

Analisi e Progetto di Algoritmi

Elia Ronchetti

@ulerich

2023/2024

Indice

1	Programmazione Dinamica - DP	6
1.1	Problemi di ottimizzazione	6
1.2	Il processo di sviluppo	6
1.3	Esempio - Fibonacci	7
1.3.1	Passaggi	7
1.4	Osservazioni sui problemi di ottimizzazione	7
1.5	LCS - Longest Common Subsequence	8
1.5.1	Definizioni di base	8
1.5.2	Istanza del problema	9
1.6	Procedura LCS	10
1.6.1	Definizione dei sottoproblemi	10
1.6.2	Equazioni di ricorrenza	10
1.6.3	Sottostruttura ottima	11
1.6.4	Equazioni di ricorrenza	11
1.6.5	Riassumendo - Per calcolare l'ottimo	13
1.6.6	Dimostrazione matematica per assurdo della sottostruttura ottima	14
1.6.7	Risoluzione Bottom-Up	14
1.6.8	Ricostruzione della soluzione ottimale	15
1.6.9	Algoritmo ricorsivo ricostruzione soluzione ottima	17
2	LIS - Longest Increasing Subsequence	19
2.1	Esempio di LIS di X	19
2.1.1	Altro esempio	20
2.2	Definizione formale e identificazione tipologia problema	20
2.3	Soluzione	20
2.3.1	Definizione dei sottoproblemi - Primo tentativo	20
2.3.2	Casi Base	20
2.3.3	Passo ricorsivo?	21
2.3.4	Test sottostruttura - Primo tentativo	21
2.4	Problema Ausiliario	21

2.4.1	Casi Base	22
2.4.2	Passo Ricorsivo	22
2.4.3	Definizione sottostruttura ottima	23
2.4.4	Passo ricorsivo (Problema Ausiliario - PA)	23
2.4.5	Equazioni di ricorrenza - PA	23
2.4.6	Sostituzione coefficienti nelle equazioni di ricorrenza	24
2.4.7	Calcolo dell'ottimo (PA)	24
2.4.8	Algoritmo DP (Bottom-up)	24
2.4.9	Algoritmo DP (bottom-up)	24
2.4.10	Ricostruzione soluzione ottimale	25
2.4.11	Algoritmo di Ricostruzione - Ricorsivo	25
3	Knapsack Problem 0/1 - Il Problema dello Zaino 0/1	26
3.1	Istanza del problema	26
3.2	Definizione formale	27
3.2.1	Soluzione del problema DP	28
3.3	Sottostruttura Ottima	28
3.3.1	Diamo un ordine agli oggetti	28
3.3.2	Sottostruttura ottima	30
3.3.3	Passo ricorsivo per (n, C)	31
3.3.4	Definizione dei sottoproblemi	31
3.4	Equazioni di ricorrenza	31
3.4.1	Sostituzione coefficienti	32
3.4.2	Calcolo dell'ottimo	32
3.5	Algoritmo DP (bottom-up)	32
3.5.1	Riempimento matrice	33
3.6	Algoritmo DP - Codice	34
3.6.1	Ricostruzione Soluzione Ottimale	35
3.6.2	Ricostruzione Soluzione Ottimale - Codice	36
4	Problema dei cammini minimi - Floyd-Warshall	38
4.1	Definizioni	38
4.1.1	Grafo	38
4.1.2	Adiacenza	39
4.1.3	Rappresentazione di un grafo	39
4.1.4	Esempio grafo orientato	40
4.1.5	Esempio grafo non orientato	40
4.1.6	Liste VS Matrice (memoria)	41
4.1.7	Liste VS Matrice (tempo)	41
4.1.8	Cammino in un grafo orientato	41
4.1.9	Grafo orientato pesato	41

