Analisi e progettazione algoritmi

Programmazione dinamica

È una tecnica algoritmica usata per risolvere problemi complessi suddividendoli in sottoproblemi più semplici, e risolvendo ciascuno di essi **solo una volta**, memorizzando poi i risultati per riutilizzarli quando necessario.

La differenza con il *divide et impera* è che la programmazione dinamica affronta problemi con **sottoproblemi sovrapposti**, e salva il risultato di ciascun sottoproblema, per evitare il ricalcolo degli stessi sottoproblemi.

Il divide et impera, invece, divide i problemi in **sottoproblemi indipendenti**, e li risolve separatamente per poi combinare i risultati.

La programmazione dinamica si usa quando il problema ha una **struttura ottima**.

La **proprietà della sottostruttura ottima** non è da usare per creare il codice; bisogna invece usare le equazioni di ricorrenza (derivanti dalle proprietà).

I problemi di **ottimizzazione combinatoria** si possono risolvere con il metodo "**brute-force"**, o "stupido", provando tutte le 2^n combinazioni; questo aumenta sia il tempo che lo spazio di esecuzione.

Longest Common Subsequence (LCS)

Data una sequenza $X = < \ldots >$ derivante da un certo alfabeto ∑, definiamo Z come una **sottosequenza** di X se e solo se

Esiste una sequenza strettamente crescente di indici $i <i_j>$ *tale che,* $\forall j \in [1,k]$, $z_j = xi_j$

Il testo del problema è quindi:

Date due sequenze X e Y, rispettivamente di m ed n numeri, si determini la lunghezza di una tra le più lunghe sottosequenze comuni al prefisso X_i e al prefisso Y_j.

Con "la lunghezza", si va a risolvere una versione ridotta del problema PB, nota come PBR.

Sottoproblemi

Il problema PBR contiene diversi sottoproblemi, ognuno dei quali non ha come input la coppia (X,Y), ma una coppia di prefissi di tali sequenze.

Quindi, il problema è definito come segue:

Date due sequenze X e Y, rispettivamente di m ed n numeri interi, si determini la lunghezza di una tra le più lunghe sottosequenze comuni al prefisso X_i e al prefisso Y_j.

Dato che $0 \leq i \leq m$ e $0 \leq j \leq n$, si ottengono **(m+1)*(n+1)** sottoproblemi. Ad ogni sottoproblemi di dimensione (i,j) è associata la variabile $\mathbf{c_{i,j}}$, ovvero la *lunghezza di una tra le più lunghe sottosequenze comuni a X_i e Y_j.* Ognuna di queste variabili c è una **blackbox**: si può utilizzare, ma non è possibile conoscerne il contenuto.

Equazioni di ricorrenza

Le equazioni di ricorrenza si scrivono **senza sapere il valore delle soluzioni dei sottoproblemi**, ma sapendo solamente che tali soluzioni esistono e si possono utilizzare.

Abbiamo le seguenti equazioni:

- **Caso base**: se $i = 0 \vee j = 0 \Rightarrow c_{i,j} = 0$
- **Passo ricorsivo**: se $i > 0$ e $j > 0$, si vanno a considerare i due prefissi X_i e Y_j.
    - I dati disponibili sono: l'input X e in particolare l'elemento x_i, l'input Y ed in particolare l'elemento y_j, e tutte le variabili $\{c_{0,0}, \ldots, c_{i-1,j}, c_{i,j-1}\}$
    - Si applica il teorema della proprietà della sottostruttura ottima della LCS
    - $\mathbf{x_i = y_j}$: se Z_k è una LCS fra X_i e Y_j, allora $z_k = x_i = y_j$, e Z_{k-1} è una LCS tra X_{i-1} e Y_{j-1}
        - $C_{i,j} = 1 + c_{i-1,j-1}$ se $x_i = y_j$
    - $\mathbf{x_i \neq y_j}$:
        - $\mathbf{z_k \neq x_i}$: si considera $c_{i-1,j}$, ossia la soluzione del sottoproblema di dimensione (i-1, j) con input X_{i-1} e Y_j
        - $\mathbf{z_k \neq y_j}$: si considera $c_{i,j-1}$, ossia la soluzione del sottoproblema di dimensione (i, j-1) con input X_i e Y_{j-1}

Siccome z_k non è noto, non possiamo sapere a priori quale dei due casi si verifichi. Quindi, consideriamo la soluzione di **entrambi i sottoproblemi** di dimensione (i-1,j) e (i,j-1), e scegliamo quella di **valore massimo**. Quindi, si può riassumere come:

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

La ricorsione permette di scomporre il problema in sottoproblemi più piccoli, ma molti sottoproblemi vengono ripetuti. Per evitare ricalcoli inutili, si può usare una **tabella**, con dimensioni *(n + 1) x (m + 1)*, dove n e m sono le lunghezze di X e Y rispettivamente. Ogni cella della tabella contiene la lunghezza della LCS tra il prefisso di X di lunghezza i, e il prefisso di Y di lunghezza j.

ESEMPIO

Consideriamo X = <a,b,c,b,d,a,b> con n=7.

Una possibile sottosequenza *Z* = <*b,c,d,b*> è tale poiché possiamo individuare indici *2, 3, 5, 7* corrispondenti a *b,c,d,b*, che sono in ordine **strettamente crescente**.

Una sequenza come *Z* = <*b,c,b,c*> non è una sottosequenza valida perché gli indici non sono in ordine strettamente crescente.

Tabella

Per costruire la tabella dell'LCS, è necessario mettere sulle righe e le colonne della tabella gli indici di ciascuna sottosequenza, con il rispettivo valore.

1. **Riempi la prima colonna e riga di zero**: corrisponde al **simbolo vuoto epsilon**, quindi non ci possono essere LCS; corrisponde al caso base dell'algoritmo
2. **Dividi in due casi**
    a. **Se $x_i = y_j$**
        i. Somma +1 dall'elemento che sta up-left rispetto a quello che stai considerando; quindi, quello ad indice (i-1, j-1)
    b. **Se $x_i \neq y_j$**
        i. Considera l'elemento sovrastante e a sinistra di quello che hai selezionato, e prendi il massimo tra i due
        ii. Se entrambi sono uguali, allora la scelta predominante è l'alto
3. L'ultima casella della tabella è la **soluzione** dell'LCS

Di seguito un esempio:

		0	1	2	3	4	5	6	7	8	**J**
		ε	2	7	4	23	21	14	1	8	Y_j
0	ε	0	0	0	0	0	0	0	0	0	
1	2	0	1 ↖	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	1 ←	
2	4	0	1 ↑	1 ↑	2 ↖	2 ↑	2 ←	2 ←	2 ←	2 ←	
3	7	0	1 ↑	2 ↖	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	
4	11	0	1 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	2 ↑	
5	21	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↖	3 ←	3 ←	3 ←	
6	14	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↖	4 ←	4 ←	
7	14	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	4 ↑	4 ↑	
7	1	0	1 ↑	2 ↑	2 ↑	2 ↑	3 ↑	4 ↑	5 ↖	5 ←	

Prefissi di una sequenza

Se X = <x_1, ..., x_n> è una sequenza con i = <1, ..., n>, il **prefisso di lunghezza i-esima** di X è la sottosequenza X_i = <x_1, ..., x_i>, ovvero *contiene tutti gli elementi fino al termine* x^i.

Il prefisso di lunghezza zero $\mathbf{X_0}$ è la sequenza vuota.

Algoritmo ricorsivo

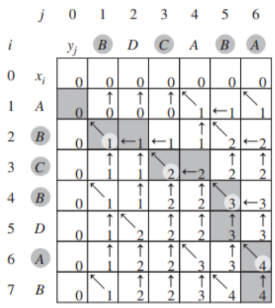
Questo algoritmo presenta una complessità in tempo esponenziale, dato che ogni sottoproblema viene risolto più volte.

Si utilizza una tecnica **bottom-up** (iterativa) per risolvere il problema: questa tecnica permette di calcolare la soluzione di ogni sottoproblema solamente una volta. Si ottiene quindi un algoritmo con *complessità in spazio maggiore*, ma complessità in *tempo minore*. Risolve il problema in $\mathrm{O}(m*n)$ occupando $\Theta(m*n)$ spazio in memoria.

```
LCS-R(X,Y,i,j)
if i = 0 v j = 0 return <>
else
     if xi = yj
          return LCS-R(X,Y,i-1,j-1) + 1
          //se è uguale, allora concatena l'elemento uguale
     else
          A = LCS-R(X,Y,i-1,j)
          B = LCS-R(X,Y,i,j-1)
          if |A| ≥ |B| return A
          else return B
```

Algoritmo iterativo per le lunghezze

```
LCS-IT(XY)
m = length(X), n = length(Y)
for (j = 0 to n)
     c[0,j] = 0
for (i = 1 to m)
     c[i,0] = 0
for (i = 1 to m)
     for (j=1 to n)
          if (xi == yj)
               c[i,j] = c[i-1, j-1] + 1
               b[i,j] = " up-left "
          else if (c[i-1, j] ≥ c[i, j-1])
               c[i,j] = c[i, j-1]
               b[i,j] = " <- "
```



Quando le lettere sono uguali, prendere il valore in diagonale e sommare 1. Se invece sono diverse, prendere il valore massimo tra alto e sinistra.

Stampa della LCS

Per stampare la LCS, si parte dalla casella più in basso a destra, e si seguono le frecce.

Es. $S_{7,6} = S_{6,6} = S_{5,5}|A = S_{4,5}|A = S_{3,4}|B|A = S_{3,3}|B|A = S_{2,2}|C|B|A = S_{2,1}|C|B|A = S_{1,0}|B|C|B|A$

```
PRINT-LCS(X,b,i,j)
if (i ≠ 0 /\ j ≠ 0)
     if (b[i,j] = “ up-left “)
          PRINT-LCS(X,b,i-1,j-1)
          print xi
     else if (b[i,j] = “ up “)
          PRINT-LCS(X,b,i,j-1)
     else
          PRINT-LCS(X,b,i-1,j)
```

Algoritmo iterative della tabella

Questo algoritmo è stato fatto nel primo esercizio di tutoraggio. Sono create due matrici: matrice **C**, che tiene conto dei risultati della LCS, e un’altra matrice **Path**, che memorizza le frecce all’interno della tabella.

```
COMPUTE-OPTIMAL-LCS(A, B)
for i in {0, …, |A|}
     C[i, 0] = 0
     Path[i,0] = ‘-‘
for j in {1, …, |B|}
     C[0,j] = 0
     Path[0,j] = ‘-‘
for i in {1, …, |A|}
     for j in {1, …, |B|}
          if A[i] == B[j]
               C[i,j] = C[i-1,j-1] + 1
               Path[i,0] = ‘D’
          else if C[i-1,j] >= C[i, j-1] //se quello a sx > sopra
               C[i,j] = C[i-1,j]
               Path[i,j] = ‘L’
          else
               C[i,j] = C[i,j-1]
               Path[i,j] = ‘U’
```

```
return Path, C
```

Versione C++

Questa versione non è stata fatta a lezione, ma è stata aggiunta perché potrebbe essere più chiara rispetto al pseudocodice.

```
char* LCS_R(char* x, char* y, int i, int j){

    if (i == 0 || j == 0) { //if we arrived at the end of the string
        char* emptyString = new char[1];
        emptyString[0] = '\0';
        return emptyString; //return an empty string
    }

    if (x[i-1] == y[j-1]) { //if the last character is the same
        char* z = LCS_R(x,y,i-1,j-1);
        int len = stringLength(z);

        char* result = new char[len+2];
        result = stringCopy(z);

        result[len] = x[i-1]; //add the same character to the end of the string
        result[len+1] = '\0';

        delete[] z; //memory allocation
        return result;
    } else {
        char* A = LCS_R(x,y,i-1,j); //search a common char in the first string
        char* B = LCS_R(x,y,i,j-1); //search a common char in the second string

        if (stringLength(A) >= stringLength(B)) {
            //if the first string is longer, then the LCS is in the first
            delete[] B;
            return A;
        } else {
            //if the second string is longer, then the LCS is in the second
            delete[]A;
            return B;
        }
    }
}
```

Sottosequenza decrescente più lunga e varianti

Ci interessa trovare la sottosequenza decrescente più lunga all'interno di una sequenza, che può essere estesa per gestire vari vincoli.

Per una sottosequenza decrescente, definiamo la relazione ricorsiva per ciascun elemento della sequenza $C_{i,j}$ con:

$$C_{i,j} = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j \,/\backslash\, x_h > x_i\} + 1$$

La spiegazione per questa equazione è la seguente:

- h deve essere compreso tra **1** (l'inizio della prima stringa), e **i** (l'indice ultimo della prima stringa)
- k deve essere compreso tra **1** (l'inizio della seconda stringa), e **j** (l'indice ultimo della seconda stringa)

- Bisogna poi mettere la **condizione che si vuole verificare**

Abbiamo varie varianti per questo.

Senza simboli consecutivi ripetuti

$C_{i,j} = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j \,/\backslash\, x_h \neq x_i\} + 1$

Senza due numeri pari consecutivi

$C_{i,j} = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j \,/\backslash\, x_h \bmod 2 \neq x_i \bmod 2\} + 1$

Con alternanza di colori nei simboli

Ogni simbolo può avere un colore $\sum \text{—>} R = \{\text{rosso}, \text{blu}\}$

$C_{i,j} = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j \,/\backslash\, \text{col}(x_h) \neq \text{col}(x_i)\} + 1$

Senza simboli blu consecutivi

Si gestisce il caso in cui $x_i = y_j$, dividendo in due sottocasi:

- **Caso 1**: $\text{col}(x_i) = \text{blu}$
  - $C_{i,j} = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j \,/\backslash\, \text{col}(x_h) \neq \text{blu}\} + 1$
- **Caso 2**: $\text{col}(x_i) \neq \text{blu}$
  - $C_{i,j} = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j\} + 1$

Senza due numeri pari consecutivi (anche entrambi dispari)

$C_{i,j} = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j \,/\backslash\, x_h \bmod 2 \neq 0 \,\backslash/\, x_i \bmod 2 \neq 0\} + 1$

LCS con numero pari di rossi

Caso base

Se $i = 0 \,\backslash/\, j = 0$, $e_{i,j,p} = \text{true}$ sse tutte le LCS di X_i, Y_j *hanno un numero pari* (se p = 0) o *dispari* (se p = 1) di rossi.

- $e_{i,j,0}$ = true se pari
- $e_{i,j,1}$ = false se dispari

Passo ricorsivo

Se $i > 0 \,/\backslash\, j = 0 \,/\backslash\, p \in \{0,1\}$

- se $x_i \neq y_j$
  - $e_{i,j,0} = \begin{cases} e_{i-1,j,0} & \text{se } c_{i-1,j} > c_{i,j-1} \\ e_{i,j-1,0} & \text{se } c_{i-1,j} < c_{i,j-1} \\ e_{i,j-1,0} \wedge e_{i-1,j,0} & \text{se } c_{i-1,j} = c_{i,j-1} \end{cases}$

Questo vale anche per i dispari, basta cambiare tutti gli **0** con **1**.

- Se $x_i = y_j$ e col(x_i) = red

  $e_{i,j,0} = e_{i-1,j-1,1} \quad e_{i,j,1} = e_{i-1,j-1,0}$

- Se $x_i = y_j$ e col(x_i) ≠ red

  $e_{i,j,0} = e_{i-1,j-1,0} \quad e_{i,j,1} = e_{i-1,j-1,1}$

Al massimo tre simboli colorati di rosso

Bisogna calcolare la lunghezza della sottosequenza più lunga tra due sequenze X_m e Y_n, permettendo **al massimo 3 simboli colorati di rosso**.

Ogni simbolo sia o rosso o blu, indicato con $\sum \longrightarrow \{R,B\}$.

Definiamo una tabella di programmazione dinamica $C_{i,j,r}$ dove $i \in \{0, \ldots, m\}$, $j \in \{0, \ldots, n\}$, $r \in \{0, \ldots, 3\}$.

Il valore $C_{i,j,r}$ rappresenta la **lunghezza della sottosequenza più lunga tra X_i e Y_j, con al massimo r simboli rossi**.

Caso base

Quando i = 0 o j = 0, quindi quando una delle sequenze è vuota, la lunghezza della sottosequenza è 0.

$$C(i,j,r) = 0 \qquad \forall R \in \{0,1,2,3\}$$

Caso ricorsivo

Per i > 0, j > 0, e qualunque r, la relazione ricorsiva si divide in due scenari.

Caso 1

Se $x_i \neq y_j$, la sottosequenza viene formata prendendo il massimo tra l'esclusione di x_i o di y_j.

$$C(i,j,r) = \max\{C(i-1, j, r), C(i, j-1, r)\}$$

Caso 2

Se $x_i = y_j$, allora abbiamo due sottocasi:

- Se **col(x_i) ≠ rosso**, la sottosequenza può estendersi senza ridurre il numero di simboli rossi
  - $C(i,j,r) = C(i-1, j-1, r) + 1$
- Se **col(x_i) = rosso e r = 0**, la sequenza non può accettare un altro simbolo rosso
  - $C(i,j,r) = C(i-1, j-1, 0)$
- Se **col(x_i) = rosso e r > 0**, la sequenza può accettare un altro simbolo rosso
  - $C(i,j,r) = C(i-1, j-1, r-1) + 1$

Lunghezza di LCS con ingombro

Dati due sequenze X e Y, lunghi rispettivamente **m** ed **n** elementi, con C come capacità residua, calcolare la LCS con un ingombro complessivo minore della capacità. Definiamo la matrice **L** per tenere conto della lunghezza, e la matrice **Z** per tenere conto degli elementi che faranno parte della sequenza.

Definiamo $L^{i,j,c} = |Z_{i,j,c}|$, e w_i come la funzione peso, che determina il peso di ciascun elemento.

Caso base

Quando i = 0 o j = 0 o c = 0, $L_{i,j,c} = 0 \,/\backslash\, Z_{i,j,c} = \text{vuoto}$.

Passo ricorsivo

∀ i > 0, j > 0, c > 0 abbiamo:

- Se $X_i = Y_j \,/\backslash\, W(X_i) > C$
  - $L_{i,j,c} = L_{i-1,j-1,c} \,/\backslash\, Z_{i,j,c} = Z_{i-1,j-1,c}$
- Se $X_i = Y_j \,/\backslash\, W(X_i) =< C$
  - $L_{i,j,c} = \max\{L_{i-1,j-1,c-W(Xi)} + 1, L_{i-1,j-1,c}\}\ (\max\{b_1, b_2\})$
  - $Z_{i,j,c} = \begin{cases} Z_{i-1,j-1,c} & \text{se } b_1 > b_2 \\ Z_{i-1,j-1,c-w(xi)} \mid X_i & \text{altrimenti} \end{cases}$
- Se $X_i \neq Y_j$
  - $L_{i,j,c} = \max\{L_{i-1,j,c}, L_{i,j-1,c}\}$

La soluzione è quindi $L_{m,n,C} \,/\backslash\, Z_{m,n,C}$

Verifica se tutte le LCS contengono almeno 3 simboli rossi

Per stabilire se tutte le LCS di X e Y contengono almeno tre simboli rossi, manteniamo una variabile $e_{i,j,r}$ che è vera, se tutte le LCS di X_i e Y_j hanno almeno r simboli rossi.

Caso base

Per i = 0 o j = 0, $e_{i,j,r}$ = *falso* ∀ r > 0

Per i = 0, o j = 0, e r = 0, $e_{i,j,r}$ = *true*

Passo ricorsivo

∀ i > 0 e j > 0, e ∀ r {0, 1, 2, 3}:

- Se $x_i \neq y_j$
  - $e_{i,j,r} = \begin{cases} e_{i,j-1,r} & \text{se } c_{i,j-1} > c_{i-1,j} \\ e_{i-1,j,r} & \text{se } c_{i,j-1} < c_{i-1,j} \\ e_{i-1,j,r} \wedge e_{i,j-1,r} & \text{se } c_{i,j-1} = c_{i-1,j} \end{cases}$
- Se $x_i = y_j$ e x non è rosso
  - $e_{i,j,r} = e_{i-1,j-1,\,r}$
- Se $x_i = y_j$ e x_i è rosso
  - Se r > 0
    - $e_{i,j,r} = e_{i-1,j-1,r-1}$
  - Se r = 0
    - $e_{i,j,0}$ = vero

Calcolare la lunghezza della LCS con esattamente 3 simboli rossi

Per le sequenze X_i, Y_j e un intero r, il valore $C_{i,j,r}$ rappresenta la **lunghezza della LCS di X_i e Y_j che contiene esattamente r simboli rossi**.

Caso base

Per i = 0, o j = 0, si fa $C_{i,j,0} = 0$, $C_{i,j,1} = C_{i,j,2} = C_{i,j,3} = -\infty$.

Passo ricorsivo

∀ i > 0, e j > 0, si calcola il valore in base a se i caratteri correnti coincidono e quanti simboli rossi sono ammessi.

$C_{i,i,r} = \max(C_{i-1,j,r}, C_{i,j-1,r})$ se $x_i \neq y_j$

$$\begin{cases} C_{i-1,j-1,r} + 1 & \text{se } x_i = y_j,\ col(x_i) \neq red \\ C_{i-1,j-1,r-1} + 1 & \text{se } x_i = y_j,\ col(x_i) = red,\ r > 0 \\ C_{i-1,j-1,0} + 1 & \text{se } x_i = y_j,\ col(x_i) = red,\ r = 0 \end{cases}$$

Proprietà della sottostruttura ottima

Definiamo i termini che ci serviranno:

- **X**: sequenza di m numeri interi
  - **X_i**: prefisso di lunghezza i ($1 \leq i \leq m$)
- **Y**: sequenza di n numeri interi
  - **Y_j**: prefisso di lunghezza j ($1 \leq j \leq n$)
- **$W_{i,j}$**: insieme di tutte le sottosequenze crescenti comuni, non necessariamente le più lunghe, di X_h e Y_k, che finiscono con $x_h = y_k$, a cui è possibile concatenare x_i/y_j
  - **$Z_{i,j}$**: una tra le più lunghe sottosequenze crescenti comuni di X_i e Y_j tale che termini con $x_i = y_j$
    - $Z_{i,j} = Z^*|x_i$; $Z_{i,j} = Z^*|y_j$
    - $Z^* \in W_{i,j}$
    - $|Z^*| = \max(W, W_{i,j}\ \{|W|\})$
  - **Z'**: qualche sottosequenza comune crescente di un prefisso più piccolo di X_i e Y_j
    - **Z'**: sottosequenza comune crescente di X_r e Y_s, termina con $x_r < x_i$
    - **z'**: ultimo elemento di Z'
      - $z' < x_i = y_j$, possibile concatenare x_i/y_j a Z'
      - $r < i$, $s < j$, $x_r = y_s = z'$

Dimostrazione

Si mostra che una soluzione Z^* del problema non può essere migliorata aggiungendo un ulteriore elemento dalla sequenza X_i o Y_j.

Si fa un ragionamento per **assurdo**: si assume che *esista una soluzione Z', ottenuta aggiungendo* x_i, che sia **migliore di** Z^*.

$Z'_{i,j} = Z' \cup \{x_i\}$. Z' è una sottosequenza comune crescente di X e Y, che termina con z'.

Si assume **per assurdo** che Z^* non sia una soluzione ottima, $|Z'| > |Z^*|$.

L'elemento z' è l'ultimo di Z', e viene aggiunto x_i/y_j alla sottosequenza. Questo implica che tutti gli elementi prima di z' devono essere minori di z'.

Esempio

$X = [1,3,5,7]$, $Y = [2,3,5,8]$, $Z^* = [3,5]$

Se aggiungiamo $x_4 = 7$, otteniamo $Z' = [3,5,7]$, la quale **non è più una LCS.**

Spiegazione originale

Afferma che una soluzione ottimale di un problema può essere costruita a partire da soluzioni ottimali dei suoi sottoproblemi. Questo è un principio chiave per la **programmazione dinamica**.

Sia Z_k la LCS delle sequenze X_m e Y_n. Esistono due casi principali che descrivono la relazione:

1. **Se $X_m = Y_n$**
   a. Se l'ultimo carattere di X e Y è lo stesso, allora questo carattere fa parte della LCS
   b. $Z_k = LCS(X_{1..m}, Y_{1..n}) = LCS(X_{1...m-1}, Y_{1..n-1}) + X_m$
2. **Se $X_m \neq Y_n$**
   a. Se l'ultimo carattere non coincide, allora la LCS non può includere X_m né Y_n.
   b. In questo caso, deve essere la LCS di una delle coppie di sottosequenze:
      i. La LCS di $X_{1..m-1}$ e $Y_{1..n}$
      ii. La LCS di $X_{1..m}$ e $Y_{1..n-1}$
   c. $Z_k = LCS(X_{1..m}, Y_{1..n}) = \max(LCS(X_{1..m-1}, Y_{1..n}), LCS(X_{1..m}, Y_{1..n-1})$
3. **Se $X_m \neq Y_n$ /\ $Z_k \neq Y_n$**
   a. Z_k è una LCS di X_m e Y_{n-1}

Dimostrazione

Dimostrazione del caso $X_m = Y_n$

Se Z_k è la LCS di X_m e Y_n, e $X_m = Y_n$, allora si può dimostrare per **assurdo** che Z_k non può escludere X_m. Se escludesse X_m, otteniamo una sottosequenza comune di $X_{1..m-1}$, e $Y_{1..n-1}$, che non può essere più lunga di Z_{k-1}.

Pertanto, Z_k deve includere X_m (e Y_n), e si può dire che:

$Z_k = Z_{k-1} + X_m$, dove Z_{k-1} è la LCS di $X_{1..m-1}$ e $Y_{1..n-1}$

Dimostrazione del caso $X_m \neq Y_n$

La LCS non può includere entrambi i caratteri. Pertanto, Z_k deve essere ottenuta o da $\mathrm{LCS}(X_{1..m-1}, Y_n)$, o $\mathrm{LCS}(X_m, Y_{1..n-1})$.

