```
procedure HATEVILLE(A)
    c[0] ← 0
    c[1] ← v₁
    d[0] ← ∅
    d[1] ← A[1]

    for i ← 2 to |A| do
        if (c[i - 1] ≥ c[i - 2] + vᵢ then
            c[i] ← c[i - 1]
            d[i] ← d[i - 1]
        else
            c[i] ← c[i - 2] + vᵢ
            d[i] ← d[i - 2] ∪ {A[i]}

return c[|A|], d[|A|]
```

È facile notare come il tempo di esecuzione dell'algoritmo sia $O(n)$, dove n è la lunghezza del vettore A. Questo perché è presente un ciclo che esegue una istruzione in tempo immediato esattamente $|A|$ volte.

2.3 Knapsack 0/1

2.3.1 Definizione del problema

Sia dato un insieme X costituito da "oggetti". A ciascun oggetto $i \in X$ è associato un valore v_i ed un ingombro w_i. Questi oggetti vanno riposti all'interno di uno zaino avente capacitá massima C.
La nozioni di valore e di ingombro vengono poi estese dai singoli elementi di X a sottoinsiemi A di X. Il valore di un insieme $A \subseteq X$ è dato dalla somma dei valori dei singoli oggetti di cui è costituito; allo stesso modo, l'ingombro di un insieme $A \subseteq X$ è dato dalla somma dei pesi dei singoli oggetti di cui è costituito. Naturalmente, si assume che un insieme di oggetti nullo ha associato sia peso sia valore pari a zero.

$$V(A) = \begin{cases} \sum_{i \in A} v_i \in A & \text{se } A \neq \emptyset \\ 0 & \text{se } A = \emptyset \end{cases} \qquad W(A) = \begin{cases} \sum_{i \in A} w_i \in A & \text{se } A \neq \emptyset \\ 0 & \text{se } A = \emptyset \end{cases}$$

Il problema **Knapsack 0/1** prevede di individuare il sottoinsieme di oggetti avente massimo valore complessivo e peso complessivo inferiore alla capacità dello zaino. Il numero totale di oggetti del sottoinsieme ed il suo peso totale (fintanto che questo è inferiore alla capacità dello zaino) non sono rilevanti ai fini della soluzione.
Il problema puó essere risolto mediante programmazione dinamica. Dato un insieme di oggetti X avente cardinalità n, ordinati per peso, sia X_i l'insieme costituito dai primi i oggetti di X secondo tale ordinamento. Sia poi C un valore intero positivo, che rappresenta la capacità dello zaino. La soluzione $S_{i,c}$ per la i-esima istanza del problema corrisponde a trovare la soluzione ottimale per il problema Knapsack 0/1 rispetto al sottoinsieme X_i e alla capacità massima c. La soluzione per l'intero insieme X è la soluzione per l'istanza X_n. Il valore totale della soluzione ottimale per la i-esima istanza del problema con capacità c viene indicata con $\text{Opt}(i, c)$.
Per calcolare la soluzione ottimale della generica istanza X_i con capacitá c, ovvero $S_{i,c}$, verrá assunto di avere a disposizione tutte le soluzioni parziali nella forma $S_{a,b}$, con $1 \leq a < i$ e $1 \leq b < j$. Pertanto, per il calcolo della soluzione ottimale dell'i, c-esima istanza del problema si hanno a disposizione l'i-esimo oggetto dello zaino X, la capacitá rimanente c e tutti i sottoinsiemi ottimali di oggetti delle istanze da $i-1, j-1$ a 0, 0. L'oggetto i puó come non puó fare parte del sottoinsieme ottimale di oggetti $S_{i,c}$.

A	V	W
A_1	1	7
A_2	1	4
A_3	1	5
A_4	1	1
A_5	1	1

Si consideri uno zaino con capacità totale pari a 10. Il sottoinsieme $\{A, B, C\}$ ha ingombro totale pari a $7 + 4 + 5 = 16$, pertanto non può essere una soluzione al problema. Il sottoinsieme $\{A, E\}$ ha ingombro totale pari a $7 + 1 = 8$, pertanto è una possibile soluzione, per quanto non sia quella ottimale. Questa è infatti data dal sottoinsieme $\{B, C, D\}$, che ha ingombro totale pari a $4 + 5 + 1 = 10$ e valore complessivo pari a $1 + 1 + 1 = 3$.

2.3.2 Programmazione dinamica: sottostruttura ottima

Si consideri l'insieme X_n composto da n oggetti. Data una soluzione ottimale S per tale insieme, vi sono due possibilità:

- n è parte della soluzione, ovvero $n \in S$ e $w_n < C$. Allora l'insieme $S' = S \setminus \{n\}$ è la soluzione ottimale per l'istanza data dall'insieme di oggetti $X \setminus \{n\} = X_{n-1}$ e da uno zaino di capacità $C' = C - w_n$.
- n non è parte della soluzione, ovvero $n \notin S$. Allora l'insieme S è la soluzione ottimale per l'istanza data dall'insieme di oggetti $X \setminus \{n\} = X_{n-1}$ e da uno zaino di capacità C.

Dimostrazione. La dimostrazione procede per casi:

- Innanzitutto, S' è compatibile con la capacità $C - w_n$. Infatti, avendosi $W(S') = W(S) - w_n$:

$$W(S) \leq C \;\Rightarrow\; W(S) - w_n \leq C - w_n \;\Rightarrow\; W(S') \leq C - w_n$$

  Inoltre, S' è l'insieme compatibile avente valore totale massimo. Si assuma infatti per assurdo che esista un insieme S'' tale che $V(S'') > V(S')$ e $W(S'') \leq C - w_n$. Allora $S'' \cup \{n\}$ è la soluzione ottimale per l'istanza X e C di valore totale maggiore di $V(S)$, e questo fatto è in contrasto con l'ipotesi che la soluzione ottimale per X sia S.
- Si assuma per assurdo che esista S', una soluzione ottimale per l'istanza data dall'insieme X_{n-1} e capacità C tale per cui $V(S') > V(S)$. Ma allora S' è una soluzione ottimale per l'istanza data dall'insieme X e dalla capacità C, che è in contrasto con l'ipotesi che la soluzione ottimale per X sia S.

2.3.3 Programmazione dinamica: equazione di ricorrenza

Il caso base dell'equazione di ricorrenza è semplice da determinare; se non vi è alcun oggetto da dover mettere nello zaino o se lo zaino non può contenere alcun oggetto, la soluzione ottimale è l'insieme vuoto.

$$S_{i,c} = \emptyset \text{ se } i = 0 \vee c = 0 \qquad \text{Opt}(i, c) = 0 \text{ se } i = 0 \vee c = 0$$

Per quanto riguarda il passo ricorsivo, possono verificarsi solamente due situazioni, mutualmente esclusive: l'i-esimo oggetto fa parte della soluzione ottimale $S_{i,c}$ oppure non ne fa parte.
Se i non fa parte di $S_{i,c}$, allora la soluzione per la i, c-esima istanza è la medesima per l'istanza $i-1, c$-esima, perché l'oggetto non viene inserito nello zaino. Se invece i fa parte di $S_{i,c}$, ovvero se l'oggetto viene inserito nello zaino, allora occorre cercare la miglior soluzione precedente a $S_{i,c}$ che sia compatibile con la presenza di i. Questa non puó che essere $S_{i-1,c-w_i}$, ovvero la soluzione per l'oggetto precedente ma la cui capacitá é diminuita del peso dell'oggetto corrente.

$$S_{i,c} = \begin{cases} S_{i-1,c} & \text{se } i \notin S_{i,c} \\ S_{i-1,c-w_i} \cup \{i\} & \text{se } i \in S_{i,c} \end{cases} \qquad \text{Opt}(i, c) = \begin{cases} \text{Opt}(i-1, c) & \text{se } i \notin S_{i,c} \\ \text{Opt}(i-1, c-w_i) + v_i & \text{se } i \in S_{i,c} \end{cases}$$

Naturalmente, non è possibile sapere a priori se l'i-esimo elemento di X faccia oppure non faccia parte di $S_{i,c}$, a meno che il peso dell'i-esimo oggetto sia maggiore della capacità c. In quel caso, si ha la certezza che $i \notin S_{i,c}$. Se invece il peso dell'oggetto è inferiore o uguale a c, allora i può fare parte di $S_{i,c}$; la scelta fra l'includere e il non includere i in $S_{i,c}$ è determinata da quale delle due restituisce una soluzione avente valore maggiore.

$$S_{i,c} = \begin{cases} S_{i-1,c} & \text{se } w_i > c \\ S_{i-1,c-w_i} \cup \{i\} & \text{se } w_i \leq c \wedge \text{Opt}(i-1, c-w_i) + v_i \geq \text{Opt}(i-1, c) \\ S_{i-1,c} & \text{se } w_i \leq c \wedge \text{Opt}(i-1, c-w_i) + v_i < \text{Opt}(i-1, c) \end{cases}$$

$$\text{Opt}(i, c) = \begin{cases} \text{Opt}(i-1, c) & \text{se } w_i > c \\ \max\{\text{Opt}(i-1, c-w_i) + v_i, \text{Opt}(i-1, c)\} & \text{se } w_i \leq c \end{cases}$$

2.3.4 Programmazione dinamica: implementazione bottom-up

L'algoritmo bottom-up viene costruito a partire dall'equazione di ricorrenza sfruttando una tabella d ed una tabella p. In ciascuna cella $p[i, j]$ viene riportato il valore della soluzione ottimale per la i, j-esima istanza del problema, mentre in ciascuna cella $d[i, j]$ viene riportata la soluzione stessa. I valori di $p[i, j]$ vengono utilizzati per calcolare i valori ottimali per le istanze successive. L'algoritmo riceve in input l'insieme V dei valori degli oggetti, l'insieme W dei pesi degli oggetti e la capacità totale C e restituisce in output la coppia $d[|V|, C], p[|V|, C]$, la soluzione del problema.

```
procedure KNAPSACK(V, W, C)
    for i ← 0 to |V| do
        d[i, 0] ← ∅
        p[i, 0] ← 0
    for c ← 0 to C do
        d[0, c] ← ∅
        p[0, c] ← 0

    for i ← 1 to |V| do
        for c ← 1 to C do
            if (W[i] > c) then
                d[i, c] ← d[i - 1, c]
                p[i, c] ← p[i - 1, c]
            else
                if (p[i - 1, c - W[i]] + V[i] ≥ p[i - 1, c]) then
                    d[i, c] ← d[i - 1, c - W[i]] ∪ {i}
                    p[i, c] ← p[i - 1, c - W[i]] + V[i]
                else
                    d[i, c] ← d[i - 1, c]
                    p[i, c] ← p[i - 1, c]

    return d[|V|, C], p[|V|, C]
```

Il tempo di esecuzione dell'algoritmo è $O(nC)$, dove n è il numero di oggetti nell'insieme X. Infatti, l'algoritmo è costituito da una coppia di cicli for innestati, dove il ciclo più esterno esegue il ciclo interno esattamente n volte, mentre il ciclo interno esegue una serie di operazioni aventi tempo di esecuzione unitario.

2.3.5 Programmazione dinamica: ricostruzione di una soluzione

É possibile ottimizzare l'algoritmo osservando come non sia necessario tenere traccia di tutte le soluzioni parziali, ma é sufficiente calcolare la soluzione ottima a partire dalla tabella dei valori ottimali. Pertanto, La procedura viene modificata eliminando ogni riferimento alla tabella d e restituendo non la coppia $d[|V|, C]$, $p[|V|, C]$, ma bensí l'intera tabella p.
Alla vecchia procedura opportunamente modificata ne viene aggiunta un'altra, `PRINT-KNAPSACK` . Questa ha in input la tabella p, l'insieme dei pesi W, una capacità c ed un indice i, mentre in output ha uno dei possibili insiemi di oggetti aventi massimo valore totale.
La prima chiamata alla procedura ha $|X|$ come valore per i e C come valore per c in modo da ottenere la soluzione per l'istanza $|X|, C$.
Se il valore di i o di c é nullo, allora la procedura termina, perché si é raggiunto il caso base in cui la soluzione é l'insieme vuoto. Se sia i che c non sono nulli ma la capacitá c é inferiore al peso dell'i-esimo oggetto, allora questo certamente non potrá essere parte della soluzione ottimale. Se sia i che c non sono nulli e la capacitá c é superiore al peso dell'i-esimo oggetto allora quest'ultimo sará parte della soluzione ottimale solamente se la sua inclusione ha comportato un valore totale maggiore rispetto alla sua esclusione.

```
procedure KNAPSACK(V, W, C)
    for i ← 0 to |V| do
        for c ← 0 to C do
            p[i, c] ← 0

    for i ← 1 to |V| do
        for c ← 1 to C do
            if (W[i] > c) then
                p[i, c] ← p[i - 1, c]
            else
                // per motivi di spazio
                temp ← p[i - 1, c - W[i]] + V[i]
                if (temp ≥ p[i - 1, c]) then
                    p[i, c] ← temp
                else
                    p[i, c] ← p[i - 1, c]

    return p
```

```
procedure PRINT-KNAPSACK(p, W, i, c)
    if (i = 0 or c = 0) then
        return

    if (c ≥ W[i]) then
        if (p[i, c] = p[i - 1, c]) then
            PRINT-KNAPSACK(p, W, i - 1, c)
        else
            PRINT-KNAPSACK(p, W, i - 1, c - W[i])
            print i
    else
        PRINT-KNAPSACK(p, W, i - 1, c)
```

Sebbene la procedura sia ricorsiva, il suo tempo di esecuzione è comunque proporzionale al numero di oggetti e alla capacità massima, perché la ricorsione è una tail-recursion.

A	**V**	**W**
A_1	1	7
A_2	1	4
A_3	1	5
A_4	1	1
A_5	1	1

0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	2	2
0	1	1	1	1	2	2	2	2	2	3
0	1	2	2	2	2	3	3	3	3	3

Capitolo 3
Tecnica di programmazione greedy

3.1 Introduzione alla tecnica greedy

La **tecnica greedy** è un tipo di approccio alla costruzione di algoritmi che risolvono problemi di ottimizzazione. La tecnica greedy permette di calcolare la soluzione ottima (di una istanza) di un problema attraverso una sequenza di scelte localmente ottime, dove ogni scelta compiuta non dipende da quelle successive. Le differenze con la tecnica di programmazione dinamica sono riassunte di seguito:

Tecnica di programmazione dinamica	**Tecnica greedy**
Restituisce il valore ottimo, a partire dal quale si ricava una soluzione ottima	Restituisce direttamente la soluzione ottima
Bottom-up	Top-down
Il problema è scomposto in molti sottoproblemi	Il problema è scomposto in pochi sottoproblemi
L'algoritmo è complesso e spesso non efficiente	L'algoritmo è semplice e efficiente
Permette di risolvere molti problemi	Permette di risolvere (in maniera esatta) pochi problemi

L'approccio greedy alla risoluzione di un problema è possibile solamente se viene dimostrata matematicamente la **proprietà della scelta greedy**, ovvero se la scelta che viene compiuta ad ogni iterazione appartiene ad una soluzione ottima del sottoproblema in esame. A sua volta, affinché tale proprietà sia vera, è necessario dimostrare che il problema possiede la proprietà della sottostruttura ottima (come per la programmazione dinamica).
Gli algoritmi basati sulla tecnica greedy vengono anche spesso utilizzati per dare soluzioni approssimate a problemi per i quali una soluzione esatta sarebbe troppo costosa e/o non necessaria. Esistono peró alcuni problemi, fra cui quelli presentati di seguito, dove algoritmi basati sull'approccio greedy restituiscono soluzioni esatte: se questo si verifica, allora é quasi garantito che l'algoritmo greedy sia quello in grado di risolvere il problema con le migliori prestazioni.

3.2 Problema interval scheduling

Sia dato un insieme $A = \{a_1, a_2, ..., a_n\}$ costituito da "attività". Ciascuna attività a_i è definita come una coppia (s_i, f_i), dove i due valori indicano rispettivamente il tempo di inizio dell'attività ed il tempo di fine.
Due attività a_i e a_j distinte si dicono *compatibili* se a_i termina prima che a_j inizi, ovvero se $f_i \leq s_j$. Più in generale, un sottoinsieme X di A è detto *compatibile* se è costituito da attività fra di loro mutualmente compatibili, o equivalentemente se non contiene alcuna coppia di elementi di A fra di loro non compatibili.

$$\text{Comp}(i,j) = \begin{cases} T & \text{se } f_i \leq s_j \\ F & \text{se } f_i > s_j \end{cases} \qquad \text{Comp}(X) = \begin{cases} T & \text{se } \nexists i,j \in X \text{ t.c. } f_i > s_j \\ F & \text{se } \exists i,j \in X \text{ t.c. } f_i > s_j \end{cases}$$

Il problema **Interval Scheduling** chiede di trovare il sottoinsieme X di A di cardinalità massima composto da attività mutualmente compatibili [1].

1. Si noti la differenza con il problema Weighted Interval Scheduling: tale problema chiede di trovare il sottoinsieme di valore totale massimo, mentre il problema Interval Scheduling chiede di trovare quello di cardinalità massima.

Il problema Interval Scheduling è risolvibile mediante programmazione dinamica. Dato un insieme $A = \{a_1, a_2, ..., a_n\}$ di attività, si ordinino tali attività per tempo di fine non decrescente. Affinché un algoritmo di programmazione dinamica possa essere applicabile è necessario aggiungere due attività "slack", a_0 e a_{n+1}, rispettivamente in prima ed in ultima posizione. Sia allora $A' = A \cup \{a_0, a_{n+1}\}$.
Sia $X_{i,j}$ il sottoinsieme di A' composto da tutte le attività che vengono dopo a_i e prima di a_j, ovvero $X_{i,j} = \langle a_{i+1}, a_{i+2}, ..., a_{j-2}, a_{j-1} \rangle$.
La soluzione $S_{i,j}$ per la i, j-esima istanza del problema corrisponde a trovare il sottoinsieme di attività mutualmente compatibili di cardinalità massima rispetto al sottoinsieme $X_{i,j}$. La cardinalità del sottoinsieme di attività mutualmente compatibili di cardinalità massima per la i, j-esima istanza del problema viene indicata con $\text{Opt}(i, j)$.
La soluzione ottimale per il problema principale è data da $S_{0,n+1}$. Infatti, tale insieme è la soluzione per l'insieme che contiene gli elementi di A' che vengono dopo a_0 e prima di a_{n+1}, che corrisponde esattamente a $A' - \{a_0, a_{n+1}\} = A$.
Il caso base si ha per l'insieme $X_{i,i+1}$, ovvero quando $i + 1 = j$. Infatti, l'insieme costituito da tutte le attività che vengono dopo a_i e prima di a_{i+1} è, per definizione, l'insieme vuoto, perché le due attivitá sono contigue.

$$S_{i,i+1} = \emptyset \qquad \text{Opt}(i, i+1) = |\emptyset| = 0$$

Per quanto riguarda il passo ricorsivo, si consideri una generica soluzione ottimale $S_{i,j}$. Tale soluzione può essere scritta come $S_{i,k} \cup \{a_k\} \cup S_{k,j}$, dove $S_{i,k}$ e $S_{k,j}$ sono le soluzioni ottimali rispettivamente delle istanze $X_{i,k}$ e $X_{k,j}$ e dove a_k è il valore che permette di restituire tali soluzioni ottimali.

$$S_{i,j} = \max\{S_{i,k} \cup \{a_k\} \cup S_{k,j} \mid i < k < j\} \qquad \text{Opt}(i, j) = \max\{\text{Opt}(i, k) + 1 + \text{Opt}(k, j) \mid i < k < j\}$$

È facile verificare che un algoritmo di programmazione dinamica basato su tale equazione avrebbe un tempo di esecuzione (almeno) quadratico. Questo perché per ottenere la soluzione ottimale per una intera istanza è necessario calcolare le soluzioni ottimali per tutte le istanze $X_{a,b}$ con $0 \leq a \leq n$ e $0 \leq b \leq n$.
Il problema Interval Scheduling può essere risolto in maniera più semplice e più efficiente applicando la tecnica greedy. Dato un insieme A di attività, l'algoritmo greedy procede come segue:

```
procedure INTERVAL-SCHEDULING(A)
1    A ← A ordinato per tempo di fine non descrescente
2    X ← {a₁}
3    k ← 1

5    for i ← 2 to |A| do
6        if sᵢ ≥ fₖ then
7            X ← {aₛ} ∪ X
8            k ← i

9    return X
```

1. Gli elementi di A vengono ordinati per tempo di fine non decrescente;
2. Viene creato l'insieme soluzione X, che inizialmente contiene soltanto a_1;
3. Sia a_k l'ultimo elemento dell'insieme X, ovvero l'elemento di X avente tempo di fine maggiore. Se l'i-esimo elemento di A é compatibile con a_k, allora viene aggiunto ad X diventando il nuovo a_k;
4. Se tutte le attività di A sono state considerate l'algoritmo termina, altrimenti si riprende dal punto precedente.