4.1.10	Esempio grafo orientato pesato	42
4.2	Il problema dei cammini minimi	42
4.2.1	L'input	42
4.2.2	L'output Matrici D e Π	43
4.3	Sottostruttura ottima (primo tentativo)	44
4.3.1	Diamo un ordine ai vertici del grafo	45
4.3.2	Equazioni di ricorrenza	46
4.3.3	Equazioni di ricorrenza	47
4.4	Algoritmo bottom-up	49
4.4.1	Algoritmo bottom-up - Codice	49
5	Algoritmi Greedy	51
5.1	Problema - Selezione attività	51
5.1.1	Soluzione con DP	52
5.1.2	Svantaggi della Soluzione tramite DP	53
5.1.3	Approccio greedy	53
5.1.4	Osservazioni sulla risoluzione Greedy	54
5.1.5	Codice Greedy	54
5.2	Greedy VS DP	55
5.2.1	Due ingredienti chiave di un algoritmo Greedy	56
5.3	Correttezza di un algoritmo Greedy	56
5.3.1	Cosa succede se sbaglio parametro?	57
5.4	Il problema dello Zaino Frazionario	57
5.5	Possibile strategia Greedy	58
5.5.1	Cambio parametro	59
5.5.2	Strategia Greedy Attuata	59
5.5.3	Codice	60
5.5.4	Proprietà della scelta Greedy (da dimostrare)	60
5.6	Algoritmi Greedy in Generale	60
5.6.1	Codice generico	60
5.7	Non tutti i problemi ammettono un algoritmi Greedy	60
5.7.1	Sistema di Indipendenza	61
5.7.2	Proprietà di scambio	62
5.7.3	Esempio 1	62
5.7.4	Esempio 2	62
5.8	Matroide Grafico M_G	63
5.8.1	Dimostrazione della proprietà dello scambio	64
5.9	Insieme massimale di un matroide	64
5.9.1	Insieme massimale di un matroide grafico	64
5.9.2	Matroide Pesato	65
5.10	Algoritmo Greedy Standard	66

5.11	Teorema di Rado	66
5.11.1	Proof	67
5.11.2	Parte 1 - (S,F) è un matroide	67
5.11.3	Parte 2 - (S,F) non è un matroide	68
5.11.4	Esempio 1 Applicazione Rado	69
5.11.5	Esempio 2 - Peso massimo archi	70
6	Minimum Spanning Tree - MST	72
6.1	Soluzione Generica	72
6.2	Definizioni Principali	73
6.2.1	Taglio	73
6.2.2	Arco che Attraversa il Taglio	73
6.2.3	Taglio che rispetta un insieme	74
6.2.4	Arco Leggero	74
6.3	Teorema dell'arco sicuro	74
6.3.1	Proof	75
6.3.2	Esempio Teorema	75
6.3.3	Soluzione generica	76
6.3.4	COROLLARIO	76
6.3.5	Implementazione GENERIC-MST	77
6.4	Algoritmo di Kruskal	77
6.4.1	Algoritmo Greedy Standard per il matroide grafico	78
6.4.2	Algoritmo Kruskal	78
6.4.3	Esempio di Esecuzione Algoritmo Kruskal	79
6.4.4	Codice Kruskal	80
6.5	Algoritmo di Prim	81
6.5.1	Idea dell'algoritmo	81
6.5.2	Proprietà dell'algoritmo	82
6.5.3	Implementazione Prim	83
6.5.4	Determinare l'MST	86
6.5.5	Riassunto PRIM-MST	86
6.5.6	Codice PRIM-MST	87
6.6	Algoritmo di Dijkstra - Cammini minimi da sorgente unica	87
6.6.1	Sottostruttura Ottima di un cammino minimo in generale	88
6.6.2	Tecnica del Rilassamento	89
6.6.3	Codice Tecnica del Rilassamento	89
6.6.4	Esecuzione Dijkstra	90
6.6.5	Esecuzione Grafica	91
6.6.6	Prova di correttezza	94

Capitolo 1

Programmazione Dinamica - DP

La programmazione dinamica (DP - Dynamic Programming) è una tecnica che (come il Divide et Impera), risolve i problemi combinando le soluzioni dei sottoproblemi.

Divide et Impera è ottimo quando i sottoproblemi da risolvere sono indipendenti, mentre DP è efficace quando i sottoproblemi non sono indipendenti e quindi hanno in comune dei sottosottoproblemi e le tecniche di risoluzione top-down risultano quindi inefficienti (chiamate ripetute). La programmazione dinamica si applica tipicamente ai **problemi di ottimizzazione**.

1.1 Problemi di ottimizzazione

Sono problemi dove ci sono molte soluzioni possibile. Ogni soluzione ha un valore e si vuole trovare una soluzione con il valore ottimo. Ci possono essere più soluzioni che raggiungono il valore ottimo.

1.2 Il processo di sviluppo

Il processo di sviluppo è diviso in 4 fasi:

- Caratterizzare la struttura di una soluzione ottima
- Definire in modo ricorsivo il valore di una soluzione ottima
- Calcolare il valore di una soluzione ottima, di solito con uno schema bottom-up (dal basso verso l'alto, risulta spesso più efficiente rispetto a top-down)

- Costruire una soluzione ottima dalle informazioni calcolate

1.3 Esempio - Fibonacci

Classico esempio è l'esecuzione di Fibonacci. Utilizzando la ricorsione pura si effettuano più volte le stesse chiamate (perchè i sotto-numeri sono gli stessi). Se invece utilizziamo la DP, con un approccio Bottom-Up ci dobbiamo chiedere, ma chi è Fibonacci di n ? è Fibonacci di $(1) + \text{Fibonacci}(2) + \dots + \text{Fibonacci}(n)$. In pratica inizio a calcolare le soluzioni dal sottoproblema più piccolo a salire, così facendo possiamo risparmiare molto tempo, al costo però di un maggiore utilizzo di spazio, dato che ho un Array che deve memorizzare i valori. Si tratta di un compromesso accettabile dato che senza usare Array il tempo di esecuzione sarebbe esponenziale.