Bisogna quindi vedere la sottosequenza più lunga tra le due, quindi:

$$Z_k = \max(\mathrm{LCS}(X_{1..m-1}, Y_{1..n}), \mathrm{LCS}(X_{1..m}, Y_{1..n-1})$$

Dimostrazione del caso $X_m \neq Y_n$ /\ $Z_k \neq X_m$

Z_k è una sottosequenza comune di X_{m-1} e Y_n. Supponiamo per assurdo che Z_k non fosse LCS di X_{m-1} e Y_n, e che quindi esista una sottosequenza comune Z' di X_{m-1} e Y_n di lunghezza maggiore di k. Allora, sarebbe anche una sottosequenza comune di X_m e Y_n, contraddicendo l'ipotesi che Z sia una LCS di X e Y.

Heaviest Common Subsequence

Questo problema è stato fatto in un esercizio del tutoraggio.

Date due sequenze $A = a_1, \ldots, a_n$, $B = b_1, \ldots, b_m$, di lunghezza n e m, e una funzione $W : \Sigma \to N^+$ che *associa ad ogni carattere un peso*.

L'obiettivo è **determinare una tra le sottosequenze comuni X tra A e B che massimizzi il peso secondo la funzione W**.

Sottoproblema

Date due sequenze A, B di lunghezza n e m, con funzione W : Σ -> N che associa ad ogni carattere un peso. Si determini una tra tutte le sequenze comuni X ***dei prefissi A_i e B_j*** *che massimizzi il peso secondo la funzione W.*

Introduciamo la variabile $c_{i,j}$ come il peso di una tra le più pesanti sottosequenze comuni di A_i, B_j.

Sottostruttura ottima

Abbiamo tutte le variabili del problema, e introduciamo la variabile $c_{s,t}$ come *il peso di una HCS di* $a_1, a_2, ..., a_s$, *e* $b_1, b_2, ..., b_t$, *con* $0 \leq s \leq i$ e $0 \leq t \leq j$.

Sia $Z = z_1, ..., z_k$ una HCS per A_i e B_j. Abbiamo anche le seguenti definizioni:

- Se $a_i = b_j$ allora $z_k = a_i = b_j$; Z_{k-1} è un HCS per A_{i-1}, e B_{j-1}, quindi
  - $c_{i,j} = c_{i-1,j-1} + W(a_i)$
- Se $a_i \neq b_j$ allora $z_k \neq a_i$ implica che Z è una HCS di A_{i-1}, B_j
- Se $a_i \neq b_j$ allora $z_k \neq b_j$ implica che Z è una HCS di A_i, B_{j-1}

Dimostrazione

- Se $a_i = b_j$
  - Supponiamo per assurdo che $\mathbf{a_i = b_j}$, ma $\mathbf{z_k \neq a_i}$. Allora Z è una sottosequenza di A_{i-1}, B_{j-1}, ma in questo caso possiamo *accodare ai a Z formando una sottosequenza comune*.
  - Siccome $W(a_i) > 0$, otteniamo $\mathbf{W^*(Z\ a_i) > W^*(Z)}$, e questo contraddice l'assunzione che Z sia una HCS di A_i, B_j.
  - Dimostriamo che Z_{k-1} è un HCS per A_{i-1}, B_{j-1}
    - Supponiamo per assurdo che esiste una sottosequenza comune **V** a A_{i-1}, B_{j-1}, con $W^*(V) > W^*(Z_{k-1})$.
    - Siccome V è una sottosequenza di A_{i-1}, B_{j-1}, si può accodare $a_i = b_j$, e formare una sottosequenza **comune a A_i, B_j più pesante di Z in quanto $W^*(V\ a_i) > W^*(Z_{k-1}a_i) = W^*(Z)$**, ottenendo una contraddizione
- Se $a_i \neq b_j$
  - Se $a_i \neq z_k$, allora Z è una **sottosequenza comune di A_{i-1}, B_j**, se esistesse una sottosequenza comune V di A_{i-1}, B_j, con peso maggiore ($W^*(V) > W^*(Z)$)
    - V sarebbe una sottosequenza di A_i, B_j, ma si **contraddice che Z è HCS di A_i, B_j**

Equazione di ricorrenza

Caso base

La sottosequenza comune è vuota, il suo peso risulta $W(\emptyset) = 0$.

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

Passo ricorsivo

$$c_{i,j} = \begin{cases} W(a_i) + c_{i-1,j-1} & \text{se } a_i = b_j \end{cases}$$

$\max\{c_{i,j-1}, c_{i-1,j}\}$ se $a_i \neq b_j$

Algoritmo iterativo

```
COMPUTE_OPTIMAL_HCS(A, B, W)
for i in {1, …, |A|}
     C[i,0] = 0
for j in {1, …, |B|}
     C[0,j] = 0
for i in {1, …, |A|}
     for j in {1, …, |B|}
          if A[i] == B[j]
               C[i,j] = C[i-1, j-1] + W(A[i])
          else
               C[i,j] = max(C[i-1, j], C[i, j-1])
return C[|A|, |B|]
```

Weighted Interval Scheduling

Commentato [FD1]: Mi son reso conto che manca la dimostrazione, tu ce l'hai?

Commentato [FC2R1]: No :/ poi chiedo a qualcuno

Commentato [f3R1]: Trovata :) poi la metto

Sia A = {1, 2, …, n} un insieme di n attività, n ∈ N. Ad ogni attività i ∈ A:

- s_i indica il tempo di inizio dell'attività
- e_i ($e_i \geq s_i$), indica il tempo di fine dell'attività
- v_i indica il valore dell'attività.

Due attività sono **compatibili** se non si sovrappongono: $[s_i, e_i) \cap [s_j, e_j) = \emptyset$.

Un insieme A di attività **contiene attività mutualmente compatibili** sse $\forall\, i,j \in A\ (i \neq j)$, $[s_i, e_i) \cap [s_j, e_j) = \emptyset$.

Definiamo la funzione **COMP: P({1, …, n}) —> {True, False}**, che associa ad ogni sottoinsieme di attività A *true* se A contiene attività mutualmente compatibili, *false* altrimenti. Quindi:

$$\text{COMP}(A) = \begin{cases} \text{True} & \text{se } \forall\, i, j \in A, i \neq j, [s_i, e_i) \cap [s_j, e_j) = \emptyset \\ \text{False} & \text{altrimenti} \end{cases}$$

Si definisce anche la funzione V: P({1, …, n}) —> R+, che:

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

Dato un insieme {1, ..., n} di attività, trovare un suo sottoinsieme di attività mutualmente compatibili di valore massimo.

- **Istanza**: $\{1, ..., n\}$ con $([s_i, e_i], v_i)\ \forall\ i \in \{1, ..., n\}$
- **Soluzione**: $S \subseteq \{1, ..., n\}$ tale che:
  - $COMP(S) = True\ /\backslash\ V(S) = \max(A \subseteq \{1, ..., n\}: COMP(A)=True)\ \{V(A)\}$

Sottoproblemi

Definiamo il **sottoproblema di dimensione i**, definito come:

*Dato un insieme {1, ..., i} di attività, trovare un suo sottoinsieme di attività mutualmente compatibili di valore massimo***.**

Si ottengono quindi **n+1** sottoproblemi, ad ognuno dei quali è associata la coppia di variabili **(OPT_i, S_i):**

- **$OPT_i = V(S_i)$,** ovvero il **valore** di un sottoinsieme di {1, ..., i} che contiene attività mutualmente compatibili di peso massimo
- **S_i**, un sottoinsieme di {1, ..., i} che contiene attività mutualmente compatibili di peso massimo, pari a OPT_i

Equazioni di ricorrenza

Caso base

$i = 0,\ OPT_i = 0,\ S_i = \emptyset$ se $i = 0$

Passo ricorsivo

$i > 0$, le attività {1, ..., n} siano ordinate rispetto ai tempi di fine.

Definiamo $\psi(i) = \max\{j \mid j < i,\ j \text{ è compatibile con } i\}$, assumendo che $\max(\emptyset) = 0$. Dividendo in casi:

- $i \notin S_i$: si considera l'insieme di attività {1, ..., i-1}
  - $OPT_i = OPT_{i-1},\ S_i = S_{i-1}$
- $i \in S_i$: $OPT_i = OPT_{\psi(i)} + v_i,\ S_i = S_{\psi(i)} \cup \{i\}$
  - Si può scrivere quindi $OPT_i = \max\{OPT_{i-1}, OPT_{\psi(i)} + v_i\}$

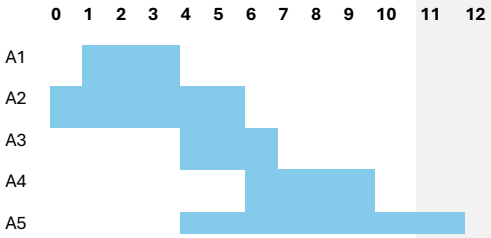
$$S_i = \begin{cases} S_{i-1} & \text{se } OPT_{i-1} \geq OPT_{\psi(i)} + v_i \\ S_{\psi(i)} \cup \{i\} & \text{altrimenti} \end{cases}$$

Esempio passo per passo

Supponiamo di avere queste attività.

Ogni attività ha un inizio, una fine, e un valore relativo. Indichiamo con **p(i)** il numero dell'evento precedente non conflittuale (con questo si intende che non si sovrappongano).

Attività	Inizio	Fine	Valore	P(i)
1	1	3	10	0
2	0	5	2	0
3	3	6	8	1
4	5	9	1	2
5	3	11	1	1
6	9	12	3	4



La formula da usare per calcolare ricorsivamente M(i) è:

$$M(i) = \max\{v(i) + M(p(i)), M(i-1)\}$$

dove:

- $v(i)$ è il valore dell'attività i
- $M(p(i))$ è il *valore massimo delle attività fino all'attività p(i)*, ossia l'ultima attività che non si sovrappone con i
- $M(i-1)$ è il valore massimo **fino all'attività precedente** i-1

Iniziamo innanzitutto a determinare le p(i) per ciascuna attività. Una volta eseguito questa operazione, è necessario svolgere i calcoli.

- $M[1] = \max\{v(1) + M[0],\ M[0]\} = \max\{10 + 0, 0\} = 10$
- $M[2] = \max\{v(2) + M[0],\ M[1]\} = \max\{2 + 0,10\} = 10$
- $M[3] = \max\{v(3) + M[1],\ M[2]\} = \max\{8 + 10, 10\} = 18$
- $M[4] = \max\{v(4) + M[2],\ M[3]\} = \max\{1 + 10, 18\} = 18$
- $M[5] = \max\{v(5) + M[1],\ M[4]\} = \max\{1 + 10, 18\} = 18$
- $M[6] = \max\{v(6) + M[4],\ M[5]\} = \max\{3 + 18, 18\} = 21$

Il massimo quindi è **21** per questo esempio.

Algoritmo ricorsivo

```
WIS-RIC(i)
if i = 0 then
     return(0, vuoto)
else
     (V1, S1) = WIS-RIC(i-1)
     (V2, S2) = WIS-RIC(phi(i))
     V2 = V2 + vi
     if V1 >= V2 then
          return (V1, S1)
```

```
     else
          return (V2, S2∪{i})
```

Algoritmo ricorsivo puro:

```
R-OPT(i)
//se non ci sono attività, il valore ottimale è 0
if i=0 return 0
//valuta se includere o no e ritorna il massimo tra:
//- includere i: valore ottimale per ultima attività non
sovrapposta p(i) + Vi
//- escludere i: valore ottimale di attività i-1
else return max{R-OTP(p(i)) + Vi, R-OPT(i-1)}
```

Algoritmo bottom-up

```
PD-OPT(n)
M[0]=0
for i = 1 to n
     M[i] = max{M[P(x)] + Vi, M[i-1]}
```

```
WIS(n)
OPT[0] = 0
S = vuoto
for i = 1 to n
     V1 = OPT[i-1]
     V2 = OPT[phi(i)] + vi
     OPT[i] = MAX(V1, V2)
     if V1 >= V2 then
          S[i] = S[i-1]
     else
          S[i] = S[phi(i)]∪{i}
return (OPT[n], S[n])
```

Stampa algoritmo:

```
PRINT-WIS(i)
if i ≠ 0
     if M[p(i)] + vi > M[i-1]
          //stampa attività non sovrapposte che sono state incluse
          PRINT-WIS(p(i), M)
          print(i)
     else
          PRINT-WIS(i-1)
```

Versione C++

Questo codice è stato fatto perché potrebbe essere più chiaro rispetto al pseudocodice. Da notare che il risultato non è giusto, ma potrebbe essere giusta in sé la logica.

```cpp
//struttura dell'evento
struct Event {
    int start; int end; int value;
    Event(int start, int end, int value) : start(start), end(end), value(value) {}
    Event() : start(0), end(0), value(0) {}

};

//aux - trova l'ultimo evento che non si sovrappone
int findLastNonConflicting(Event events[], int i){
    for (int j = i-1; j >= 0; j--){
        if(events[j].end <= events[i].start){
            return j;
        }
    }
    return -1;
}

//aux - confronta la fine degli eventi
bool compareEvents(const Event &a, const Event &b){
    return a.end < b.end;
}

long WIS(Event events[], int n){

    //ordinamento degli eventi in base al tempo di fine
    std::sort(events, events + n, compareEvents);

    long* dp = new long[n];
    dp[0] = events[0].value;

    for (int i = 1; i < n; i++){
        long includeValue = events[i].value;
        int lastIndex = findLastNonConflicting(events, i);

        if (lastIndex != -1)
            includeValue += dp[lastIndex];

        dp[i] = max(includeValue, dp[i-1]);
    }

    long result = dp[n-1];
    delete[]dp;
    return result;
}
```

Longest Increasing Subsequence

Data una sequenza X di m numeri, si determini la **lunghezza** di **una** tra le più lunghe sottosequenze crescenti di X.

Quello che si andrà a risolvere è una versione ridotta del problema PB, che è la seguente:

Data una sequenza X di m numeri interi, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti di X.

Il problema contiene diversi sottoproblemi ognuno dei quali non ha come input la sequenza X, ma un suo **prefisso**. Quindi, il sotto problema è definito come *data una sequenza X di n numeri, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti di X_i.*

Dato che $0 \leq i \leq n$ si ottengono (n+1) sottoproblemi. Ad ogni sottoproblema è associata una variabile $\mathbf{c_i}$, ovvero la lunghezza di una tra le più lunghe sottosequenze crescenti di X_i.

Dobbiamo introdurre un problema ausiliario perché non c'è alcun modo per poter comprendere se l'elemento x_i possa essere accodato alle sottosequenze crescenti relative ai sottoproblemi di dimensione minore a i.

Problema ausiliario

Dobbiamo introdurre un **problema ausiliario**: per ogni prefisso X_i della sequenza, con 1 e n, si calcola la **lunghezza** di una delle più lunghe sottosequenze crescenti che termina con l'elemento x_i. Serve poiché, per ogni prefisso, dobbiamo verificare quali degli elementi precedenti possono essere concatenati alla sottosequenza crescente.

Definiamo c_i come la *lunghezza della più lunga sottosequenza crescente di X_i che termina con x_i.*

Data una sequenza X di n numeri, si determini la lunghezza di una più lunga sottosequenza crescente di X la quale termina con x_n.

Ogni sottoproblema di dimensione i è associato alla variabile c_i (lunghezza di una più lunga sottosequenza crescente di X_i la quale termina con x_i).

Anche il problema ausiliario contiene m+1 diversi sottoproblemi, ognuno associato ad una diversa variabile. È definito come:

Data una sequenza X di m numeri interi, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti di X_i, e che termina con x_i.

Solo imponendo che la soluzione c_s si riferisca ad una sottosequenza crescente di X_s che termina con x_s è possibile stabilire se un altro elemento x_u, con $u > s$, possa essere accodato a tale sottosequenza.

Equazioni di ricorrenza

- **Caso base**: se $i = 1$ allora $c_1 = 1$ ($X_1 = \langle x_1 \rangle$)
    - Quando il prefisso considerato è la sequenza vuota, oppure è una sequenza composta da un singolo elemento

- Il caso base i = 0 risulta non necessario, in quanto è possibile utilizzare il caso i=1

- **Passo ricorsivo**: se i > 1 allora
  - $c_i = \max\{ c_j \mid 1 \leq j < i \wedge x_j < x_i \} + 1$
    - I dati disponibili per calcolarlo sono: input X (ed in particolare l'elemento x_i) e tutte le varibili $\{c_0, \ldots, c_{i-1}\}$.

Algoritmo in pseudocodice

Può essere implementato in modo efficiente con un approccio **bottom-up**, memorizzando i risultati parziali in un array C di dimensione n. Questo array contiene le *lunghezze delle sottosequenze crescenti per ciascun prefisso X_i*. Questo permette di risolvere il problema in $O(m^2)$ occupando $\Theta(m)$ spazio in memoria.

```
LIS(X, i)
if i = 1 return 1
else
     maxLength = 1
     for j = 1 to i – 1
          if X[j] < X[i]
               maxLength = max(maxLength, LIS(X, j) + 1)
     return maxLength
```

```
LIS(X)
     n = length(X)
     C[1] = 1
     for i = 2 to n
          tmp = 0
          for j = 1 to i-1
               if X[j] < X[i] and C[j] > tmp
                    tmp = C[j]
                    ind = j
          C[i] = tmp + 1
          B[i] = ind
          if C[i] > max
               max = C[i]
```

Stampa sottosequenza crescente

```
PRINT_LIS(i,b)
if b[i] ≠ 0
     PRINT_LIS(b[i], b)
print(X[i])
```

Versione C++

Questa versione del codice è stata fatta perché potrebbe essere più chiara da comprendere rispetto al pseudocodice.

```
int* LIS(char* x){

    int n = stringLength(x);
    int* c = new int[n]; //lunghezza della LIS
    int* b = new int[n]; //indice del predecessore | -1 se non ci sono
    int maxLength = 0;
    int endIndex = 0;
    int i = 0;


    //filling the two arrays
    for (i = 0; i < n; i++){
        //puts all 1s in the c array, all -1s in the b array
        c[i] = 1; b[i] = -1;
    }

    for (i = 1; i < n; i++){
        //confronto l'elemento corrente selezionato con il precedente
        for (int j = 0; j < i; j++){
            //se x[j] < x[i] (sequenza crescente quindi),
            //e la lunghezza della sequenza fino a j più l'elemento
            //corrente è maggiore di c[i], allora aggiorna c[i] e b[i]
            if (x[j] < x[i] && c[j] +1 > c[i]){
                c[i] = c[j] + 1;
                b[i] = j;
            }
        }

      b[6] = 4 (x[4] = 5)
      b[4] = 2 (x[2] = 4)
      b[3] = 0 (x[0] = 3)
      - va indietro in questo modo perchè sta evitando i -1 nell'array
      - i = b[1] -> i = -1 (esci)


   */
    for (i = endIndex; i >= 0; i=b[i]){
        result[k--] = x[i] - '0'; //conversione in intero dalla stringa iniziale
        //-1 = non ci sono più predecessori, si interrompe
        if (b[i] == -1) break;
    }

    for (i = 0; i < maxLength; i++){
        if(result[i] < 0) result[i] = 0;
    }

    result[maxLength] = -1;

    delete[] c;
    delete[] b;
    return result;
}
```

Longest Increasing Common Subsequence

È un'estensione del problema LIS, che cerca di determinare la più lunga sottosequenza crescente comune a due sequenze X_m e Y_n.

> Date due sequenze X e Y, rispettivamente di m ed n numeri interi, si determini una tra le più lunghe sottosequenze crescenti comuni a X e Y.

Quello che si risolve è una versione ridotta del problema, definita come:

> Date due sequenze X e Y, rispettivamente di m ed n numeri interi, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti comuni a X e Y.

Problema ausiliario

La versione ridotta contiene diversi sottoproblemi, ognuno dei quali non ha come input la coppia (X,Y), ma una coppia di prefissi di tali sequenze. Il sottoproblema è definito come:

> Date due sequenze X e Y, rispettivamente di m ed n numeri interi, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti comuni al prefisso X_i e al prefisso Y_j.

Si ottengono (m*1)+(n*1) sottoproblemi, e ad ogni sottoproblema è associata una variabile $c_{i,j}$ = lunghezza di una tra le più lunghe sottosequenze crescenti comuni a X_i e Y_i.

Date solamente le variabili $\{c_{0,0}, \ldots, c_{i-1,j}, c_{i,j-1}\}$, e i prefissi X_i e Y_j, non c'è alcun modo per **poter comprendere se gli elementi x_i e y_i, nel caso fossero uguali, possano essere accodati alle sottosequenze crescenti relative ai sottoproblemi di dimensione minore a (i,j)**.
Introduciamo quindi il problema ausiliario definito come:

> Date due sequenze X e Y, rispettivamente di m ed n numeri interi, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti comuni a X_i e a Y_j, e che termina con x_i e y_j.

Si definisce C[i,j] come la lunghezza della più lunga sottosequenza crescente comune che termina con x_i e y_j.

Equazioni di ricorrenza

- CASO BASE: Se $\mathbf{x_i} \neq \mathbf{y_j}$ allora, con i > 0 /\ j > 0
  - C[i,j] = 0

- CASO RICORSIVO: Se $x_i = y_j$ allora
  - $C[i,j] = \max\{C_{h,k} \mid 1 \leq h < i \,/\backslash\, 1 \leq k < j \,/\backslash\, x_h < x_i\} + 1$

Una volta calcolati i valori $c_{1,1}$, $c_{2,1}$, etc, si hanno a disposizione tutte le lunghezze delle sottosequenze comuni massimale fra qualche prefisso di X, e qualsiasi prefisso di Y che terminano con l'ultimo elemento di entrambi i prefissi.

Abbiamo quindi:

$c_{i,j} = 1 + \max\{c_{hk} \mid 1 <= h < i,\ 1 <= k < j,\ x_h < x_i\}$

Di seguito, un esempio:

		1	2	3	4	5	6	7	8	J
		2	7	4	23	21	14	1	8	y_j
1	2	1	0	0	0	0	0	0	0	
2	4	0	0	2	0	0	0	0	0	
3	7	0	2	0	0	0	0	0	0	
4	11	0	0	0	0	0	0	0	0	
5	21	0	0	0	0	3	0	0	0	
6	14	0	0	0	0	0	3	0	0	
7	1	0	0	0	0	0	0	1	0	
i	x_i									

Algoritmo in pseudocodice

Questo algoritmo è implementato con tecnica bottom-up, e permette di risolvere il problema on $O(m^2 * n^2)$ occupando $\Theta(m*n)$ spazio in memoria.

```
LICS(X, Y)
m = length(X)
n = length(Y)
C[0…m, 0…n] = 0 //inizializzazione
for i = 1 to m
     for j = 1 to n
          if X[i] = Y[j]
               max_length = 0
               for h = 1 to i-1
                    for k = 1 to j-1
                         if X[h] < X[i] and C[h, k] > max_length
                              max_length = C[h, k]
               C[i, j] = max_length + 1
          else
               C[i, j] = 0
max_LICS = 0
for i = 1 to m
     for j = 1 to n
          if C[i, j] > max_LICS
```

```
                max_LICS = C[i, j]
return max_LICS
```

Sottosequenza crescente massima (LIS con vincoli)

Per ogni istanza $X_i = \{x_1, \ldots, x_i\}$, l'obiettivo è **trovare la sottosequenza crescente massima che termina in x_i**.

Sia Z_i^* la sottosequenza crescente massima di X_i che termina con x_i. Abbiamo quindi:

- **Caso base**: $Z_1^* = \{x_1\}$
- **Formula ricorsiva**
  - $\forall\ i > 1$, $C_i = \max\{C_j \mid 1 \leq j < i \land x_j < x_i\}$
  - $Z_i = Z^*$ se questa sottosequenza termina in x_i.

Teorema

Sia Z_i la soluzione del i-esimo sottoproblema. Supponiamo che la tesi valga per tutti i $j = \{1, \ldots, i\text{-}1\}$. Sia Z_i la sottosequenza crescente massima che termina in x_i, tale che $Z_i = Z^* \cup \{x_i\}$.

Supponiamo per assurdo che Z^* non sia la soluzione del i-esimo sottoproblema. Esisterebbe allora un'altra sottosequenza Z', che termina anch'essa in x_i, tale che **$|Z'| > |Z|$**. Ciò implicherebbe che Z non è ottimale, il che è *una contraddizione*. Pertanto, Z^* deve essere la soluzione corretta.

Hateville (simile a WIS)

È un problema molto simile a WIS; consiste nel trovare il numero massimo di sotto-problemi tra due elementi, che seguono un pattern specifico.

Abbiamo un'istanza n, con $X = \{1, \ldots, n\}$ per ogni abitante i appartenente a X_n, dove la donazione di ciascun abitante deve essere *maggiore o uguale* di zero.

Il nostro obiettivo è **trovare l'insieme della donazione** che massimizza il valore complessivo, usando

$$D(S) = \max\{D(A), A \subseteq X_n\},\ \text{comp}(A) = \text{true}$$

Funzione di compatibilità

La funzione di compatibilità verifica *se un sottoinsieme A di X_n soddisfa le condizioni del problema*.

Formalmente, $\text{comp}(A)$ è vera se e solo se $\forall\ i \in A$, $i\text{-}1 \notin A$, e $i + 1 \notin A$.

In altre parole, **due abitanti consecutivi non possono far parte dello stesso sottoinsieme**.

Funzione delle donazioni

La funzione delle donazioni D(A), definita per ogni sottoinsieme $A \subseteq X_n$, assegna un valore positivo a ciascun insieme non vuoto, pari alla somma delle donazioni dei suoi elementi:

$D(A) = \sum d_i \quad$ se $A \neq \emptyset$

$D(\emptyset) = 0 \quad$ se $A = \emptyset$

Sottoproblema

Scomponiamo il problema principale in sottoproblemi. L'input del sottoproblema i-esimo è $X_i = \{1, \ldots, i\}$, che rappresenta **un'istanza del problema più grande**.