Tale algoritmo è effettivamente un algoritmo greedy, perché la scelta localmente ottima viene considerata anche globalmente ottima. Tuttavia, affinché questo possa considerarsi corretto, occorre dimostrare la validità della proprietà della scelta greedy, ovvero che l'attività che viene per prima in una certa iterazione (soluzione ottimale locale) è effettivamente membro della soluzione ottimale globale. A dire il vero, dato che ad ogni iterazione l'insieme A viene ridotto e l'attività che viene considerata è sempre la prima, per dimostrare la validità della proprietà della scelta greedy è sufficiente dimostrare che l'attività a_1, quella che nell'ordine temporale viene prima di tutte, è sempre parte della soluzione ottima.

Proprietà della scelta greedy per il problema Interval Scheduling. Sia dato un insieme di attività $A = \langle a_1, a_2, ..., a_n \rangle$ ordinate per tempo di fine non decrescente. L'attività a_1 è sempre parte della soluzione ottima.

Dimostrazione. Sia X la soluzione ottima per l'insieme A. Sia poi a'_1 l'attività in X avente minor tempo di fine. Naturalmente, se a'_1 coincide con a_1, allora la dimostrazione è terminata; altrimenti, si sostituisca in X l'attività a'_1 con a_1. Il nuovo insieme X ha mantenuto la stessa cardinalità ed è ancora costituito da attività mutualmente compatibili, pertanto il nuovo X è un sottoinsieme massimo di attività compatibili di A che include ora a_1.

Si noti come l'algoritmo che risolve il problema Interval Scheduling applicando la tecnica greedy è nettamente più veloce di quello che lo risolve applicando la programmazione dinamica. Infatti, indicando con n la cardinalità di A, si ha che il loop principale (righe 5-8) esegue esattamente $|n|$ volte un blocco di codice con tempo di esecuzione unitario, mentre l'ordinamento di A avviene (assumendo di usare un algoritmo di ordinamento efficiente) in tempo $O(n\log(n))$. Pertanto, asintoticamente, il tempo di esecuzione complessivo viene ad essere $O(n\log(n)) + O(n) = O(n\log(n))$.

3.3 Problema dello zaino frazionario

Siano dati n oggetti $X = \langle 1, 2, ..., n \rangle$, un valore intero C e due funzioni V e W. Tali funzioni associano a ciascun oggetto un numero intero, rispettivamente un valore ed un peso. La definizione di peso e di valore viene poi generalizzata ad un sottoinsieme di oggetti come la somma dei pesi e dei valori di ciascun oggetto di cui è costituito.

$$V : X \mapsto N \qquad V(i) = v_i$$

$$W : X \mapsto N \qquad W(i) = w_i$$

Il **problema dello Zaino Frazionario** chiede di trovare una sequenza di quantità di ciascun oggetto che permette di massimizzare il valore complessivo di tale sequenza ed al contempo essere inferiore o uguale alla capacità dello zaino.
Più formalmente, indicando con $P = \langle p_1, p_2, ..., p_n \rangle$ con $p_i \in [0, 1]$ per ogni $1 \leq i \leq n$ la sequenza di percentuali con cui ciascun i-esimo oggetto viene aggiunto ad una certa soluzione, il problema chiede di trovare la sequenza P^* tale per cui:

$$\sum_{i=1}^{n} p^*_i w_i \leq C \qquad P^* = \left\{ P \mid \max\left\{ \sum_{i=1}^{n} p_i v_i \right\} \right\}$$

> Sia dato l'insieme di tre oggetti $X = \langle (10, 20), (9, 8), (8, 5) \rangle$ ed una capacità $C = 20$. Una soluzione ottimale al problema dello Zaino Frazionario è data dalla sequenza $P^* = \langle 0.35, 1, 1 \rangle$, avente valore complessivo $V(P^*) = 0.35^*10 + 1^*9 + 1^*8 = 20.5$ e peso complessivo $W(P^*) = 0.35^*20 + 1^*8 + 1^*5 = 20$.

Il problema può essere risolto applicando la tecnica greedy. Innanzitutto, dato un certo insieme di oggetti $X = \langle 1, 2, ..., n \rangle$, per ciascun i-esimo oggetto sia $s_i = v_i \ / \ w_i$ il *valore specifico* [2] di tale oggetto, ovvero il suo valore per unità di peso.
Si ordini l'insieme X in ordine non crescente rispetto ai valori specifici degli oggetti di cui è costituito. Ad ogni iterazione dell'algoritmo greedy viene selezionata la quantità dell'i-esimo oggetto che viene scelta (non superiore ad 1) e la capacità dello zaino viene ridotta di tanto quanto è il peso di tale quantità. L'algoritmo prevede di prendere, per ciascun oggetto, la massima quantità compatibile con la capacità residua dello zaino, fermandosi quando è stata raggiunta la capacità massima.

```
procedure FRACTIONAL-KNAPSACK(n, V, W, C)
 1    for i ← 1 to n do
 2        S[i] ← V[i] / W[i]

 3    S ← S ordinato per valore specifico non crescente
 4    V ← V ordinato secondo S
 5    W ← W ordinato secondo S
 6    i ← 1
 7    c ← C

 8    while i ≤ n and c > 0 do
 9        P[i] ← c / W[i]
10        if (P[i] > 1) then
11            P[i] ← 1
12        c ← c - P[i] * W[i]
13        i ← i + 1

14    return P
```

Tale algoritmo è effettivamente un algoritmo greedy, perché la scelta localmente ottima viene considerata anche globalmente ottima. Tuttavia, affinché questo possa considerarsi corretto, occorre dimostrare la validità della proprietà della scelta greedy, ovvero che la percentuale di oggetto che viene scelta per ciascuna iterazione (soluzione ottimale locale) è effettivamente membro della soluzione

2. Il nome "valore specifico" è dato in analogia con il concetto di **peso specifico**, ovvero il peso di un metro cubo di una certa sostanza.

ottimale globale. A dire il vero, dato che ad ogni iterazione viene sempre considerato il primo oggetto e la capacità dello zaino diminuisce sempre di una quantità relativa al peso di quest'ultimo, per dimostrare la validità della proprietà della scelta greedy è sufficiente dimostrare che la massima quantità compatibile con l'intera capacità C del primo oggetto, quello che ha il valore specifico maggiore di tutti, è sempre parte della soluzione ottima.

Proprietà della scelta greedy per il problema dello Zaino Frazionario. Sia dato un insieme di oggetti $X = \langle 1, 2, ..., n \rangle$ ordinati per valore specifico non crescente ed una capacità C. La massima percentuale del primo oggetto compatibile C che può essere aggiunta alla soluzione ottima è sempre parte di quest'ultima.

Dimostrazione. La percentuale che viene scelta per il primo oggetto è $p_1 = \min(C \ / \ w_1, 1)$, la massima possibile. Sia P una soluzione ottima: dato che l'algoritmo sceglie sempre la massima percentuale possibile, la percentuale p'_1 che si trova in P può essere esclusivamente uguale oppure inferiore a p_1.
Se p'_1 e p_1 sono uguali, la dimostrazione è terminata. Se p'_1 è inferiore a p_1, è possibile costruire una nuova soluzione in cui la percentuale del primo oggetto è uguale a p_1. La proporzione di uno o più oggetti aggiunti successivamente sarà ridotta di conseguenza. Dato che il valore specifico del primo oggetto è superiore o uguale a quello dei successivi non è possibile ottenere una soluzione avente valore inferiore. Si conclude quindi che p_1 è sempre parte di una soluzione ottima.

3.4 Matroidi e sistemi di indipendenza

Siano dati un insieme finito S ed un insieme non vuoto F, sottoinsieme dell'insieme potenza di S. La coppia (S, F) è detta **sistema di indipendenza** se, per ciascun elemento di F, anche tutti i sottoinsiemi di tale elemento appartengono ad F.

$$(S, F \subseteq \text{Pow}(S)) \text{ con } A \in F \wedge B \subset A \Rightarrow B \in F \ \forall A \in F, \forall B \subset A$$

Ciascun elemento dell'insieme F è detto **sottoinsieme indipendente** di S. Si noti come, affinché una coppia (S, F) possa essere un sistema di indipendenza, l'insieme F deve almeno contenere l'insieme vuoto, perché l'insieme vuoto è sottoinsieme di ogni insieme.

Si considerino gli insiemi S e F così definiti:

$$\begin{cases} S = \{1, 2, 3\} \\ F \subset \text{Pow}(S) = \{\emptyset, \{1\}, \{3\}, \{1, 3\}\} \end{cases} \qquad \text{Pow}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

La coppia (S, F) costituisce un sistema di indipendenza. Infatti, tutti i sottoinsiemi dell'insieme vuoto sono in F (sé stesso), tutti i sottoinsiemi dell'insieme $\{1\}$ sono in F (l'insieme vuoto), tutti i sottoinsiemi dell'insieme $\{3\}$ sono in F (l'insieme vuoto) e tutti i sottoinsiemi dell'insieme $\{1, 3\}$ sono in F ($\{1\}$, $\{3\}$ e l'insieme vuoto).

Un sistema di indipendenza (S, F) è detto **matroide** se possiede la **proprietà di scambio**, ovvero se per qualsiasi coppia di insiemi in F, dove il primo ha un elemento in meno del secondo, esiste almeno un elemento del secondo insieme non presente nel primo tale che aggiunto a quest'ultimo restituisce un insieme che appartiene ancora ad F.

$$(S, F \subseteq \text{Pow}(S)) \text{ con } \forall A, B \in F \text{ t.c. } |B| = |A| + 1 \ \exists b \in B - A \text{ t.c. } \{b\} \cup A \in F$$

Siano S un insieme finito generico e F il sottoinsieme dell'insieme potenza di S che contiene tutti gli elementi di Pow(S) con cardinalità inferiore o uguale a k, con k numero naturale fissato.
La coppia (S, F) costituisce un sistema di indipendenza perché, preso un qualsiasi insieme $f \in F$, tutti i suoi sottoinsiemi hanno cardinalità inferiore a f, che a sua volta ha cardinalità inferiore o uguale a k. Pertanto, anche tutti i sottoinsiemi di f fanno parte di F.
Inoltre, la coppia (S, F) costituisce un matroide. Siano infatti due insiemi qualsiasi $A, B \in F$ tale che $|B| = |A| + 1$. Avendo A un elemento in meno di B, si ha $|A| < k$. Aggiungendo un qualsiasi elemento $b \in B - A$ all'insieme A si ottiene l'insieme $\{b\} \cup A$ che ha cardinalità $|A| + 1 = |B|$, che per ipotesi è un valore inferiore o uguale a k, e che quindi appartiene ad F.

Sia $M = (S, F)$ un matroide. Un elemento $s \in S$ è detto **estensione** dell'insieme $A \in F$ se $s \notin A$ e $A \cup \{s\} \in F$, ovvero aggiungendo s ad A (ed A non contiene s) si ottiene ancora un sottoinsieme indipendente di S. Se non esiste alcun elemento s che possa essere aggiunto ad A tale che $A \cup \{s\} \in F$, si dice che A è **massimale**.

Tutti i sottoinsiemi indipendenti massimali in un matroide hanno la stessa cardinalità.

Dimostrazione. Sia $M = (S, F)$ un matroide. Si supponga per assurdo che vi siano (almeno) due sottoinsiemi indipendenti massimali in M, A e B, tali che $|A| < |B|$.
Essendo M un matroide ed essendo $|A| \neq |B|$, deve essere a questi applicabile la proprietà di scambio. Ovvero, deve esistere un elemento $b \in B - A$ tale che $\{b\} \cup A \in F$, tuttavia questo contraddice l'ipotesi che A sia massimale e che quindi non possa esistere alcun b. Si deduce quindi che due (o più) sottoinsiemi A e B indipendenti massimali in uno stesso matroide aventi cardinalità diversa non possano esistere.

Sia (S, F) un sistema di indipendenza e sia C un sottoinsieme di S. La coppia (C, F_C) con $F_C = \{A \in F \text{ t.c. } A \subseteq C\}$ è il **sottosistema di indipendenza indotto** da C.

Un qualsiasi sottosistema indotto (C, F_C) di un matroide (S, F) è esso stesso un matroide se i suoi insiemi massimali hanno la stessa cardinalità.

Un sistema di indipendenza (S, F) è (anche) un matroide se e solo se per ogni $C \subseteq S$ gli insiemi massimali di (C, F_C) hanno la stessa cardinalità.

Un **matroide pesato** è un matroide $M = (S, F)$ a cui viene aggiunta una funzione peso W, la quale associa ad ogni elemento dell'insieme S un numero reale strettamente positivo. Tale funzione viene estesa ad un generico insieme A membro di F come la somma dei pesi di tutti gli elementi contenuti in A. Un qualsiasi sottoinsieme indipendente di un matroide pesato avente peso massimo è detto **sottoinsieme ottimo** del matroide.

$$W : S \mapsto \mathbb{R}^+ \qquad W(s) = w_s \; s \in S \qquad W(A) = \sum_{a \in A}^{|A|} w_a \; A \in F$$

3.5 Algoritmo Greedy Standard

Il legame fra matroidi e problemi risolvibili mediante tecnica greedy è molto forte. Infatti, molti problemi per i quali un metodo greedy fornisce soluzioni ottime possono essere riformulati come la ricerca di uno dei sottoinsiemi ottimi di un matroide pesato. L'algoritmo che, fornitogli in input un matroide pesato, permette di ricavarne un sottoinsieme ottimo prende il nome di **algoritmo greedy standard**.

Tale algoritmo riceve in input un matroide pesato $M = (S, F, W)$ e restituisce un sottoinsieme ottimo A. L'algoritmo è effettivamente un algoritmo greedy perché esamina uno dopo l'altro in ordine di peso ogni elemento $x \in S$ e lo aggiunge immediatamente all'insieme A se $A \cup \{x\}$ è esso stesso un sottoinsieme indipendente.

```
procedure STANDARD-GREEDY-ALGORITHM(S, F, W)
1    A ← ∅
2    S ← S ordinato per peso W decrescente

3    for i ← 1 to |S| do
4        if ({sᵢ} ∪ A) ∈ F then
5            A ← {sᵢ} ∪ A

6    return A
```

L'algoritmo greedy standard restituisce sempre un sottoinsieme indipendente.

Dimostrazione. Si faccia riferimento allo pseudocodice dell'algoritmo greedy standard, presentato sopra. L'insieme che questo restituisce, A, é inizialmente l'insieme vuoto, che per definizione é un sottoinsieme indipendente. La riga 4 controlla se, aggiungendo un elemento s_i ad A, si ottiene un membro di F, ovvero se $A \cup \{s_i\}$ é a sua volta un sottoinsieme indipendente: se questo accade, la riga 5 sostituisce A con $A \cup \{s_i\}$. Poiché l'insieme A viene inizializzato come sottoinsieme indipendente e poiché ogni esecuzione del corpo del ciclo principale se modifica A lo lascia indipendente, per principio di induzione l'insieme restituito dall'algoritmo sará certamente un sottoinsieme indipendente.

Per adattare l'algoritmo greedy standard ad un certo problema occorre solamente, se è possibile farlo, determinare una condizione (riga 4) che permetta di discriminare gli elementi dell'istanza del problema che fanno parte del sottoinsieme ottimo da quelli che non ne fanno parte. Il restante corpo dell'algoritmo è sostanzialmente sempre lo stesso. Molti algoritmi greedy (ma non tutti) sono infatti riducibili all'algoritmo greedy standard, dove la generica funzione di valutazione dell'istanza viene sostituita da una funzione specifica che varia da problema a problema.
Il tempo di esecuzione dell'algoritmo greedy standard può essere calcolato osservando che questo può essere diviso in tre parti: l'ordinamento dell'insieme S (riga 2), il ciclo for (riga 3) e la condizione di appartenenza dell'elemento in esame alla soluzione del problema (righe 4 e 5). La prima ha tempo di esecuzione pari a quella di un normale ordinamento, ovvero $O(n\log(n))$, mentre la seconda ha tempo di esecuzione proporzionale alla dimensione dell'input, ovvero $O(n)$. La terza non ha un tempo di esecuzione predefinito, ma varia da problema a problema, e viene indicata con un generico $O(f(n))$. Questa viene esaminata n volte, pertanto il tempo di esecuzione complessivo della seconda e della terza parte è $O(n) \times O(f(n)) = O(nf(n))$. Si ha quindi che il tempo di esecuzione dell'intero algoritmo è dato da $O(n\log(n)) + O(nf(n))$.

Teorema di Rado (I). Un sistema di indipendenza è (anche) un matroide se, per qualsiasi funzione peso, l'algoritmo greedy standard fornisce il sottoinsieme ottimo per tale sistema.

Dimostrazione. Siano (S, F) un matroide e W una qualsiasi funzione peso. Sia poi $X = \{s_1, s_2, ..., s_p\}$ la soluzione fornita dall'algoritmo greedy standard per tale matroide, con gli elementi disposti in ordine decrescente di peso, ovvero $W(s_1) \geq W(s_2) \geq ...W(s_p)$. Naturalmente, essendo il risultato dell'applicazione dell'algoritmo greedy standard, l'insieme X è massimale per (S, F).
Sia allora $X' = \{s_1', s_2', ..., s_p'\}$ un altro insieme massimale, distinto da X, avente anch'esso gli elementi ordinati in ordine decrescente di peso. Avendo X e X' la stessa cardinalità, è possibile mettere in corrispondenza biunivoca l'i-esimo elemento di X con l'i-esimo elemento di X'. Il teorema è dimostrato se, per un qualsiasi insieme X' massimale per (S, F), il peso $W(X)$ è almeno pari al peso $W(X')$.
Se ciascun elemento di X ha peso maggiore del corrispettivo elemento di X' allora certamente $W(X) \geq W(X')$. Ma questo significa che X, l'insieme restituito dall'algoritmo greedy standard, ha peso almeno pari a quello di un generico insieme massimale X', ed il teorema è dimostrato.
Si consideri il caso più generico in cui non tutti gli elementi di X hanno peso maggiore dei loro corrispettivi in X'. Ricordando che gli elementi di X e di X' sono disposti in ordine di peso, deve esistere almeno un elemento con indice k tale per cui tutti gli elementi di X con indice inferiore a k hanno peso maggiore dei rispettivi elementi di X' mentre il k-esimo elemento di X ha peso inferiore al k-esimo elemento di X'.
In particolare, deve certamente aversi $W(s_k) \leq W(s_{k-1})$ e $W(s_k') \leq W(s_{k-1}')$. Ricordando però che vale $W(s_k) < W(s_k')$, si ha allora $W(s_k) < W(s_k') \leq W(s_{k-1}') \leq W(s_{k-1})$. Si consideri allora $C = \{s \in S \text{ t.c } W(s) \geq W(s_k')\}$, il sottoinsieme di S che contiene tutti gli elementi con peso maggiore o uguale al peso di s_k'. La coppia (C, F_C) forma un sottosistema indotto per C.
L'elemento s_k' appartiene certamente a C, perché evidentemente $W(s_k') \geq W(s_k')$. Ma allora $X' \cap C = \{s_1', s_2', ..., s_{k-1}', s_k'\}$. D'altro canto, l'elemento s_k non può appartenere a C, perché per ipotesi è stato assunto che $W(s_k) < W(s_k')$. Ma allora $X \cap C = \{s_1, s_2, ..., s_{k-1}\}$. Questo significa che $X' \cap C$ ha un elemento in più di $XcapC$, e quindi $|X' \cap C| > |X \cap C|$. Questo contraddice però l'ipotesi che X e X' siano insiemi massimali per (S, F), ed implica che non possa esistere un k tale per cui $W(s_k) < W(s_k')$.
Occorre allora concludere che ciascun elemento di X deve per forza avere peso maggiore o uguale al peso del rispettivo elemento di X'. Ma questa situazione coincide con la situazione precedente, ed il teorema è dimostrato.