La soluzione del sottoproblema è un sottoinsieme $S_i \subseteq X_i$, che massimizza la somma delle donazioni, *rispettando la funzione di compatibilità*:

$$D(S_i) = \max\{D(A)\}, \text{ dove } A \subseteq X_i, \text{comp}(A) = \text{true}$$

Risoluzione

- **Caso base** i = 0
  - $X_0 = \emptyset$
  - $S_0 = \emptyset$ ➔ $OPT(0) = D(S_0) = 0$
- **Caso i = 1**
  - $X_1 = \{1\}$
  - $S_1 = \{1\}$, quindi $OPT(1) = d_1$

Passiamo adesso al **passo ricorsivo**; considerando il sottoproblema all'istante i, con input X_i, e supponendo di *conoscere le soluzioni ottimali dei sottoproblemi precedenti* (S(i) <-> OPT(i)).

A questo punto, si distingue tra due casi:

- Se $\mathbf{i \in S_i}$, allora $S_i = S(i\text{-}2) \cup \{i\}$
- Se $\mathbf{i \notin S_i}$, allora $S_i = S(i\text{-}1)$

Pertanto, la soluzione ottimale si può esprimere con:

$$OPT(i) = \max\{OPT(i - 2) + d_i, OPT(i - 1)\}$$

Algoritmo bottom-up

```
C[0, …, n] ; C[i] <- OPT(i)
PD_HV(n)
     C[0] = 0 //caso base
     C[1] = d1 //caso base
     for i = 2 to n
          C[i] = max{C[i-2] + di, C[i-1]}
```

Print hateville

```
PRINT_HV(n)
     if i = 1 print(1)
     else
          if C[i-2] + di ≥ c[i-1]
               PRINT_HV(i-2)
               PRINT(i)
          else
               PRINT_HV(i-1)
```

Dimostrazione della proprietà

Il teorema dimostra la struttura ricorsiva della soluzione ottimale per ogni sottoproblema i-esimo, con i ≥ 2. Si assume di conoscere le soluzioni dei problemi da 0 a i-1, indicate con S_0, …, S_{i-1}, e si vuole determinare S_i, la soluzione del problema corrente.

Il teorema afferma che:

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

Caso 1

Caso 1: $i \notin S_i$

Si dimostra che, **se i non appartiene a S_i, allora $S_i = S_{i-1}$**.

Supponiamo per assurdo che S_{i-1} non sia la soluzione ottimale per il problema i-esimo, ossia che esista un insieme $S' \neq S_{i-1}$ tale che $D(S') > D(S_{i-1})$.

Dato che i !∈ S_i, l'insieme S' è contenuto in {1, ..., i-1}, e inoltre S' è compatibile (ovvero rispetta la funzione *comp(S') = true*).

Tuttavia, se $D(S') > D(S_{i-1})$, ciò implica che S_{i-1} non sarebbe la soluzione ottimale per il problema con istanza (i-1)-esima, il che **contraddice** l'ipotesi che S_{i-1} sia la soluzione ottimale del sottoproblema (i-1)-esimo.

Questa contraddizione che, $\mathbf{se\ i\ !\in S_i\ allora\ S_i = S_{i-1}}$

Caso 2

Caso 2: $i \in S_i$

In questo caso, si dimostra che $\mathbf{S_i = S_{i-2} \cup \{i\}}$. Procediamo per assurdo, supponendo che $S_{i-2} \cup \{i\}$ non sia la soluzione ottimale del problema i-esimo.

Esista un insieme $S' \neq S_{i-2} \cup \{i\}$ tale che:

$$D(S'') > D(S_{i-2} \cup \{i\}).$$

Dato che S'' include i, possiamo scrivere $S' = S'' \cup \{i\}$, dove S'' è un sottoinsieme compatibile che rispetta la condizione *comp(S'') = true*.

Poiché *i-1 !∈ S''*, S'' è **compatibile**, e quindi $D(S'') > D(S_{i-2})$ porterebbe a una contraddizione, poiché **ciò implicherebbe che S_{i-2} non è la soluzione ottimale del sottoproblema (i-2)-esimo**.

L'ipotesi per assurdo è falsa, e possiamo concludere che $\mathbf{S_i = S_{i-2} \cup \{i\}}$.

Distanza di edit

Il problema del calcolo della distanza minima tra due sequenze $X = (x_1, ..., x_m)$, e $Y = (y_1, ..., y_n)$ si concentra nel determinare il *numero minimo di operazioni necessarie per trasformare la sequenza X nella sequenza Y*. Le operazioni ammesse sono:

- **Inserisci(a)**: inserisce l'elemento *a* nella posizione corrente di X
- **Cancella(a)**: elimina l'elemento *a* dalla posizione corrente di X
- **Sostituisci(a,b)**: sostituisce l'elemento *a* con l'elemento *b* nella posizione corrente di X

Sottoproblema

Il sottoproblema è definito come $S(X_i, Y_j)$, dove $0 \leq i \leq m$, e $0 \leq j \leq n$, ed è il **numero minimo di operazioni per trasformare i primi i-esimi caratteri di X, nei primi j-esimi caratteri di Y (quindi di trasformare X_i in Y_j)**.

Si ottengono **(m*1)(n*1) sottoproblemi**, e a ciascuno di essi è associata una variabile

$S_{i,j}$ = numero minimo di operazioni elementari che permette di trasformare X_i in Y_j

Ognuna di queste variabili è da considerare come una *black-box*.

Caso base

- Se $i = 0 \wedge j = 0$: entrambi i prefissi sono la sequenza vuota, allora $S_{i,j} = 0$
- Se $i = 0 \wedge j > 0$: ci vuole un numero di operazioni elementari pari alla *lunghezza del prefisso Y_j*, quindi $S_{i,j} = j$
- Se $j = 0 \wedge i > 0$: ci vuole un numero di operazioni elementari pari alla *lunghezza del prefisso X_i*, quindi $S_{i,j} = i$

Si può riassumere come:

$$S_{i,j} = \begin{cases} 0 & \text{se } i = 0 \wedge j = 0 \\ j & \text{se } i = 0 \wedge j > 0 \\ i & \text{se } i > 0 \wedge j = 0 \end{cases}$$

Passo ricorsivo

- **Caso 1**: se $x_i = y_j$, non è richiesta alcuna operazione, e la distanza rimane invariata rispetto a quella dei sottoproblemi precedenti:
  - $S(X_i, Y_j) = S(X_{i-1}, Y_{j-1})$
- **Caso 2**: se $x_i \neq y_j$, si possono applicare una di queste operazioni:
  - **Inserisci y_j**: la distanza è quella di $S(X_i, Y_{j-1})$ più 1, dato che abbiamo inserito un nuovo carattere
    - $S(X_i, Y_j) = S(X_i, Y_{j-1}) + 1$
      - 1 rappresenta il contributo dell'operazione $\mathbf{insert(y_i)}$
  - **Cancella x_i**: la distanza è quella di $S(X_{i-1}, Y_j)$ più 1, poichè un carattere è stato eliminato
    - $S(X_i, Y_j) = S(X_{i-1}, Y_j) + 1$
      - 1 rappresenta il contributo dato dall'operazione $\mathbf{delete(x_i)}$
  - **Sostituisci x_i con y_j**: la distanza sarà quella di $S(X_{i-1}, Y_{j-1})$ più 1, dato che un carattere è stato sostituito
    - $S(X_i, Y_j) = S(X_{i-1}, Y_{j-1}) + 1$

- 1 rappresenta il contributo dato dall'operazione **replace(x_i, y_j)**

- La soluzione ottimale è
  - **$S(X_i, Y_j) = \min\{S\{X_i, Y_{j-1}) + 1, S(X_{i-1}, Y_j) + 1, S(X_{i-1}, Y_{j-1}) + 1\}$**

Equazione di ricorrenza

La distanza ottimale $S(x_i, y_j)$ è data da:

- $S(X_{i-1}, Y_{j-1})$ se $x_i = y_j$
- $\min\{S(X_i, Y_{j-1}) + 1, S(X_{i-1}, Y_j) + 1, S(X_{i-1}, Y_{j-1}) + 1\}$ se $x_i \neq y_j$

Algoritmo bottom-up

		0	1	2	3	4	5	6	J
		E	P	R	E	S	T	O	y_j
0	E	0	1	2	3	4	5	6	
1	R	1	1	1	2	3	4	5	
2	I	2	2	2	2	3	4	5	
3	S	3	3	3	3	2	3	4	
4	O	4	4	4	4	3	3	3	
5	T	5	5	5	5	4	3	4	
6	T	6	6	6	6	5	4	4	
7	O	7	7	7	7	6	5	4	
i	x_i								

1. Si **inizializza** il bordo della matrice: d[i,0] = i, d[0,j] = j, ∀ I e ∀ j
2. Si riempie passo per passo, si fa solo con la prima lettera di RISOTTO
   a. d[1,1]: confronta R con P. Sono diversi, quindi si **sostituisce** con la formula $\min\{d[i,j-1], d[i-1,j], d[i-1,j-1]\}$
      i. d[1,1] = min(d[1,0], d[0,1], d[0,0]) + 1 = min(1,1,0) + 1 = 1
   b. d[1,2]: confronta R con PR. R **corrisponde** a R, quindi non si esegue nessuna operazione
      i. d[1,2] = d[0,1] = 1
   c. d[1,3]: confrontiamo R con PRE, sono diversi
      i. d[1,3] = min(d[3,0],d[2,1],d[0,2])+1 = min(3,1,2)+1 = 2
   d. Si continua per il resto della riga
   e. d[2,1]: confronta I con P; sono diversi, quindi:
      i. d[2,1] = min(d[1,1], d[2,0],d[1,0])+1 = min(1,2,1)+1 = 2

DA CONTROLLARE indici – sono corretti, ma ChatGPT li scrive all'inverso del prof

```
EDIT(X,Y)
m = length(X), n = length(Y)
```

```
for j = 0 to n
     d[0, j] = j
for i = 1 to m
     d[i, 0] = i
for i = 1 to m
     for j = 1 to n
          If Xi = Yj
               d[i,j] = d[i-1, j-1]
          else
               d[i,j] = min{d[i, j-1], d[i-1, j], d[i-1, j-1]} + 1
```

Algoritmo ricorsivo

```
ED_RIC(i,j)
if i = 0 OR j = 0
     if i = 0 return j
     else return i
else
     if xi = yi
          return ED_RIC(i-1, j-1)
     else
          ins = 1 + ED_RIC(i, j-1)
          del = 1 + ED_RIC(i-1, j)
          rep = 1 + ED_RIC(i-1, j-1)
          return MIN(ins, del, rep)
```

Interleaving di due stringhe

L'**interleaving** di due stringhe X e Y è una nuova stringa Z che contiene tutti i caratteri di X e Y, mantenendo l'ordine relativo dei caratteri di entrambe.

Ad esempio, se X = "abc" e Y = "def", una possibile interleaving potrebbe essere *"adbcef"*. È un problema di combinatoria sulle stringhe, risolto con la programmazione dinamica.

Date tre sequenze, definite sull'alfabeto $\sum$, $X = (x_1, ..., x_m)$ di lunghezza m, $Y = (y_1, ..., y_n)$ di lunghezza n, $W = (w_1, ..., w_{m+n})$ di lunghezza m+n, stabilire se **W è un interleaving di X e Y**, ovvero se X e Y si possono trovare come due sottosequenze disgiunte in W.

Sottoproblema

Il sottoproblema di dimensione i,j è definito come:

Date tre sequenze X, Y e W, rispettivamente di lunghezza m, n e m+n, si determini se W_{i+j} è interleaving di X_i e Y_j.

Si ottengono (m+1)*(n+1) sottoproblemi, a cui ciascuno è associata una variabile. Indichiamo $S_{i,j}$ = true se **la stringa W_{i+j} è un interleaving di X_i e Y_j**.

Equazioni di ricorrenza

Caso base

$$S_{i,j} = \begin{cases} \text{true} & \text{se } i = 0 \wedge j = 0 \\ s_{i,j-1} & \text{se } i = 0 \wedge j > 0 \wedge w_j = y_j \\ \text{false} & \text{se } i = 0 \wedge j > 0 \wedge w_j \neq y_j \\ s_{i-1,j} & \text{se } i > 0 \wedge j = 0 \wedge w_i = x_i \\ \text{false} & \text{se } i > 0 \wedge j = 0 \wedge w_i \neq x_i \end{cases}$$

Passo ricorsivo

$$s_{i,j} = \begin{cases} s_{i,j-1} & \text{se } i = 0 \wedge j > 0 \wedge w_j = y_j \\ s_{i-1,j} & \text{se } i > 0 \wedge j = 0 \wedge w_i = x_i \\ s_{i-1,j} & \text{se } w_{i+j} = x_i \wedge w_{i+j} \neq y_j \\ s_{i,j-1} & \text{se } w_{i+j} \neq x_i \wedge w_{i+j} = y_j \\ s_{i-1,j} \vee s_{i,j-1} & \text{se } w_{i+j} = x_i \wedge w_{i+j} = y_j \end{cases}$$

Soluzione del problema

		0	1	2	3	4	5	J
		E	M	A	M	M	A	y_j
0	E	T	F	F	F	F	F	
1	C	T	F	F	F	F	F	
2	I	T	T	T	F	F	F	
3	A	F	T	T	T	T	T	
4	O	F	T	F	F	F	T	
i	x_i							

i+j	1	2	3	4	5	6	7	8	9

W_{i+j}	C	I	M	A	A	M	M	A	O

Algoritmo ricorsivo

```
INT(i,j)
if i = 0 /\ j = 0 return true
if w_{i+j} ≠ x_i /\ w_{i+j} ≠ y_j return false
if i = 0 /\ j > 0
     if w_j = y_j
          return INT(i, j-1)
     else
          return false
if i > 0 /\ j = 0
     if w_i = x_i return INT(i-1, j)
     else return false
if w_{i+j} = x_i /\ w_{i+j} ≠ y_j
     return INT(i-1,j)
if wi+j ≠ x_i /\ w_{i+j} = y_j
     return INT(i,j-1)
if wi+j = xi /\ wi+j = yj
     return INT(i-1, j) \/ INT(i, j-1)
```

Algoritmo bottom-up

Questo algoritmo occupa, come tempo, O(m*n) occupando Θ(m*n) spazio in memoria.

```
INT(X,Y,Z)
s[0,0] = true
for i = 1 to m
     if w_i = x_i
          s[i,0] = s[i-1, 0]
     else
          s[i,0] = false
for j = 1 to n
     if w_j = y_j
          s[0,j] = s[0, j - 1]
     else
          s[0,j] = false
for i = 1 to m
     for j = 1 to n
          if w_{i+j} ≠ x_i /\ w_{i+j} ≠ y_j
               s[i,j] = false
          if w_{i+j} = x_i /\ w_{i+j} ≠ y_j
               s[i,j] = s[i-1, j]
```

```
            if w_{i+j} ≠ x_i /\ w_{i+j} = y_j
                 s[i,j] = s[i, j-1]
            if w_{i+j} = x_i /\ w_{i+j} = y_j
                 s[i,j] = s[i-1, j] \/ s[i, j-1]
return s[m,n]
```

Stringa palindroma

Trovare il **numero minimo di caratteri** da aggiungere a una stringa S affinché diventi un **palindromo**. Il problema si suddivide in sottoproblemi.

Si introduce la funzione f : ∑* —> N: ∀ S ∈ ∑*, f(S) = n. minimo di caratteri da aggiungere per rendere S palindroma.

Sia S = (s_1, s_2, ..., s_n) una stringa di larghezza n definita su un alfabeto ∑.

Sottoproblema

Si definisce S_{ij} = (s_1, ..., s_j) la sottostringa della stringa S ottenuta da S, considerando tutti i simboli di S *dalla posizione i alla posizione j*.

Dato che $1 \leq i \leq n$, $1 \leq j \leq n$, si ottengono **n * n sottoproblemi**, e ad ognuno di questi si associa una variabile.

$m_{i,j}$ = numero minimo di caratteri da aggiungere alla sottostringa $S_{i,j}$ per renderla palindroma

Si possono presentare vari casi:

- S = E, S è palindroma, f(S) = 0
- S = a, ∀ A ∈ ∑, S è palindroma, f(S) = 0
- |S| ≥ 2, allora S = aS'b con: a il primo carattere di S, b l'ultimo carattere di S, S' sottostringa di carattere
    - a = b, S è palindroma se lo è anche S
        - f(S) = f(S')
    - a ≠ b:
        - S —> a[[S'b]]a, f(S) = 1 + f(S'b)
        - S —> b[[aS']]b, f(S) = 1 + f(aS')
        - **Riepilogo**: $1 + \min\{f(S'b), f(aS')\}$

Equazioni di ricorrenza

Caso base

Con i ≥ j.

- i = j: $S_{i,j}$ è composta da un solo carattere, quindi è palindroma; $m_{i,j} = 0$
- i > j: $S_{i,j}$ è vuota, quindi è palindroma: $m_{i,j} = 0$

Passo ricorsivo

Con i < j

- $m_{i,j} = \begin{cases} m_{i+1,j-1} & \text{se } i < j \ /\backslash \ s_i = s_j \\ 1 + \min\{m_{i-1,j}, m_{i,j-1}\} & \text{se } i < j \ /\backslash \ s_i \neq s_j \end{cases}$

Soluzione con matrice

S = C A S A

		C	A	S	A	
		1	2	3	4	j
C	1	0	1	2	1	
A	2	0	0	1	0	
S	3	0	0	0	1	
A	4	0	0	0	0	
	i					

- ogni cella sulla diagonale rappresenta una singola lettera **ed è sempre 0** perché ogni singola lettera è di per sé un palindromo
- Celle (i,j) per i = i+1 (sottostringhe di lunghezza 2)
  - Per “ca”, c ed a non sono uguali, quindi **la cella (1,2) contiene 1**
- Celle (i,j) per i = i+2 (sottostringhe da lunghezza 3)
  - Per “cas”, servono almeno due operazioni per renderla palindromica, quindi **la cella (1,3) contiene 2**

Algoritmo ricorsivo

```
PAL_RIC(i,j)
if i ≥ j return 0
else
     if si = sj return PAL_RIC(i+1, j-1)
     else
          s1 = LCS_RIC(i, j-1)
          s2 = LCS_RIC(i+1, j)
          return MIN(s1, s2)
```

Algoritmo bottom-up

Impiega O(n*n) occupando Θ(n*n) spazio in memoria.

```
LCS(S)
n = |S|
for j=1 to n
     for i=j to n
          m_{i,j} = 0
for i=n-1 down to 1
     for j = i+1 to n
          if s_i = s_j
               m_{i,j} = mi+1,j-1
          else
               m_{i,j} = 1 + min{m_{i+1,j}, m_{i,j-1}}
return c[1,n]
```

Commentato [f4]: @f.dongiovanni1@campus.unimib.it
Domanda: è giusto che sia i+1,j-1; o deve essere i-1,j-1?

Knapsack problem

È un problema di ottimizzazione combinatoria, in cui si cerca di selezionare un sottoinsieme di oggetti da un insieme dato, ciascuno con un valore e un peso, in modo da massimizzare il valore totale, senza superare un limite di capacità prestabilito.

Sia C > 0 la capacità di uno zaino, e X un insieme di *n* oggetti, dove: v_i indica il valore dell'oggetto, w_i indica il peso dell'oggetto.

Si introduce la funzione V: P({1, ..., n}) —> R+, che associa ad ogni sottoinsieme di oggetti A il suo valore complessivo.

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

Si introduce la funzione W: P({1, ..., n}) —> R+, che associa ad ogni sottoinsieme di oggetti A il suo ingombro complessivo.

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

Istanza

Un insieme di oggetti X = {1,2, ..., n}, dove ogni oggetto i ha:

- Un valore $v_i \in N$, utilità o guadagno
- Un peso $w_i \in N$, costo

Bisogna scegliere un sottoinsieme di oggetti tale che il peso totale non superi C. Bisogna quindi *massimizzare il valore totale V(S) del sottoinsieme $S \subseteq X$, soggetto al vincolo di capacità $W(S) \leq C$.*

Abbiamo:

- **Istanza**: $X_i = \{1, ..., i\}$ con $(v_j, w_j)\ \forall\ j \in \{1, ..., i\}$
- **Soluzione**: $S_i \subseteq \{1, ..., i\}$ tale che:
  - $W(S_i) \leq C\ /\backslash\ V(S_i) = \max(A \subseteq \{1, ..., i\} : W(A) \leq C)\ \{V(A)\}$

Si ottengono n+1 sottoproblemi, ad ognuno dei quali è associata una **coppia di variabili**:

- **OPT_i** = $V(S_i)$: valore di un sottoinsieme di {1, ..., i} di ingombro complessivo ≤ C, e di massimo valore complessivo
- **S_i**: sottoinsieme di {1, ..., i} di ingombro complessivo ≤ C, e di massimo valore complessivo pari a OPT_i.

Sottoproblema

$OPT_{(i, c)}$ come il valore massimo, ottenibile usando solo i primi i oggetti, e con una capacità massima residua c.

Il sottoproblema è definito come:

Dato un insieme i di oggetti, trovare un suo sottoinsieme di ingombro complessivo ≤ C, e di massimo valore complessivo

- **$OPT_{i,c} = V(S_{i,c})$**: valore di un sottoinsieme di {1, ..., i} di ingombro complessivo ≤ c, e di massimo valore complessivo
- **$S_{i,c}$**: sottoinsieme di {1, ..., i} di ingombro complessivo ≤ c, e di massimo valore complessivo pari a $OPT_{i,c}$

Sottostruttura ottima

Sia X un insieme di oggetti, W il peso massimo consentito per lo zaino, $V: P(X) \to R$ la funzione che indica il valore corrispondente ad un sottoinsieme di oggetti $A \subseteq X$.

Consideriamo $c_{i,h}$ il peso per il sottoproblema i,h Knapsack su $A \subseteq X = \{x_1, ..., x_i\}$, con $0 \leq i \leq n$, $0 \leq h \leq W$. Sia $Z_{i,h} = \{z_1, ..., z_h\}$ la sua soluzione. Abbiamo quindi:

- $w_i > h$: non possiamo prendere l'oggetto x_i, di conseguenza $x_i \notin Z_{i,h}$; abbiamo che $Z_{i,h} = Z_{i-1,h}$
- $w_i \leq h \wedge x_i \in Z_{i,h}$, l'elemento x_i fa parte della soluzione, quindi $Z_{i,h} = Z_{i-1,\, h-wi} \cup \{x_i\}$
- $w_i \leq h \wedge x_i \notin Z_{i,h}$, l'elemento x_i non fa parte della soluzione, quindi $Z_{i,h} = Z_{i-1,\, h}$

Equazioni di ricorrenza

Caso base

(i,c) con i = 0 e ∀ c; X_0; $S_{i,c} = 0 \longrightarrow OPT_{i,c} = 0$

(i,c) con c = 0 e ∀ i; X_i; $S_{i,c} \subseteq X_i \longrightarrow OPT_{i,c} = 0$

Passo ricorsivo

i > 0 /\ c > 0, $X_i = \{1, \ldots, i\}$

- Se $w_i > c$ (ingombro > capacità), $i \notin S_{i,c}$
  - X_{i-1}; $S_{i,c} = S_{i-1,\, c} \longrightarrow OPT_{i,c} = OPT_{i-1,c}$
- Se $w_i \leq c$
  - $S_{ic} = \begin{cases} S_{i-1,c-wi} \cup \{i\} & \text{se } i \in S_{ic} \longrightarrow OPT_{i-1,c-wi} + V_i \geq OPT_{i-1,c} \\ S_{i-1,c} & \text{se } i \notin S_{ic} \longrightarrow OPT_{i-1,c-wi} + V_i < OPT_{i-1,c} \end{cases}$

Il risultato è quindi $\mathbf{OPT_{i,c} = max\{OPT_{i-1,c-wi} + V_i,\, OPT_{i-1,c}\}}$.

Al massimo 3 oggetti rossi nello zaino

Vogliamo avere almeno tre oggetti rossi nello zaino; ha $X_n = \{1, \ldots, n\}$ ∀ $V_i \in N_0$, $W_i \in N_0$.

Gli input dell'algoritmo sono X_n, C (capacità), r (numero di elementi rossi) ∈ {0, 1, 2, 3}.

Caso base

Dobbiamo vedere i casi base, vedendo quando una delle variabili vale zero, mentre tutte le altre variabili hanno un qualsiasi valore.

i = 0, ∀ C, ∀ r
- $S_{i,c,r} = \emptyset \longrightarrow OPT_{i,c,r} = \emptyset$

- ∀ i, c = 0, ∀ r
  - $S_{i,c,r} = \emptyset \longrightarrow OPT_{i,c,r} = \emptyset$
- ∀ i, ∀ c, r = 0
  - $S_{i,c,r} = \emptyset \longrightarrow OPT_{i,c,r} = \emptyset$

Passo ricorsivo

Il passo ricorsivo vale per ogni valore di i,c, r, $X_i = \{1, \dots, i\}$. Bisogna dividerlo in due casi:

- Se $W_i > c$, $i \notin S_{i,c}$
  - X_{i-1}, $S_{i,c} = S_{i-1,c} \Rightarrow OPT_{i,c} = OPT_{i-1,c}$
- Se $W_i \leq c$
  - $$S_{i,c,r} = \begin{cases} S_{i-1,c-wi,\,r} \cup \{i\} & \text{se } i \in S_{ic} \wedge \text{col}(x_1) = \text{blu} \Rightarrow OPT_{i-1,c-wi,r} + V_i \geq OPT_{i-1,c} \\ S_{i-1,c-wi,r-1} \cup \{i\} & \text{se } i \in S_{ic} \wedge \text{col}(x_1) = \text{red} \Rightarrow OPT_{i-1,c-wi,r-1} + V_i \geq OPT_{i-1,c} \\ S_{i-1,c,r} & \text{se } i \notin S_{ic} \Rightarrow OPT_{i-1,c-wi,r} + V_i < OPT_{i-1,c} \end{cases}$$

La soluzione è quindi $OPT_{icr} = \max\{OPT_{i-1,c-wi,r-1} + v_i, OPT_{i-1,c,r}\}$.

Knapsack a coppie

L'obiettivo è avere due stringhe X_n e Y_m, tali che **nelle coppie di S non ci sono elementi ripetuti né di X_n né di Y_m**, mantenendo ovviamente il valore della capacità.

Istanza

$X_n = \{x_1, \dots, x_m\}$, $Y_n = \{y_1, \dots, y_n\}$, $c \in N+$, con la funzione w su X_n, e v su Y_m.

Algoritmo ricorsivo

		C	A	S	A	
		1	2	3	4	j
C	1	0	1	2	1	
A	2	0	0	1	0	
S	3	0	0	0	1	
A	4	0	0	0	0	
	i					

```
KP_RIC(i, c)
if i = 0 \/ c = 0
     return (0, vuoto)
else
     if wᵢ > c
          return KP_RIC(i-1, c)
     else
```

```
        (V1, S1) = KP_RIC(i-1, c)
        (V2, S2) = KP_RIC(i-1, c-wi)
        V2 = V2 + vi
        If V1 >= V2
             return (V1, S1)
        else
             return (V2, S2∪{i})
```

Algoritmo iterativo

Si utilizza la tecnica bottom-up, permette di risolvere il problema in O(n * C) occupando Θ(n * C + n^2 * C) spazio.

```
KP(n,C)
for i = 0 to n
     OPT[i,0] = 0
     S[i,0] = vuoto
for c = 0 to C
     OPT[0,c] = 0
     S[0,c] = vuoto
for i = 1 to n
     for c = 1 to C
          if wᵢ > c
               OPT[i,c] = OPT[i-1, c]
               S[i,c] = S[i-1, c]
          else
               V₁ = OPT[i-1, c]
               V₂ = OPT[i-1, c-wᵢ] + vᵢ
               OPT[i,c]= MAX(V₁, V₂)
               if V₁ >= V₂
                    S[i,c] = S[i-1, c]
               else
                    S[i,c] = S[i-1, c-wᵢ]∪{i}
     return (OPT[n,C], S[n,C])
```

Algoritmo di stampa

```
PRINT(i,c,r)
if i > 0 /\ c > 0
     if wᵢ > c
          PRINT(i-1,c,r)
     else
          if col(i) = red /\ r > 0
               if OPT_{i-1,c-wi,r-1} + vᵢ ≥ OPT_{i-1,c,r}
                    PRINT(i-1,c-wi,r-1)
```

```
            print(i)
        else
            PRINT(i-1,c,r)
    if col(i) = red /\ r = 0
        PRINT(i-1, c, r)
        print(i)
    else
        PRINT(i-1,c,r)
```

Subset sum

Richiede di determinare **se esiste un sottoinsieme di un insieme dato di numeri che sommi a un valore target k.**

Istanze del problema

- **Input**
  - Un insieme $X_n = \{x_1, x_2, ..., x_n\}$ di numeri interi non negativi
  - Un intero target k
- **Output**
  - Verifica se esiste un sottoinsieme di X_n tale che la somma degli elementi del sottoinsieme sia uguale a k

Sottoproblema

L'obiettivo è verificare, per ogni valore parziale di somma *k*, se può essere raggiunto usando i primi *i* elementi dell'insieme *X*.

Si utilizzano variabili boolean per indicare se è possibile ottenere una determinata somma usando un sottoinsieme degli elementi finora considerati.

Istanza

$X_i = \{x_1, ..., x_i\}, k \in \{0, ..., k\}, i \in \{0, ..., n\}$

Equazioni di ricorrenza

Caso base

Con $i = 0$, $x_0 = \emptyset$, $k \in \{0, ..., K\}$

- $e_{i,k}$ = true se k = 0 (sempre possibile una somma pari a 0 usando un sottoinsieme vuoto)
- $e_{0,k}$ = false se k > 0

Passo ricorsivo

Con i > 0

- Se $X_i > k$: $e_{i,k} = e_{i-1,k}$
- Se $X_i \leq k$
  - Se X_i è utilizzato se $X_i \in Y$ => $e_{i,k} = e_{i-1,k-xi}$
  - Se X_i non è utilizzato se $X_i \notin Y$ => $e_{i,k} = e_{i-1,k}$

Algoritmo di ricostruzione

```
RICOS(i,k,E)
if e_ik = false
     print ("non esiste")
else
     if i > 0 /\ k > 0
          if x_i > k RICOS(i-1,k,e)
     else
          if e_i-1, k-x_i = true
               RICOS(i-1, k - x_i)
               print(x_i)
          else
               RICOS(i-1, k, e)
```

Grafi

I grafi sono costituiti da un **insieme di nodi o vertici** V, rappresentati come punti/cerchi, e un **insieme di archi** E, che collegano i nodi e rappresentano le relazioni tra di essi.

Quindi, un grafo G è una coppia G = (V,E) dove: V = $\{v_1, \ldots, v_n\}$ è l'insieme dei vertici, $E \in \subseteq V \times V$ è l'insieme degli archi.

Possono essere come tipi:

- **Grafo non orientati**: gli archi *non hanno una direzione*
- **Grafo orientato**: gli archi *hanno una direzione*; esiste un arco da u a v, ma non necessariamente al contrario
- **Grafo pesato**: ad ogni arco è associato un *peso o costo*; è comune nei problemi di cammini minimi

Per calcolare i cammini minimi, si usano due algoritmi, quello di *Floyd-Warshall*, e di *Djikstra*.

Categorie

Questa parte è stata presa dal MEGA, e dovrebbe essere un riassunto dei vari tipi di grafi:

- **DAG – grafo aciclico orientato**: grafo diretto che non ha cicli diretti, dove comune scegliamo un vertice del grafo, non possiamo tornare ad esso percorrendo gli archi del grafo. Si può dire **aciclico** se una visita in profondità non presenta archi all'indietro.
- **Albero libero**: Sia G = (V,E) un grafo orientato; allora si definisce G albero libero se:
  - Il numero di archi è uguale al numero di vertici meno uno
  - Il grafo è connesso
- **Grafo connesso**: un grafo orientato si dice *connesso* se per ogni coppia di vertici v e w, esiste almeno un cammino non diretto che collega v e w
- **Albero orientato**: Sia T un grafo orientato; allora T è un albero orientato se *T è connesso*, *T non contiene cicli non orientati*, *esiste un unico nodo n tale che il numero degli archi entranti in n è pari a 0*, e *per ogni altro nodo del grafo diverso da n si ha che il numero degli archi entranti in n è 1*.
- **Albero binario**: albero in cui i nodi hanno grado compreso tra 0 e 2.
  - Con albero si intende un *grafo non diretto, connesso ed aciclico*
  - Con grado di un nodo si intende il *numero di sotto alberi del nodo*, che è uguale al numero di figli del nodo

Floyd-Warshall

È applicato su un **grafo senza cappi per trovare i cammini minimi** tra tutte le coppie di vertici. Si imposta +∞ come valore di distanza quando non esiste un cammino tra due vertici. L'obiettivo è di riempire la matrice dalle distanze minime tra ogni coppia di vertici.

Istanze

- **V**: insieme dei vertici con |V| = n
- **E**: insieme degli archi
- **W**: funzione peso, W: E -> R, assegnando un *valore positivo o negativo a ciascun arco*
  - **Matrice W** è una matrice n x n che rappresenta i pesi dei cammini minimi

$$W_{i,j} = \begin{cases} \emptyset & \text{se } i = j \\ W(i,j) & \text{se } i \neq j \wedge (i,j) \in E \\ \infty & \text{se } i \neq j \wedge (i,j) \notin E \end{cases}$$

I vertici intermedi sono appartenenti all'insieme {1, ..., k}

Possono esistere dei *cicli*, ma non cappi; i pesi invece possono essere **positivi o negativi**.

Soluzione

$\forall$ i,j $\in V^2$, il peso di un cammino minimo dal vertice al vertice j. Si calcolano n^2 valori, uno per ciascuna coppia, $\forall (i,j) \in V^2$.

Si trova quindi $D = (d_{ij})_{(i,j) \in V^2}$

Approccio top-down

Si definisce V(i,j,S) come il **peso del cammino minimo da i a j**, con vertici intermedi appartenenti all'insieme S.

Si creano sottoinsiemi progressivi, dove si calcola per i vertici intermedi {1, ..., v-1}, scendendo progressivamente nell'insieme dei vertici intermedi. Il caso base si ha quando *S=Ø*, cioè senza vertici intermedi, e si usa direttamente il **peso w(i,j)**.

Sottoproblema

Per tutte le coppie di vertici, calcolare il peso di un cammino minimo da i a j, con vertici intermedi che appartengono a {1, ...,k}, k $\in$ {0, ..., n}.

Caso base

{1, ..., k}, k = 0, Ø

$$D_0 = (d_{i,j}^{0}) = W_{i,j} = \begin{cases} 0 & \text{se } i = j \\ W(i,j) & \text{se } i \neq j \ /\backslash \ (i,j) \in E \text{ (esiste un cammino)} \\ +\infty & \text{se non esiste un cammino} \end{cases}$$

Passo ricorsivo

Si assume di aver già calcolato $D^0 = W$, $D^{(1)}$, ..., $D^{(k-1)}$.

$\forall$ i, j appartenente a V^2, troviamo $d_{i,j}^{k}$, e D^k.

Se k non è un vertice intermedio nel cammino da i a j:

- $d_{i,j}^{k} = d_{i,j}^{k-1}$

- $d_{i,j}^k$: i -> j è decomponibile in **2 sottocomuni**: i -> k, k -> j
  - $d_{i,j}^k = d_{i,k}^{k-1} + d_{k,j}^{k-1}$
  - $d_{i,j}^k = \min\{d_{i,j}^{k-1}, d_{i,k}^{k-1}, d_{k,j}^{k-1}\}$

Se k è invece un vertice intermedio, allora si ha $d_{ij}^k = \min\{d_{ij}^{k-1}, d_{ik}^{k-1}, d_{kj}^{k-1}\}$.

Pseudocodice

```
F-W(W,n)
D(0) = W
for k = 1 to n
     for i = 1 to n
          for j = 1 to n
               d_ij^k = min{d_ij^k-1, d_ik^k-1, d_kj^k-1}
return D(n)
```

Algoritmo di stampa

In $D^{(k)}$ abbiamo solo i pesi minimi, e la matrice π ha tutti i predecessori; infatti, π_{ij} è il predecessore di j in un cammino minimo da i a j.

```
PRINT-PATH(π, i, j)
if i = j print i
else if π_ij ≠ NULL
     PRINT-PATH(π, i, π_ij)
     print j
else print “no cammino da i a j”
```

Calcolo dei predecessori

L’obiettivo è di identificare il **predecessore** di un vertice j per ogni vertice i, in un cammino minimo tra i e j. Si usa una matrice π dei predecessori, con vertici intermedi.

Definiamo ∀ i,j ∈V2 => π_{ijk}

Caso base

Il caso base è senza vertici intermedi.

$$\Pi_{ij}^0 = \begin{cases} \text{NULL} & \text{se } i = j \vee (i,j) \notin E \\ i & \text{se } i \neq j \wedge (i,j) \in E \end{cases}$$

Passo Ricorsivo

Con k > 0, $\pi^k = (\pi_{ij}^{k-1})$ (i,j) ∈ V, dove *k non è un vertice intermedio nel cammino minimo da i a j*.

Se invece *k è un vertice intermedio in un cammino minimo da i a j,* allora:

$$\Pi_{ij}^{k} = \begin{cases} \pi_{ij}^{k-1} & \text{se } d_{ij}^{k-1} < d_{ik}^{k-1} + d_{kj}^{k-1}, \text{ k non è vertice intermedio} \\ \Pi_{kj}^{k-1} & \text{se } d_{ij}^{k-1} > d_{ik}^{k-1} + d_{kj}^{k-1}, \text{ k è vertice intermedio} \end{cases}$$

Varianti di problemi

Non due vertici consecutivi di colore uguale

Dato un grafo $G = (V, E, W, col)$, orientato e pesato, con:

- V: insieme dei vertici
- E: insieme degli archi
- W: peso degli archi
- $col : V \rightarrow \{rosso, blu\}$: funzione che associa ad ogni vertice un colore

L'obiettivo è trovare un cammino minimo tra i vertici i e j, in cui non ci sono due vertici consecutivi dello stesso colore.

La soluzione finale è data da D_n, dove n è il numero totale di vertici.

Sottoproblema

Definiamo $D^k = d_{ij}^k$, dove d_{ij}^k è il peso del cammino minimo tra i e j, utilizzando solo vertici intermedi tra $\{1, ..., k\}$.

Caso base

Quando non ci sono vertici intermedi, ovvero k = 0:

$$d_{ij}^{0} = \begin{cases} 0 & \text{se } i = j \\ w_{ij} & \text{se } (i,j) \in E, col(i) \neq col(j) \\ +\infty & \text{altrimenti} \end{cases}$$

Passo ricorsivo

Se k non è un vertice intermedio, allora $d_{ij}^k = d_{ij}^{k-1}$.

Altrimenti, se k è un vertice intermedio, allora $d_{ij}^k = \min(d_{ij}^{k-1}, d_{ik}^{k-1}, d_{kj}^{k-1})$.

Pseudocodice

Il codice non è stato fatto a lezione, ma è stato fatto da ChatGPT.

```
MinCammino(G)
for i,j in V x V
     if i = j them
          d[0][i][j] = 0
     else if (i,j) ∈E /\ col(i) ≠ col(j)
          d[0][i][j] = w[i][j]
     else
          d[0][i][j] = +∞

for int k = 1 to n
     for (i,j) in V x V
          d[k][i][j] = min(d[k-1][i][j], d[k-1][i][k] + d[k-
     1][k][j])
return d[n]
```

Non due archi consecutivi dello stesso colore

Dato un grafo orientato e pesato G = (V, E, W, col) senza cicli di peso negativo e senza cappi, *trovare il peso del cammino minimo tra due vertici i e i, tale che* ***non ci siano due archi consecutivi dello stesso colore***.

Definiamo $d_{ij}^{a,b}$ come il **peso del cammino da i a j, in cui il primo arco ha colore a, e l'ultimo arco ha colore b**, con a,b ∈{rosso, blu}.

Soluzione

La soluzione finale è data da D_n, una matrice che contiene tutte le distanze minime $d_{ij}^{a,b}$, per ogni coppia di colori a e b.

Caso base

k = 0, ∀ coppia di vertici i,j ∈ V^2:

$$d_{i,j}^{a,b} = \begin{cases} 0 & \text{se } i = j \\ w_{ij} & \text{se } (i,j) \in E,\ col(i) \neq col(j) \\ +\infty & \text{altrimenti} \end{cases}$$

Passo ricorsivo

Per ogni vertice intermedio *k*:

- se k non è un vertice intermedio: $d_{ij}^{a,b} = d_{ij}^{a-1,b-1}$
- Se k è un vertice intermedio: $d_{ij}^{a,b} = \min(d_{ij}^{a,b}, d_{ik}^{a,c} + d_{kj}^{c,b})$, dove **c** rappresenta **il colore dell'arco intermedio**.

Esattamente n archi rossi

Abbiamo (V, E, col) ,$col : E \text{ —> } \{red, blue\}$, con le stesse notazioni degli altri problemi.

$\forall$ i,j $\in V_2$, stabilire se esiste un cammino da i a j con **esattamente 3 archi rossi**.

Sottoproblema

Dato (k, r), con $k \in \{0, ...,n\}$, e $r \in \{0, ..., R\}$, definiamo la variabile $D_{k,r} = d_{ij}^{k,r}$, che vale **true** se e solo se esiste un cammino da i a j con esattamente *r archi rossi* e con vertici intermedi $\in \{1, ..., k\}$.

Caso base

(k,r) con k = 0 /\ $\forall$ e E = {0,1,2,R}, $\forall$ (i,j) $\in V^2$, abbiamo:

$$e_{ij}^{(0,0)} = \begin{cases} true & \text{se } i = j \\ true & \text{se } i \neq j \wedge (i,j) \in E \wedge col(i,j) \neq red \\ false & \text{altrimenti} \end{cases}$$

$$e_{ij}^{(0,1)} = \begin{cases} false & \text{se } i = j \\ true & \text{se } i \neq j \wedge (i,j) \in E \wedge col(i,j) = red \\ false & \text{altrimenti} \end{cases}$$

Quindi anche $e_{ij}^{(0,2)} = e_{ij}^{(0,3)} = false$

Passo ricorsivo

(k,r) con k > 0 /\ r $\in$ {0, 1, ..., R}

Dividiamo in:

- Se k non è un vertice intermedio: $e_{ij}^{(k,r)} = e_{ij}^{(k-1,r)} = E_1$
- Se k è un vertice intermedio
    - $e_{ij}^{k,r}$ = un arco deve essere rosso V $(e_{ik}^{(k-1,r)} \wedge e_{kj}^{(k-1,r)}) = E_2$

Se uniamo, abbiamo $e_{ij}^{k,r} = E_1 \text{ V } E_2$

Come **soluzione** quindi abbiamo $\forall$ i,j $\in V^2$ $e_o^{(n,3)}$.

Numero di archi rossi pari

Dato (V, E, W, col), col : E —> {red, blue}, $\forall$ i,j $\in V^2$, calcolare il **peso di un cammino minimo** da i a j, nel quale il **numero di archi rossi è pari**.

Sottoproblema

Definiamo (k,p), k $\in$ {0, …, n}, p $\in$ {0,1}, e definiamo $d_{ij}^{(k,p)}$.

Tale variabile è il peso di un cammino minimo da i a j nel quale il numero di archi è **pari se p = 0**, **dispari se p = 1**, e con vertici intermedi $\in$ {1, …, k}.

Caso base

(k, p) con k = 0 /\ p $\in$ {0,1}

$$d_{ij}^{(0,0)} = \begin{cases} 0 & \text{se } i = j \\ W_{ij} & \text{se } i \neq j \wedge (i,j) \in E \wedge col(i,j) \neq red \\ \infty & \text{altrimenti} \end{cases}$$

$$d_{ij}^{(0,1)} = \begin{cases} 0 & \text{se } i = j \\ W_{ij} & \text{se } i \neq j \wedge (i,j) \in E \wedge col(i,j) \neq red \\ \infty & \text{altrimenti} \end{cases}$$

Passo ricorsivo

(k,P) con k > 0 /\ $\forall$ p $\in$ {0,1}

Distinguiamo in:

- se k non è un vertice intermedio
  - $d_{ij}^{(k,p)} = d_{ij}^{(k-1,p)} = E_1$
- Se k è un vertice intermedio
  - P = 0
    - $d_{ij}^{(k,0)} = V\ \{(d_{ik}^{(k-1,0)} \wedge d_{kj}^{(k-1,0)})\ V\ (d_{ik}^{(k-1,1)} \wedge d_{kj}^{(k-1,1)}\} = E_2$
  - P = 1
    - $d_{ij}^{(k,1)} = V\ \{(d_{ik}^{(k-1,0)} \wedge d_{kj}^{(k-1,1)}) \wedge (d_{ik}^{(k-1,1)} \wedge d_{kj}^{(k-1,0)})\} = E_3$

$$d_{ij}^{(k,p)} = \begin{cases} e_1\ V\ e_2 \\ e_1\ V\ e_3 \end{cases}$$

La soluzione è quindi $\forall$ i,j $\in$ VxV $d_{ij}^{(n,0)}$, $D^{(n,0)}$.

Alterna il colore dei vertici

Dato (V, E, W, f, g), f : V —> {nero, lilla}, g : E —> {rosso, blu}, calcolare il peso di un cammino minimo da i a j nel quale **il numero di archi rossi è pari, e che alterna il colore dei vertici**.

Sottoproblema

(k,p), k ∈ {0, ..., n }, p ∈ {0, 1} ∀ i,j.

$d_{ij}^{(k,p)}$ è il peso di un cammino minimo da i a j nel quale il numero di archi rossi è pari se **p = 0**, dispari se **p = 1**, e che alterna il colore dei vertici, e con vertici intermedi ∈ {1, ..., k}.

Caso base

(k, p) con k = 0 /\ p ∈ {0,1}

$$d_{ij}^{(0,0)} = \begin{cases} 0 & \text{se } i = j \\ W_{ij} & \text{se } i \neq j \wedge (i,j) \in E \wedge g(i,j) \neq R \wedge f(i) \neq f(j) \\ \infty & \text{altrimenti} \end{cases}$$

$$d_{ij}^{(0,1)} = \begin{cases} \infty & \text{se } i = j \\ W_{ij} & \text{se } i \neq j \wedge (i,j) \in E \wedge g(i,j) \neq R \wedge f(i) \neq f(j) \\ \infty & \text{altrimenti} \end{cases}$$

Passo ricorsivo

(k, p) con k > 0 /\ p ∈ {0,1}

- se k non è un vertice intermedio
  - $d_{ij}^{(k,p)} = d_{ij}^{(k-1,p)} = E_1$
- Se k è un vertice intermedio
  - P = 0
    - $d_{ij}^{(k,0)} = V\ \{(d_{ik}^{(k-1,0)} \wedge d_{kj}^{(k-1,0)})\ V\ (d_{ik}^{(k-1,1)} \wedge d_{kj}^{(k-1,1)}\} = E_2$

Cammino minimo con lunghezza minore di L

Abbiamo come istanza (V, E, W), con L∈N+. La soluzione è trovare il peso di un cammino minimo da i a j, con vertici intermedi E {0, ..., k}, e con lunghezza ≤ L.

Sottoproblema

Definiamo il sottoproblema come $d_{i,j}^{k,l}$, come il peso di un cammino minimo da i a j con vertici intermedi, e con lunghezza ≤ L.

Caso base

Con k = 0 /\ l∈{0, ..., L}

$$d_{ij}^{k,l} = \begin{cases} 0 & \text{se } i = j \\ \infty & \text{se } i \neq j \ /\backslash \ (i,j) \notin E \\ W_{i,j} & \text{se } i \neq j \ /\backslash \ (i,j) \in E \end{cases}$$

$$d_{ij}^{0,0} = \begin{cases} 0 & \text{se } i = j \\ \infty & \text{se } i \neq j \ /\backslash \ (i,j) \in E \\ \infty & \text{se } i \neq j \ /\backslash \ (i.j) \notin E \end{cases}$$

Passo ricorsivo

Con k > 0 /\ l∈ {0, ..., L}

- se k non appartiene al cammino
  - $d_{ij}^{k,l} = d_{ij}^{(k-1,\ l)} = E1$
- Se k appartiene al cammino
  - $d_{ij}^{k,l} = \min\{d_{ik}^{(k-1,l1)}, d_{kl}^{(k-1,\ l2)}\} = E2$
  - Con $0 \leq l_1 \leq L$, $0 \leq l_2 \leq L$, $l_1+l_2 \leq L$

Sequenza dei valori degli archi crescente

Abbie o come istanza (V, E, W, f), f : E -> N. Dobbiamo trovare il peso di un cammino minimo da i a j nel quale la sequenza dei valori degli archi è strettamente crescente.

Problema ausiliario

Dobbiamo introdurre un problema ausiliario, dove verifichiamo il valore associato degli archi.

$\forall$ (i,j)∈ V^2, $\forall$ (a,b)∈ $f(E)^2$, calcoliamo il peso di un cammino minimo da i a j, nel quale la sequenza dei **valori associati agli archi è strettamente crescente**, con valore associato al primo arco (variabile **a**), e con valore associato all'ultimo arco (variabile **b**).

Caso base

$\forall (i,j) \in V^2$, $\forall (a,b) \in f(E)^2$

$$d_{i,j}^{0,a,b} = \begin{cases} 0 & \text{se } i = j \\ w_{i,j} & \text{se } i \neq j \wedge f(i,j) = a = b \\ \infty & \text{altrimenti} \end{cases}$$

Passo ricorsivo

Con $k > 0$

- Se k non appartiene al cammino
  - $d_{ij}^{k,a,b} = d_{ij}^{k-1,a,b} = E_1$
- Se k appartiene al cammino
  - $d_{ij}^{k,a,b} = d_{ik}^{k-1,a,c} + d_{kj}^{k-1,d,b} = E_2$, con $c < d$

Quindi la soluzione è $d_{i,j}^{k,a,b} = \min\{E_1, E_2\}$. Dato il problema ausiliario, dovremmo avere anche una soluzione al problema originale.

$$d_{ij}^{pb} = \begin{cases} 0 & \text{se } i = j \\ \min\{d_{ij}^{n,a,b}\} & \text{altrimenti} \end{cases}$$

Presenti entrambi i colori

Dato un insieme (V, E, W, col), col: E -> {red, blu e}, determinare il **peso di un cammino minimo** da un nodo di partenza a un nodo di arrivo in un grafo, in modo che **ci siano archi di un colore specifico, sia di un altro colore**.

Sottoproblema

(K, p, q), con: $K \in \{0, n\}$, $p \in \{True, False\}$, $q \in \{True, False\}$, $\forall (i,j) \in V^2$.

Definiamo dij(k,p,q) come il *peso di un cammino minimo* da i a j nel quale è *presente un arco rosso sse* ***p = T****, nel quale è presente un arco blu sse* ***q = T****, e con vertici intermedi* $\in \{1, ..., K\}$.