Teorema di Rado (II). Se l'algoritmo greedy standard fornisce, per qualsiasi funzione peso, il sottoinsieme ottimo per un sistema di indipendenza, allora tale sistema di indipendenza è (anche) un matroide.

Dimostrazione. Sia (S, F) un sistema di indipendenza che non è un matroide. Allora esiste un insieme $C \subseteq S$ e due insiemi massimali $A, B \in (C, F_C)$ con cardinalità diversa. Sia $|B| = p$ e $|A| > |B|$. La seconda parte del teorema è dimostrata se è possibile costruire almeno una funzione peso per la quale l'algoritmo greedy standard applicato a (S, F) restituisce un insieme non ottimo. Si consideri a tal proposito la funzione peso W così costruita:

$$W(s) = \begin{cases} p+2 & \text{se } s \in B \\ p+1 & \text{se } s \in (A-B) \\ 1 & \text{altrimenti} \end{cases}$$

L'algoritmo greedy standard restituirebbe l'insieme B, perché la funzione peso attribuisce peso maggiore agli elementi di B. L'insieme A ha almeno $p+1$ elementi, perché per ipotesi $|A| > |B|$ e $|B| = p$. Inoltre, ciascun elemento presente in A ma non in B (almeno uno certamente esiste) ha peso $p+1$. Pertanto, il peso complessivo di A è certamente almeno pari a $(p+1)(p+1)$. L'insieme B ha, per ipotesi, p elementi ciascuno di peso $p+2$, pertanto il peso complessivo di B è $p+2$. Si noti però come:

$$W(A) \geq (p+1)(p+1) = (p+1)^2 \geq p^2 + 2p + 1 > p^2 + 2p = p(p+2) = W(B)$$

Ovvero, $W(A) > W(B)$. Questo significa che, per questa funzione peso, l'algoritmo greedy standard non restituisce l'insieme di peso maggiore per il sistema di indipendenza (S, F), ed il teorema è provato.

Il Teorema di Rado implica che se un problema può essere formulato come un matroide pesato allora esiste un algoritmo greedy, basato sull'algoritmo greedy standard, che lo risolve per qualsiasi funzione peso. Se invece non è possibile, potrebbe comunque esistere un algoritmo greedy basato sull'algoritmo greedy standard in grado di risolvere il problema, ma solo per alcune specifiche funzioni peso, non tutte. Inoltre, il teorema non dice nulla a proposito degli algoritmi greedy che non sono basati sull'algoritmo greedy standard.

Sia $V = v_1, v_2, ..., v_n$ un insieme di vettori m-dimensionali. Ciascun vettore v_i è costituito da $m-1$ componenti e da un peso, assegnato da una funzione W. Si richiede di determinare il sottoinsieme di vettori linearmente indipendenti che ha peso massimo, applicando l'algoritmo greedy standard.
Sia S l'insieme finito di vettori, e sia poi F la famiglia dei sottoinsiemi di S composti da vettori linearmente indipendenti. Affinché l'algoritmo greedy standard sia applicabile, la tripla (S, F, W) deve essere codificabile come un matroide, che a sua volta richiede di dimostrare che la tripla è un sistema di indipendenza.
Sia $A \in F$ un qualsiasi insieme di vettori linearmente indipendenti estratti da F. Se da A vengono eliminati uno o più vettori, si ottiene un insieme A' ancora costituito da vettori linearmente indipendenti, pertanto anche A' appartiene ad F. Pertanto, (S, F, W) è un sistema di indipendenza.
Siano $A, B \in F$ due insiemi di vettori linearmente indipendenti estratti da F, dove $|A| = |B| + 1$. È sempre possibile scegliere un elemento $b_i \in B$ tale per cui $A \cup \{b_i\} \in F$, perché se non esistesse tale vettore allora l'insieme B non sarebbe costituito da vettori linearmente indipendenti.
È quindi possibile risolvere il problema utilizzando l'algoritmo greedy standard; è sufficiente utilizzare l'appartenenza ad un insieme di vettori linearmente indipendenti come condizione di appartenenza.

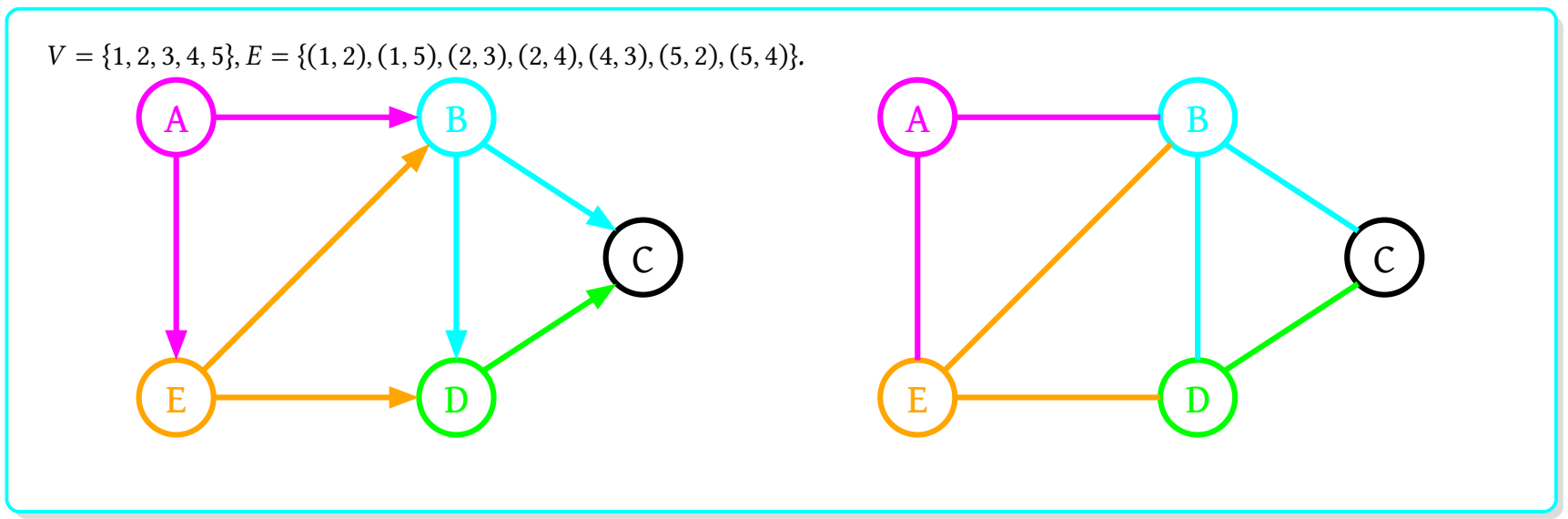
```
procedure greedy_ind_vectors(V)
    X ← ∅
    V ← V ordinato per peso W decrescente

    for i from 1 to n do
        if ({vᵢ} ∪ X contiene vettori linearmente indipendenti) then
            X ← {vᵢ} ∪ X

    return X
```

Nonostante l'algoritmo greedy standard restituisca sempre il sottoinsieme ottimo di peso massimo, è comunque possibile utilizzarlo per risolvere problemi di ottimizzazione di minimo (ammesso che questo sia risolvibile mediante tecnica greedy). Una volta convertito il problema in esame in un matroide pesato $M = (S, F, W)$ equivalente, viene selezionato il peso $w_0 \in W$ di valore massimo. A questo punto la funzione peso W viene sostituita con un'altra funzione W', il cui elemento i-esimo w_i' è dato dalla differenza fra w_0 e w_i, l'i-esimo elemento di W.
In questo modo, ordinare gli elementi di S rispetto a W' in ordine decrescente equivale ad ordinarlo rispetto a W in ordine crescente, e di conseguenza restituire il sottoinsieme ottimo massimale rispetto a W' equivale a restituire l'insieme ottimo minimale rispetto a W.

Capitolo 4
Strutture dati

4.1 Strutture dati per grafi

Si definisce **grafo** una coppia $G = (V, E)$, costituita da un insieme $V = \{v_1, v_2, ..., v_n\}$ di elementi detti **vertici** e da un insieme $E = \{e_1, e_2, ..., e_m\}$ di **archi**, ovvero di coppie ordinate di elementi di V. Il numero di elementi di V e di E indicano la **dimensione** del grafo. Per convenzione, i vertici del grafo vengono numerati a partire da 1.
Ogni arco $e_k = (v_i, v_j)$ indica l'esistenza di una relazione R tra i due vertici v_i e v_j. Se la relazione R è simmetrica, ovvero se per qualsiasi coppia (v_i, v_j) l'esistenza di $v_i R v_j$ implica l'esistenza di $v_j R v_i$, il grafo si dice **non orientato**. Se invece la relazione non è simmetrica, il grafo si dice **orientato**. Un vertice si dice **adiacente** ad un altro vertice se esiste un arco che ha tali vertici come elementi. Ovvero, v_i è adiacente a v_j in un grafo $G = (V, E)$ se $(v_i, v_j) \in E$. Un arco (v_i, v_i), ovvero un arco che unisce un vertice con sé stesso, viene chiamato **cappio**.
Un grafo può venire rappresentato in forma estensionale elencando gli elementi dei due insiemi di cui è costituito. In alternativa, può essere rappresentato graficamente riportando i nodi come cerchi numerati e gli archi come frecce che connettono tali cerchi. Se il grafo è orientato, le frecce hanno una punta, mentre se non è orientato sono piatte.



Una prima struttura dati in grado di contenere le informazioni relative ad un grafo é la cosiddetta **lista di adiacenza**. Questa consiste in un vettore L di dimensione $|V|$, dove ciascun elemento $L[i]$ è una lista che contiene tutti i vertici rispetto a cui l'i-esimo vertice del grafo è adiacente.
Le liste di adiacenza riportano solamente le informazioni relative a quali nodi e quali archi esistono all'interno del grafo, ed infatti il numero di elementi che contiene è $|V| + |E|$. D'altro canto, per sapere se una certa coppia di vertici del grafo che rappresenta sono adiacenti è una operazione eseguibile in tempo lineare, perché occorre "scorrere" le liste relative ai due vertici fino a trovare (se esistono) una corrispondenza. Pertanto, le liste di adiacenza sono una codifica vantaggiosa per i grafi **sparsi**, ovvero i grafi il cui numero di archi è di molto inferiore al numero di vertici.
Una seconda struttura dati in grado di contenere le informazioni relative ad un grafo é la cosiddetta **matrice di adiacenza**. Questa consiste in una matrice M di dimensione $|V \times V|$, dove ciascuna cella $M[i, j]$ contiene il valore 1 se l'i-esimo ed il j-esimo vertice del grafo sono adiacenti, e 0 altrimenti.
La matrice di adiacenza deve riportare esplicitamente informazioni in merito all'esistenza o alla non esistenza di ogni arco, ed infatti il numero di elementi che contiene è $|V|^2$. D'altro canto, sapere se una certa coppia di vertici del grafo che rappresenta sono adiacenti è una operazione eseguibile in tempo costante, perché è sufficiente leggere il valore della cella che ha tale vertici per indici. Pertanto, le matrici di adiacenza sono una codifica vantaggiosa per i grafi **densi**, ovvero i grafi il cui numero di archi è vicino al numero di vertici.
Si noti come le due strutture dati siano del tutto equivalenti. Ovvero, se é nota la rappresentazione in forma di lista di adiacenza per un grafo, é sempre possibile convertirla in una matrice di adiacenza che rappresenta lo stesso grafo, e viceversa.

$V = \{1, 2, 3, 4, 5\}, E = \{(1, 2), (1, 5), (2, 3), (2, 4), (4, 3), (5, 2), (5, 4)\}$

$$
\begin{aligned}
&A \Rightarrow B \Rightarrow E \\
&B \Rightarrow C \Rightarrow D \\
&C \\
&D \Rightarrow C \\
&E \Rightarrow B \Rightarrow D
\end{aligned}
$$

	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	1	0
C	0	0	0	0	0
D	0	0	1	0	0
E	0	1	0	1	0

$$
\begin{aligned}
&A \Rightarrow B \Rightarrow E \\
&B \Rightarrow A \Rightarrow C \Rightarrow D \Rightarrow E \\
&C \Rightarrow B \Rightarrow D \\
&D \Rightarrow B \Rightarrow C \Rightarrow E \\
&E \Rightarrow A \Rightarrow B \Rightarrow D
\end{aligned}
$$

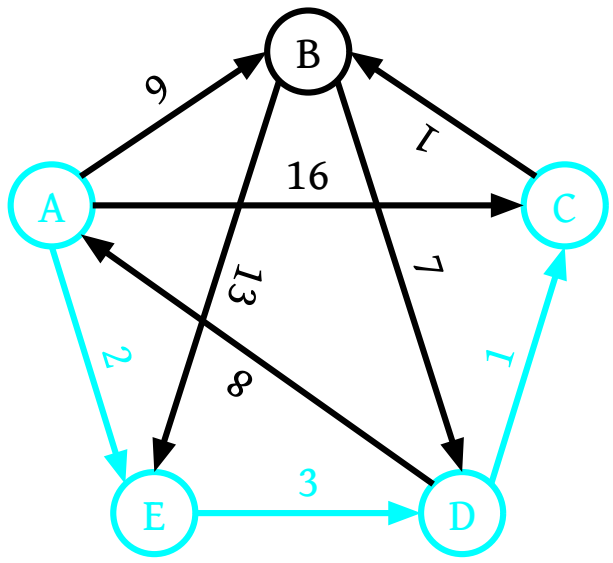
	A	B	C	D	E
A	0	1	0	0	1
B	1	0	1	1	1
C	0	1	0	1	0
D	0	1	1	0	1
E	1	1	0	1	0

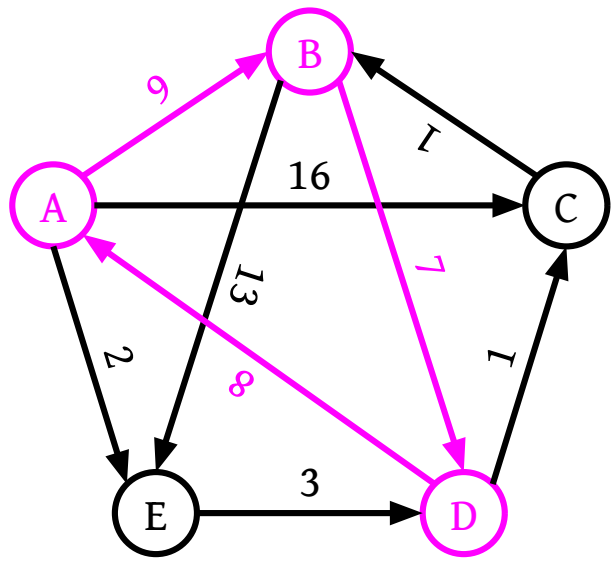
Dato un grafo $G = (V, E)$, prende il nome di **cammino** una qualsiasi sequenza ordinata di vertici tali per cui, ad eccezione dell'ultimo, ciascuno di essi ha un arco con il suo successore. Formalmente, un cammino dal vertice v_a al vertice v_b é dato dalla k-upla $\langle v_a, v_{a+1}, ..., v_{b-1}, v_b \rangle$, dove per ciascuna coppia di vertici v_i e v_{i+1} con $a \leq i < b$ si ha che (v_i, v_{i+1}) é un elemento di E.
Il numero di archi di un cammino meno uno viene detto **lunghezza** del cammino. Un cammino dove il primo vertice coincide con l'ultimo prende il nome di **ciclo**. Un cammino in cui ogni suo vertice compare una sola volta, ovvero che non contiene alcun ciclo, prende il nome di **cammino semplice**. Tutti i vertici che compongono un cammino ad eccezione del primo e dell'ultimo si dicono **vertici intermedi**.
Un **grafo orientato pesato** è costituito da una tripla $G = (V, E, W)$ dove, oltre all'insieme $V = \{v_1, ..., v_n\}$ di vertici e all'insieme $E = \{e_1, ..., e_m\}$ di archi, figura la funzione $W : V \times V \mapsto \mathbb{R}^+$. Tale funzione restituisce, per ciascuna coppia di vertici in $V \times V$, un numero positivo w_{ij} chiamato **peso** dell'arco (v_i, v_j). Per convenzione, se l'arco (v_i, v_j) non è presente nel grafo, si ha $W(v_i, v_j) = \infty$, mentre se $v_i = v_j$ si ha $W(v_i, v_j) = 0$. Il peso complessivo di un cammino è dato dalla somma dei pesi degli archi che costituiscono il cammino.

$$
W(v_i, v_j) = \begin{cases} 0 & \text{se } v_i = v_j \\ \infty & \text{se } v_i \neq v_j \wedge (v_i, v_j) \notin E \\ w_{ij} & \text{altrimenti} \end{cases}
$$

$$
W(\langle v_a, v_{a+1}, ..., v_{b-1}, v_b \rangle) = \sum_{i=a}^{b-1} W(v_i, v_{i+1})
$$

Il cammino blu ha peso 2 + 3 + 1 = 6, mentre il cammino rosa ha peso 8 + 9 + 7 = 24. Il primo è un cammino semplice, mentre il secondo no.





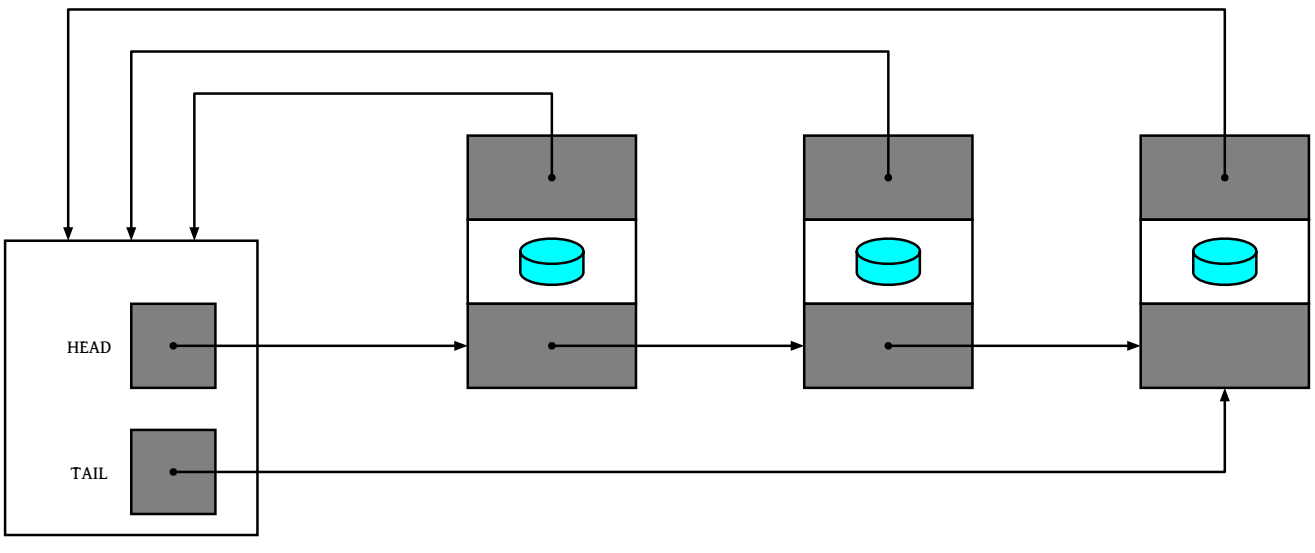
4.2 Strutture dati per insiemi disgiunti

Una **struttura dati per insiemi disgiunti** mantiene una collezione $\{S_1, S_2, ..., S_k\}$ di insiemi dinamici disgiunti, ovvero insiemi di cardinalità non fissata i cui elementi non possono trovarsi in più di un insieme contemporaneamente. Ciascun insieme è identificato da un **rappresentante**, che è un elemento dell'insieme. Quale elemento questo debba essere è lasciato all'implementazione: può sia essere un elemento qualunque, sia un elemento che possiede una certa caratteristica all'interno dell'insieme (l'elemento più piccolo, l'elemento che ha associato il valore maggiore, ecc...).
Si vuole costruire una struttura dati che supporti le seguenti tre operazioni:

- `MAKE-SET(x)` crea un nuovo insieme il cui unico elemento nonché rappresentante è x. Poiché gli insiemi sono disgiunti, x non può trovarsi in un altro insieme;
- `UNION(x, y)` unisce gli insiemi (univoci) che contengono gli elementi x e y, siano questi S_x e S_y, in un unico insieme, che è l'unione di questi due, che viene poi aggiunto alla collezione. Naturalmente, si assume che S_x e S_y siano disgiunti prima di applicare l'operazione. Il rappresentante di $S_x \cup S_y$ diviene uno qualsiasi degli elementi di S_x o di S_y [1]. Al fine di mantenere la proprietà di disgiunzione degli insiemi della collezione, occorre eliminare dalla collezione S_x e S_y;
- `FIND-SET(x)` restituisce il rappresentante dell'insieme (univoco) che contiene x.