Caso base

(K,p,q), con $K = 0 \wedge p \in \{T,F\}$, $q \in \{T, F\}$

- $d_{ij}^{0,T,T} = \infty$, perché sarebbero presenti sia un arco rosso che un arco blu
- $d_{ij}^{0,T,F} = \infty$ se $i = j$

$$\begin{cases} W_{i,j} & \text{se } i \neq j \wedge (i,j) \in E \wedge col(i,j) = red \\ \infty & \text{altrimenti} \end{cases}$$

- $d_{ij}^{0,F,T} = \begin{cases} \infty & \text{se } i = j \\ W_{i,j} & \text{se } i \neq j \wedge (i,j) \in E \wedge col(i,j) = blue \\ \infty & \text{altrimenti} \end{cases}$
- $d_{ij}^{0,F,F} = \begin{cases} \infty & \text{altrimenti} \\ W_{i,j} & \text{se } i \neq j \wedge (i,j) \in E \\ \infty & \text{altrimenti} \end{cases}$

Passo ricorsivo

(K, p, q) con $K > 0 \wedge p \in \{T, F\} \wedge q \in \{T, F\}$

- se K non appartiene al cammino
  - $d_{i,j}^{(K,p,q)} = d_{i,j}^{(K-1,p,q)} \Rightarrow E_0$
- Se k appartiene al cammino
  - p = q = T
    - $d_{i,j}^{(K,T,T)} = \min\{d_{ik}^{(K-1, r1, b1)} + d_{kj}^{(K-1, r2, b2)}\} \Rightarrow E_1$
      - $(r_1,b_1,r_2,b_2) \in \{T,F\}^4$ t.c. $(r_1 = T \vee r_2 = T) \wedge (b_1 = T \vee b_2 = T)$
      - Almeno una delle variabili che dice rosso o blu deve essere vera
  - p = T, q = F
    - $d_{ij}^{(K, T, F)} = \min\{d_{ik}^{(K-1,r1,b1)} + d_{kj}^{(K-1,r2,b2)}\} \Rightarrow E_2$
      - $(r_1,b_1,r_2,b_2) \in \{T,F\}^4$ t.c. $(r_1 = T \vee r_2 = T) \wedge (b_1 = b_2 = F)$
  - p = F, q = T
    - $d_{ij}^{(K, F, T)} = \min\{d_{ik}^{(K-1,r1,b1)} + d_{kj}^{(K-1,r2,b2)}\} \Rightarrow E_3$
      - $(r_1,b_1,r_2,b_2) \in \{T,F\}^4$ t.c. $(r_1 = r_2 = F) \wedge (b_1 = T \vee b_2 = T)$
  - p = F, q = F
    - $d_{ij}^{(K,F,F)} = d_{ik}^{(K-1,F,F)} + d_{kj}^{(K-1,F,F)} \Rightarrow E_4$

La soluzione finale per il passo ricorsivo è dunque:

$$d_{ij}^{(K,p,q)} = \begin{cases} \min\{E_0, E_1\} & \text{se } p = q = T \\ \min\{E_0, E_2\} & \text{se } p = T \text{ e } q = F \\ \min\{E_0, E_3\} & \text{se } p = F \text{ e } q = T \\ \min\{E_0, E_4\} & \text{se } p = q = F \end{cases}$$

3 coppie di vertici consecutivi rossi

Dato (V, E, W, col), col: V -> {red, blue}, bisogna determinare il peso di un cammino minimo da i a j nel quale vi sono **3 coppie di vertici consecutivi rossi**.

Sottoproblema

Definiamo il sottoproblema con (K, r), con K∈{0, ..., n}, e r∈{0, ..., R}. $d_{ij}^{(K,R)}$ è il *peso di un cammino minimo da i a j in cui vi sono r coppie di vertici rossi, e con vertici intermedi ∈{1, ..., K}*.

Caso base

(K, r), con k ∈ {0, ..., n}, e ∀ e ∈ {0, ..., R}.

- $d_{i,j}^{0,0} = \begin{cases} 0 & \text{se } i = j \\ W_{i,j} & \text{se } i \neq j \wedge (i,j) \in E \wedge (col(i) \neq red \wedge col(j) \neq red)) \\ \infty & \text{altrimenti} \end{cases}$
- $d_{ij}^{0,1} = \begin{cases} \infty & \text{se } i = j \\ W_{i,j} & \text{se } i \neq j \wedge (i,j) \in E \wedge (col(i) = red \wedge col(j) = red)) \\ \infty & \text{altrimenti} \end{cases}$
- $d_{ij}^{0,2} = d_{ij}^{0,3} = \infty$

Passo ricorsivo

(K, r) con K > 0, e ∀ r ∈ {0, ..., R}

- K non è un vertice intermedio in un cammino desiderato (ovvero K non appartiene al cammino)
  - $d_{ij}^{k,r} = d_{ij}^{(k-1,r)} \Rightarrow E_1$
- K è un vertice intermedio in un cammino desiderato (ovvero K appartiene al cammino)
  - $d_{ij}^{k,r} = \min\{d_{ik}^{(K-1,r1)} + d_{kj}^{(K-1,r2)}\} \Rightarrow E_2$
    - $(r_1, r_2) \in \{0, ..., r\}^2$ t.c. $r_1 + r_2 = r$

La soluzione del passo ricorsivo è quindi $d_{ij}^{k,r} = \min\{E_1, E_2\}$.

La soluzione finale è ∀ $(i,j) \in V^2$ $d_{ij}^{(n,3)}$.

Strutture dati per insiemi disgiunti

Sono utilizzate per **gestire un insieme di elementi divisi in sottoinsiemi distinti**, dove ogni elemento appartiene un solo sottoinsieme.

Si supponga di avere X elementi, i quali si possono raggruppare in insiemi disgiunti S_1, S_2, ..., S_k, con ciascun sottoinsieme contenente un certo numero di elementi.

Queste strutture dati sono comunemente usate in algoritmi che *richiedono l'identificazione di componenti connesse in un grafo non orientato*, come negli algoritmi di *Kruskal* e *Prim*, per la costruzione di alberi di copertura di peso minimo.

Di seguito, un esempio.

Consideriamo un insieme $X = \{x_1, ..., x_n\}$ con $S_1, ..., S_k$ come *collezione di insiemi disgiunti*. Le operazioni, quindi, possono essere illustrate come segue:

- Se $x \in S_i$, $y \in S_j$, $i \neq j$, l'operazione $\mathrm{Union}(x, y)$ combina S_i e S_j in un unico insieme
- $\mathrm{Find\_set}(x)$ restituirà il rappresentante di S_i, consentendo di verificare l'appartenenza.

Rappresentazione dell'insieme

Ogni insieme è rappresentato da un elemento designato come **rappresentante**: non importa quale sia il rappresentante specifico di un insieme, purchè sia unico per quell'insieme.

Tipo di dato astratto

Un tipo di dato astratto per insiemi disgiunti deve supportare principalmente queste operazioni:

- **Make_Set(x)**: crea un insieme contenente solo l'elemento x. Ha una complessità $O(1)$, ed è utilizzata per inizializzare ogni elemento come appartenente a un insieme disgiunto.
- **Union(x,y)**: unisce i due insiemi disgiunti contenenti x e y, garantendo che gli elementi siano raggruppati sotto un unico rappresentante. Se x e y appartengono già allo stesso insieme, l'operazione non cambia la struttura. Ha una complessità O(n), da ricordarsi che bisogna aggiornare anche i riferimenti al rappresentante nuovo.
- **Find_set(x)**: trova l'insieme contenente x, e restituisce un rappresentante di quell'insieme. Ogni insieme ha un rappresentante unico, e questa operazione consente di identificare a quale gruppo appartiene un elemento. Ha una complessità O(n), ma se tengo un puntatore anche alla testa della lista, allora diventa O(1)

Algoritmo per il calcolo delle componenti connesse in un grafo non orientato

Per determinare le componenti connesse in un grafo, possiamo applicare il seguente algoritmo, che utilizza la struttura dati per insiemi disgiunti; questo algoritmo itera su ciascun vertice e arco del grafo:

1. **Inizializzazione**: $\forall$ vertice v in G, crea un insieme tramite $\mathrm{Make\_set}(v)$
2. **Unione degli insiemi**: $\forall$ arco (u,v) in G, verifica se i vertici u e v appartengono a insiemi differenti. Se si, si esegue $\mathrm{Union}(u, v)$
3. **Risultato**: alla fine, i vertici appartenenti allo stesso insieme rappresentano componenti connesse del grafo.

Esempio lezione

Questo è l'esempio svolto a lezione.

$\forall\ x \in X$, MAKESET(x) = {x}. Dati $x,y \in X$, con S_x intersecato $S_y \neq 0$,

S_x = insieme al quale appartiene x

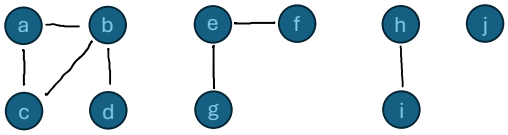
S_y = insieme al quale appartiene y

Con questo si crea un insieme, con un solo elemento, nodo del grafo

UNION(x,y) : Z, $\forall\ x,y \in X^2$ tale che S_x intersecato $S_y = \emptyset$, UNION(x,y) = $S_x \cup S_y$

$\forall\ x \in X$, FIND_SET: X -> X, $\forall\ x \in X$ FIND_SET(X) = rapp(S_x)

Con questo si uniscono tutti gli insiemi che contengono nodi connessi tra loro da un arco



$R \subseteq V^2$, (V,E) non orientato; R = {(u,v)$\in V^2$ => esiste un cammino da u a v}, dove v è raggiungibile da n. Abbiamo quindi la seguente tabella:

Arco elaborato	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}	{e}	{f}	{g}	{h}	{i}	{j}	
(e,g)	{a}	{b,d}	{c}	{e,g}	{f}	{h}	{i}	{j}		
(a,c)	{a,c}	{b,d}	{e,g}	{f}	{h}	{i}	{j}			
(h,i)	{a,c}	{b,d}	{e,g}	{f}	{h,i}	{j}				
(a,b)	{a,b,c,d}	{e,g}	{f}	{h,i}	{j}					

(e,f)	{a,b,c,d}	{e,f,g}	{h,i}	{j}						
(b,c)	FIND_SET(B) = FIND_SET(C)									

Pseudocodice

```
CONNECTED_COMPONENT(G), G = (V,E)
for each v∈V
     MAKE_SET(v)
for each (u,v)∈E
     if FIND_SET(u) ≠ FIND_SET(v)
          UNION(u,v)
SAME_COMP(u,v)
     if FIND_SET(u) = FIND_SET(v)
          return true
     else return false
```

Strutture dati per insiemi disgiunti v2

Lista concatenata

Questa parte è stata fatta a lezione il 25 novembre, con lo stesso titolo – dovrebbe essere una diversa implementazione o qualcosa di simile.

Per rappresentare insiemi disgiunti, si utilizza una struttura basata su liste concatenate ed alberi. Ogni insieme è costituito da una lista di elementi, i quali sono strutturati.

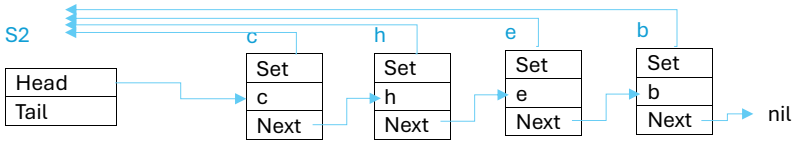
Rappresentazione degli insiemi

Ogni insieme è rappresentato come una lista concatenata, con un **head** che punta all'inizio della lista, e un **tail** che punta alla fine della lista. Gli oggetti nella lista hanno i seguenti *attributi*:

- `x.set`: riferimento all'insieme a cui appartiene l'elemento
- `x.key`: il valore dell'elemento
- `x.next`: puntatore all'elemento successivo nella lista

Set
Key
Next

Il **rappresentante** è il primo oggetto della lista, e se dovessimo rappresentarla visivamente, con l'esempio S = {c, h, e, b}.



Operazioni

Make-set

Crea un insieme con un singolo nodo radice *x*.

```
MAKE_SET(x)
new S
x.set = S
x.next = NIL
s.tail = x
s.head = x
```

Find-set

Trova la radice di x puntando fino alla radice.

```
FIND_SET(x)
return x.set.headà
```

Commentato [FD5]: È giusta questa descrizione? Sembra che .set porti direttamente al primo nodo, senza seguire i puntatori uno ad uno

Commentato [FC6R5]: Si hai ragione, dovrebbe essere invece che punta direttamente al primo

Union

Aggiunge la lista contenente x in coda alla lista contenente y.

```
UNION(x,y)
S1 = x.set
S2 = y.set
if S1.length ≥ S2.length
     S1.tail.next = S2.head
     S1.tail = S2.tail
     Z = S2.head
     while Z ≠ NIL
          Z.set = S1
          Z = Z.next
     S1.length = S1.length + S2.length
```

Commentato [FD7]: Non capivo cosa si intendesse, poi ho capito. Cosa sarebbero “e” e “g”?

Commentato [FC8R7]: Ho messo la spiegazione sbagliata del prof, prima ha scritto UNION(e,g), e poi ha messo x e y. Così dovrebbe essere giusta

Problema generale

L’algoritmo esegue m operazioni in totale, tra MAKE_SET, FIND_SET, UNION:

- **n** di queste sono MAKE_SET(le prime n) al più

- **n-1** delle quali sono UNION

Avremmo quindi un codice simile a questo, con un totale di aggiornamenti di:

$$\sum_{i=1}^{n-1} i = ((n-1)(n-2))/2 = \Theta(n^2)$$

```
for i=1 to n
     MAKE_SET(Xi)
for i=1 to n-1
     UNION(Xi+1, Xi)
```

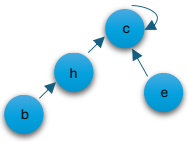
Dopo log+c volte, l'insieme risultante ha almeno *k elementi*; siccome l'insieme più grande ottenibile ha n elementi, allora **log n** è il massimo numero di volte che x viene aggiornato.

Foreste di insiemi disgiunti

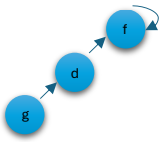
Ogni insieme è un albero radicato. Ogni nodo contiene un elemento. Ogni nodo punta solo al *padre*, e la radice *a se stessa*. La radice **contiene il rappresentante**.

Di seguito, due esempi.

{b,c,e,h}



{d,f,g}



Operazioni

In queste operazioni, per eseguire ogni UNION, occorre eseguire FIND_SET 2 volte. Una sequenza di **n-1 UNION** può creare una catena lineare di n nodi.

Introduciamo **rank**, che è il limite superiore per l'altezza di un nodo all'interno dell'albero che rappresenta un insieme. Con l'altezza di un nodo si intende il *numero di archi in un cammino più lungo dal nodo ad una foglia*.

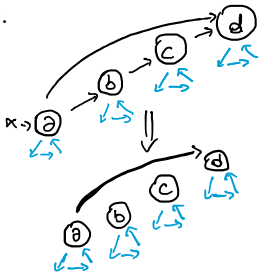
Make_set

```
MAKE_SET(x)
     x.p = x
     x.rank = 0
```

Find_set

In questo caso viene effettuata la **compressione dei cammini**.

```
FIND_SET(x)
     if x.p = x
          return x.p
     else return FIND_SET(X.p)
```



Union

```
UNION(e,g)
     LINK(FIND_SET(x), FIND_SET(y))


LINK(x,y)
     if x.rank > y.rank
          y.p = x
     else x.p = y
          if x.rank = y.rank
               y.rank = y.rank + 1
```

Commentato [FC9]: Non sono sicuro se questo else sia giusto

Algoritmi greedy

Rappresentano una classe di algoritmi usati per risolvere problemi di ottimizzazione, in cui l'obiettivo è **massimizzare o minimizzare** una certa funzione. Segue questi principi:

- **Scelta locale ottima**: in ogni passo, l'algoritmo sceglie l'opzione che sembra migliore sul momento, senza considerare soluzioni future o alternative
- **Non reversibilità**: una volta fatta una scelta, non viene mai riconsiderata; questo porta spesso a una *complessità computazionale inferiore* rispetto ad altri approcci.

La differenza rispetto a una programmazione dinamica è che *un algoritmo greedy effettua scelte locali ottimali che puntano a risolvere il sotto problema senza ripercorrere scelte precedenti*, mentre nella programmazione dinamica si risolvono i sottoproblemi in sequenza, basandosi su soluzioni già calcolate. L'unica cosa che li accomuna è che entrambi hanno la proprietà della sottostruttura ottima (*poiché la soluzione ottimale al problema globale può essere costruita partendo dalle soluzioni ottimali dei sottoproblemi*).

Di seguito, un esempio – sia data una serie di elementi con valori ed ingombri. L'obiettivo è *massimizzare il valore totale senza superare un limite di ingombro*, in questo caso K = 45.

La strategia che si usa è di **selezionare gli elementi con il massimo valore rispettando il limite d'ingombro**. Come esempi di soluzioni possiamo avere:

- $A = \{1\} \rightarrow A = \{1\}, OPT = 100$

	VAL	ING
1	100	30
2	80	25
3	60	20

- $S = \{2,3\}$ → $OPT = 140$

Sistema di indipendenza

Un sistema di indipendenza è una struttura matematica che consiste in una coppia (E,F) dove **E** è l'insieme finito degli **elementi**, e invece $F \subseteq P(E)$ è l'insieme degli **insiemi indipendenti** (famiglia di sottoinsiemi di E).

La famiglia F deve soddisfare le seguenti proprietà:

- **Proprietà dell'insieme vuoto**: l'insieme vuoto è considerato indipendente
- **Proprietà dell'ereditarietà/chiusura**: $\text{se } A \in F, B \subseteq A, \text{allora } B \in F$. Ogni sottoinsieme di un insieme indipendente è anch'esso indipendente.

Matroide

Un sistema di indipendenza (E,F) è detto **matroide** se per ogni c $\subseteq$ E gli insiemi massimali di <c, F_c> hanno la **stessa** cardinalità, e se soddisfa la proprietà di scambio:

$$A,B \in F, \text{con } |B| > |A|, \text{esiste un elemento } b \in B \setminus A \text{ t.c. } A \cup \{b\} \in F.$$

Data una famiglia F di insiemi indipendenti, se un insieme B in F ha una cardinalità maggiore di un altro insieme A in F, è possibile trovare un elemento in B, che non appartiene ad A, da aggiungere ad A mantenendo l'indipendenza. Questa struttura soddisfa due proprietà:

- **Ereditarietà**. L'insieme vuoto appartiene ad F, e tutti i sottoinsiemi propri di un insieme di F appartengono anch'essi ad F
  - $\emptyset \in F, \text{se } I \in F \text{ e } J \subseteq I, \text{allora } J \in F$
- **Scambio**: se due insiemi di F non hanno la stessa cardinalità, allora un elemento che sta soltanto nel sottoinsieme più grande può essere aggiunto all'insieme più piccolo, e l'insieme risultante appartiene ancora ad F.

Sia <E,F> un sistema di indipendenza

Matroide di un grafo

M è un matroide di un grafo non orientato G = (V,E), se S corrisponde all'insieme E degli archi; mentre, un insieme I di archi sta in F se, e solo se, *gli archi in I non formano un ciclo*.

Matroide pesato

È pesato se **c'è un peso positivo associato ad ogni elemento x ∈ S**. Il peso di un insieme $I \in F$ è dato dalla *somma dei pesi degli elementi che appartengono ad I*.

Il metodo greedy verifica le proprietà della scelta greedy e della sottostruttura ottima, e che pertanto trova sempre l'insieme indipendente massimale di peso massimo.

La funzione di costo è quindi:

$$w: S \longrightarrow R^+; w: F \longrightarrow R^+, w(F) = \sum_{e_i \in F} w(e_i)$$

Dimostrazione del teorema di rado

Ottimalità di GREEDY_MAX

Supponiamo che (E,F) sia un matroide. Sia $w: E \rightarrow R$ una funzione di peso qualunque, e consideriamo $S = \{b_1, ..., b_p\}$, dove i pesi degli elementi sono ordinati in modo decrescente:

$$w(b_1) \geq w(b_2) \geq ... \geq w(b_p)$$

Definiamo S come l'insieme restituito dall'algoritmo *GREEDY_MAX applicato a (E,F,w)*.

Poiché $S \in F$, S è massimale in (E,F), sia ora $A \in F$ un **altro insieme massimale**. Si vuole dimostrare che $\mathbf{w(S) \geq w(A)}$.

Essendo A massimale, (E,F) un matroide, si ha che $|A| = |S| = p$, $\text{con } A = \{a_1, ..., a_p\}$.

Dimostriamo che $\forall\, i = 1, ..., p$ vale $w(b_i) \geq w(a_i) \Rightarrow w(S) \geq w(A)$. Supponiamo *per assurdo* che esista un indice $i \in \{1, ...,p\}$ tale che $w(b_i) < w(a_i)$, il che è una **contraddizione**.

Definiamo ora:

Insiemi	**Proprietà matroidi**
$D = \{b_1, ..., b_{k-1}\} \subseteq S$, con i primi *k-1* elementi di S	$S \in F \Rightarrow D \in F$
$D' = \{a_1, ..., a_{k-1}, a_k\} \subseteq A$	$A \in F \Rightarrow D' \in F, \lvert D'\rvert = \lvert D\rvert + 1$

Secondo la **proprietà di scambio dei matroidi**, esiste un elemento $a_j \in D' \setminus D$ t.c. $D \cup \{a_j\} \in F$.

Essendo D costituito dai primi k-1 elementi *selezionati da GREEDY_MAX*, l'elemento b_k è necessariamente il k-esimo elemento selezionato, e allora: $w(b_k) \geq w(a_j)$.

Con $j \leq k$ otteniamo $w(b_k) \geq w(a_k)$.

Dimostrazione della direzione inversa

Se (E,F) non è un matroide, allora esiste una funzione di peso $w: E \rightarrow R$ tale che *GREEDY_MAX non restituisce una soluzione massima associata a (E,F) e w.*

Supponiamo che (E,F) non sia un matroide. Esistono due insiemi $A,B \in F$ tali che

$$|B| = |A| + 1, |B| > |A|, \forall\, b \in B \setminus A \text{ l'insieme } A \cup \{b\} \notin F$$

Sia W: E→R la funzione così definita $\forall\, x \in E$:

$$w(x) = \begin{cases} 1 + \frac{1}{2}|A| & \text{se } x \in A \\ 1 & \text{se } x \in B \setminus A \\ 0 & \text{se } x \in (A \cup B)^C \end{cases}$$

Considerando una funzione peso per ogni elemento:

- All'interno di S, l'insieme restituito da GREEDY_MAX, ci sono **sicuramente tutti gli elementi di A**
- In S potrebbero esserci anche elementi che appartengono a $(A \cup B)^C$

Per la definizione di w, abbiamo:

- $w(S) = w(A) = |A| * (1 + 1/(2|A|)) = |A| + \frac{1}{2}$
- $w(B) \geq |B| = |A| + 1 > |A| + \frac{1}{2} = w(S),\ b \in F$

Esempi

Esempio di non matroide

Consideriamo questi elementi con valori ed ingombri. Sia K = 45 il massimo ingombro consentito.

Definiamo $E = \{x_1, x_2, x_3\}$, e il sistema di indipendenza (E,F), dove: $F = \{A \subseteq E \mid ing(A) \leq K\}$ è l'**insieme delle soluzioni ammissibili**, ovvero sottoinsiemi di E il cui ingombro totale è minore o uguale a K.

Come *esempi in F* abbiamo $B = \{x_2, x_3\} \in F$, $ing(B) = 45$; $A = \{x_3\} \in F$, con $ing(A) = 30$. In questo caso, $|B| = 2 > |A| = 1$.

Verifichiamo quindi *la proprietà di scambio*: verifichiamo se esiste un elemento $b \in B \setminus A$ t.c. $A \cup \{b\} \in F$. Come possibili combinazioni abbiamo:

- $A \cup \{x_1\} = \{x_1, x_3\} \notin F$ *poiché ing = 50 > K*
- $A \cup \{x_2\} = \{x_2, x_3\} \notin F$ *poiché ing = 55 > K*

Non esiste quindi un elemento $b \in B \setminus A$ t.c. $A \cup \{b\} \in F$. quindi, (E,F) **non è un matroide**.

Esempio 1 (Sistema di indipendenza)

Sia V uno spazio vettoriale e $|E| < \infty$; consideriamo

$$E = \{A \subseteq E \mid A \text{ un insieme di vettori linearmente indipendenti}\}$$

In questo caso abbiamo che, *se* $A \subseteq F$ *e* $B \subseteq A$*, allora* $B \subseteq F$.

Esempio 2 (Sistema di indipendenza)

Consideriamo il problema Knapsack. Sia $E = \{x_1, \ldots, x_m\}$, $ing : E \rightarrow N$.

Definiamo $F = \{A \subseteq E \mid ing(A) \leq K\}$. Abbiamo quindi la *proprietà di chiusura*: se $A \in F$, e $B \subseteq A$, allora $B \in F$, dato che $ing(B) \leq ing(A) \leq K$.

Esempio 3 (Matroide di un Grafo)

Rappresentiamo una foresta in un grafo.

Sia $G = (V, E)$, con $V = \{1, 2, \ldots, 7\}$, ed E l'insieme degli archi rappresentati nel grafo:

$$E = \{(1,2), (2,3), (1,3), (4,5), (5,7), (6,7), (4,6)\}$$

Definiamo F come $\{A \subseteq E \mid (V,A) \text{ è una foresta}\}$. Effettuiamo quindi le seguenti verifiche:

- $E = \{(1,2), (1,3), (2,3), (4,5), (5,7), (6,7), (4,6)\}$, $E \in F \rightarrow$ no, poiché contiene cicli
- $A = \{(1,2)\} \in F$? Si
- $A = \{(1,2), (1,3), (4,5), (6,7)\} \in F$? Si

Esempio knapsack frazionario

Definiamo k = 45, e prendiamo tutte le attività che possono essere inserite, partendo dal primo elemento.

- $ing(X_1) \leq k$, quindi *possiamo prenderlo*
- $ing(X_2) \geq (k\text{-}ing(X_1))$, quindi *prendiamo soltanto la parte frazionaria che ci sta nel knapsack*

Quindi abbiamo: $\{X_1, 15/25\ X_2\}$, $ING = 148$

	VAL	ING	VAL/ING
1	100	30	3,3
2	80	25	3,2
3	60	20	3

Esempio 4 (massimale)

Definiamo <E,F> come un sistema indipendente, e una *funzione peso* w: E —> R, $A \subseteq E$, $A \subset F$.

In questo caso, A è **massimale** quando $\forall\ e \in E \setminus A$, $A \cup \{e\} \notin F$.

La soluzione è M ∈ F **massimale** tale che $w(M) = \max(A \in F, A \text{ massimale}) \{w(A)\}$.

Un insieme A è **massimale** (rispetto a F) quando non è possibile aggiungere nessun altro elemento di $E \setminus A$ ad A mantenendo l'indipendenza.

Esempio 5 (cardinalità k)

Definiamo <E,F>; $\forall A, B \in F$ con $|B| > |A|$ esiste $b \in B \setminus A$ tale che $A \cup \{b\} \in F$.

Con $F = \{A \subseteq E \mid |A| \leq k\}$, che vuol dire che F include tutti i sottoinsiemi di E con cardinalità al massimo k, guardiamo due domande:

- <E,F> è un sistema indipendente? Bisogna soddisfare la **proprietà di chiusura**
  - $\forall A \in F, \forall B \subseteq A \longrightarrow B \in F$ (chiusura)
    - Supponiamo che $A \in F$, allora $|A| \leq k$
    - Se $B \subseteq A$, allora $|B| \leq |A| \leq k \longrightarrow$ soddisfa il vincolo $|B| \leq k \longrightarrow$ implica $B \in F$
- <E,F> è un matroide? Bisogna soddisfare la proprietà di chiusura (ereditarietà), e quella dell'**aumento**
  - $\forall A,B \in F, |B| = |A| + 1$ esiste $b \in B \setminus A$ tale che $A \cup \{b\} \in F$ (aumento)
    - Supponiamo $A,B \in F, |B| = |A|+1$
    - Per costruzione di F, sappiamo che $|A| \leq k$, e $|B| \leq k$
      - Questo implica che $|A| + 1 \leq k$, possiamo aggiungere un elemento ad A senza superare la soglia k
      - Poiché $B \setminus A \neq \emptyset$, esiste almeno un elemento $b \in B \setminus A$
      - Aggiungendo b ad A, otteniamo $A \cup \{b\}$
      - La cardinalità di $A \cup \{b\}$ quindi sarà $|A| + 1$, dato che $|A|+1 \leq k$, allora $A \cup \{b\} \in F$

Esempio 6

Si consideri un grafo non orientato G = (V,E), <E,F>, $F = \{A \subseteq E \mid (V,A) \text{ è una foresta}\}$, ed è un sistema indipendente, quindi un *matroide*. Una **foresta** è un grafico **aciclico**; se una foresta contiene t componenti connesse, queste componenti saranno ciascuna un albero (ovvero un grafo senza cicli).

Dato un grafo (V,C) che è una foresta, essa contiene $|V| - |C|$ alberi.

DIMOSTRAZIONE: definiamo t come il numero di alberi, e_i, v_i il numero di archi e di vertici dell'i-esimo albero; in questo caso, $e_i = v_i - 1$ se ho un albero.

Se una foresta F è composta da t alberi, (t = **numero di componenti connesse)**

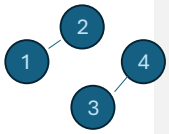
$$|C| = \sum_{i=1}^{t} e_i = \sum_{i=1}^{t} (v_i - 1) = \sum_{i=1}^{t} v_i - \sum_{i=1}^{t} 1 = |V| - t$$

Commentato [FC10]: Ho messo adesso l'esempio corretto, era giusto con |C|, e contiene |V| - |C| alberi. Se hai bisogno ti mando la foto degli appunti

Quindi $|C| = |V| - t \Rightarrow t = |V| - |C|$

- Si somma il numero totale di archi di ogni albero
- Il numero totale degli archi dell'i-esimo albero è uguale a v_{i-1}
- Scompongo le due sommatorie

Ogni parte del grafo separata è quindi un **albero**. Come si vede nella figura, il nodo 1 e il nodo 2 formano un albero, mentre il nodo 3 e 4 ne formano un altro. Come esempio di applicazione della formula, applichiamo $|E| = 4\ \text{nodi} - 2\ \text{alberi} = 2\ \text{edge}$.



Teorema di rado

Sia (E,F) un sistema indipendente. (E,F) è un matroide se e solo se l'algoritmo greedy risolve il problema di massimo peso per ogni funzione peso w: $E \longrightarrow R^+$.

In particolare, afferma che:

- Se **(E, F) è un matroide**, allora possiamo utilizzare l'algoritmo greedy per trovare il sottoinsieme $A \in F$ che massimizza il peso w; questo sottoinsieme A sarà quindi ottimale
- Se **un sistema indipendente (E, F) soddisfa il teorema greedy**, ovvero l'algoritmo riesce sempre a trovare la soluzione ottimale per qualsiasi funzione peso w, allora (E,F) è un matroide

Esempi di sistema di indipendenza su un grafo

Sia G = (V,E) un grafo con:

- $V = \{1,2,3,4,5,6,7\}$, ovvero l'insieme dei nodi
- $E = \{(1,2), (2,3), (1,3), (4,5), (5,7), (6,7), (4,6)\}$, ovvero l'insieme degli archi

Definiamo il sistema di indipendenza (E,F) dove $F = \{A \subseteq E \mid (V,A)\ \text{una foresta}\}$, ovvero *F contiene tutti i sottoinsiemi di archi che formano una foresta* (grafo aciclico e non orientato).

Definiamo $A = \{(1,2), (2,3), (4,5), (5,7), (6,7)\} \subseteq E$, e in questo caso, *$A \in F$ poiché il grafo è una foresta*.

Massimale

L'insieme A è detto **massimale** rispetto a F se $\forall e \in E \setminus A,\ A \cup \{e\} \notin F$. A è quindi indipendente, e non può essere ulteriormente esteso senza perdere la proprietà di indipendenza.

Problema di massimo associato a un sistema di indipendenza

Sia (E,F) un sistema di indipendenza, e sia $w : E \rightarrow R_+$ una funzione **peso**. Definiamo il problema di massimo come quello di trovare $A \in F$ tale che massimizzi il peso complessivo, quindi:

$$w(A) = \sum_{e \in A} w(e) \text{ sia } \textbf{massimo} \text{ tra tutti gli insiemi indipendenti in } F.$$

Algoritmo greedy

Associato ad un sistema d'indipendenza <E,F>, e ad una funzione peso $w : E \rightarrow R^+$. è un algoritmo per i sistemi d'indipendenza matroidi (Solo per matroidi!).

```
GREEDY-MAX(E,F,W)
S = Ø
Q[1, …, n] = (e₁, …, eₙ)
Q = SORT(Q) //ordinamento decrescente rispetto ai pesi
     for i = 1 to n
          if S ∪ {Q[i]} ∈ F
               S = S∪{Q[i]}
return S
//S = insieme indipendente massimale rispetto ai pesi
```

Problema del massimo assortimento

Dato un insieme E, e una funzione di peso $w : E \text{ —> } R$, si vuole individuare un sottoinsieme $S \subseteq E$ che ***massimizza il peso totale*** $w(S) = \sum_{e \in S} w(e)$, rispettando alcune condizioni di ammissibilità.

Se B non è massimale, esiste B' $\in$ F **massimale** con $B' \in F$, con B' $\subseteq$ B, tale che:

$$B' = B \cup \{e\}, \quad w(B') = w(B) + w(e) \geq w(B) \geq w(S)$$

Problema di minimo associato

Il problema di minimo associato si basa su un sistema <E,F> con una funzione peso $w: E \text{ —> } R^+$.

Istanza: Sia <E,F> un sistema indipendente, con una funzione peso $w: E \text{ —> } R^+$

Soluzione: Trovare un sottoinsieme massimale $S \in F$ tale che **minimizzi il peso totale**:

$$w(s) = \min_{(A \in F,\ A \text{ massimale})} \{w(A)\}$$

Algoritmo

```
GREEDY_MIN(E,F,W)
     S = Ø
     Q[1, …, n]  = (e₁, …, eₙ)
     Q = SORT(Q) //ordina in base al peso crescente
     for i = 1 to n
          if S∪{Q[i]} ∈ F
               S = S∪{Q[i]}
     return S
```

Proposizione associata

Se <E,F> è un **matroide** allora, $\forall\, w : E \longrightarrow R^{+}$, l'algoritmo `GREEDY_MIN(E,F,W)` calcola una **soluzione al problema di minimo associato** a <E,F> e W.

Dimostrazione

Sia F un matroide, $\forall\, A,B \in F$, se A e B massimali, si ha che $|\mathbf{A}| = |\mathbf{B}|$.

Questo implica che **tutti i sottoinsiemi massimali hanno la stessa cardinalità h**.

Cerco arbitrariamente un peso $w : E \longrightarrow R$ definita come

$$w'(e) = k - w(e), \text{ dove } k > max_{(e \in E)} \{w(e)\}$$

Sia $A \in F$ un massimale di <E,F>, allora

$$w'(A) = \sum_{x \in A} w'(x) = \sum_{x \in A} (k - w(x)) = \sum_{x \in A} k - \sum_{x \in A} w(x) = |A| * k - w(A) = h*k - w(A)$$

$$\text{quindi } w'(A) = h*k - w(A)$$

h*k è costante, quindi per aumentare w'(A) bisogna diminuire w(A).

$w(A)$ è **minimo** se e solo se **w'(A) è massimo**. Quindi, calcolare un massimale $A \in F$ che minimizza w(A) è equivalente a trovare un massimale $A \in F$ che massimizza w'(A).

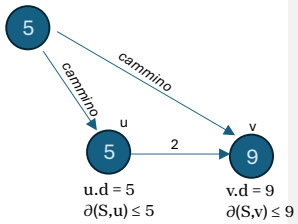
$$S \in F \text{ è massimale, con}$$

$$w(S) = \min_{(A \in F,\, A \text{ massimale})} \{w(A)\} = \max_{(A \in F,\, A \text{ massimale})} \{w'(A)\}$$

L'algoritmo GREEDY_MAX(E,F,W') risolve il problema max{w'(A)}, in quanto <E,F> è un matroide. `GREEDY_MAX(E,F,W') = GREEDY_MIN(E,F,W)`.

Tecnica del rilassamento

Il rilassamento di un arco consiste **nell'aggiornare la distanza di un vertice** v se si trova un **cammino più corto** passando per un arco (u,v). La distanza v.d è aggiornata solo se la nuova distanza u.d + w(u,v) è **minore** del valore attuale di v.d.




```
RELAX(u,v,w)
if v.d > u.d + w(u,v)
     v.d = u.d + w(u,v)
     v.π = u
```

Proprietà di convergenza

Sia S → U → V un cammino minimo da S a V, e sia (U,V) un **arco che viene rilassato**. Se, prima del rilassamento, U.d = ∂(S,U), allora, dopo il rilassamento, V.d = ∂(S,V).

Dimostrazione

Dopo il rilassamento abbiamo:

$$V.d \leq U.d + w(U,V) = \partial(S,U) + w(U,V) = \partial(S,V)$$

Per la **proprietà del limite superiore**, sappiamo che $V.d \geq \partial(S,V)$. Quindi, diventa $V.d = \partial(S,V)$.

Inizializzazione delle distanze e dei predecessori

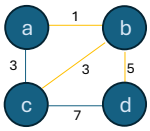
Serve a impostare le condizioni iniziali per calcolare le distanze minime dal nodo sorgente s agli altri nodi del grafo G.

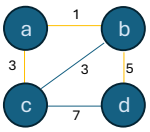
1. Inizializza le distanze v.d: $\forall$ nodo $v \in V$ imposta $v.d = \infty$. Inizialmente non conosciamo alcun cammino che collega il nodo sorgente s al nodo v
2. Per il **nodo sorgente**, imposta s.d = 0
3. **Imposta i predecessori**: per ogni nodo $v \in V$ imposta $v.\pi$ = NIL

```
for each v ∈ V
     v.d = ∞
     v.π = NIL
s.d = 0 [nota: s.d = ∂(s,s)]
```

Minimum spanning tree

Dato un grafo $G = (V,E)$ non orientato e connesso, e $w : E \longrightarrow R$, un **minimum spanning tree** è un sottoinsieme di archi (V,T) con $T \subseteq E$, che è *aciclico*, che *connette tutti i nodi in V*, e (v,t) **minimizza il peso** totale tale che $w(T) = min_{(A \subseteq E,\ (V,A)\ non\ connessa)}\ \{w(A)\}$. Di seguito, un esempio.





$T = \{(a,b), (b,c), (b,d)\}$

$T = \{(a,c), (b,a), (b,d)\}$

Per il grafo G = (V,E) non orientato, definiamo $F = \{A \subseteq E \mid (V,A)\ \text{aciclico e foresta}\}$; allora $<E,F>$ è un matroide. Ha le seguenti proprietà:

- **Aciclico e connesso**: un sottoinsieme $A \subseteq E$ è un albero se è sia aciclico sia connesso
- **Massimale e aciclico**: un sottoinsieme A massimale è un albero, e vale $|A| = |V| - 1$

Un albero è:

- Un **grafo connesso** con $|A| = |V| - 1$
- Un grafo **connesso e aciclico** con $|A| = |V| - 1$
- Un grafo **aciclico e massimale** rispetto alla proprietà di aciclicità

Algoritmo

Dato un minimum spanning tree (V,T), $T \subseteq E$ il quale è aciclico e connesso, (V,T) è una foresta, $T \in F$, l'algoritmo `GREEDY_MIN(E,F,W)` **calcola un minimum spanning tree seguendo questi passaggi**:

- Inizializza $S = \emptyset$
- Ordina gli archi Q in ordine crescente di peso
- $\forall$ arco $(u,v) \in E$:
  - Se $S \cup \{(u,v)\}$ non forma un ciclo, allora $S = S \cup \{(u,v)\}$
- Restituisci S

```
GREEDY_MIN(E,F,W')
S = Ø
Q = (q₁, …, qₙ}
Q = SORT(Q)
for each (u,v) ∈ E
     if S ∪ {(u,v)} ∈ F
          S = S ∪ {(u,v)}
```

```
if FIND_SET(u) ≠ FIND_SET(v)
     UNION(u,v)
```

Questo algoritmo ha costo $O(|E| \log|E|)$.

Algoritmo Kruskal

Fa crescere $S \subseteq E$ in modo tale che (V,S) **è una foresta**, e $S \subseteq T$, T è un MST. L'arco (u,v) che si aggiunge a S collega **u**, che è in una componente connessa, a **v** ,che è in un'altra componente connessa (una componente connessa è un albero).

Quindi, costruisce un MST aggiungendo iterativamente gli archi con peso minimo. Durante ogni iterazione, *l'arco aggiunto connette due componenti connesse del grafo, trasformandole in una singola componente*.

È quindi un esempio di algoritmo greedy che costruisce un albero di copertura minimo. Si può vedere anche come una **specializzazione di un algoritmo generico per MST**, dove l'arco che viene aggiunto all'insieme di archi S è il *peso minimo tra quelli che connettono due componenti connesse di (V,S)*, dove V è l'insieme di vertici del grafo. La **condizione** che deve essere rispettata è che **l'arco (u,v) deve essere sicuro per S**.

Dimostrazione del prof

Basta scegliere $(V_c, V - V_c)$ come taglio; esso rispetta S.

Un arco di peso minimo che collega C ad un'altra componente connessa di (V,S) è un arco leggero che attraversa il taglio?

(V,S) è sempre una foresta, ma, a differenza di Kruskal, (V', S) è un albero dove V' è formato dai vertici di S. L'algoritmo fa crescere l'albero, ovvero che *incrementa V' di un vertice e S di un arco*.

Viene aggiunto ad S un arco di peso minimo tra un vertice che esiste nell'albero, e un vertice che non esiste nell'albero.

Un tale arco è **(u,v)** con $\mathbf{U \in V'}$, $\mathbf{v \in V \setminus V'}$, quindi attraversa il taglio $(U', V \setminus V')$. Siccome $(V', V \setminus V')$ rispetta S, (u,v) è il più leggero, **(u,v) è sicuro** e quindi $S \cup \{(u,v)\} \subseteq$ di un MST.

$$(V', S) \rightarrow (V' \cup \{v\}, S \cup \{(u,v)\})$$

Arco sicuro per un insieme

Sia (V,E) non orientato e connesso, $w : E \longrightarrow R^+$, siano $T \subseteq E$ tale che (V,T) è un MST.

Sia $S \subseteq T$ e (V,S) una foresta.

Un arco $(u,v) \in E$ è detto **sicuro** per S, se $S \cup \{(u,v)\} \subseteq T'$, dove (V, T') è un MST. Ovvero, se si aggiunge un altro arco a S, esso è ancora un sottoinsieme di T' (ovvero è contenuto nel MST o è al massimo l'MST stesso).

Dato un grafo non orientato e connesso $G = (V,E)$, un **albero di copertura** (*spanning tree*) è un sottografo (V, T) dove $T \subseteq E$, che soddisfa queste condizioni:

- (V, T) è connesso, ovvero *esiste un cammino tra qualsiasi coppia di vertici*
- (V, T) non contiene cicli
- Include tutti i vertici di V

```
GENERIC_MST(g)
     S = Ø
     while “S non forma un albero di copertura”
          trova un arco (u,v) sicuro per S
          S = S ∪ {(u,v)}
     return S
```

Teorema di Frattalis Francescanius

Sia $G = (V,E)$ non orientato e connesso, $w : E \longrightarrow R^{+}$. Abbiamo $S \subseteq T$ con (V,T) un MST.

$C = (V_c, E_c)$ componenti connesse in una foresta (V,S), sia (u,v) di peso minimo che collega C ad un’altra componente connessa di (V,S). (u,v) è un arco sicuro per S.

Secondo teorema

Se in un grafo consentito connesso non orientato $G = (V,E)$ con una funzione di peso $w: E \rightarrow R$, si considera un albero di copertura minimo T; ogni arco che attraversa un taglio $(V’, V \setminus V’)$ con il peso minimo è sicuro per T.

Dimostrazione

Consideriamo il grafo $G = (V,E)$ non orientato e connesso, e una funzione di peso $w: E \rightarrow R$. Sia (V,T) un MST, e $A \subseteq T$.

Consideriamo un taglio $(V’, V \setminus V’)$ dove A non attraversa il taglio, e un arco (u,v) che attraversa il taglio con peso minimo.

Dobbiamo dimostrare che l’**aggiunta di (u,v) ad A non riduce il peso complessivo** dell’albero di copertura.

Immaginiamo che (u,v) non sia parte di T; se così fosse, ci sarebbe un ciclo che si forma aggiungendo (u,v) a T.

Poiché (u,v) attraversa il taglio esiste un arco (x,y) in T che attraversa lo stesso taglio **ma che non è in A**. Rimuovendo (x,y) e aggiungendo (u,v), otteniamo un nuovo albero **T’** con un peso che è minore o uguale a quello. Pertanto, **(u,v) è sicuro per T**.

Spiegazione del prof

Abbiamo un grafo $G = (V,E)$ non ordinato, con $w : E \rightarrow R$, (V,T) è un MST con $A \subseteq T$. Abbiamo un taglio $(V', V \setminus V')$.

$(u,v) \in E$ attraversa il taglio se $U \in V'$, $v \in V \setminus V'$ (o viceversa), con il taglio dell'esempio.

(V,E) non orientato e connesso, $w : E \rightarrow R$, (V,T) è un MST, $A \subseteq T$. Abbiamo un taglio $(V', V \setminus V')$ che rispetta A, un arco (u,v) leggero che attraversa il taglio; (u,v) è sicuro per A.

Bisogna quindi far vedere che $\mathbf{A \cup \{(u,v)\} \subseteq T'}$ con **(V,T')** un MST; assumiamo che $(u,v) \notin T$.

In T c'è un cammino p semplice da u a v, e si forma un ciclo aggiungendo (u,v) a T.

$U \in V'$, $v \in V \setminus V'$ => esiste $(x,y) \in T$ con $(x,y) \in p$, (x,y) attraversa il taglio; $(X,y) \notin A$ perché il taglio rispetta A.

Eliminando (x,y) da T, T si spezza in due componenti; aggiungendo (u,v) si ricompongono formando un albero di connessione T':

$$T' = T \setminus \{(x,y)\} \cup \{(u,v)\}$$

Facciamo vedere che (V,T') è un MST; essendo (u,v) leggero abbiamo $w(u,v) \leq (x,y)$; quindi:

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

Essendo T MST abbiamo che $w(T) \leq w(T')$; pertanto otteniamo $w(T') = w(T)$, e quindi anche **(V,T') è un MST**.

Resta da dimostrare che $A \cup (u,v)\} \subseteq T'$; $A \subseteq T'$ poiché $A \subseteq T$, $(x,y) \notin A$; il taglio rispetta A, e (x,y) lo attraversa. Pertanto, $A \cup \{(u,v)\} \subseteq T'$.

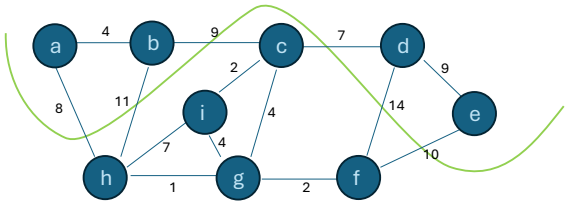
Taglio

Un **taglio** è una partizione $(V', V \setminus V')$ di V, con $V' \subset V$. Un arco $(u, v) \in E$ attraversa il taglio se:

$$u \in V' \text{ e } v \in V \setminus V', \text{ oppure viceversa.}$$

Un arco è detto **leggero** se ha peso minimo tra tutti gli archi che attraversano il taglio.

Un taglio rispetta $S \subseteq E$ se nessun arco di S attraversa il taglio. Di seguito, un **esempio**:



(a,h), (h,b), (c,b), (c,d), (d,f), (e,f) **attraversano il taglio**

(c,d) è l'arco **leggero**

Algoritmo di Prim

È un algoritmo greedy per la costruzione di un MST. Prim cresce l'albero di copertura aggiungendo un vertice alla volta.

Inizialmente (V,S) è una foresta. Tuttavia, a differenza dell'algoritmo di *Kruskal*, che costruisce il MST aggiungendo archi e unendo componenti connesse, in **Prim** (V', S) è sempre un **albero singolo** che cresce progressivamente.

V' contiene i vertici dell'albero in crescita, e S contiene gli archi selezionati. L'algoritmo fa crescere l'albero incrementando V' di un vertice, e S di un arco.

Viene aggiunto ad S un arco di peso minimo tra un vertice che appartiene all'albero, e un vertice che non appartiene all'albero. Un tale arco è (u,v), dove $U \in V'$, $v \in V \setminus V'$, quindi *attraversa il taglio* $\{V', V \setminus V'\}$.

Siccome $\{V', V \setminus V'\}$ rispetta S, **(u,v) è leggero**; (u,v) è sicuro, $S \cup \{(u,v)\} \subseteq$ di un MST.

1. Si inizia con un insieme vuoto di vertici V' = Ø, e un albero S vuoto
2. Ogni volta, si sceglie **l'arco di peso minimo che collega un vertice in V' a un vertice in V \ V'**; questo arco viene aggiunto all'albero S, e il vertice collegato viene aggiunto a V'.
3. L'algoritmo continua fino a che tutti i vertici sono stati aggiunti all'albero

Dimostrazione del prof

Per individuare (u,v) $\forall v \in V \setminus V'$, v.key è il minimo valore tra i pesi degli archi che collegano v a vertici dell'albero (V', S), ∞ se non vi sono archi.

$\forall v \in V \setminus V'$, si sceglie il vertice con valore minore, quindi un **min-heap** (coda con priorità) contenente $V \setminus V'$. Definiamo v.π come il predecessore di v in (V,T), che serve per incrementare V' → $V \setminus V' \cup \{v\}$.

```
MST-PRIM(G, w, r), r ∈ V qualunque
for each u ∈ V
     u.key = ∞
     u.π = NIL
r.key = 0 //primo vertice
Q=V, V'=Ø all'inizio, quindi V\V'=Q
//Q = mini heap da cui bisogna estrarre gli z
while Q ≠ Ø
     v = EXTRACT-MIN(Q) //estrai il v con la key minore
```

Commentato [FD11]: R sarebbe il vertice inziale nella costruzione dell'MST?

Commentato [FD12]: Qui chatgpt mi dice che è R.key = 0, proprio perché essendo il primo vertice non ha predecessori

Commentato [FC13R12]: Lasciamolo cosi per ora, chiedo in giro

Teo e il besozzi sul gruppo mi hanno detto r.key, sicuramente loro ne sanno di più

```
for each z ∈ Adj[v] //adiacenti
     //Se un arco offre un percorso più economico
     //per aggiungere z all'albero, si aggiorna
     if z ∈ Q AND w(z,v) < z.key
          z.π = v
          z.key = w(z,v) //key = peso
          DECREASE(Q, z, w(z,v))
```

Algoritmo di Dijkstra

È usato per **trovare il cammino minimo da un vertice sorgente s a tutti gli altri vertici di un grafo orientato e pesato** con pesi non negativi.

Dato un grafo orientato pesato con pesi non negativi, la sorgente $s \in V$.

Istanza: $G = (V,E)$ orientato, $w : E \rightarrow R_+$, $s \in V$.

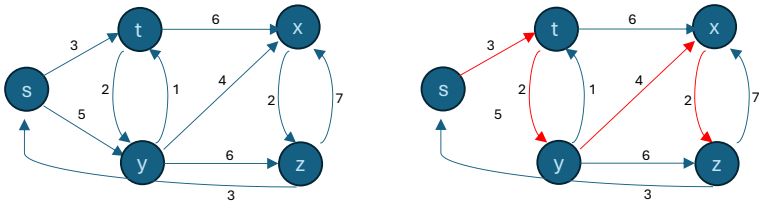
Soluzione: $\forall\ v \in V$, $\partial(s,v)$ = *peso di un cammino minimo da s a v*, si costruirà quindi un albero di cammini minimi con radice s.