4.2.1 Implementazione mediante liste concatenate

Una prima implementazione di una struttura dati per insiemi disgiunti prevede di rappresentare ciascun insieme mediante una lista concatenata. L'oggetto di ciascun insieme ha gli attributi `head`, che punta al primo oggetto della lista, e `tail`, che punta all'ultimo oggetto. Ogni oggetto nella lista contiene un elemento dell'insieme, un puntatore al successivo oggetto della lista e un puntatore che ritorna all'oggetto dell'insieme. All'interno di ciascuna lista concatenata, gli oggetti possono apparire in qualsiasi ordine. Il rappresentante è l'elemento dell'insieme nel primo oggetto della lista.
Con questa rappresentazione, entrambe le operazioni `MAKE-SET` e `FIND-SET` possono essere implementate per avere tempo di esecuzione unitario. Per implementare `MAKE-SET` è sufficiente creare una nuova lista concatenata il cui unico oggetto è x. Per implementare `FIND-SET` occorre seguire il puntatore da x per arrivare all'oggetto del suo insieme e poi ritornare all'elemento nell'oggetto cui punta `head`.



La più semplice implementazione dell'operazione `UNION(x, y)` richiede un tempo di esecuzione nettamente superiore a quello di `MAKE-SET(x)` e di `FIND-SET(x)`. Per eseguire `UNION(x, y)` viene aggiunta la lista di y alla fine della lista di x, ed il rappresentante di x diviene rappresentante dell'insieme risultante. Questo richiede di aggiornare il puntatore all'oggetto dell'insieme per ogni oggetto che si trovava in y, e questa operazione richiede tempo lineare nella lunghezza della lista di y.
Si supponga di avere un insieme di n oggetti $x_1, x_2, ..., x_n$. Si eseguano n operazioni `MAKE-SET` seguite da $n-1$ operazioni `FIND-SET`. Il tempo per eseguire le n operazioni `MAKE-SET` è lineare in n. Poiché l'i-esima operazione `UNION` aggiorna i oggetti, il numero totale di oggetti aggiornati da tutte le $n-1$ operazioni `UNION` è $\sum_{i=1}^{n-1} i = O(n^2)$. Il numero totale di operazioni è $2n-1$, pertanto ciascuna operazione richiede in media un tempo $\Theta(n)$.
Nel caso peggiore, l'implementazione dell'operazione `UNION` richiede un tempo medio $\Theta(n)$ per chiamata, perché potrebbe venire unita una lista più lunga ad una più corta. Se si introducesse un campo aggiuntivo a ciascuna lista che ne riporta la lunghezza (è sufficiente aggiungere un contatore che viene aggiornato ad ogni `UNION`), sarebbe sempre possibile scegliere di unire la lista più corta a quella

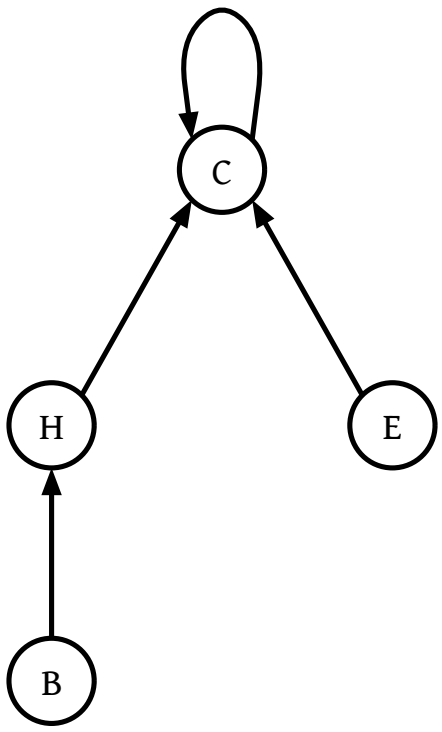
1. In alcune implementazioni si richiede che il rappresentante dell'insieme unione sia sempre un elemento del primo insieme o sempre un elemento del secondo insieme.

più lunga. Con questa euristica, chiamata **euristica dell'unione pesata**, una singola operazione `UNION` richiederebbe ancora un tempo $\Omega(n)$ se entrambi gli insiemi hanno $\Omega(n)$ elementi.

> Utilizzando la rappresentazione degli insiemi disgiunti mediante liste concatenate e sfruttando l'euristica dell'unione pesata, una sequenza di m operazioni `MAKE-SET`, `FIND-SET` e `UNION`, dove n di queste sono `MAKE-SET`, richiede un tempo complessivo $O(m + n\log(n))$.

4.2.2 Implementazione mediante foreste

Una implementazione più efficiente degli insiemi disgiunti prevede di utilizzare degli alberi radicati. Ciascun albero rappresenta un insieme, ciascun nodo contiene un oggetto ed il nodo radice contiene l'oggetto rappresentante. Ogni nodo che non sia la radice punta al nodo padre, mentre il nodo radice ha un puntatore a sé stesso.
Con questa implementazione, l'operazione `MAKE-SET` ha ancora tempo di esecuzione unitario, perché consiste nel creare un albero avente un solo nodo, con un puntatore a sé stesso, che contiene un oggetto (che sarà rappresentante del relativo insieme). L'operazione `FIND-SET` scorre i puntatori fino a trovare un oggetto che puntatore a sé stesso, che per definizione è la radice dell'albero; tale cammino semplice verso la radice prende il nome di **cammino di ricerca**. L'operazione `UNION` consiste semplicemente nello "sganciare" il puntatore dalla radice a sé stessa di uno dei due alberi e collegarlo all'altra radice.



L'operazione di `UNION` così come descritta finora non fornisce un miglioramento in termini di prestazioni in confronto a `UNION` rispetto all'implementazione con liste concatenate. Infatti, applicando $n-1$ volte `UNION` può creare un albero che è una catena lineare di n nodi. Inoltre, l'operazione `FIND-SET` così definita ha tempo lineare, mentre la versione dell'implementazione mediante liste aveva tempo unitario.
Per migliorare il tempo di esecuzione dell'operazione `UNION` viene introdotta l'euristica **unione per rango**. Questa prevede di fare in modo che, quando è necessario unire due alberi, l'albero da cui viene sganciato il puntatore sia sempre quello con meno nodi. Per conoscere questa informazione non é necessario che ogni nodo tenga traccia della dimensione del sottoalbero che ha tale nodo per radice: è infatti sufficiente che memorizzi il proprio **rango**, ovvero il numero di archi nel cammino semplice più lungo fra tale nodo e una foglia sua discendente. Tale valore viene sfruttato come limite superiore all'altezza del nodo. Durante l'operazione di `UNION` fra due alberi, l'unione per rango prevede che la radice con il rango più piccolo venga fatta puntare alla radice con il rango più grande.
Quando `MAKE-SET` crea un insieme, il rango iniziale dell'unico nodo nel corrispondente albero è `0`. Quando viene applicata `UNION` a due alberi si presentano due possibili casi: le radici dei due alberi hanno ranghi diversi oppure hanno lo stesso rango. Nel primo caso, la radice di rango più basso viene collegata alla radice di rango più alto, ottenendo un solo albero e lasciando tutti i ranghi inalterati. Nel secondo caso, viene arbitrariamente scelta una delle due radici come radice del nuovo albero ed il rango di questa viene incrementato di uno.

Dato un nodo `x`, siano `x.rank` il valore intero che ne riporta il rango, e sia `x.p` il padre del nodo `x`. La procedura `LINK`, una subroutine chiamata da `UNION`, riceve in input i puntatori a due radici.

```
procedure MAKE-SET(x)
1    x.p ← x
2    x.rank ← 0



procedure UNION(x, y)
3    LINK(FIND-SET(x), FIND-SET(y))
```

```
procedure LINK(x, y)
 4    if (x.rank > y.rank) then
 5        y.p ← x
 6    else if (x.rank = y.rank) then
 7        x.p ← y
 8        y.rank ← y.rank + 1
 9    else
10        x.p ← y
```

Per migliorare il tempo di esecuzione dell'operazione `FIND-SET` viene invece introdotta l'euristica **compressione del cammino**. Questa fa in modo che ciascun nodo del cammino di ricerca punti direttamente alla radice lasciando inalterati i ranghi dei nodi.

```
procedure FIND-SET(x)
1    if (x ≠ x.p) then
2        x.p ← FIND-SET(x.p)
3    return x.p
```

La procedura `FIND-SET` è un **metodo a doppio passaggio**. Durante il primo passaggio, risale il cammino di ricerca per trovare la radice, mentre durante il secondo passaggio discende il cammino di ricerca per aggiornare i modi in modo che puntino direttamente alla radice. Ogni chiamata di `FIND-SET(x)` restituisce `x.p` nella riga 3. Se `x` è la radice, allora la riga 2 non viene eseguita e viene restituito `x.p` che è uguale a `x`. Questo è il caso in cui la ricorsione tocca il fondo. Altrimenti, viene eseguita la riga 2 e la chiamata ricorsiva con il parametro `x.p` restituisce un puntatore alla radice. La riga 2 aggiorna il nodo `x` in modo che punti direttamente alla radice, mentre la riga 3 restituisce questo puntatore.
Utilizzando sia l'unione per rango sia la compressione del cammino è possibile ottenere un tempo di esecuzione per le operazioni `UNION` e `FIND-SET` approssimativamente lineare. Infatti, date m operazioni (di cui n `MAKE-SET` e $n-1$ `UNION`) su n oggetti, il tempo di esecuzione nel caso peggiore è $O(m\alpha(n))$, dove $\alpha(n)$ è la **funzione di Ackermann inversa**. Questa funzione cresce estremamente lentamente, tanto che per qualsiasi applicazione pratica il valore restituito da tale funzione è ≤ 4. Pertanto, il tempo di esecuzione è ragionevolmente approssimabile come $O(m\alpha(n)) \approx O(4m) = O(m)$.
Una delle tante applicazioni delle strutture dati per insiemi disgiunti consiste nel determinare le componenti connesse di un grafo non orientato. La procedura `CONNECTED-COMPONENTS` usa le operazioni degli insiemi disgiunti per calcolare le componenti connesse di un grafo. Una volta che `CONNECTED-COMPONENTS` ha preprocessato il grafo, la procedura `SAME-COMPONENT` è in grado di determinare se due vertici sono nella stessa componente connessa.

```
procedure CONNECTED-COMPONENTS(V, E)
1    foreach v ∈ V
2        MAKE-SET(v)
3    foreach (u, v) ∈ E
4        if (FIND-SET(u) ≠ FIND-SET(v)) then
5            UNION(u, v)
```

```
procedure SAME-COMPONENT(u, v)
6    if (FIND-SET(u) = FIND-SET(v)) then
7        return T
8    else
9        return F
```

Capitolo 5
Cammini minimi su grafi

5.1 Cammini minimi da sorgente unica: algoritmo di Dijkstra

Sia dato un grafo $G = (V, E, W)$ non orientato, pesato e connesso. Sia poi $v_s \in V$ un vertice "privilegiato", chiamato **vertice sorgente**. Dato il vertice generico $v_k \in V$, sia P_{sk} il cammino su G che ha inizio in v_s e fine in v_k avente il peso complessivo piú piccolo possibile, detto **cammino minimo** da v_s a v_k.
Il **problema dei cammini minimi da sorgente unica** richiede di trovare, rispetto ad un vertice fissato $v_s \in G$, uno qualsiasi dei cammini minimi che hanno inizio in v_s e fine in ciascun vertice $v_k \in S - \{v_s\}$. Dati due vertici v_i e v_j di un grafo pesato G, viene indicato con $\delta(v_i, v_j)$ il peso di uno qualsiasi dei cammini minimi che hanno inizio in v_i e fine in v_j.
Un algoritmo che permette di risolvere il problema dei cammini minimi da sorgente unica è l'**Algoritmo di Dijkstra**. Tale algoritmo é applicabile esclusivamente a grafi pesati connessi, non orientati e aventi una funzione peso che restituisce solo valori strettamente positivi. Prima di descrivere l'algoritmo, é necessario compiere alcune osservazioni in merito alla natura dei cammini minimi.

Proprietà della sottostruttura ottima per il problema dei cammini minimi. Dato un grafo pesato $G = (V, E, W)$, siano v_h e v_k due vertici di G. Sia poi P_{hk} un cammino minimo su G che ha inizio nel vertice v_h e fine nel vertice v_k. Si considerino due indici i, j qualsiasi tali per cui $h \leq i \leq j \leq k$: ogni sottocammino P_{ij} di P_{hk}, avente inizio in v_i e fine in v_k, è a sua volta un cammino minimo.

Dimostrazione. Il cammino P_{hk} può essere separato in tre sottocammini: il cammino P_{hi} da v_h a v_i, il cammino P_{ij} da v_i a v_j ed il cammino P_{jk} da v_j a v_k.

$$P_{hk} = \langle v_h, v_{h+1}, ..., v_i, ..., v_j, ..., v_{k-1}, v_k \rangle = P_{hi} \rightsquigarrow P_{ij} \rightsquigarrow P_{jk} \qquad W(P_{hk}) = W(P_{hi}) + W(P_{ij}) + W(P_{jk})$$

Si supponga per assurdo che esista un cammino P'_{ij} tale che $W(P'_{ij}) < W(P_{ij})$. Sia $P'_{hk} = P_{hi} \rightsquigarrow P'_{ij} \rightsquigarrow P_{jk}$: questo è un cammino che ha inizio in v_h e fine in v_k il cui peso complessivo è inferiore a quello di P_{hk}:

$$W(P'_{ij}) < W(P_{ij}) \;\Rightarrow\; W(P'_{ij}) + W(P_{hi}) + W(P_{jk}) < W(P_{ij}) + W(P_{hi}) + W(P_{jk}) \;\Rightarrow\; W(P'_{hk}) < W(P_{hk})$$

Ma allora P_{hk} non puó essere il cammino minimo che ha inizio in v_h e fine in v_k, perché esiste almeno un cammino (P'_{hk}) che ha inizio e fine negli stessi vertici ma peso complessimo inferiore. Questo é peró in contrasto all'ipotesi secondo la quale P_{hk} sia il cammino minimo che ha tali vertici per estremi. Si deduce allora che il cammino P'_{ij} non puó esistere, e che quindi P_{ij} é effettivamente il cammino minimo che ha inizio in v_i e fine in v_j.

Dati due vertici u e v, sia w il predecessore di v lungo il cammino minimo che ha inizio in u e fine in v, ovvero il penultimo vertice che costituisce il cammino. Per indicare che v_i é il predecessore di v_j rispetto ad un determinato cammino viene usata la notazione $\pi(v_j) = v_i$.
É possibile scomporre tale cammino minimo nel sottocammino che ha inizio in u e fine in w unito all'arco che unisce w a v. Sulla base della proprietá della sottostruttura ottima sopra riportata, deve aversi che il peso di un cammino minimo fra u e v é uguale alla somma fra il peso di un cammino minimo fra u e w ed il peso dell'arco che unisce w e v.

$$\delta(s, v) = \delta(s, u) + W(u, v)$$

Nel caso in cui u non sia il predecessore di v sul cammino minimo da s a v, ma esiste comunque un arco (u, v) che li unisce, deve aversi:

$$\delta(s, v) \leq \delta(s, u) + W(u, v)$$

Questo perché se si mantiene il medesimo $\delta(s, u)$, il valore $W(u, v)$ non può che essere maggiore del peso dell'arco minimo che unisce u a v, altrimenti coinciderebbe con tale arco.

Dato un grafo non orientato e pesato $G = (V, E, W)$ ed una sorgente $s \in V$, l'algoritmo di Dijkstra associa a ciascun vertice v di G un campo d ed un campo π: il primo contiene un limite superiore per $\delta(s, v)$, mentre il secondo riporta uno dei vertici con i quali v è congiunto da un arco. L'algoritmo inizializza, per ogni vertice, il campo π a NULL ed il campo d a ∞, tranne per s, a cui viene assegnato 0.

```
procedure INITIALIZE-SINGLE-SOURCE(V, s)
 1    foreach v ∈ V do
 2        v.d ← ∞
 3        v.π ← NULL
 4    s ← 0
```

Alla fine dell'esecuzione, per ciascun vertice v il campo d contiene $\delta(s, v)$, ovvero l'effettivo peso di un cammino minimo dalla sorgente a v, mentre il campo π riporta il predecessore di v in tale cammino minimo. Questo viene fatto operando una serie di "miglioramenti", detti **rilassamenti**.
Presi due vertici u e v, se $v.d$ è maggiore della somma fra $u.d$ ed il peso dell'arco che unisce u e d, allora significa che il cammino che ha inizio nella sorgente e fine in v passando per u come predecessore è un cammino con peso inferiore, e quindi migliore, di quello finora trovato che abbia inizio nella sorgente e fine in v. Il rilassamento consiste in nient'altro che la sostituzione di $v.d$ con $u.d + W(u, v)$ e di $v.\pi$ con u.

```
procedure RELAX(u, v, W)
 1    if (v.d > u.d + W(u, v)) then
 2        v.d ← u.d + W(u, v)
 3        v.π ← u
```

In particolare, l'algoritmo di Dijkstra opera esattamente un solo rilassamento per ciascun vertice. Dato un grafo non orientato, pesato e connesso $G = (V, E, W)$, il procedimento è riportato di seguito:

1. A ciascun vertice del grafo viene associato un campo d, inizializzato a ∞, ed un campo π, inizializzato a NULL.
2. Viene scelto il noto sorgente, indicato con s, il cui campo d viene impostato a 0. Dopodiché, tutti i vertici vengono inseriti in una coda di min-priority, dove la priorità è determinata dal valore del campo d;
3. Viene estratto il vertice avente campo d con valore minore, sia questo v;
4. Per ciascun vertice u adiacente a v viene operato il rilassamento: se il valore $v.d$ è maggiore della somma fra $u.d$ e $W(u, v)$, allora $v.d$ viene sostituito con $u.d + W(u, v)$ e $v.\pi$ viene sostituito con u;
5. Se la coda non è vuota si riprende l'esecuzione dal punto 3, altrimenti l'algoritmo termina.

Una volta che l'algoritmo é terminato, é possibile costruire a ritroso un cammino minimo per ciascun vertice risalendo di predecessore in predecessore fino alla sorgente, unico vertice che avrá ancora NULL come campo predecessore.

```
procedure DIJKSTRA(V, E, W, s)
1    INITIALIZE-SINGLE-SOURCE(V, s)
2    Q coda di min priority
3    foreach v ∈ V do
4        ENQUEUE(v, Q)

5    S ← ∅
6    while Q ≠ ∅ do
7        u ← POP(Q)
8        S ← S ∪ {u}
9        foreach v ∈ ADJ(u) do
10           RELAX(u, v, W)
```

```
procedure PRINT-MINPATH(v)
 1   do
 2       print v
 3       v.π ← v
 4   while v.π ≠ NULL
```

Correttezza dell'algoritmo di Dijkstra. Dato un grafo connesso, non orientato e pesato $G = (V, E, W)$, sia $s \in V$ un vertice eletto a sorgente. L'algoritmo di Dijkstra restituisce correttamente la lunghezza di un cammino minimo dalla sorgente s a tutti i vertici in V.