Abbiamo $G' = (V', E')$ con $V' \subseteq V$, $E' \subseteq E$ tale che:

- V' è un insieme di vertici raggiungibili da s
- G' è un albero con radice s
- $\forall\ v \in V'$ l'unico cammino da s a v in G' è un cammino minimo da s a v

1. Si inizializza la distanza di ogni vertice a ∞, tranne quella del **vertice sorgente** che viene settata a 0 (come in `INITIALIZE-SINGLE-SOURCE`)
2. Per ogni vertice, eseguiamo il *rilassamento degli archi*. Se il cammino trovato passando per un arco ha un peso inferiore a quello precedentemente registrato per un vertice, aggiorniamo il valore della distanza
3. L'algoritmo continua *selezionando il vertice con la distanza più bassa non ancora visitato*, e rilassando tutti gli archi uscenti da questo vertice
4. La complessità dell'algoritmo è $O(|V| \log|V| + |E| \log|V|)$, grazie all'uso della min-heap per l'estrazione del vertice con la distanza minima

Un esempio grafico:



Dimostrazione

Dopo ogni passo di rilassamento, la distanza $v.d$ di ogni vertice v rappresenta la distanza più corta possibile da s a v. Questo è vero per ogni passo dell'algoritmo, grazie al fatto che **rilassiamo gli archi in modo sistematico**, partendo dai vertici con la distanza minima. Ogni volta che un vertice v è visitato, la sua distanza $v.d$ è uguale al cammino minimo da s a v, e non cambierà più in seguito, poiché rilassamenti successivi non potranno ridurre il valore di $v.d$.

Algoritmo

Mantiene $V' = \{v \in V \mid \partial(s,v).= v.d\}$. Seleziona ripetutamente $u \in V \setminus V'$, con stima minima. Abbiamo $V \setminus V'$ MIN-HEAP! $Q = V \setminus V'$; $V' \rightarrow V' \cup \{u\}$, rilassa gli archi uscenti da u.

```
DIJKSTRA(G, w, s)
INITIALIZE-CS(G,s)
V' = Ø
Q = V
while Q ≠ Ø
     u = EXTRACT-MIN(Q)
     V' = V' ∪ {u}
     for each v ∈ Adj[u] //adiacenti
          RELAX(u,v,w)
          if v.d è diminuito
               DECREASE(Q,v,v.d)
```

Dimostrazione con limite superiore

Questa dimostrazione giustifica la correttezza dell'algoritmo di Dijkstra in termini della **proprietà del limite superiore delle distanze**, indicate come $v.d$, che vengono aggiornate tramite rilassamenti durante l'algoritmo.

L'obiettivo è di **dimostrare che durante l'esecuzione dell'algoritmo, per ogni vertice v del grafo**:

1. $v.d \geq \partial(s,v)$, dove $\partial(s,v)$ è la distanza minima da s a v
2. Quando $v.d = \partial(s,v)$, il valore di v.d non cambia più nei passi successivi
3. $v.d$ non può mai scendere sotto $\partial(s,v)$, né aumentare una volta raggiunto $\partial(s,v)$

Caso base

$\mathbf{m} = \mathbf{0}$; dopo l'inizializzazione:

- Per il nodo sorgente s, $s.d = 0$, che è uguale a $\partial(s,s) = 0$. $s.d \geq \partial(s,s)$ è **vero**.

- Per tutti gli altri vertici $v \neq s$, $v.d = \infty$ che è certamente maggiore o uguale alla distanza minima da s a v => $\partial(s,v)$

La proprietà, quindi, vale per $m = 0$.

Passo induttivo

Assumiamo che dopo m passi di rilassamento, valga che $\forall\ x \in V$, $x.d \geq \partial(s,x)$. Bisogna dimostrare che la proprietà vale anche per $\mathbf{m+1}$.

Si rilassa un arco (u,v). L'operazione di rilassamento consiste nel verificare se $v.d > u.d + w(u,v)$. Se si, aggiorniamo $v.d = u.d + w(u,v)$.

1. **v.d non cambia dopo il rilassamento**
   Se v.d non cambia, l'attuale valore di v.d è già minore o uguale a $u.d + w(u,v)$. Poiché abbiamo $u.d \geq \partial(s,u)$, abbiamo $v.d \geq \partial(s,v)$. **La proprietà rimane valida**.

2. **v.d cambia dopo il rilassamento**
   v.d viene aggiornato a $u.d + w(u,v)$. bisogna verificare che il nuovo valore di v.d sia ancora maggiore o uguale a $\partial(s,v)$.
   Con la definizione di rilassamento abbiamo $v.d = u.d + w(u,v)$
   Per ipotesi induttiva sappiamo che $u.d \geq \partial(s,u)$. Quindi:
   $$v.d = u.d + w(u,v) \geq \partial(s,u) + w(u,v)$$
   Abbiamo $\partial(s,u) + w(u,v) \geq \partial(s,v)$, perchè la distanza minima da s a v deve passare attraverso qualche cammino. Quindi $v.d \geq \partial(s,v)$.

Con $v.d = \partial(s,v)$ il valore di v.d non può più cambiare:

1. La distanza v.d non può mai diventare inferiore a $\partial(s,v)$, perché il rilassamento mantiene sempre $v.d \geq \partial(s,v)$
2. Una volta che $v.d = \partial(s,v)$, non può aumentare, poiché il rilassamento può solo ridurre il valore di v.d e non aumentarlo
3. Quindi, mantiene sempre il limite superiore sulle distanze fino a trovare quelle minime

Correttezza

L'algoritmo di Dijkstra termina con $U.d = \partial(S,U)\ \forall\ U \in V$, dove $\partial(S,U)$ rappresenta la distanza minima tra S e U.

È sufficiente dimostrare che $U.d = \partial(S,U)$ al momento in cui U viene aggiunto a V', l'insieme dei vertici già processati.

Assumiamo per assurdo che esista un vertice $U \in V$ tale che $U.d \neq \partial(S,U)$ quando U viene aggiunto a V'.

Sia U il vertice estratto da Q, la coda di priorità, e aggiunto a V' tale che $U.d \neq \partial(S,U)$.

- Poiché $S.d = \partial(S,S) = 0$, è evidente che $U \neq S$
- Quando U è estratto, $V \neq \emptyset$, ovvero almeno S è già in V'.
- Deve esistere un cammino da S a U, altrimenti $\partial(S,U) = \infty$, il che implicherebbe $U.d = \infty$, contraddicendo l'ipotesi $U.d \neq \partial(S,U)$.

Esiste dunque un **cammino minimo p da S a U**. Prima di aggiungere U a V', il cammino p collega $S \in V'$ a $U \in V \setminus V'$.

Sia Y il primo vertice lungo p tale che $Y \in V \setminus V'$, e sia X il predecessore di Y. Dalle proprietà del rilassamento:

- Quando X è stato aggiunto a V', vale $X.d = \partial(S,X)$
- Successivamente, l'arco (X,Y) è stato rilassato, quindi $Y.d = \partial(S,Y)$

Poiché Y precede U nel cammino minimo p, vale $\partial(S,Y) \leq \partial(S,U)$. Pertanto, $Y.d = \partial(S,Y) \leq \partial(S,U)$ (**).

Quando U viene estratto da Q, abbiamo $U.d \leq Y.d$, poiché U è il vertice con distanza minima in Q. Combinando con (**), possiamo sostituire le disuguaglianze con uguaglianze, ottenendo

$$U.d = \partial(S,U)$$

Questo contraddice l'ipotesi iniziale $U.d \neq \partial(S,U)$. Quindi l'algoritmo di Dijkstra è corretto.

Complessità

Con array semplice:

- Ogni aggiornamento (RELAX) o inserimento in Q: O(1)
- Operazione di estrazione `EXTRACT_MIN`: $O(|V|)$
- Complessità totale: $O(|V|) + O(|E|) + O|V|^2) = O|V|^2 + |E|)$ poiché $|E| < |V|^2$

Con min-heap

- Inserimento e aggiornamento: $O(\log |V|)$
- Complessità totale: $O((|V| + |E|) \log |V|) = O(|E| \log |V|)$

Teoria della complessità computazionale

Formula booleana

Una formula booleana ϕ è costruita utilizzando n variabili booleane $x_1, \ldots, x_n$, e m connettivi logici, quali $\wedge$, $\vee$, $\neg$, $\rightarrow$, $\leftarrow\rightarrow$, e parentesi.

Un'**assegnazione di verità** consiste nell'assegnare un valore $\{0,1\}$ a ciascuna variabile booleana; una formula ϕ è **soddisfacibile** se esiste almeno un'assegnazione di verità che la rende vera, quindi $\phi = 1$.

Di seguito, un esempio. Abbiamo $\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_2) \vee x_4)) \wedge \neg x_2$.

Un'assegnazione possibile che soddisfa ϕ è: $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 1$.

Problema 3-CNF-SAT

Una formula è in 3-CNF se ogni disgiunzione (delimitata da parentesi) contiene **al massimo 3 letterali**, ad esempio:

$$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_4 \vee x_5 \vee x_6)$$

Problema

Dobbiamo verificare se una formula booleana in 3-CNF è **soddisfacibile**, ovvero se **esiste un'assegnazione di verità per le variabili che rende la formula vera**.

Formalmente, abbiamo come **input** una formula booleana ϕ in 3-CNF. Come **output** abbiamo **1** se ϕ è soddisfacibile, 0 altrimenti.

3-CNF-SAT è completo

Per dimostrare che il problema 3-CNF-SAT (il problema della soddisfacibilità di una formula booleana 3-CNF) è completo, dobbiamo dimostrare che:

- **3-CNF-SAT è in NP**: data un'assegnazione delle variabili, possiamo verificare in tempo polinomiale se soddisfa la formula
- **SAT $\leq_p$ 3-CNF-SAT:** possiamo ridurre (vedi "riduzioni") una formula arbitraria in CNF a una formula in 3-CNF in tempo polinomiale, preservando la soddisfacibilità. Questo implica che **il problema SAT si riduce al problema 3-CNF-SAT**.

Di conseguenza, il problema 3-CNF-SAT è **NP-completo**.

Seconda parte 3-CNF

Questa parte è stata fatta dopo in un'altra lezione.

Una formula booleana ϕ è costruita utilizzando variabili booleane connettivi logici e parentesi. Una formula è in CNF (formula congiuntiva) se è **espressa come una congiunzione AND di clausole**, dove ciascuna clausola è una **disgiunzione OR di letterali**.

Commentato [FC14]: Da qua fino alla fine del vertex cover non ho praticamente la minima idea di cosa sia, perché il prof l'ha fatta estremamente veloce, e come vedi, molto complessa

Una formula è in forma 3-CNF se *ogni clausola ha esattamente 3 letterali distinti*. Un esempio:

$$\phi = (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee v_3)$$

Problema 3-CNF-SAT

È definito come segue:

- **Input**: Q: $I \rightarrow \{0,1\}$, dove I è l'insieme delle formule ϕ in forma 3-CNF
- **Output**: 1 se ϕ è soddisfacibile, cioè se esiste un'assegnazione delle variabili booleane che rende vera la formula; 0 altrimenti

NP-completezza

3-CNF-SAT appartiene a NP

Dato $\phi \in I$, $\text{con } Q(\phi) = 1$, un **certificato** è l'assegnazione y dei valori booleani alle variabili di ϕ *(che restituiscono Q(ϕ) = 1)*. L'algoritmo A verifica che y rende vera ϕ *(A(ϕ,y) = 1)*, sostituendo i valori nelle variabili e calcolando il valore di ϕ. Questo processo richiede tempo polinomiale rispetto alla lunghezza di ϕ, e l'assegnazione y ha lunghezza $O(|x|)$, dove x è il numero di variabili.

Riduzione da SAT a 3-CNF-SAT

La riduzione da SAT a 3-CNF-SAT trasforma una formula arbitraria ϕ in SAT in una formula equivalente ϕ in 3-CNF, preservando la soddisfacibilità e lavorando in tempo polinomiale.

Per dimostrare che 3-CNF-SAT è NP-completo, dobbiamo definire una funzione $f : I \rightarrow I'$, calcolabile in tempo polinomiale, tale che $SAT(\phi) = 1 \leftrightarrow 3\text{-CNF-SAT}(f(\phi)) = 1$.

Per trasformarlo, seguiamo i seguenti passaggi:

1. Costruzioni del parsing tree per ϕ: ogni foglia rappresenta un letterale, mentre i nodi interni rappresentano i connettivi logici
2. Introduzione di variabili y_i per rappresentare l'output di ogni nodo interno
3. Riscrittura di ϕ come una congiunzione tra la variabile associata alla radice y_1 e le clausole che descrivono l'operazione di ciascun nodo

$$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$$

$$\phi' = y_1 \wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \wedge \ldots$$

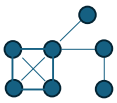
Cricca

Dato G = (V,E) non orientato, una cricca è un **sottoinsieme V' ⊆ V di vertici ogni coppia dei quali è collegato da un arco in E**, ossia ∀ v e v' ∈ V' con v' ≠ v esiste (v, v') appartenente ad E, ossia:

(V', Ev') è un sottografo completo di G, |V'| è la dimensione della cricca.

Di seguito un esempio.

Grafo non orientato completo



Grafo non orientato che contiene una cricca di dimensione 4

Come problema di decisione abbiamo I = {(G,k), G grafo non orientato, k ∈ N con k > 1}, S = {0,1}, con ∀(G,k) ∈ I, CLIQUE(G,k) = 1 se e solo se esiste una cricca di dimensione k.

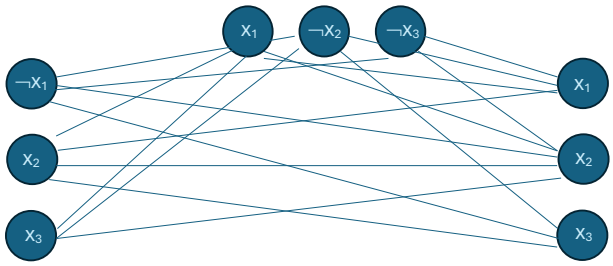
NP-completezza

1. Mostriamo che **CLIQUE ∈ NP**, ossia **è verificabile in tempo polinomiale**
   a. ∀(G,k) ∈ I tale che CLIQUE(G,k) = 1 esiste **un certificato y** dove y = V' è la cricca di dimensione k in G
   b. L'algoritmo A verifica che y = V' è una cricca controllando che, **∀ v e v' ∈ V, v ≠ v', (v, v') appartenente ad E**
      i. A controlla che |V'| = k; la verifica è svolta in tempo polinomiale in n = |(G, k)|.
      ii. Inoltre, |y| = |V'| = k = O(n)
2. Dimostriamo che 3-CNF-SAT $\leq_p$ CLIQUE
   a. Siccome 3-CNF-SAT è NP-COMPLETO, si conclude che anche CLIQUE è NP-COMPLETO; occorre mostrare che **esiste f : I → I' calcolabile in tempo polinomiale**, dove I e I' sono gli insiemi delle istanze di 3-CNF-SAT e CLIQUE, tale che ∀ϕ ∈ I, 3-CNF-SAT(ϕ) = 1 ←→ CLIQUE(f(ϕ)) = 1
   b. Occorre definire f, ossia ∀ϕ ∈ I, f(ϕ) ∈ I' (f(ϕ) = ….); sia dunque ϕ ∈ I e una **formula booleana** in forma 3-CNF.
      i. ϕ = $C_1 \wedge C_2 \wedge \ldots \wedge C_k$ con k clausole, una coppia (G,k) tale che 3-CNF-SAT(ϕ) = 1 ←→ CLIQUE(f(ϕ)) = 1.
      ii. ∀r = 1, …, k, C_r contiene esattamente 3 letterali distinti (l_1^r, l_2^r, l_3^r), C_r = ($l_1^r \vee l_2^r \vee l_3^r$)
      iii. Costruiamo (G,k) tale che è uguale a f(ϕ). k è proprio il numero di clausole di ϕ. 3-CNF-SAT(ϕ) = 1 ←→CLIQUE(G, ϕ) = 1, dove nella prima *ϕ è soddisfacibile*, e nella seconda *in G c'è una cricca V' con |V'| = k*.

iv. $\forall C_r = (l_1^r \vee l_2^r \vee l_3^r)$ in ϕ, inseriamo v_1^r, v_2^r, v_3^r in V. $(v_i^r, v_j^s) \in E$ se e solo se valgono le seguenti condizioni:
    1. v_i^r e v_j^s si trovano in triple distinte, cioè $r \neq s$
    2. l_i^r non è la negazione di l_j^s
v. G può essere calcolato a partire da ϕ in tempo polinomiale nella dimensione di ϕ. Per fare questo, occorre **riempire una matrice $3k$ x $3k$**.
vi. Di seguito, un esempio:

$\phi = (x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3)$

$C_1 = x_1 \vee \neg x_2 \vee \neg x_3$; $C_2 = \neg x_1 \vee x_2 \vee x_3$; $C_3 = x_1 \vee x_2 \vee x_3$



Essenzialmente: colleghi x_1 a x_2 e x_3, x_2 a x_1 e x_3, ovvero non a sé stesso. Inoltre, non sono collegati ai loro complementari (es. colleghi x_2 e x_2 se sono di clausole diverse; mentre non collegheremo mai x_2 e $\neg x_2$).

c. Dimostriamo che ϕ è soddisfacibile se e solo se ha una cricca di dimensione K
    i. Sia ϕ soddisfacibile, allora $\forall r\ C_r$ contiene un letterale l_i^r a cui è assegnato valore 1, e l_i^r è associato a v_i^r.
    ii. $\forall r = 1, \ldots, k$ selezioniamo 1 vertice v_i^r associato ad un letterale in C_r di valore 1. Otteniamo un insieme V' di K vertici.
    iii. Mostriamo che V' è una cricca
        1. Bisogna mostrare che $\forall (v, v') \in V, v \neq v'$ si ha $(v, v') \in E$.
        2. Scelti $v, v' \in V'$ con $v \neq v'$, necessariamente $v = v_i^r$ e $v' = v_j^s$ con $r \neq s$. Inoltre, ai corrispondenti l_i^r e l_j^s è assegnato valore 1 dall'assegnazione di verità per ϕ; pertanto, **l_i^r non è la negazione di l_j^s**.
        3. Pertanto, $(v_i^r, v_i^s) \in E$, quindi V' è una CLIQUE di dimensione K, con $\text{CLIQUE}(G, k) = 1$
    iv. Supponiamo ora che $\text{CLIQUE}(G, k) = 1$, ossia G ha una cricca V' di dimensione K, e nessun arco in G collega vertici della stessa tripla

1. Quindi V' contiene un vertice per ogni tripla, e V' deve avere k vertici
2. Possiamo assegnare 1 ad ogni letterale l_i^r tale che $v_i^r \in V'$ senza temere di assegnare 1, sia ad un letterale e alla sua negazione
3. Pertanto, ogni clausola di ϕ è soddisfatta; **ad ogni variabile che non corrisponde ad un vertice nella cricca si può assegnare un valore arbitrario**.

Vertex cover

Dato un grafo non orientato $G = (V,E)$, un **vertex cover** è un sottoinsieme di vertici $V' \subseteq V$ tale che *ogni arco di G è incidente ad almeno un vertice in V'*.

Quindi, dato $I = \{(G,K) \mid G \text{ non orientato}, K \in N \text{ con } K \geq 1\}$, $S = \{0,1\}$. $\forall (G,K) \in I$ VERTEX-COVER$(G,K) = 1$ se e solo se **esiste una copertura di G di dimensione K**.

Copertura di vertici di un grafo non orientato

Sia $G = (V,E)$ un grafo non orientato; una copertura di vertici di G è **$V' \subseteq V$ tale che $\forall (u,v) \in E$** si ha che **$u \in V'$ e/o $v \in V'$**. Ciascun vertice copre i suoi archi incidenti, e una copertura di G è un insieme di vertici che copre tutti gli archi.

$|V'|$ è detta la dimensione della copertura V'.

Complemento di un grafo non orientato

Sia $G = \{V,E\}$ un grafo non orientato; il complemento di G è il grafo

$$\underline{G} = (V,E^*) \text{ dove } E^* = \{(u,v) \mid u,v \in V, u \neq v, (u,v) \notin E\}$$

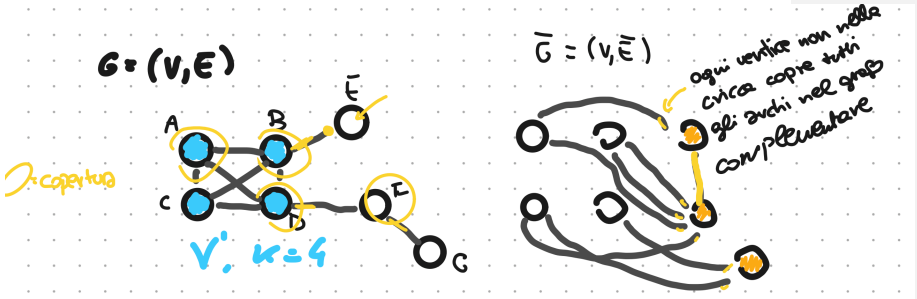
Quindi, E^* contiene tutti gli archi che non erano presenti in E, ma che potrebbero esistere dato che $u \neq v$.

NP-completezza

1. Mostriamo che **VERTEX-COVER è NP**, cioè verificabile in tempo polinomiale
   a. $\forall (G,K) \in I$, VERTEX-COVER$(G,K) = 1$, il **certificato** è $y = V'$, dove V' è la **copertura di G di dimensione k**. L'algoritmo di verifica sarà un algoritmo A che controlla che $|V'| = k$, e poi controlla, *per ogni arco $(u,v) \in E$ che $u \in V'$ e/o $v \in V'$*.

b. Tale verifica può essere svolta in **tempo polinomiale** in $n = |(G,K)|$; inoltre $|y| = |V'| = k = O(n)$

2. Dimostriamo che **CLIQUE** $\leq_p$ **VERTEX-COVER**, CLIQUE = I e VERTEX-COVER = I'
   a. Siccome CLIQUE è NP-DIFFICILE, discenderà che VERTEX-COVER è NP-DIFFICILE
      i. Vogliamo dimostrare che **esiste f : I → I' calcolabile in tempo polinomiale** tale che $\forall(G,k) \in I$ CLIQUE(G,k) = 1 se e solo se VERTEX-COVER(f(G,k)) = 1
   b. $\forall(G,k) \in I$, $f(G,k) = (\underline{G}, |V| - k)$, dove $\underline{G} = (V,\underline{E})$ è il **complemento di G**.
   c. Dato $(G,k) \in I$ $f(G,k)$ si costruisce in tempo polinomiale nella $|(G,k)|$.
3. Dimostriamo che **$\forall(G,k) \in I$ CLIQUE(G,k) = 1 ←→ VERTEX-COVER(f(G,k)) = 1**
   a. G ha una cricca di dimensione k ←→ $\underline{G}$ ha una copertura di vertici di dimensione $|V| - k$
   b. "→": Sia $(G, k) \in I$, e $f(G, k) = (\underline{G}, |V| - k)$. Supponiamo che G abbia una **cricca V'** di dimensione k.



      i. Mostriamo che **V \ V' è una copertura di vertici** di $\underline{G} = (V, \underline{E})$ ($\underline{E} = E^*$).
         1. Sia $(u,v) \in \underline{E}$, quindi $(u,v) \notin E$
         2. Siccome V' è una cricca, almeno uno tra u e v non appartiene a V'; *se entrambi appartenessero a V', allora si avrebbe $(u,v) \in E$.*
            a. Avendo (u,v) nella cricca, vuol dire che entrambi dovrebbero essere collegati (perché i vertici in una cricca devono essere collegati tra di loro con tutti gli archi possibili)
         3. Quindi, almeno uno tra u e $v \in V \setminus V'$
         4. L'arco (u,v) è coperto da $V \setminus V'$.
      ii. Siccome $(u,v) \in \underline{E}$, è stato scelto in modo arbitrario, allora per ogni arco $(u,v) \in \underline{E}$ è coperto da un vertice in $V \setminus V'$
      iii. Quindi $V \setminus V'$ è una **copertura di vertici** per $\underline{G}$. Inoltre, $|V \setminus V'| = |V| - k$.
      iv. La copertura è di dimensione **$|V| - k$**
   c. "←": Sia $(G,k) \in I$, e sia $f(G,k) = (G, |V| - k)$.

i. Supponiamo che G abbia una copertura di vertici $V' \subseteq V$ con $|V'| = |V| - k$, allora $\forall(u,v) \in V$, se $(u,v) \in E \rightarrow u \in V'$ e/o $v \in V'$.
ii. Tale implicazione è **equivalente** a: $\forall u,v \in V, u \notin V' \wedge v \notin V' \rightarrow (u,v) \in E$
iii. Ovvero, $\forall u,v \in V, u \in V \setminus V' \wedge v \in V \setminus V' \rightarrow (u,v) \in E$.
iv. Cioè **$V \setminus V'$ è una cricca in G**, e ha dimensione $|V| - |V'| = |V| - (|V| - K) = k$

Problema Q

Un problema Q è rappresentato da una relazione binaria tra due insiemi: **I** (insieme delle istanze), **S** (insieme delle soluzioni). Formalmente, quindi, si ha $Q \subseteq I \times S$; questo significa che ogni istanza $i \in I$ può essere associata a una o più soluzioni $s \in S$. Il problema Q può essere anche visto come una funzione $Q: I \rightarrow S$. Abbiamo come esempio il **shortest path**.