Dimostrazione. Sia $\langle v_1 = s, v_2, ..., v_n \rangle$ l'ordine con cui i vertici del grafo vengono estratti dalla coda durante l'esecuzione dell'algoritmo di Dijkstra. L'algoritmo è corretto se, quando viene estratto il k-esimo vertice, si ha $v_k.d = \delta(s, v_k)$ per tutti i $k = 1, 2, ..., n$. Tale proprietà può essere dimostrata per induzione.
Il caso base è dimostrato se $s.d = \delta(s, s)$ quando viene estratto s. Dato che per raggiungere s da s non occorre percorrere alcun cammino, la distanza da s a sé stesso è 0. L'algoritmo inizializza a 0 il valore di $s.d$, quindi quando s viene estratto il valore di $s.d$ è effettivamente 0.
Per quanto riguarda il passo ricorsivo, occorre mostrare che per il vertice v_k, quando viene estratto, vale $v_k.d = \delta(s, v_k)$ assumendo che la relazione valga per v_{k-1}. Si supponga per assurdo che questo non sia vero, e che quindi $v_k.d \neq \delta(s, v_k)$ quando l'algoritmo estrae v_k dalla coda.
Il vertice v_k non può essere s, perché è stato appena mostrato che il teorema per s è valido. Deve allora essere un vertice estratto dopo s, e deve necessariamente esistere un cammino che va da s a v_k, dato che altrimenti si avrebbe $\delta(s, v_k) = \infty$ e questo contraddirebbe l'ipotesi assunta per assurdo. Esistendo (almeno) un cammino da s a v_k, esisterà anche (almeno) un cammino minimo da s a v_k.
Sia v_y il primo vertice lungo il cammino minimo da s a v_k ad essere ancora parte della coda di priorità, e sia v_x il predecessore di v_y: per definizione, questo deve essere giá stato estratto dalla coda di prioritá. Il cammino minimo da s a v_k può essere allora scomposto come $s \rightsquigarrow v_x \rightarrow v_y \rightsquigarrow v_k$. Si noti come v_k potrebbe coincidere con v_y, così come s potrebbe coincidere con v_x, pertanto i sottocammini $s \rightsquigarrow v_x$ e $v_y \rightsquigarrow v_k$ potrebbero essere vuoti.
Alla fine dell'iterazione in cui viene estratto v_x, essendo v_x predecessore di v_y si avrà $v_y.d = \delta(s, v_y)$ (lo stesso accadrá per tutti gli altri vertici adiacenti a v_x). Poiché v_y precede v_k nel cammino minimo che ha inizio in s e fine in v_k e poiché tutti i pesi degli archi sono non negativi, deve aversi $\delta(s, v_y) \leq \delta(s, v_k)$. Essendo poi v_k non ancora estratto, si avrá $v_k.d = \infty$. Riassumendo:

$$v_y.d = \delta(s, v_y) \leq \delta(s, v_k) \leq v_k.d$$

Tuttavia, poiché entrambi i vertici v_k e v_y si trovavano ancora nella coda quando v_k viene estratto, si ha $v_k.d \leq v_y.d$. Ma allora le disuguaglianze sopra espresse sono di fatto delle uguaglianze:

$$v_y.d = \delta(s, v_y) = \delta(s, v_k) = v_k.d$$

Da cui si ha $\delta(s, v_k) = v_k.d$, che è in contraddizione con l'ipotesi assunta per assurdo. Occorre quindi concludere che effettivamente $\delta(s, v_k) = v_k.d$ quando tale vertice viene estratto, e di conseguenza il teorema è provato.

Per calcolare il tempo di esecuzione dell'algoritmo di Dijkstra, si assuma innanzitutto di stare utilizzando una struttura dati efficiente, ad esempio un min-heap binario, per implementare la coda di prioritá Q.
Si osservi come `INITIALIZE-SINGLE-SOURCE` abbia chiaramente tempo di esecuzione lineare nel numero dei vertici, dato che compie una operazione avente tempo costante esattamente una sola volta per ciascuno di essi. Allo stesso modo, l'inserimento di tutti i vertici all'interno della coda di prioritá ha tempo di esecuzione lineare nel numero degli stessi. La riga 5 che inizializza l'insieme S é una operazione con tempo di esecuzione unitario, e quindi trascurabile. Riassumendo, il tempo di esecuzione complessivo delle righe da 1 a 5 é $O(|V|)$.
Il ciclo `while` alla riga 6 esegue il suo corpo una sola volta per ciascun vertice, dato che nella coda di priorità vengono inseriti tutti i vertici e ad ogni iterazione uno di questi viene sempre estratto. Estrarre un vertice da una coda di priorità (riga 7) è una operazione che ha tempo di esecuzione logaritmico nel numero dei vertici. D'altro canto, inserire il vertice estratto nell'insieme S (riga 8) é una operazione con tempo di esecuzione unitario.
Dato che ciascun vertice viene inserito in S esattamente una sola volta, l'algoritmo ispeziona ciascun arco esattamente una sola volta. La procedura di rilassamento su una coppia di archi ha tempo di esecuzione unitario: dato che questa viene operata esattamente tante volte quanti sono i vertici del grafo, il ciclo `foreach` ha tempo di esecuzione lineare nel numero degli archi.
Il tempo di esecuzione complessivo della seconda parte dell'algoritmo (righe da 6 a 10) viene allora ad essere $O(|E|\log(|V|))$, al quale va sommato il tempo di esecuzione di `INITIALIZE-SINGLE-SOURCE`, che é $O(|V|)$. Dato che si sta considerando un grafo connesso, é possibile maggiorare $|V|$ con $|E|$. Il tempo di esecuzione complessivo dell'algoritmo di Dijkstra viene allora ad essere $O(|V|)+O(|E|\log(|V|)) = O(|E|\log(|V|)) = O(|E|\log(|E|))$.

5.2 Cammini minimi da ogni sorgente: algoritmo di Floyd-Warshall

Dato un grafo $G = (V, E, W)$ senza cappi, orientato e pesato, il **problema dei cammini minimi da ogni sorgente** richiede, per tutte le possibili coppie di vertici $(v_i, v_j) \in V \times V$, di trovare il cammino che ha inizio in v_i e fine in v_j avente il minimo peso.
Il problema potrebbe essere risolto applicando al grafo l'algoritmo di Dijkstra tante volte quanti sono i vertici del grafo, cambiando ad ogni iterazione il vertice sorgente. Si noti peró come questo sia possibile solamente se la funzione peso del grafo restituisce valori strettamente positivi e se il grafo non é orientato.
Il problema dei cammini minimi da ogni sorgente può essere formulato come un problema di ottimizzazione di minimo:

- Per ciascuna coppia di vertici (v_i, v_j), l'insieme delle soluzioni possibili è dato da tutti i possibili cammini che hanno inizio in v_i e fine in v_j;
- La funzione obiettivo è il peso del cammino;
- Il valore ottimo per v_i e v_j è il peso del cammino minimo che ha inizio in v_i e fine in v_j;
- La soluzione ottimale è data da uno qualsiasi dei cammini di peso minimo che hanno inizio in v_i e fine in v_j;

Dato un grafo orientato e pesato $G = (V, E, W)$, se ne numerino i vertici in maniera univoca. Supponendo che i vertici di G siano $V = \{v_1, v_2, ..., v_n\}$, se ne consideri un sottoinsieme $K = \{v_1, v_2, ..., v_k\}$ per un k generico. Presi due vertici $v_i, v_j \in V$, si indichi con P_{ij}^k un cammino minimo da v_i a v_j i cui vertici intermedi sono stati estratti dall'insieme K.
Parametrizzando il problema rispetto ai k vertici intermedi, si ottiene un algoritmo di programmazione dinamica chiamato **Algoritmo di Floyd-Warshall**. Questo risolve il problema dei cammini minimi da ogni sorgente per grafi orientati la cui funzione peso puó restituire valori negativi L'algoritmo può essere descritto in maniera informale come segue:

1. Per tutte le possibili coppie di vertici (v_i, v_j) si calcoli S_{ij}^0, la lunghezza del cammino minimo da v_i a v_j che non ha alcun vertice intermedio. Tali lunghezze possono essere calcolate direttamente dai dati del problema;
2. Si utilizzi tale informazione per calcolare, per tutte le possibili coppie di vertici (v_i, v_j), S_{ij}^k, la lunghezza di un cammino minimo da v_i a v_j i cui vertici intermedi sono estratti dall'insieme $\{v_1, v_2, ..., v_k\}$ (pur non utilizzando necessariamente ciascuno di tali vertici);
3. L'algoritmo termina quando viene calcolata, per tutte le possibili coppie di vertici (v_i, v_j), S_{ij}^n, la lunghezza del cammino minimo da v_i a v_j i cui vertici intermedi sono estratti dall'intero insieme V. É poi possibile, sulla base di questa e di altre informazioni, ricostruire un cammino minimo.

Il caso base dell'equazione di ricorrenza si ha rispetto a P_{ij}^0, il cammino minimo da v_i a v_j che non ha alcun vertice intermedio, l'arco che connette direttamente v_i e v_j. Se i due vertici sono coincidenti, ovvero se $v_i = v_j$, l'unico cammino possibile è il cammino degenere che va da v_i a sé stesso, di peso nullo. Tale cammino non solo effettivamente non possiede alcun vertice intermedio, rispettando la definizione, ma essendo l'unico cammino possibile è anche certamente quello di peso minimo. Se i due vertici v_i e v_j sono distinti ma non è presente nel grafo un arco fra i due, si ha per convenzione che la loro distanza é infinita.

$$S_{ij}^0 = \begin{cases} \infty & \text{se } v_i \neq v_j \wedge \langle v_i, v_j \rangle \notin E \\ 0 & \text{se } v_i = v_j \\ W(v_i, v_j) & \text{altrimenti} \end{cases}$$

Per quanto riguarda la relazione di ricorrenza, si consideri P_{ij}^k, il cammino minimo da v_i a v_j i cui vertici intermedi sono estratti dall'insieme $\{v_1, v_2, ..., v_k\}$. Possono verificarsi due situazioni: v_k è oppure non è uno dei vertici intermedi del cammino.
Se v_k non è uno dei vertici del cammino, allora questo equivale a dire che i vertici intermedi di P_{ij}^k sono estratti dall'insieme $\{v_1, v_2, ..., v_{k-1}\}$, che è lo stesso insieme da cui sono stati estratti i vertici intermedi di P_{ij}^{k-1}. Questo significa che un cammino minimo che va da i a j i cui vertici intermedi sono stati estratti da $\{v_1, v_2, ..., v_{k-1}\}$ è anche un cammino minimo che va da i a j i cui vertici intermedi sono stati estratti da $\{v_1, v_2, ..., v_k\}$.
Se invece v_k è uno dei vertici intermedi di P_{ij}^k, allora tale cammino può essere certamente diviso in due parti: $v_i \rightsquigarrow v_k$ e $v_k \rightsquigarrow v_j$. Il primo sottocammino è a sua volta un cammino minimo che ha inizio in v_i e fine in v_k, mentre il secondo sottocammino è un cammino minimo che ha inizio in v_k e fine in v_j.
Dato che entrambi i sottocammini provengono da un cammino i cui vertici intermedi sono stati estratti dall'insieme $\{v_1, v_2, ..., v_k\}$, anche questi avranno i loro vertici intermedi estratti da tale insieme. Dato che però in nessuno dei due sottocammini figura v_k come vertice intermedio, essendo sempre agli estremi, allora è possibile affermare con certezza che i vertici intermedi di entrambi i sottocammini sono estratti dall'insieme $\{v_1, v_2, ..., v_{k-1}\}$.

$$S^k(i,j) = \begin{cases} S^{k-1}(i,j) & \text{se } v_k \notin P_{ij}^k \\ S^{k-1}(i,k) + S^{k-1}(k,j) & \text{se } v_k \in P_{ij}^k \end{cases}$$

Naturalmente non è possibile sapere a priori se v_k faccia o non faccia parte della k-esima soluzione ottimale. La scelta migliore fa l'includere o il non includere v_k nella soluzione dipende da quale rende ottimale, ovvero minimo, il peso del cammino risultante.

$$S^k(i,j) = \min\{S^{k-1}(i,j), S^{k-1}(i,k) + S^{k-1}(k,j)\}$$

L'algoritmo bottom-up viene costruito a partire dall'equazione di ricorrenza sfruttando tante matrici D quanti sono i vertici del grafo. All'interno di $D^{(k)}[i,j]$, la cella $[i,j]$ della k-esima matrice, contiene la lunghezza del cammino minimo che ha inizio in v_i e fine in v_j i cui vertici intermedi sono stati estratti dall'insieme $v_1, v_2, \ldots, v_k$. L'algoritmo riceve la matrice W dei pesi degli archi e restituisce in output la matrice $D^{(n)}$, la soluzione del problema.
Si noti come la matrice $D^{(0)}$ coincida esattamente con W, la matrice passata in input, perché entrambe riportano le lunghezze dei cammini (unici, se esistono) costituiti da un solo arco [1].

```
procedure FW(W)
n ← W.rows
D⁽⁰⁾ ← W

for k ← 1 to n do
    D⁽ᵏ⁾ nuova matrice n x n
        for i ← 1 to n do
            for j ← 1 to n do
                if (D⁽ᵏ⁾[i, j] ≤ D⁽ᵏ ⁻ ¹⁾[i, k] + D⁽ᵏ ⁻ ¹⁾[k, j]) then
                    D⁽ᵏ⁾[i, j] ← D⁽ᵏ ⁻ ¹⁾[i, j]
                else
                    D⁽ᵏ⁾[i, j] ← D⁽ᵏ ⁻ ¹⁾[i, k] + D⁽ᵏ ⁻ ¹⁾[k, j]

return D⁽ⁿ⁾
```

È facile notare come il tempo di esecuzione dell'algoritmo sia $O(|V|^3)$, perché sono presenti tre cicli dove in quello più interno viene eseguita una operazione con costo unitario.
Per la restituzione di una soluzione ottima al problema è necessario calcolare, oltre alla matrice $D^{(n)}$, anche la **matrice dei predecessori** $\Pi^{(n)}$. In ciascuna cella i, j è riportato NULL se v_i coincide con v_j oppure se non esiste un cammino fra v_i e v_j, altrimenti riporta il predecessore di v_j in qualche cammino minimo che ha inizio in v_i.
Tale matrice, così come la matrice $D^{(n)}$, viene calcolata parametrizzando il problema rispetto ai k vertici intermedi. Sia allora $\Pi^{(k)}$ la matrice dei predecessori dove i vertici intermedi dei cammini a cui fa riferimento sono stati estratti dall'insieme $\{v_1, v_2, \ldots, v_k\}$. I valori di ciascuna i, j-esima cella di tale matrice sono calcolati a partire da un'equazione di ricorrenza.
Il caso base dell'equazione di ricorrenza si ha con $\Pi^{(0)}[i,j]$, il predecessore di v_j in $P_{ij}^{(k)}$, ovvero nel cammino minimo da v_i a v_j che non ha alcun vertice intermedio. Se i due vertici sono coincidenti, ovvero se $v_i = v_j$, il vertice v_j non ha un predecessore in $P_{ij}^{(0)}$, perché si sposta da sé stesso in sé stesso. Allo stesso modo, se $P_{ij}^{(0)}$ non esiste, il vertice v_j non ha un predecessore in tale cammino. Se i due vertici sono distinti e $P_{ij}^{(0)}$ esiste, allora deve aversi che $\Pi^{(0)}[i,j] = v_i$, perché $P_{ij}^{(0)}$ è costituito da un solo arco che connette i due vertici direttamente.

$$\Pi^{(0)}[i,j] = \begin{cases} \text{NULL} & \text{se } (v_i \neq v_j \wedge \langle v_i, v_j \rangle \notin E) \vee v_i = v_j \\ v_i & \text{altrimenti} \end{cases}$$

Per quanto riguarda il passo ricorsivo, se $S^k(i,j) = S^{k-1}(i,j)$, ovvero se il cammino minimo da v_i a v_j avente i vertici intermedi estratti dall'insieme $\{v_1, v_2, \ldots, v_{k-1}\}$ ha la stessa lunghezza di quello avente i vertici intermedi estratti dall'insieme $\{v_1, v_2, \ldots, v_k\}$, allora è possibile inferire che il cammino minimo $P_{ij}^{(k)}$ ed il cammino minimo $P_{ij}^{(k-1)}$ abbiano lo stesso penultimo vertice, ovvero che $\Pi^{(k)}[i,j] = \Pi^{(k-1)}[i,j]$. Questo perché, se i due cammini hanno la stessa lunghezza, allora significa che esiste una soluzione ottimale (non necessariamente l'unica) comune alla k-esima e alla $k-1$-esima istanza del problema.
Se invece $S^k(i,j) = S^{k-1}(i,k) + S^{k-1}(k,j)$, sebbene non sia noto quale sarà il predecessore di v_j nel cammino minimo, certamente tale cammino avrá v_k tra i suoi vertici intermedi. Per questo motivo, dovrá valere $\Pi^{(k)}[i,j] = \Pi^{(k-1)}[k,j]$, ovvero il predecessore del vertice

1. Si noti inoltre come i valori della i-esima matrice dipendano esclusivamente da quelli della $i-1$-esima: questo significa che sarebbe possibile ottimizzare ulteriormente il costo spaziale dell'algoritmo tenendo traccia, in ciascuna i-esima iterazione, solamente dei valori della $i-1$-esima matrice e non di tutte le precedenti.

v_j nel cammino minimo avente i vertici intermedi estratti dall'insieme $\{v_1, v_2, ..., v_{k-1}\}$ che va da v_k a v_j sará lo stesso (quale che sia) del predecessore del vertice v_j nel cammino minimo avente i vertici intermedi estratti dall'insieme $\{v_1, v_2, ..., v_k\}$ che va da v_i a v_j.

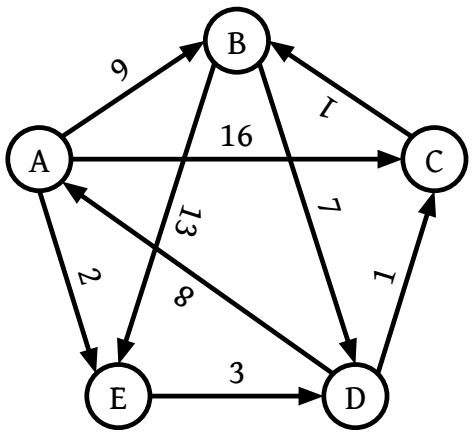
$$\Pi^{(k)}[i,j] = \begin{cases} \Pi^{(k-1)}[i,j] & \text{se } S^k(i,j) = S^{k-1}(i,j) \\ \Pi^{(k-1)}[k,j] & \text{se } S^k(i,j) \neq S^{k-1}(i,j) \end{cases}$$

Il calcolo delle matrici dei predecessori può venire fatto direttamente durante la costruzione delle matrici D. Una volta che è stata calcolata $\Pi^{(n)}$ è possibile ottenere una soluzione ottima. Alla vecchia procedura opportunamente modificata ne viene aggiunta un'altra, `PRINT-FW` . A questa viene passata in input la matrice $\Pi^{(n)}$ e restituisce in output il cammino minimo per ciascuna coppia di vertici a cui tale matrice fa riferimento. La subroutine `PRINT-PATH` al suo interno riceve in input $\Pi^{(n)}$ ed una coppia di indici i e j e restituisce il cammino minimo da v_i a v_j.

```
procedure FW(W)
    n ← W.rows
    D⁽⁰⁾ ← W
    for i ← 1 to n do
        for j ← 1 to n do
            if (D⁽⁰⁾[i, j] ≠ ∞ and i ≠ j) then
                Π⁽⁰⁾ [i, j] ← i
            else
                Π⁽⁰⁾ [i, j] ← NULL

    for k ← 1 to n do
        D⁽ᵏ⁾ nuova matrice n x n
        Π⁽ᵏ⁾ nuova matrice n x n
        for i ← 1 to n do
            for j ← 1 to n do
                if (D⁽ᵏ - ¹⁾[i, j] ≤ D⁽ᵏ - ¹⁾[i, k] + D⁽ᵏ - ¹⁾[k, j]) then
                    D⁽ᵏ⁾[i, j] ← D⁽ᵏ - ¹⁾[i, j]
                    Πᵏ[i, j] ← Π⁽ᵏ - ¹⁾[i, j]
                else
                    D⁽ᵏ⁾[i, j] ← D⁽ᵏ - ¹⁾[i, k] + D⁽ᵏ - ¹⁾[k, j]
                    Π⁽ᵏ⁾[i, j] ← Π⁽ᵏ - ¹⁾[k, j]

    return D⁽ⁿ⁾, Π⁽ⁿ⁾
```

```
procedure PRINT-FW(Π)
    for i ← 1 to Π.rows do
        for j ← 1 to Π.columns do
            PRINT-PATH(Π, i, j)

procedure PRINT-PATH(Π, i, j)
    if (i = j) then
        print i
    else (if Π[i, j] = NULL) then
        print NULL
    else
        PRINT-PATH(Π, i, Π[i, j])
        print j
```

Il tempo di esecuzione della procedura di stampa è comunque $O(|V|^3)$, perché i due cicli innestati eseguono una procedura tail-ricorsiva che ha tempo di esecuzione lineare nel numero di vertici del grafo.



	A	B	C	D	E
A	0	7	6	5	2
B	15	0	8	7	13
C	16	1	0	8	14
D	8	2	1	0	10
E	11	5	4	3	0

	A	B	C	D	E
A	NULL	C	D	E	A
B	D	NULL	D	B	B
C	D	C	NULL	B	B
D	D	C	D	NULL	A
E	D	C	D	E	NULL

La procedura `PRINT-FW` è molto simile alle procedure usate per ottenere i cammini minimi negli alberi indotti dalle visite ai grafi. Infatti, ciascuna riga i della matrice Π definisce un **sottografo dei predecessori** $G_i = (V_{i\pi}, E_{i\pi})$, ovvero un albero di cammini minimi avente v_i come radice. $V_{i\pi}$ è definito come l'insieme dei vertici di G con predecessori diversi da NULL a cui viene aggiunto i. L'insieme degli archi orientati $E_{i\pi}$ è l'insieme degli archi indotto dai valori π per i vertici in $V_{i\pi}$:

$$V_{i\pi} = \{j \in V \mid \pi_{ij} \neq \text{NULL}\} \cup \{i\} \qquad E_{i\pi} = \{(\pi_{ij}, j) \in E \mid v \in (V_{i\pi} \,/\, \{i\})\}$$

L'algoritmo Floyd-Warshall può essere utilizzato anche per calcolare la **chiusura transitiva** di un grafo $G = (V, E)$, definita come il grafo $G^* = (V, E^*)$ dove

$$E^* = \{(i, j) : \text{esiste un cammino dal vertice } i \text{ al vertice } j \text{ in } G\}$$

Il metodo consiste nell'assegnare un peso 1 ad ogni arco di E e nell'eseguire normalmente l'algoritmo di Floyd-Warshall. Sia $n = |V|$: se esiste un cammino dal vertice i al vertice j, si avrà $D^n[i, j] < n$, altrimenti $D^n[i, j] = \infty$.
Un metodo simile che permette di raffinare ulteriormente il costo in termini di tempo e spazio prevede di sostituire le operazioni aritmetiche usate nell'algoritmo di Floyd-Warshall con gli operatori logici AND e OR. Dato un grafo $G = (V, E)$ con $|V| = n$, sia T^k con $k \leq n$ una matrice la cui cella $[i, j]$ contiene il valore 1 se esiste un cammino da v_i a v_j in G i cui vertici intermedi sono tutti estratti dal sottoinsieme $\{v_1, v_2, ..., v_k\}$ di V e 0 altrimenti. Le celle di T^k sono definite a partire dalla seguente equazione di ricorrenza:

$$T^0[i, j] = \begin{cases} 0 & \text{se } i \neq j \wedge (i, j) \notin E \\ 1 & \text{se } i = j \vee (i, j) \in E \end{cases} \qquad T^k[i, j] = T^{k-1}[i, j] \vee \left(T^{k-1}[i, k] \wedge T^{k-1}[k, j]\right)$$

Come nell'algoritmo di Floyd-Warshall, vengono calcolate le matrici T^k per valori crescenti di k.

```
procedure FW(V, E)
    n ← |V|
    T(0) nuova matrice n x n
    for i ← 1 to n do
        for j ← 1 to n do
            if (i = j or (i, j) ∈ E) then
                T(0) ← 1
            else
                T(0) ← 0

    for k ← 1 to n do
     T(k) nuova matrice n x n
     for i ← 1 to n do
         for j ← 1 to n do
             T(k)[i, j] ← T(k - 1)[i, j] or (T(k - 1)[i, k] and T(k - 1)[k, j])

 return T(n)
```

Sebbene il tempo di esecuzione e lo spazio occupato dall'algoritmo siano teoricamente rimasti invariati, i calcolatori sono comunque in grado di eseguire tale algoritmo più velocemente perché le operazioni logiche vengono tradotte in istruzioni assembly molto efficienti.

Capitolo 6
Minimum Spanning Tree

6.1 Problema del Minimum Spanning Tree

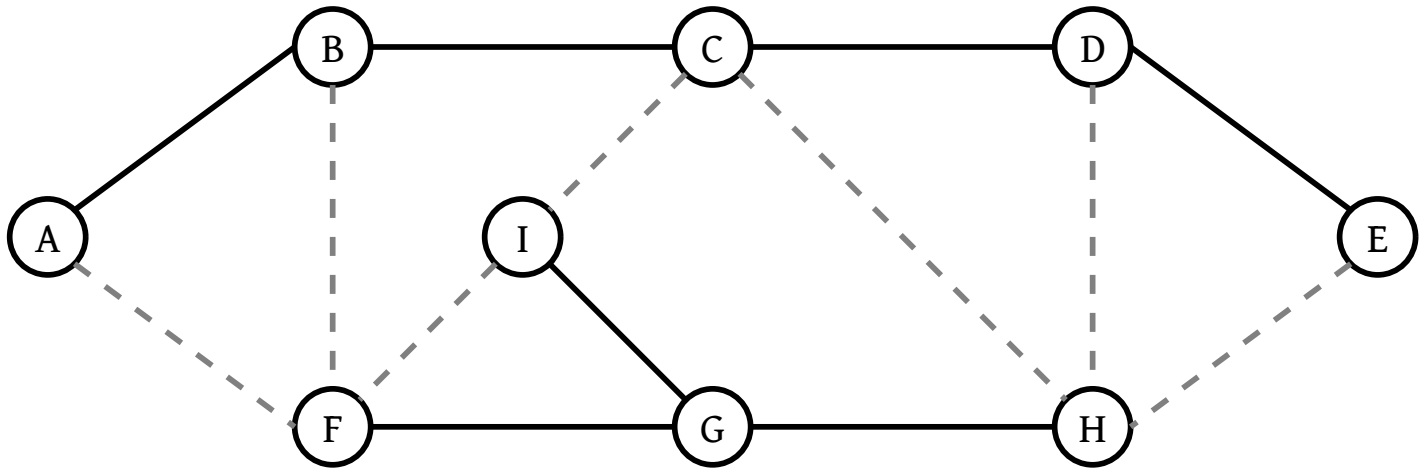
Dato un grafo $G = (V, E)$ non orientato e connesso, si dice **matroide grafico** la coppia di insiemi $M_G = (S, F)$, dove S coincide con l'insieme degli archi E e F è costituito da tutti i sottoinsiemi di S che non presentano cicli, ovvero da tutti ed i soli sottoinsiemi di S che definiscono dei sottografi che formano una **foresta**.

Un matroide grafico è un matroide.

Dimostrazione. Dato un grafo non orientato e connesso $G = (V, E)$, sia $M_G = (S, F)$ il matroide grafico a questo associato. Si dimostri innanzitutto che M_G è un sistema di indipendenza: preso un qualsiasi $A \in F$, sia B un suo sottoinsieme. Tale sottoinsieme viene costruito eliminando da A uno o più archi; essendo però A privo di cicli per ipotesi, anche B dovrà esserlo, e quindi anche B è un elemento di F. Questo significa che qualsiasi sottoinsieme di un elemento di F fa a sua volta parte di F, pertanto M_G è un sistema di indipendenza.
Siano poi $A, B \in F$ due sottoinsiemi tali per cui $|B| = |A| + 1$, i quali definiscono rispettivamente i sottografi $G_A = (V, A)$ e $G_B = (V, B)$. Per definizione tali sottografi sono delle foreste, costituite rispettivamente da $|V|-|A|$ e da $|V|-|B|$ alberi. La foresta G_B contiene esattamente un albero in meno della foresta G_A, infatti:

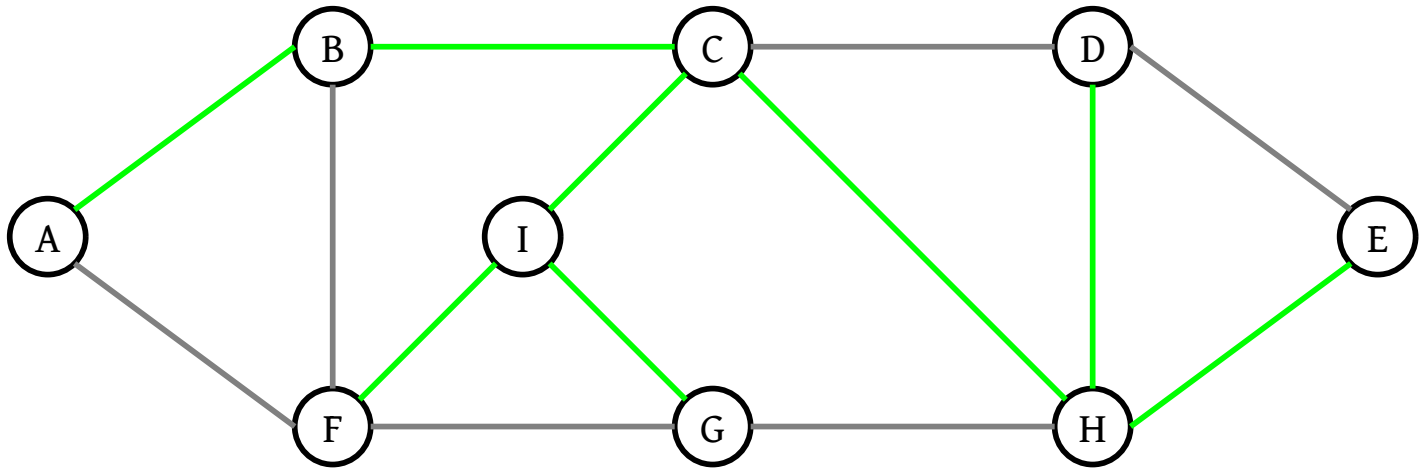
$$|V|-|A|-(|V|-|B|) = |V|-|A|-|V| + |B| = -|A| + |A| + 1 = 1$$

Questo implica che, essendo G_A e G_B costituite dagli stessi vertici, esiste in G_B un arco (u, v) che connette due vertici u e v che in G_B si trovano in un solo albero ma che in G_A si trovano in due alberi diversi. Allora tale arco può essere aggiunto a G_A senza creare un ciclo, ottenendo quindi un nuovo sottografo i cui vertici formano ancora un insieme che appartiene ad F. Questa è però precisamente la definizione di proprietà di scambio, e quindi è possibile affermare che M_G sia un matroide.



Si consideri il grafo $G = (V, E)$ sopra presentato. A partire dall'insieme F, che contiene tutti i sottoinsiemi di E che non hanno cicli. Da F è estratto il sottoinsieme di archi $A = \{(a, b), (b, c), (c, d), (d, e), (i, g), (f, g), (g, h)\}$. Il sottografo $G_A = (V, A)$ definisce una foresta di $|V|-|A| = 2$ alberi, il primo formato dagli archi $\{(a, b), (b, c), (c, d), (d, e)\}$ ed il secondo dagli archi $\{(i, g), (f, g), (g, h)\}$

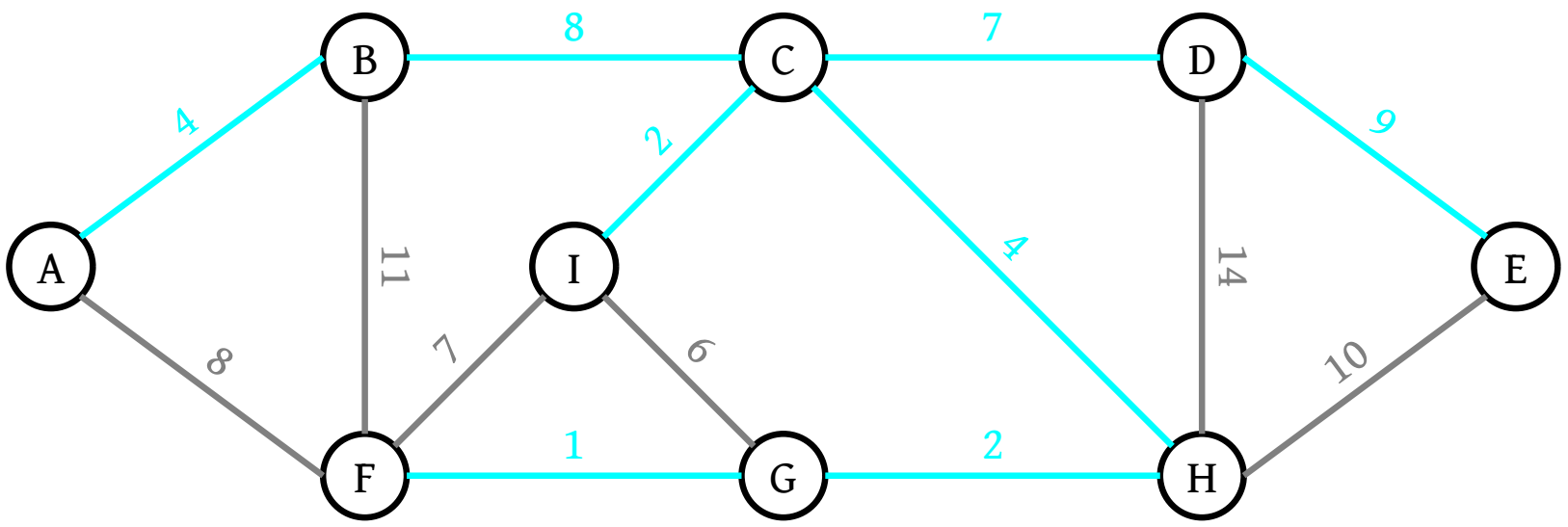
Dato il matroide grafico $M_G = (S, F)$, l'insieme $A \in F$ è massimale se non è possibile aggiungere un arco ad A (che non faccia già parte di A) senza formare un ciclo. Questo equivale a dire che il sottografo G_A indotto da A è una foresta formata da un solo albero, avente $|V|-1$ archi che connette tutti i vertici del grafo. Un albero con queste caratteristiche viene chiamato **albero di copertura**, o **Spanning Tree** (**ST**).



L'insieme di archi sopra evidenziato è uno spanning tree per il grafo dell'esempio precedente.

Sia dato un grafo connesso non orientato pesato $G = (V, E)$, con W funzione che associa un intero positivo a ciascuna coppia di archi di G. Il **problema del minimum spanning tree** (**MST problem**) richiede di trovare l'albero T, sottoinsieme di E, tale per cui $G_T = (V, T)$ è lo spanning tree per il grafo G che ha peso complessivo minimo. Il peso di uno spanning tree è dato dalla somma dei pesi di tutti gli archi di cui è composto.

$$\forall v \in V,\ \exists (u, v) \in T \qquad\qquad W(T) = \sum_{(u,v)\in T} W(u, v) \text{ minimo}$$



Rispetto al grafo sopra riportato, quello evidenziato in blu è il MST.

È evidente come il problema possa essere risolto mediante tecnica greedy, perché un insieme massimale di un matroide grafico è uno spanning tree. Si ha quindi che il minimum spanning tree è dato dall'insieme massimale del matroide grafico con peso minimo. In termini molto generici, l'algoritmo può essere descritto come segue:

1. Viene inizializzato un insieme A vuoto;
2. Viene scelto l'arco (u, v) di modo che $A \cup (u, v)$ sia un sottoinsieme dell'insieme T degli archi del MST. Un arco che aggiunto ad A restituisce un insieme di archi ancora sottoinsieme dell'MST è detto **arco sicuro** per A;
3. Se $A = T$, ovvero se $G_A = (V, T)$ è il MST, l'algoritmo termina, altrimenti si torna al punto precedente.

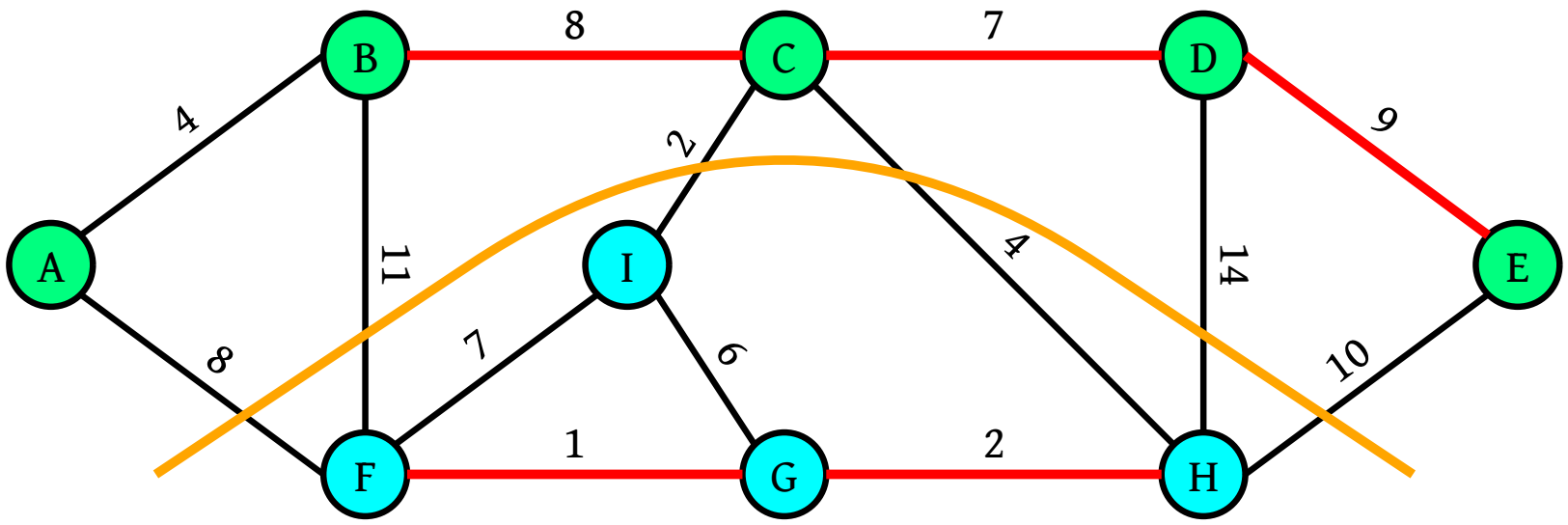
```
procedure GENERIC-MST(G, W)
1    A ← ∅

2    while G_A ≠ MST do
3        trova arco (u, v) sicuro per A
4        A ← A ∪ {(u, v)}
```

```
5    return A
```

L'algoritmo è effettivamente un algoritmo greedy perché ad ogni iterazione viene operata una scelta localmente ottima assumendo che lo sia anche complessivamente.
Si noti come la condizione di terminazione dell'algoritmo, ovvero valutare se il sottografo $G_A = (V, A)$ è il MST, sia piuttosto vaga, dato che non è direttamente computabile. A tal proposito, si presti attenzione all'insieme A: in una qualsiasi iterazione, tale insieme definisce una foresta $G_A = (V, A)$ per un grafo $G = (V, E)$ avente $|V|-|A|$ alberi.
Dato che per definizione lo Spanning Tree si ha quando la foresta è costituita da un solo albero, la condizione di terminazione dell'algoritmo può essere formulata come un controllo sulla differenza fra il numero di vertici del grafo ed il numero di alberi della foresta. In particolare, tale condizione è verificata quando la differenza fra i due è pari ad 1, perché questo significa che è rimasto un solo albero.

```
procedure GENERIC-MST(G, W)
1    A ← ∅

2    while |V| - |A| > 1 do
3        trova arco (u, v) sicuro per A
4        A ← A ∪ {(u, v)}
5    return A
```

Per rendere l'algoritmo effettivamente utilizzabile è infine necessario capire come compiere la scelta greedy, ovvero definire un metodo che discrimini quando un arco è un arco sicuro.
Dato un grafo $G = (V, E)$, viene detto **taglio** una qualsiasi partizione di V in due insiemi, V' e $V-V'$. Un arco $(u, v) \in E$ si dice che **attraversa il taglio** se il vertice u appartiene a V' e v appartiene a $V-V'$. L'arco che attraversa il taglio avente peso minimo è detto **arco leggero**. Dato un sottoinsieme $A \subseteq E$, si dice che un taglio **rispetta** l'insieme A se nessun arco di A attraversa tale taglio.