- **Insieme delle istanze**: $I = \{(G = (V, E), u,v \mid G$ è un grafo non orientato; $u,v \in V\}$
- **Insieme delle soluzioni**: $S = \{u_1 = u, u_2, \ldots, u_n = v) \mid$ cammino di lunghezza minima da u a v in $G\}$.

In questo caso, un'istanza $i = (G, u, v)$ potrebbe essere associata a più soluzioni, poiché possono esistere più cammini minimi distinti da u a v.

Problema di decisione

È una versione particolare di un problema Q, in cui l'insieme delle soluzioni è binario $S = \{0,1\}$. Quindi, $Q : I \rightarrow \{0,1\}$.

Un esempio è il **PATH**:

- **Insieme delle istanze**: $I = \{(G = (V,E), u,v,k) \mid G$ è un grafo non orientato, $u,v \in V, k \in N^+\}$
- **Insieme delle soluzioni:** $S = \{0,1\}$

$\forall i \in I$, il problema PATH(i) restituisce **1** se esiste un cammino da u a v in G che utilizza al più k archi. L'insieme delle istanze per cui PATH(i) = 1 è indicato con:

$$L = \{i \in I \mid Q(i) = 1\}$$

Problema di decisione decidibile

È un problema $Q : I \rightarrow \{0,1\}$ per cui **esiste un algoritmo A che calcola Q**. Questo implica che, $\forall i \in I$, l'algoritmo stabilisce correttamente se Q(i) = 1, oppure se Q(i) = 0.

Problema di decisione decidibile in tempo polinomiale

Un problema di decisione $Q : I \rightarrow \{0,1\}$ è decidibile in tempo polinomiale se **esiste un algoritmo A che calcola Q in tempo $O(n^k)$** per qualche $k \in R_+$.

Qui, n rappresenta la lunghezza $|i|$ dell'istanza i, e A garantisce una soluzione in tempo $O(n^k)$.

Problema di decisione verificabile

Un problema $Q : I \rightarrow \{0,1\}$ è **verificabile** se esiste un algoritmo A che, dato in input, un'istanza $x \in I$, una certificazione $y \in C$, restituisce un valore $\{0,1\}$ tale che:

- $\forall x \in I$ con $Q(x) = 1$, esiste almeno un $y \in C$ che rende $A(x,y) = 1$
- $\forall x \in I$ con $Q(x) = 0$, non esiste alcun $y \in C$ che rende $A(x,y) = 1$

In questo contesto, y è detto **certificato**, ed è una prova sufficiente a dimostrare che $Q(x) = 1$.

Problema verificabile

Dato $Q : I \rightarrow \{0,1\}$, Q è verificabile se e solo se **esiste un algoritmo A con 2 input** $x \in I$, $y \in C$ e **ritorna un valore in {0,1}** tale che:

$$\forall x \in I \text{ con } Q(x) = 1 \text{ esiste } y \in C \text{ tale che } A(x,y) = 1$$

y è detto **certificato**, qualcosa che offre ad A abbastanza informazioni per provare che $Q(x) = 1$. Se $x \in I$ con $Q(x) = 0$, non ci deve essere alcun y che porti a provare che $Q(x) = 1$.

Problema verificabile in tempo polinomiale

Un problema $Q : I \rightarrow \{0,1\}$ è verificabile in tempo polinomiale se **esiste un algoritmo A che verifica Q in tempo $O(n^k)$**, dove n è la lunghezza $|x|$ dell'istanza x, e $|y| = O(|x|^c)$ per qualche $c > 0$.

Classe P

La classe P è definita come l'**insieme di tutti i problemi di decisione decidibili in tempo polinomiale**; quindi

$$P = \{ Q \mid Q \text{ un problema di decisione decidibile in tempo polinomiale}\}.$$

Ad esempio, il problema PATH appartiene alla classe P.

Si può dire che un problema appartiene a P se è "facile da risolvere".

Classe NP

La classe NP è definita come l'**insieme di tutti i problemi di decisione verificabili in tempo polinomiale**:

$$\text{NP} = \{Q \mid Q \text{ un problema di decisione verificabile in tempo polinomiale}\}$$

Ad esempio, SAT appartiene alla classe NP.

Si può dire che un problema appartiene a NP se è "facile da verificare", ovvero, assegnate delle variabili, possiamo verificare velocemente se funzionano. Ma trovare questa assegnazione potrebbe richiedere molto più tempo.

Problema SAT

Abbiamo come insieme delle istanze $I = \{\phi \mid \phi \text{ è una formula booleana}\}$, con l'insieme delle soluzioni $S = \{0,1\}$.

$\forall\, \phi \in I$, il problema $\text{SAT}(\phi)$ restituisce 1 se esiste un'assegnazione che soddisfa ϕ; altrimenti, restituisce 0.

SAT come problema verificabile

Il problema SAT è verificabile:

- y: un'assegnazione di verità che soddisfa ϕ
- A: un algoritmo che verifica se un'assegnazione y soddisfa ϕ in tempo polinomiale

Riduzioni

Una riduzione tra due problemi Q e Q' permette di **trasformare qualunque istanza di Q in un'istanza di Q' in modo tale che la soluzione dell'istanza di Q' corrisponda a una soluzione per l'istanza originale di Q**.

Se possiamo ridurre Q a Q', denotiamo questa relazione come $Q \leq_p Q'$; questo indica che Q non è più difficile di Q', almeno in termini di complessità computazionale.

Un esempio:

Un'equazione lineare $Ax+b = 0$ può essere ridotta a un'equazione quadratica del tipo $0x^2 + bx + c = 0$, perché *risolvere la seconda fornisce una soluzione anche alla prima*.

Commentato [FC15]: Spero sia giusto questo, non ne ho idea

Definizione formale

Siano Q e Q' due problemi decisionali:

- $Q : I \rightarrow \{0,1\}$ dove I è l'insieme delle istanze di Q
- $Q' : I' \rightarrow \{0,1\}$, dove I' è l'insieme delle istanze di Q'

Diciamo che $Q \leq_p Q'$, cioè che Q è riducibile a Q', se esiste una funzione $f: I \rightarrow I'$ calcolabile in **tempo polinomiale** tale che $\text{per ogni } x \in I$:

$$Q(x) = 1 \leftrightarrow Q'(f(x)) = 1$$

Questa riduzione implica che, se sappiamo risolvere Q', possiamo risolvere anche Q trasformando le istanze di Q in istanze di Q'.

Proprietà

$\text{Se } Q \leq_p Q', \text{ e } Q' \text{ è decidibile, allora anche } Q \text{ è decidibile.}$

$\text{Se } Q \leq_p Q', Q' \in P, \text{ allora } Q \in P.$

NP-completezza

Un problema è detto NP-completo se soddisfa **entrambe** le seguenti condizioni:

- $Q \in NP$: il problema **appartiene alla classe NP**, ovvero le sue soluzioni possono essere verificate in tempo polinomiale
- **È almeno difficile quanto tutti i problemi in NP**: Per ogni problema $Q' \in NP$ esiste una **riduzione polinomiale** $\mathbf{Q' \leq_p Q}$; per ogni problema in NP può essere trasformato in un'istanza di Q. $Q \text{ è NP-difficile.}$

I problemi NP-completi sono, intuitivamente, i problemi più difficili della classe NP. Se fosse possibile risolvere un qualsiasi problema NP-completo in tempo polinomiale, allora ogni problema in NP sarebbe risolvibile in tempo polinomiale, il che implicherebbe che $P = NP$.

Teorema di Cook-Levin

Il problema SAT è stato il primo problema dimostrato essere NP-completo. Il teorema afferma che **SAT** appartiene a NP, e che:

$$\forall Q' \in NP, Q' \leq_p SAT$$

Quindi, ogni problema in NP può essere in tempo polimoniale a SAT; quindi, rende SAT un problema fondamentale per la teoria della complessità.

Metodo per dimostrare che un problema è NP-completo

Per dimostrare che un problema Q è NP-completo, sono sufficienti i seguenti passi:

- Mostrare che $\mathbf{Q \in NP}$, cioè che una soluzione a Q può essere verificata in tempo polinomiale
- Trovare un problema NP-completo noto, *ad esempio SAT*, e dimostrare che $\text{SAT} \leq_p Q$. Se si può ridurre SAT a Q in tempo polinomiale, allora **Q è almeno tanto difficile quanto SAT**, e quindi è NP-completo.

Dizionario

Insieme dinamico che supporta le operazioni di **inserimento** (aggiungere un elemento), **ricerca** (trovare un elemento associato ad una chiave), **cancellazione** (rimuove un elemento). L'obiettivo è di *ottimizzare i costi delle operazioni rispetto a implementazioni semplici* (es. liste concatenate o binari di ricerca).

Lo si rappresenta con una **tabella**, ovvero una sequenza di elementi e_i ciascuno dei quali ha:

- Una chiave K_i, composta da un numero di caratteri o bit
- Un'informazione I_i, associata alla chiave K_i

Ogni elemento e_i è memorizzato come un oggetto x, che contiene $X.\text{Key}$ (la chiave K_i), e $x.I$ (l'informazione I_i).

Tavole ad indirizzamento diretto

Assumiamo che ogni elemento abbia una chiave appartenente all'universo delle chiavi $U = \{0, 1, \ldots, m\text{-}1\}$, e che **due elementi distinti non possono avere la stessa chiave**.

Per rappresentare il dizionario si usa un **array** $T[0, \ldots, m\text{-}1]$ detto *tavola di indirizzamento diretto*. Ogni posizione k di T contiene un puntatore all'elemento con chiave $K = k$, ed è composta da K e I_k.

Abbiamo le seguenti operazioni:

```
DIRECT-ADDRESS_INSERT(T,x)
//Inserisce un elemento x nella tavola T
T[x.key] = x

DIRECT-ADDRESS-SEARCH(T,k)
//trova elemento con chiave k
return T[k]

DIRECT-ADDRESS-DELETE(T,x)
```

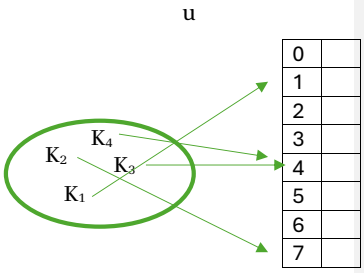
```
//rimuove l'elemento x
T[x.key] = NIL
```

Il costo di ciascuna operazione è $O(1)$, ma ha un problema di **spreco di memoria**: se m (dimensione dell'universo) è grande rispetto al numero di chiavi utilizzate $|K|$, c'è un notevole spreco di memoria.

Tavole hash

Le tavole hash rappresentano un compromesso tra tempo e memoria. Lo spazio occupato dipende da $|K|$ e non da $|U|$, occupa $\phi(|K|)$ come memoria, e il tempo di ricerca nel caso medio è di $\phi(1)$.

Usando l'indirizzamento diretto, un elemento con chiave k è memorizzato in $T[k]$; invece, con l'hashing, un elemento con la chiave è memorizzato in $T[h(K)]$, dove h è detta **funzione di hash**. Quindi, la funzione mappa una chiave K a un indice della tabella T con $h : U \rightarrow \{0, 1, ..., n-1\}$ $\forall k \in U$, $h(K)$ è la cella della tavola hash T.



$h(K_1) = 1$, $h(K_2) = 7$, $h(K_3) = 4$, $h(K_4) = 4$

Collisione: $K_3 \neq K_4$ ma stesso valore hash

Si dice che l'elemento di chiave k è mappato nella cella $h(K)$, e $h(K)$ è il valore hash di K.

Come **vantaggi** abbiamo la riduzione dello spazio necessario (infatti $m << |U|$). Come **svantaggi** invece abbiamo la possibile perdita della corrispondenza diretta chiave-posizione, causando **collisioni** (due chiavi $K_1 \neq K_2$ con $h(K_1) = h(K_2)$).

Gestione delle collisioni

Possono essere risolte in due modi.

Metodo del concatenamento

Ogni cella j contiene una **lista concatenata** degli elementi con lo stesso valore hash, quindi con $h(K) = j$.

Questo metodo ha due operazioni:

- **`CHAINED-HASH-INSERT(T,x)`**: inserisce un elemento x in testa alla lista associata a $T[h(x.Key)]$
- **`CHAINED-HASH-DELETE(T,x)`**: cancella x dalla lista associata a $T[h(x.Key)]$

Per l'inserimento e la cancellazione il tempo di esecuzione è di $O(1)$, mentre per la ricerca abbiamo:

- **Caso peggiore**: $O(n)$, n = numero di elementi nella tabella
- **Caso medio**: $O(1 + \alpha)$, dove $\alpha = n/m$, chiamato anche *fattore di carico*
    - Le prestazioni dipendono da come h distribuisce mediamente le chiavi tra le m celle.

Indirizzamento aperto

In caso di collisione, si cerca una posizione libera all'interno della tabella stessa, evitando l'uso di liste, e neanche elementi memorizzati fuori dalla tabella.

La tavola può riempirsi, e abbiamo $\alpha \leq 1$ sempre, con $n \leq m$.

Non usando i puntatori, permette di avere più memoria extra; usiamo questa memoria per consentire valori di m più grandi.

Probe function

Al posto di usare liste, si esplora la tavola in una sequenza definita da una funzione di probe:

$$h(k,i) : u \times \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$$

dove i è il tentativo corrente. La sequenza di ispezione è $<h(k,0), h(k,1), ..., h(k, m-1)>$. Si vuole che tale sequenza sia una permutazione di $\{0, ..., m-1\}$.

In tal modo, se esiste una posizione libera si trova, e ogni posizione può essere considerata come una possibile cella in cui inserire una nuova chiave. Assumiamo che per semplicità non vi siano dati satelliti.

I **vantaggi** sono che non c'è alcun uso di liste, e tutta la memoria è dedicata alla tavola. Gli **svantaggi** sono che la cancellazione è complicata.

Vogliamo inoltre che ogni chiave abbia la stessa probabilità di avere come sequenza di ispezione una delle **m! permutazioni** di $\{0, ..., m-1\}$; si vorrebbe quindi che venissero generate m! permutazioni differenti al variare delle chiavi.

Assicurano quindi che, almeno, $\forall\ k \in U$ $<h(K,0), ..., h(K, m-1)>$ è una **permutazione** di $\{0, ..., m-1\}$.

Inserimento

```
HASH-INSERT(T,K)
i = 0
repeat
     j = h(K,i)
```

```
     if T[j] == NIL
          T[j] = K
          return j
     else i++
until i == m
error “overflow della tabella hash”
```

Ricerca

```
HASH-SEARCH(T,K)
i = 0
repeat
     j = h(K, i)
     if T[j] == k
          return j
     i++
until T[j] == NIL OR i == m
return NIL
```

Cancellazione

È difficile da eseguire, poichè quando si cancella una chiave dalla cella i **non possiamo inserire NIL in T[i]:** facendo così, sarebbe impossibile ritrovare qualunque chiave per il cui inserimento si è passati da i, e T[i] era occupata. Abbiamo quindi tre soluzioni:

- Inserire “deleted” al posto di NIL nella cella i
- Modificare INSERT per trattare tale cella come vuota
- Lasciare SEARCH così com’è, che ignora il valore “deleted”

Strategie di probe

Ispezione lineare

Sia $h' : U \rightarrow \{0, ..., m-1\}$ una funzione di hash ordinaria, con h’ chiamata **funzione hash ausiliaria**.

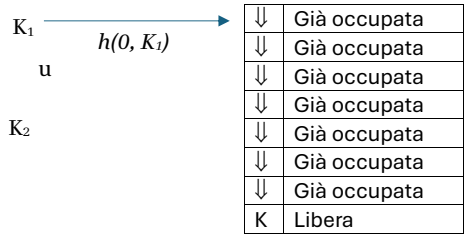
Definiamo $h : U \times \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$ $\forall (K,i) \in U \times \{0, ..., m-1\}$,

$$\mathbf{h(K,i) = (h'(K) + i) \bmod m}$$

Vengono generate delle sequenze di ispezione del tipo $<h'(K) \bmod m, (h'(K)+1 \bmod m, ...>$.

Siccome la prima cella ispezionata $h'(K)$ determina l’intera sequenza ed i possibili valori di h’ sono m, si hanno in totale **m sequenze di ispezione distinte**.

Il problema principale è l’**addensamento primario**: si formano lunghe file di celle occupate che aumentano il tempo di ricerca.



Ispezione quadratica

Dato $h : U \times \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$ $\forall K,I \in U \times \{0, ..., m-1\}$,

$$\mathbf{f(K, i) = (h'(K) + c_1 i + c_2 i^2) \bmod m}.$$

Di seguito un esempio.

$c_1 = c_2 = 1$, $m = 20$, h' e K qualunque

Si ha dunque $h(K,0) = h'(K)$, $h(K,1) = h'(K)+2$, $h(K,2) = h'(K) + 6$, $h(K,3) = h(K) + 12$, $h'(K,4) = h(K,0) = h'(K)$

Con $c_1 = c_2 = ½$ e $m = 2^s$, viene usata l'intera tabella, e si ha che $\forall I \in \{0, ..., m-1\}$, i valori $h(K,i)$ sono **tutti distinti**.

Dunque, la prima posizione $h(K,\phi) = h'(K)$ determina **l'intera sequenza di scansione**, e le sequenze distinte sono quindi m. Se per due chiavi K_1, K_2 ($\text{con } K_1 \neq K_2$) si ha che $h'(K_1) = h'(K_2)$, allora le due sequenze di ispezione coincidono.

Doppio hashing

Siano $h_1, h_2 : u \rightarrow \{0, ..., m-1\}$ due funzioni di hashing ausiliarie. La **probe function** nel doppio hasing parte da $h : U \times \{0, ..., m-1\} \rightarrow \{0, ..., m-1\}$, $\forall (K,i) \in U \times \{0, ..., m-1\}$

$$h(K,i) = h_1(K) + i * h_2(k) \bmod m$$

La sequenza di ispezione dipende dalla chiave k in due modi:

- Tramite la **posizione iniziale** $h_1(k) \bmod m$
- La **distanza** tra due posizioni successive, ovvero $(i * h_2(k) \bmod m)$

Il valore di $h_2(K)$ deve essere **primo** con m $\forall k$, ossia $MCD(h_2(K), m) = 1$.

Infatti, se $MCD(h_2(K), m) = d > 1$ per qualche chiave K, allora la ricerca di tale chiave *andrebbe ad esaminare solo una porzione 1/d della tabella*. Come si fa?

- Se m è potenza di 2 basta definire h_2 che restituisce sempre un dispari
- Scegliere m primo, e h_2 che ha come valore un intero positivo $< m$
- Un esempio: $h_1(K) = K \bmod m$, $h_2(K) = 1 + (K \bmod m')$, $m' = m-1$, $m' = m-2$

In tal modo, **ogni coppia $h_1(K)$ e $h_2(K)$ produce una distinta sequenza di scansione**. Si hanno $\Theta(m^2)$ sequenze distinte anziché $\Theta(m)$ sequenze distinte.

Se avessimo l'hashing uniforme, avremmo i seguenti casi:

- **La ricerca è senza successo**. Il numero atteso di ispezioni è $1/(1-\alpha)$, con α = fattore di carico.
- **La ricerca è con successo**. Il numero atteso di ispezioni è $(1/\alpha) \ln (1/(1-\alpha))$
- **Inserimento**. Richiede in media un numero di ispezioni $\leq 1/(1-\alpha)$

Hashing uniforme semplice

Qualsiasi elemento ha la stessa probabilità di essere inviato tramite h in una qualsiasi delle m celle, indipendentemente dalle celle in cui sono inviati gli altri elementi.

Sia X una variabile aleatoria su u $\forall j \in \{0, ..., m-1\} \rightarrow p(h(x) = j) = 1/m$.

Sotto questa ipotesi, vale che se n_j è la lunghezza della lista $T[j]$, allora $E[n_j] = \alpha = n/m$.

Dimostriamo. Siano $x_1, ..., x_n$ gli elementi inseriti di chiavi $K_1, ..., K_n$. Definiamo la variabile aleatoria $Z_{ij} = 1$ se $h(K_i) = j$, 0 altrimenti. La lunghezza attesa di una qualunque lista $j \in [0, ..., m-1]$ è

$$E[\sum_{i=1}^{n} Z_{ij}] = \sum_{i=1}^{n} E[Z_{ij}] = n \cdot \frac{1}{m} = \frac{n}{m} = a$$

La ricerca senza successo o con successo richiede tempo pari a $\Theta(1+\alpha)$.

Dimostrazione senza successo

Nell'ipotesi di hashing uniforme, qualsiasi chiave K non ancora memorizzata ha la stessa probabilità di essere associata ad una qualsiasi delle m celle. Quindi, il tempo medio di attesa per scoprire che la chiave non è presente è dato dalla lunghezza attesa di $T[K]$, ossia

$E[n * h(K)] = \alpha \rightarrow O(1+\alpha)$, con: $\mathbf{1}$ = tempo di calcolo di $h(K)$ e accesso alla lista
$\boldsymbol{\alpha}$ = tempo di scansione della lista

Finchè $n = O(m)$, la tabella non si riempie troppo, ed abbiamo $\alpha = O(m) / m = O(1)$

Progettazione funzione hash

Una funzione hash è una serie di tecniche euristiche per ottenere buone prestazioni; una buona funzione hash dovrebbe **soddisfare approssimativamente l'ipotesi dell'hashing uniforme semplice**, e dove **ogni chiave ha la stessa probabilità di essere inviata in una qualunque delle m celle** (indipendentemente dalla cella in cui viene inviata qualunque altra chiave).

Di solito, **non è possibile verificare tale condizione**, poiché raramente è nota la distribuzione di probabilità secondo la quale vengono estratte le chiavi da u.

Possono essere **note** però, come nel seguente esempio: le chiavi sono numeri reali casuali K distribuiti in modo indipendente e uniforme in $[0,1)$, quindi $0 \leq K < 1$. Allora $h(K) = [K_m]$. Questo *soddisfa la condizione di hashing unificato semplice*.

Quando non è nota?

Dobbiamo quindi ricorrere ad euristiche per ridurre al minimo il fatto che *chiavi correlate hanno lo stesso valore hash*.

Come approccio quindi usiamo la divisione hash, che è indipendente da qualsiasi regolarità presente nei dati. Quindi, le chiavi sono interpretate come numeri naturali.

Metodo della divisione

Dato $h : U = N \rightarrow \{0, ..., m\text{-}1\}$ $\forall k \in N\ h(k) = k \bmod m$, con m sconosciuto.

Ma $m \neq 2^p$, poiché se lo fosse, $h(K)$ rappresenterebbe i bit meno significativi di K. Meglio rendere invece il valore di hash dipendente da tutti i bit.

È meglio scegliere quindi come m un *numero primo non troppo vicino ad una potenza di 2*.

Metodo della moltiplicazione

Scegliamo $0 < A < 1$, e $h(K) = \lfloor m(AK - [AK]) \rfloor = \lfloor (AK \bmod 1) \rfloor$.

Il **vantaggio** è che m non è critico; scegliamo $m = 2^p$, così la tabella è facile da implementare in un calcolatore. Come valore di A, si può scegliere $\mathbf{A \approx Vd\text{-}1/2 \approx (\sqrt{5} - 1)/2}$.

Hashing universale

Si vuole evitare che, con una funzione hash prefissata, un **avversario scelga chiavi che causano collisioni** (quindi che scelga le n chiavi in modo che vengano inviate tutte nella stessa scegliere), possiamo **scegliere h casualmente da una famiglia di funzioni hash, in modo che sia indipendente dalle chiavi che saranno memorizzate**. Questo approccio è conosciuto come **hashing universale**, e all'inizio dell'algoritmo si sceglie una funzione h in una classe di funzioni.

$\partial H = \{h : U \rightarrow \{0, ..., m\text{-}1\}\}$, ∂H è universale se $\forall K, I \in U\ (K \neq I)$ vale che:

$$|\{h \in \partial H \mid h(K) = h(I)\}| \leq |\partial H| / m$$

In altri termini, **se h è scelta casualmente da ∂H che è universale** allora la **probabilità** di una collisione nel caso in cui $h(K)$ e $h(I)$ fossero scelte in modo casuale e indipendente dall'unione $\{0, ..., m-1\}$ è $P(h(K)) = h(I) \leq 1/m$.

Sia h scelta casualmente da ∂H con ∂H universale, e sia utilizzata per inserire n chiavi in una tavola T di dimensione m, applicando il concatenamento per risolvere le collisioni.

Allora, se K non è in T, $E[n * h(K)] \leq \alpha$, $n*h(K)$ è la *lunghezza della lista in cui è inserita K*.

Se K è in T, $E[n * h(K)] \leq 1 + \alpha$.

È **impossibile** quindi per un avversario scegliere una sequenza di operazione che forzi il caso peggiore. Vale infatti che, utilizzando l'hashing universale e il concatenamento per risolvere le collisioni, in una tavola con celle inizialmente vuote, occorre tempo atteso $\Theta(n)$ per eseguire una sequenza di n operazioni `INSERT` / `SEARCH` / `DELETE` che contiene $O(m)$ `INSERT`).

Famiglia universale

È un insieme di funzioni hash con la proprietà che, per ogni coppia di chiavi distinte k_1 e k_2, la probabilità che $h(k_1) = h(k_2)$ sia al massimo $1/m$, con m = dimensione della tabella hash.

Sia p primo, con $p > |u|$, $Z_p = \{0, ..., p-1\}$, $Z_p^* = \{1, ..., p\}$. Sia una qualunque chiave $K \in U$, quindi $K \in Z_p$, con $p > m$.

Ora, $\forall a \in Z_p^*$, $\forall b \in Z_p$ definiamo $h(a,b) = u \rightarrow \{0, ..., m-1\}$. $\forall k \in U$, $h(a,b)(k) = ((aK+b) \bmod p) \bmod m$.

Vale il seguente **teorema**:

$\partial H = \{h_1^b : u \rightarrow \{q, ..., m-1\}, q \in Z_p^*, b \in Z_p\}$ ci sono $p(p-1)$ funzioni, ∂H è **universale**.