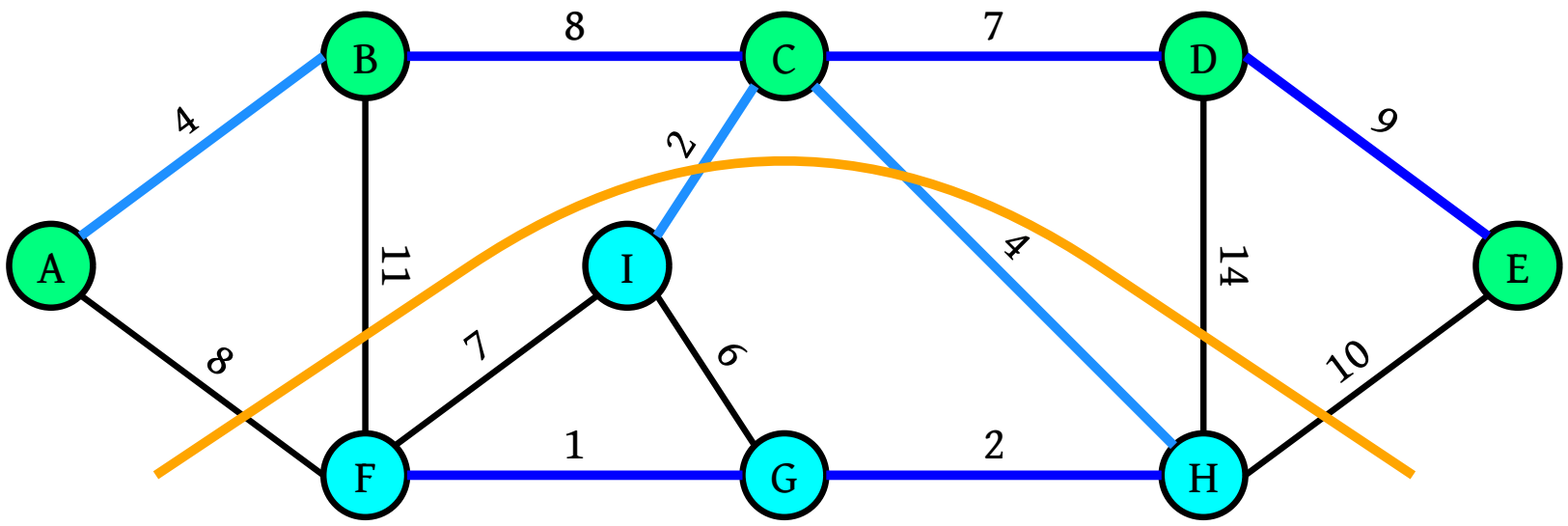
Si consideri il grafo sopra presentato. Il taglio evidenziato partiziona i vertici del grafo in due sottoinsiemi, $V = \{A, B, C, D, E\}$ e $V-V' = \{F, G, H, I\}$. Tale taglio rispetta il sottoinsieme di archi $A = \{(B, C), (C, D), (D, E), (F, G), (G, H)\}$. Gli archi che attraversano il taglio sono $(A, F), (B, F), (C, I), (C, H), (D, H), (E, H)$; fra questi, l'arco (C, I) è l'arco leggero per il taglio, essendo quello fra questi di peso minimo.

Teorema dell'arco sicuro. Siano dati un grafo connesso non orientato e pesato $G = (V, E, W)$, un sottoinsieme A dell'insieme T di archi del MST e un qualsiasi taglio che rispetti A. L'arco leggero (u, v) del taglio è certamente un arco sicuro per A, ovvero $A \cup \{(u, v)\} \subseteq T$.

Dimostrazione. Se (u, v) è l'unico arco che attraversa il taglio il teorema è dimostrato, dato che è l'unico arco in grado di connettere le due componenti distinte nel MST.
Si consideri invece il caso in cui esista almeno un altro arco che attraversa il taglio. Si supponga per assurdo che l'arco leggero (u, v) del taglio non sia un arco sicuro per A, ovvero che aggiungendo (u, v) all'insieme A non si ottiene un sottoinsieme di T. Deve allora esistere un altro arco che attraversa il taglio e che appartiene a T, sia questo (x, y).
L'arco (x, y) non può appartenere ad A, perché per definizione il taglio rispetta A. Se da T viene rimosso l'arco (x, y) si ottengono due alberi separate; se viene poi aggiunto (u, v) i due alberi si ricongiungono per ottenere l'albero di connessione, non necessariamente minimo, $T' = T - \{(x, y)\} \cup \{(u, v)\}$. Poiché per ipotesi (u, v) è l'arco leggero per il taglio, il peso di (u, v) deve essere inferiore al peso di (x, y). Allora:

$$W(T') = W(T) - W(x, y) + W(u, v) \leq W(T)$$

Essendo però T un albero di connessione minimo, deve aversi $W(T) \leq W(T')$; si ha quindi $W(T') \leq W(T) \wedge W(T) \leq W(T')$, ma questo significa $W(T) = W(T')$, e quindi anche T' è un albero di connessione minimo.
Dato che per ipotesi $A \subseteq T$ e $(x, y) \notin A$, allora $A \subseteq T'$. Questo significa che $A \cup \{(u, v)\} \subseteq T'$. Ma questa è la definizione di arco sicuro, ed avendo appena mostrato che T' è un MST deve aversi che (u, v) è un arco sicuro per A.



Il sottoinsieme di archi $A = \{(B, C), (C, D), (D, E), (F, G), (G, H)\}$ è parte degli archi del MST. Per questo motivo, il teorema precedente stabilisce che l'arco leggero del taglio che rispetta A è un arco sicuro per A, e infatti l'arco (C, I) è uno degli archi del MST.

Corollario del teorema dell'arco sicuro. Siano dati un grafo connesso non orientato e pesato $G = (V, E)$ ed un sottoinsieme A dell'insieme T di archi del MST che definisce una foresta $G_A = (V, A)$. Sia $C = (V_C, A_C)$, con $V_C \subseteq V$ e $A_C \subseteq A$ uno qualsiasi degli alberi di G_A. Il taglio che divide i vertici di G_A nei due sottoinsiemi V_C e $V - V_C$ rispetta l'insieme A. Inoltre, un arco leggero per tale taglio è anche arco sicuro per l'insieme A.

Dimostrazione. Il taglio $(V_C, V - V_C)$ rispetta A e (u, v) è un arco leggero per questo taglio, quindi (u, v) è sicuro per A.

Sia A il sottoinsieme degli archi del MST di una generica iterazione dell'algoritmo, e sia G_A la foresta indotta dall'insieme A. Il corollario appena mostrato stabilisce che per trovare un arco sicuro per A, e che può essere aggiunto a quest'ultimo, è sufficiente scegliere uno qualsiasi degli alberi di G_A e trovare l'arco di peso minimo che collega un vertice che appartiene a tale albero con un vertice che non vi appartiene.
Ci si chiede allora come determinare in una qualsiasi iterazione (ed in maniera efficiente) quale sia l'arco di peso minimo da aggiungere all'insieme di archi del MST in costruzione. Per farlo vi sono diversi approcci, e ciascuno di questi è associato ad un algoritmo diverso.

6.2 Ricavare il Minimum Spanning Tree: algoritmo di Kruskal

L'**Algoritmo di Kruskal** è un algoritmo greedy che permette di ricavare il MST di un grafo a partire dall'algoritmo greedy standard.
Innanzitutto, sia $G = (V, E, W)$ un grafo connesso, non orientato e pesato; a questo è possibile associare un matroide grafico $M_G = (S, F)$. Come già detto, un qualsiasi insieme massimale di archi M_G è uno spanning tree, pertanto il MST è quell'insieme massimale avente peso minimo.
L'algoritmo greedy standard risolve problemi di ottimizzazione di massimo, ma la ricerca di un MST (come dice il nome) è un problema di ottimizzazione di minimo. A tale scopo occorre, come già visto in precedenza, sostituire alla funzione peso W la funzione peso W', dove ciascun valore i-esimo è dato dalla differenza fra il massimo valore in W e l'i-esimo valore di W.
Rifacendosi all'algoritmo greedy standard, l'insieme che sarà passato in input all'algoritmo (e che viene poi ordinato) sarà l'insieme degli archi del grafo. Si ricordi che tale insieme va ordinato, secondo ordine decrescente, rispetto a W', e non rispetto a W. In realtà questo equivale ad ordinare gli elementi rispetto a W ma in ordine inverso (in ordine crescente), pertanto è possibile applicare tale semplificazione.

```
procedure KRUSKAL-MST(E)
1    A ← ∅
2    E ← <e₁, …, eₙ> ordinati per peso crescente

3    for i ← 1 to n do
4        if eᵢ è arco sicuro then
5            A ← A ∪ {eᵢ}

6    return A
```

Per sapere se l'i-esimo arco è un arco sicuro occorre applicare il corollario al teorema dell'arco sicuro: un arco è sicuro se connette due alberi distinti di G_A.

```
procedure KRUSKAL-MST(V, E, W)
1    A ← ∅
2    E ← <e₁, …, eₙ> ordinati per peso crescente

3    for i ← 1 to n do
4        (u, v) ← eᵢ
5        if u e v ∉ stesso albero in G_A then
6            A ← A ∪ {eᵢ}

7    return A
```

L'algoritmo funziona perché, ordinando gli elementi in ordine crescente, è garantito che fra tutti i possibili archi che connettono due alberi distinti verrà sempre scelto quello di peso minimo.
L'ultimo passo consiste nel determinare che struttura dati usare per salvare le informazioni relative agli alberi. Questo può essere fatto in maniera efficiente mediante una union-find e le relative primitive: per ogni vertice viene creato un insieme che contiene solo quel vertice, ed i confronti vengono fatti su questi insiemi. In particolare, se vengono trovati due insiemi disgiunti significa che non vi è alcun vertice in comune ai due insiemi, e che quindi è stato trovato un arco che unisce due alberi distinti. A questo punto è sufficiente sostituire i due insiemi con la loro unione.

```
procedure KRUSKAL-MST(V, E, W)
 1    A ← ∅
 2    E ← <e₁, …, eₙ> ordinati per peso crescente
 3    foreach v ∈ V do
 4        MAKE_SET(v)

 5    for i ← 1 to n do
 6        (u, v) ← eᵢ
 7        if FIND_SET(u) ≠ FIND_SET(v) then
 8            A ← A ∪ {eᵢ}
 9            UNION(u, v)
```

```
10    return A
```

Per quanto riguarda il tempo di esecuzione dell'algoritmo, è possibile distinguere tre sezioni dello stesso: l'ordinamento degli archi rispetto al peso (riga 2), la costruzione dei set (righe 3 e 4) ed il loop principale (righe da 5 a 9):

1. L'ordinamento avviene con il consueto tempo logaritmico, $O(n\log(n))$. Dato che, nello specifico, l'ordinamento è fatto sugli archi di un grafo $G = (V, E, W)$, e che il numero di tali archi è dato dalla cardinalità di E, si ha che l'ordinamento degli archi ha tempo di esecuzione $O(|E|\log(|E|))$.
2. È possibile assumere che la costruzione dei set avvenga in tempo lineare per le proprietà della struttura dati union-find. Dato che viene costruito un set per ogni vertice, il tempo di esecuzione di tale costruzione è $O(|V|)$.
3. Il tempo di esecuzione del loop principale dipende interamente dal tempo di esecuzione delle primitive della union-find. Il confronto fra due set e l'unione di due set avviene in tempo α, con α costante fissata. Dato che il confronto avviene tante volte quanti sono gli archi del grafo, e dato che le unioni avvengono al massimo tante volte quanti sono i confronti, il tempo di esecuzione del loop principale è $O(|E|\alpha)$.

Il tempo di esecuzione complessivo è dato dalla somma dei tre tempi di esecuzione parziali, ovvero $O(|E|\log(|E|)) + O(|V|) + O(|E|\alpha)$. Dato che si sta considerando un grafo connesso, il numero di archi è necessariamente superiore al numero di vertici, pertanto è possibile effettuare una maggiorazione e sostituire $|V|$ con $|E|$. Si ha quindi $O(|E|\log(|E|)) + O(|E|) + O(|E|\alpha)$, che asintoticamente equivale a $O(|E|\log(|E|)) + O(|E|\alpha)$. A sua volta, la costante α è certamente inferiore a $\log(|V|)$, che è a sua volta certamente inferiore a $\log(|E|)$. È quindi possibile maggiorare il tempo di esecuzione come $O(|E|\log(|E|)) + O(|E|\log(|E|))$, ovvero $O(2|E|\log(|E|))$, ma che asintoticamente equivale semplicemente a $O(|E|\log(|E|))$.

6.3 Ricavare il Minimum Spanning Tree: algoritmo di Prim

L'**Algoritmo di Prim** è un algoritmo greedy che permette di ricavare il MST di un grafo che non si rifà all'algoritmo greedy standard. In termini molto generali, l'idea dell'algoritmo è la seguente:

1. Dato un grafo $G = (V, E, W)$, viene scelto arbitrariamente un vertice r, e si considera l'albero (degenere) C formato da questo vertice isolato;
2. Viene trovato l'arco di peso minimo che connette un vertice in C con un vertice v non in C e si aggiunge v a C;
3. L'algoritmo viene ripetuto fino ad esaurire tutti i vertici.

L'algoritmo funziona perché ad ogni iterazione i vertici del grafo vengono divisi in due sottoinsiemi: i vertici dell'albero in costruzione (l'insieme V_C) e tutti quelli rimanenti (l'insieme $V - V_C$). Per definizione, tale taglio rispetta l'insieme degli archi di C, pertanto l'arco di peso minimo che connette un vertice in V_C con un vertice in $V - V_C$ è certamente un arco sicuro per l'arco in costruzione.
Per determinare quale sia l'arco corretto da considerare, l'algoritmo associa a ciascun vertice v del grafo $G = (V, E, W)$ due campi: un campo chiave (v.key) ed un campo predecessore ($v.\pi$. Il campo chiave contiene il peso dell'arco che, rispetto all'iterazione corrente, è il minimo trovato, mentre il campo predecessore contiene il vertice con il quale il vertice corrente ha l'arco con peso più piccolo. Dato un grafo $G = (V, E, W)$, si ha allora:

1. I campi chiave di tutti i vertici vengono inizializzati a ∞, mentre i campi predecessore vengono inizializzati a NULL;
2. Viene scelto un vertice arbitrario, al quale viene assegnato come campo chiave il valore 0. Dopodiché, tutti i vertici vengono inseriti in una coda di min-priority, dove il valore della chiave definisce l'ordine (la priorità) con cui i vertici vengono estratti. In questo modo, il vertice estratto è sempre quello con valore del campo chiave più piccolo;
3. Viene operata un'estrazione ed il vertice v estratto viene confrontato con i vertici a questo adiacenti che ancora si trovano nella coda. Per ciascuno di questi, se il relativo campo chiave contiene un valore maggiore di v.key allora come campo predecessore viene posto v e come campo chiave viene posto il peso dell'arco che connette v a tale vertice;
4. Se la coda di priorità non è vuota, l'algoritmo riprende dal punto 2, altrimenti si procede oltre, perché è possibile assumere che a tutti i vertici tranne quello di partenza è stato univocamente assegnato un predecessore;
5. L'insieme degli archi del MST viene costruito riportando, per ciascun vertice, l'arco (quale che sia) che ha con il rispettivo predecessore.

```
procedure PRIM-MST(V, E, W, r)
```

```
 1    foreach v ∈ V do
 2        v.key ← ∞
 3        v.π ← NULL
 4    r.key ← 0

 5    Aggiungi tutti i vertici di V alla coda Q

 6    while Q ≠ ∅ do
 7        u ← POP(Q)
 8        foreach v ∈ ADJ(u) do
 9            if v ∈ Q and W(u, v) < v.key then
10                v.key ← W(u, v)
11                v.π ← u

12    A ← ∅
13    foreach v ∈ V do
14        A ← A ∪ {(v, v.π)}

15    return A
```

Per quanto riguarda il tempo di esecuzione dell'algoritmo, è possibile distinguere quattro sezioni dello stesso: la costruzione della coda di priorità (righe da 1 a 5), il loop principale (righe da 6 a 11) e la ricostruzione della soluzione (righe da 12 a 14)

1. Il campo chiave ed il campo predecessore vengono aggiunti a ciascun vertice in tempo lineare. Allo stesso modo, il tempo di esecuzione dell'aggiunta dei vertici alla coda è proporzionale al numero di vertici, perché tutti i vertici inizialmente hanno la stessa priorità e quindi possono essere aggiunti in ordine casuale. Il tempo di esecuzione della costruzione della coda è allora $O(|V|)$;
2. Il loop principale viene eseguito una volta per ciascun vertice del grafo. L'estrazione del vertice in cima alla coda è una operazione che richiede tempo di esecuzione logaritmico, perché per ricavare il vertice con campo chiave più piccolo occorre effettuare una ricerca binaria. Allo stesso modo, l'analisi di ciascun vertice adiacente al vertice corrente viene compiuta in tempo logaritmico. Il tempo di esecuzione del loop principale è allora $O(|V|\log(|V|))$;
3. La ricostruzione della soluzione viene eseguita in tempo proporzionale al numero dei vertici, perché l'aggiunta di un singolo vertice avviene in tempo costante e ciascun vertice viene analizzato esattamente una sola volta. Si ha allora che il tempo di esecuzione della ricostruzione della soluzione è $O(|V|)$.

Il tempo di esecuzione complessivo è dato dalla somma dei tre tempi di esecuzione parziali, ovvero $O(|V|) + O(|V|\log(|V|) + O(|V|)$, che asintoticamente equivale a $O(|V|) + O(|V|\log(|V|)$. Essendo il tempo logaritmico più influente sul tempo complessivo rispetto a quello lineare, si ha che l'equazione equivale asintoticamente a $O(|V|\log(|V|))$. Dato che si sta considerando un grafo connesso, il numero di archi è necessariamente superiore al numero di vertici, pertanto è possibile effettuare una maggiorazione e sostituire $|V|$ con $|E|$. Si ha allora che il tempo di esecuzione dell'algoritmo di Prim è $O(|E|\log(|E|))$.

Capitolo 7
Visite di grafi

7.1 Visita in ampiezza

Dato un grafo $G = (V, E)$ ed un vertice distinto $s \in V$ detto **sorgente**, una **visita in ampiezza**, o **Breadth-First Search** (**BFS**) ispeziona tutti gli archi di G per individuare tutti quei vertici di G che è possibile raggiungere seguendo un cammino che ha inizio in s. Concettualmente, una BFS opera come segue:

1. Viene visitata la sorgente s, ovvero il vertice che ha distanza 0 dalla sorgente;
2. Vengono visitati uno dopo l'altro tutti i vertici adiacenti ad s, ovvero quei vertici che distano 1 dalla sorgente;
3. Vengono visitati uno dopo l'altro tutti i vertici adiacenti ai vertici adiacenti ad s, ovvero quei vertici che distano 2 dalla sorgente, ecc...

L'algoritmo per la visita in ampiezza di un grafo necessita di assegnare degli attributi ai suoi vertici. Per ciascun vertice v, gli attributi sono i seguenti:

1. Un colore, indicato con $v.color$, che rappresenta lo stato del vertice all'iterazione corrente. I possibili colori sono tre:

   - Bianco: il vertice non è stato ancora visitato;
   - Grigio: il vertice è stato **scoperto**, ovvero è stato visitato ma alcuni vertici a questo adiacenti non lo sono ancora stati;
   - Nero: il vertice è stato visitato e tutti i vertici a questo adiacenti sono stati visitati;

2. La distanza dalla sorgente, indicata con $v.d$;
3. Il predecessore di tale vertice nel cammino dalla sorgente, indicato con $v.\pi$.

All'inizio della visita, a ciascun vertice viene assegnato il colore bianco, un valore d pari a ∞ (ad eccezione della sorgente s, che ha $s.d = 0$) ed un valore π pari a NULL. Alla fine della visita, a tutti i vertici raggiungibili dalla sorgente sarà assegnato il colore nero ed i relativi valori di d e di π. Se alla fine della visita sono ancora presenti dei vertici di colore bianco che hanno ∞ come attributo distanza e NULL come attributo predecessore, allora significa che non vi è modo di raggiungerli a partire dalla sorgente.
L'algoritmo per la visita in ampiezza di un grafo è presentato di seguito. Questo, dopo aver inizializzato gli attributi di ciascun vertice (righe 1-6) fa uso di una coda di priorità (FIFO queue), nella quale vengono messi i vertici non appena vengono visitati e nella quale rimangono fino a quando estratti per esplorarne gli adiacenti. Si supponga di avere a disposizione una procedura denominata `ADJACENT` che restituisce una lista contenente tutti i vertici adiacenti al vertice passato come argomento.

```
procedure BFS(V, E, s)
 1    foreach v ∈ V do
 2       v.color ← W
 3       v.d ← ∞
 4       v.π ← NULL

 5   s.color ← G
 6   s.d ← 0

 7   Q ← coda FIFO (inizialmente vuota)
 8   ENQUEUE(Q, s)

 9   while Q ≠ ∅ do
10        v ← HEAD(Q)
11        foreach u ∈ ADJACENT(v) do
12            if (u.color = W) then
13                u.color ← G
14                ENQUEUE(Q, u)
15                u.d ← v.d + 1
16                u.π ← v
17        v.color ← B
18        DEQUEUE(Q)
```

L'algoritmo continua fintanto che vi sono ancora vertici nella coda di priorità. Il primo vertice v fra questi viene estratto dalla coda, ed il loop interno (righe 11-16) analizza ciascun vertice adiacente a v. Ognuno di questi viene esplorato, assegnandogli il colore grigio,

incrementando di uno la sua distanza dalla sorgente e ponendo v come suo predecessore. Dopodiché, viene assegnato a v il colore nero e viene rimosso dalla coda.
L'algoritmo termina quando la coda di priorità è vuota, ovvero quando non esistono più vertici grigi, perché significa che tutti i vertici visitabili sono stati visitati e tutti i vertici rimasti bianchi non sono raggiungibili. È infatti facile verificare che ad un vertice può venire assegnato il colore grigio solamente *durante* la visita, mai prima e mai dopo.
Il costo di inizializzazione dell'algoritmo in termini di tempo di esecuzione è $O(|V|)$, perché viene compiuto un numero costante di operazioni per ciascun vertice. Una singola operazione sulla coda può considerarsi eseguita in un tempo costante, perchè questa non necessita di essere riordinata quando un elemento viene aggiunto o rimosso. Ciascun vertice viene inserito e/o estratto dalla coda non più di una sola volta. Questo significa che il numero massimo di volte che l'algoritmo ispeziona un vertice è $2E$, e che quindi il costo del loop principale dell'algoritmo in termini di tempo di esecuzione è $O(|E|)$. Il tempo di esecuzione complessivo dell'algoritmo viene allora ad essere $O(|V| + |E|)$.
Per un grafo $G = (V, E)$ con sorgente $s \in V$, è detto **sottografo dei predecessori** di G il grafo $G_\pi = (V_\pi, E_\pi)$ dove

$$V_\pi = \{v \in V : v.\pi \neq \text{NULL}\} \cup \{s\} \qquad E_\pi = \{(v.\pi, v) : v \in V_\pi - \{s\}\}$$

Il sottografo dei predecessori G_π è un **albero BF** se V_π è formato dai vertici raggiungibili da s e, per ogni $v \in V_\pi$, c'è un solo cammino semplice da s a v in G_π che è anche un cammino minimo da s a v in G. Un albero BF ha la sorgente come radice ed al suo livello i-esimo si trovano tutti i vertici che distano i dalla sorgente. Si noti come una struttura di questo tipo sia effettivamente un albero, perché è un grafo connesso e $|E_\pi| = |V_\pi| - 1$
Una volta operata una visita in ampiezza e aver assegnato a ciascun vertice i valori per i tre attributi, è possibile costruire l'albero BF associato alla visita. Per farlo, occorre costruire un algoritmo che restituisca un insieme V_π in cui sono presenti tutti i vertici che sono stati raggiunti dalla visita in ampiezza ed un insieme E_π in cui sono presenti tutti gli archi che dalla sorgente portano a tali vertici. Tale algoritmo avrà in input la sorgente s e l'insieme V', ovvero l'insieme dei vertici V dove ciascuno di questi è stato arricchito con i valori dei tre attributi.
I vertici raggiunti dalla visita in ampiezza sono la sorgente e tutti i vertici che hanno un valore diverso da NULL come campo predecessore, pertanto è sufficiente aggiungere a V_π tutti i vertici aventi queste caratteristiche. Per quanto riguarda E_π, questo sarà costituito da tutti gli archi che connettono un vertice ad un suo predecessore, perché ripercorrendo a ritroso tutti i predecessori da ciascun vertice alla sorgente si ottengono tutti i cammini che vanno dalla sorgente a tale vertice.

```
procedure BUILD-BF-TREE(V', s)
1    Eπ ← ∅
2    Vπ ← {s}

3    foreach v ∈ (V' - {s}) do
4        if (v.π ≠ NULL) then
5            Vπ ← Vπ ∪ {v}
6            Eπ ← Eπ ∪ {(v.π, v)}

7    return Eπ, Vπ
```

7.2 Visita in profonditá

Dato un grafo $G = (V, E)$, una **visita in profondità**, o **Depth-First Search** (**DFS**) visita il grafo di vertice adiacente in vertice adiacente fintanto che è possibile, raggiungendo tutti i vertici del grafo. Concettualmente, una DFS opera come segue:

1. Viene visitata la sorgente s;
2. Viene visitato il primo adiacente $a1$ di s, poi viene visitato il primo adiacente $a2$ di $a1$, poi visita il primo adiacente $a3$ di $a2$, ...
3. Quando viene raggiunto un vertice che non ha adiacenti da visitare si risale al predecessore e la visita riparte (se possibile) da un altro adiacente di tale predecessore;
4. Ogni volta che non ci sono adiacenti da visitare si risale al predecessore;
5. Quando la visita risale alla sorgente e s non ha più adiacenti da visitare, si sceglie una nuova sorgente e la visita riparte;
6. La visita termina quando non ci sono più vertici disponibili per essere scelti come nuova sorgente.

L'algoritmo per la visita in profondità di un grafo necessita di assegnare degli attributi ai suoi vertici. Per ciascun vertice v, gli attributi sono i seguenti:

1. Un colore, indicato con $v.color$, che rappresenta lo stato del vertice all'iterazione corrente. I possibili colori sono tre:

   - Bianco: il vertice non è stato ancora scoperto;
   - Grigio: il vertice è stato scoperto;
   - Nero: il vertice e tutti i vertici a questo adiacenti sono stati visitati;

2. Il predecessore di tale vertice nel cammino dalla sorgente, indicato con $v.\pi$;
3. L'istante temporale in cui il vertice è stato scoperto, detto **tempo di scoperta** ed indicato con $v.d$;
4. L'istante temporale in cui il vertice è stato completamente esplorato, detto **tempo di completamento** ed indicato con $v.f$.

All'inizio della visita, a ciascun vertice viene assegnato il colore bianco, un tempo di scoperta pari a 0, un tempo di completamento pari a 0 ed un valore π pari a NULL. Dopo la visita tutti i vertici avranno assegnato il colore nero, perché ogni vertice è sempre eleggibile a sorgente (quindi anche un vertice isolato può comunque venire raggiunto). Inoltre, dopo la visita, il campo π sarà rimasto NULL solamente per quei vertici che sono stati scelti come sorgente, perché per definizione le sorgenti non hanno un predecessore. Infine, un vertice può avere assegnato il colore grigio solamente *durante* la visita, mai prima e mai dopo.
L'algoritmo per la visita in profondità di un grafo è presentato di seguito. Questo, dopo aver inizializzato gli attributi di ciascun vertice (righe 1-5) e aver inizializzato una variabile globale `time` a zero, chiama la subroutine `DFS-VISIT` per operare la visita in profondità vera e propria.

```
procedure DFS(G)
 1    foreach v ∈ V do
 2        v.color ← W
 3        v.π ← NULL
 4        v.d ← 0
 5        v.f ← 0
 6    time ← 0
 7    foreach v ∈ V do
 8        if v.color = W then
 9            DFS-VISIT(G, v)
```

```
procedure DFS-VISIT(G, u)
10    time ← time + 1
11    u.d ← time
12    u.color ← G
14    foreach v ∈ adj(u) do
15        if v.color = W then
16            v.π ← u
17            DFS-VISIT(G, v)
18    time ← time + 1
19    u.f ← time
20    u.color ← B
```

Il valore di `time` viene incrementato di uno ogni volta che viene chiamata la subroutine `DFS-VISIT` per un vertice u (riga 10), e viene assegnato al valore di $u.d$ (riga 11). Tale subroutine inizia una visita in profondità con u come radice; quando questa termina, al vertice u sarà assegnato un tempo di scoperta (l'istante in cui u è diventato un vertice grigio) ed un tempo di completamento (l'istante in cui u è diventato un vertice nero). Naturalmente, deve aversi $u.f < u.d$. La differenza fra $u.f$ e $u.d$ restituisce il lasso di tempo che è stato necessario per esplorare completamente il vertice u.
A questo punto (righe 14-17) vengono esplorati tutti i vertici adiacenti ad u, impostando u come predecessore di ciascuno di questi. Quando non è più possibile procedere con una visita a partire da u, a questo viene assegnato il colore nero ed il relativo tempo di completamento (righe 18-20), dopodiché `DFS` opera una nuova visita in profondità con uno dei vertici ancora bianchi (se ce ne sono) come sorgente. L'algoritmo termina quando a tutti i vertici del grafo è assegnato il colore nero.
Il tempo di esecuzione per l'inizializzazione (righe 1-5) è $O(|V|)$, dato che vengono eseguite delle operazioni con tempo di esecuzione unitario una volta per ogni vertice. La subroutine `DFS-VISIT` viene chiamata esattamente una volta per ciascun vertice del grafo, perché `DFS-VISIT` viene invocata soltanto se un vertice è bianco e la sua prima istruzione prevede di assegnarvi il colore grigio. Durante un'esecuzione di `DFS-VISIT`, il ciclo alle righe da 14 a 17 viene eseguito una volta per ciascun vertice adiacente al vertice in esame. Dato che il numero complessivo di tutti i vertici adiacenti ad un vertice è al massimo pari al numero di archi del grafo, tale ciclo ha un tempo di esecuzione pari a $O(|E|)$. Il tempo di esecuzione complessivo dell'algoritmo è allora $O(|V|) + O(|E|) = O(|V| + |E|)$. Diversamente da ciò che accade in una visita in ampiezza, una visita in profondità induce una foresta di alberi, non un albero solo, perché questa può essere ripetuta da più sorgenti. Il **sottografo dei predecessori** di una visita in profondità è quindi definito come $G_\pi = (V, E_\pi)$, dove l'insieme dei vertici V coincide con l'insieme dei vertici del grafo e:

$$E_\pi = \{(v.\pi, v) : v \in V \wedge v.\pi \neq \text{NULL}\}$$

Il sottografo dei predecessori di una visita in profondità forma una **foresta DF**, composta da uno o più **alberi DF**. Tale sottografo forma effettivamente una foresta perché la struttura degli alberi DF rispecchia la struttura delle chiamate ricorsive di `DFS-VISIT`. Ovvero, dati due vertici u e v, si ha $u = v.\pi$ se e soltanto se `DFS-VISIT(v)` è stata chiamata durante una visita degli adiacenti di u. Inoltre, il vertice v è un discendente del vertice nella foresta DF se e soltanto se v viene scoperto durante il lasso di tempo in cui u è un vertice grigio.
Un'interessante proprietà della visita in profondità è che i tempi di scoperta e di completamento hanno una **struttura di parentesi**. Ovvero, se per ciascun vertice v, si rappresenta il suo tempo di scoperta come una parentesi aperta ed il suo tempo di completamento

come una parentesi chiusa, allora l'insieme di tutti i tempi di tutti i vertici produce una formula ben formata per la grammatica delle parentesi.

Teorema delle parentesi. Dato un grafo $G = (V, E)$ orientato e non pesato, si operi una visita in profondità su tale grafo. Si consideri una coppia di vertici u e v ed i relativi intervalli $(u.d, u.f)$ e $(v.d, v.f)$:

- $(u.d, u.f)$ contiene $(v.d, v.f)$. Allora u e v si trovano in uno stesso albero della foresta DF, ed in particolare u è un predecessore di v;
- $(v.d, v.f)$ contiene $(u.d, u.f)$. Allora u e v si trovano in uno stesso albero della foresta DF, ed in particolare v è un predecessore di u;
- $(v.d, v.f)$ e $(u.d, u.f)$ sono intervalli disgiunti. Allora u e v si trovano in due alberi distinti della foresta DF.

Dimostrazione. Si consideri la situazione $u.d < v.d$ (la situazione opposta è del tutto analoga, basta invertire i vertici). Possono presentarsi due casi:

- $v.d < u.f$. Questo significa che v è stato scoperto dopo il tempo di scoperta di u ma prima del tempo di fine esplorazione di u, il che significa che v è un discendente di u. Inoltre, poiché v è stato scoperto prima di u, verrà completamente esplorato prima della completa esplorazione di u. Ne segue che i vari tempi coinvolti sono disposti nell'ordine:

  $$u.d < v.d < v.f < u.f$$

  Che equivale a dire che l'intervallo $(v.d, v.f)$ è interamente contenuto nell'intervallo $(u.d, u.f)$
- $u.f < v.d$. In questo caso, i due intervalli sono completamente disgiunti, in quanto l'ordine ottenuto è il seguente:

  $$u.d < u.f < v.d < v.f$$

  Infatti, questo equivale a dire che nessuno dei due vertici è stato scoperto mentre l'altro era grigio. Ne segue che anche u e v non sono discendenti l'uno dell'altro.

Un'altra proprietà della visita in profondità è che questa può essere utilizzata per classificare gli archi del grafo in input. È possibile definire quattro tipi di archi in base alla foresta DF prodotta da una visita in profondità di un grafo:

- **Arco d'albero**, indicato con la lettera **T**. Sono gli archi nella foresta DF. Un arco (u, v) è un arco d'albero se v viene scoperto la prima volta durante l'esplorazione di (u, v);
- **Arco all'indietro**, indicato con la lettera **B**. Sono quegli archi (u, v) che collegano un vertice u ad un antenato v in un albero DF;
- **Arco in avanti**, indicato con la lettera **F**. Sono quegli archi (u, v) che collegano un vertice u ad un vertice v in un albero DF;
- **Arco trasversale**, indicato con la lettera **C**. Possono connettere i vertici nello stesso albero DF, purché un vertice non sia antenato dell'altro, oppure possono connettere vertici di alberi DF differenti.

L'algoritmo per la visita in profondità contiene abbastanza informazioni per classificare gli archi che incontra. Infatti, ogni arco (u, v) può essere classificato in base al colore del vertice v che viene raggiunto quando l'arco viene ispezionato per la prima volta:

1. Se v è bianco, allora (u, v) è un arco d'albero. Questo è vero per definizione: se gli archi d'albero sono gli archi dove il vertice di arrivo non è stato ancora scoperto, allora tale vertice è un vertice bianco;
2. Se v è grigio, allora (u, v) è un arco all'indietro. Si noti infatti come i vertici grigi formino sempre una catena lineare di discendenti che corrisponde allo stack delle chiamate attive di `DFS-VISIT`; il numero di vertici grigi è uno in più della profondità dell'ultimo vertice scoperto. L'ispezione procede sempre a partire dal vertice grigio più profondo, quindi un arco che raggiunge un altro vertice grigio raggiunge un antenato;
3. Se v è nero e $(u.d < v.d)$, allora (u, v) è un arco in avanti. Questo perché v è stato completamente esplorato ma u è stato scoperto prima di v, quindi v è un discendente di u in uno stesso albero;
4. Se v è nero e $(u.d > v.d)$, allora (u, v) è un arco trasversale. Questo perché v è stato completamente esplorato ma u è stato scoperto dopo v, quindi v e u non possono fare parte dello stesso albero.

```
procedure DFS(G)
 1    foreach v ∈ V do
 2        v.color ← W
 3        v.π ← NULL
 4        v.d ← 0
 5        v.f ← 0
 6    time ← 0
 7    foreach v ∈ V do
 8        if v.color = W then
 9            DFS-VISIT(G, v)
```

```
procedure DFS-VISIT(G, u)
10    time ← time + 1
11    u.d ← time
12    u.color ← G
13    foreach v ∈ adj(u) do
14        if v.color = W then
15            (u, v) ← "T"
16            v.π ← u
17            DFS-VISIT(G, v)
18        else
19            if v.color = G then
20                (u, v) ← "B"
21            else
22                if u.d < v.d then
23                    (u, v) ← "F"
24                else
25                    (u, v) ← "C"
26    time ← time + 1
27    u.f ← time
28    u.color ← B
```

In un grafo non orientato potrebbe presentarsi una ambiguità nella classificazione degli archi, perché (u, v) e (v, u) sono di fatto lo stesso arco. In tal caso, l'arco viene classificato come il primo tipo della lista di classificazione che può essere applicato.

```
procedure DFS(G)
 1    foreach v ∈ V do
 2        v.color ← W
 3        v.π ← NULL
 4        v.d ← 0
 5        v.f ← 0
 6    time ← 0
 7    foreach v ∈ V do
 8        if v.color = W then
 9            DFS-VISIT(G, v)
```

```
procedure DFS-VISIT(G, u)
10    time ← time + 1
11    u.d ← time
12    u.color ← G
13    foreach v ∈ adj(u) - {u.π} do
14        if v.color = W then
15            (u, v) ← "T"
16            v.π ← u
17            DFS-VISIT(G, v)
18        else
19            (u, v) ← "B"
20    time ← time + 1
21    u.f ← time
22    u.color ← B
```

Il motivo per cui in un grafo non orientato non si presentano tutte e quattro le casistiche è spiegato nel seguente teorema.

> Un grafo non orientato, dopo una visita in profondità, ha solo archi d'albero e archi all'indietro.

L'algoritmo per la ricerca in profondità può venire adattato per l'esecuzione di un ordinamento topologico. Dato un grafo orientato, diretto e aciclico (*Directed Acyclic Graph*, *DAG*), un **ordinamento topologico** del DAG $G = (V, E)$ è un ordinamento lineare di tutti i suoi vertici tale che, se G contiene un arco (u, v), allora u compare prima di v in tale ordinamento. L'algoritmo per l'ordinamento topologico può essere descritto a grandi linee come segue:

1. Per ogni vertice viene effettuata una normale ricerca in profondità, in modo da calcolare il tempo di completamento $v.f$ per ciascun vertice v;
2. Una volta completata l'ispezione di un vertice, tale vertice viene inserito in uno stack;
3. Vengono stampati i vertici contenuti nello stack (nell'ordine in cui sono stati inseriti).

```
procedure DFS(G)
 1    foreach v ∈ V do
 2        v.color ← W
 3        v.π ← NULL
 4        v.d ← 0
 5        v.f ← 0
 6    time ← 0
 7    S ← []
 8    foreach v ∈ V do
 9        if v.color = W then
10            DFS-VISIT(G, v)
11    while not IS-EMPTY(S) do
12        POP(S)
```

```
procedure DFS-VISIT(G, u)
13    time ← time + 1
14    u.d ← time
15    u.color ← G
16    foreach v ∈ adj(u) do
17        if v.color = W then
18            v.π ← u
19            DFS-VISIT(G, v)
20    time ← time + 1
21    u.f ← time
22    u.color ← B
23    PUSH(S, u)
```

Il tempo di esecuzione dell'algoritmo per l'ordinamento topologico è $O(|V|+|E|)$. L'ordinamento in profondità, anche se viene esteso con l'aggiunta di una operazione di push, richiede comunque tempo $O(|V| + |E|)$, perché tale operazione ha tempo di esecuzione immediato.

Dato che la stampa degli elementi della coda è una operazione immediata che viene effettuata esattamente $|V|$ volte, si ha che il tempo di esecuzione complessivo è asintoticamente pari a $O(|V|) + O(|V| + |E|) = O(|V| + |E|)$.