1.3.1 Passaggi

A livello pratico dobbiamo:

1. Scomporre il problem in sottoproblemi di dimensione inferiore
2. Formulare la soluzione in maniera ricorsiva - Equazioni di Ricorrenza
3. Usare una strategia bottom-up (non top-down)
4. Memorizzare i risultati in una opportuna struttura dati
5. Individuare il "luogo" che contiene la soluzione del problema (nel caso di Fibonacci l'ultima cella a destra)

DP risulta vantaggiosa quando il numero di chiamate distinte è polinomiale (il numero totale di chiamate è esponenziale).

1.4 Osservazioni sui problemi di ottimizzazione

Per ogni istanza del problema esiste un insieme di soluzioni possibili (feasible solutions), più soluzioni perchè le soluzioni ottime possono essere diverse. Esiste una funzione obiettivo che associa un valore ad ogni soluzione possibile e restituisce come OUTPUT una soluzione possibile (soluzione ottimale) per cui il valore restituito dalla funzione obiettivo è massimo/minimo (valore ottimo).

1.5 LCS - Longest Common Subsequence

Si tratta di un problema che ha come istanza due sequenze di valori e richiede di trovare la più grande sottosequenza comune fra di esse. Si tratta di un problema di ottimizzazione, per questo usare DP è un'ottima idea.

1.5.1 Definizioni di base

Sequenza Successione di elementi topologicamente ordinati, presi da un insieme Σ .

Per esempio $X = \langle 2, 4, 10, 5, 9, 11 \rangle$, più in generale:

- $X = \langle x_1, x_2, \dots, x_m \rangle \rightarrow$ sequenza di $m = |X|$ elementi

Prefisso di lunghezza i Primi i elementi della sequenza:

- $X = \langle x_1, x_2, \dots, x_i \rangle \rightarrow$ prefisso di lunghezza i di X

Dato $X = \langle 2, 4, 10, 5, 9, 11 \rangle$ per esempio $X_3 = \langle 2, 4, 10 \rangle$.

i-esimo elemento Indichiamo con $X[i]$ l'i-esimo elemento x_i della sequenza X.

Sottosequenza Una qualsiasi successione di elementi (anche non consecutivi) di una sequenza che però rispettino l'ordine sulla sequenza.

Per esempio data una sequenza $X = \langle 2, 4, 10, 5, 9, 11 \rangle$

- $Z = \langle 4, 5, 9 \rangle$ è una sottosequenza di X
- $Z = \langle \rangle = \epsilon$ è una sottosequenza di X
- $\langle 9, 5, 4 \rangle$ NON è una sottosequenza di X

Definizione formale di sottosequenza Data $X = \langle x_1, x_2, \dots, x_m \rangle$, una sequenza $Z = \langle z_1, z_2, \dots, z_k \rangle$ ($k \leq m$) è sottosequenza di X se esiste una successione di k indici interi $i_1 < i_2 < \dots < i_k$ tali che $X[i_j] = z_j$ per j compreso tra 1 e k.

Esempio sottosequenza Dato $X = \langle 2, 4, 10, 5, 9, 11 \rangle$, $Z = \langle 4, 5, 9 \rangle$ è una sottosequenza di X.

Sottosequenza comune di X e Y è una sottosequenza sia di X che di Y.

$$X = \langle 1, 13, 5, 3, 1, 12, 8, 11, 6, 10, 10 \rangle$$

$$Y = \langle 1, 5, 5, 2, 3, 1, 12, 8, 8, 10 \rangle$$

$$S = \langle 5, 3, 1, 8, 10 \rangle$$

S è sottosequenza comune di X e Y.

LCS è la più lunga sottosequenza comune Z di X e Y.

Esempio di LCS

$$X = \langle 2, 10, 5, 3, 1, 12, 8, 30, 11, 6, 10, 13 \rangle$$

$$Y = \langle 2, 5, 10, 2, 3, 1, 30, 12, 6, 8, 10 \rangle$$

$$\langle 2, 10, 3, 1, 12, 8, 10 \rangle \text{ è LCS di X e Y}$$

La LCS è una soluzione ottimale, mentre la sua lunghezza (7) è il valore ottimo.

1.5.2 Istanza del problema

P: date due sequenze $X = \langle x_1, x_2, \dots, x_n \rangle$ e $Y = \langle y_1, y_2, \dots, y_n \rangle$, trovare la più lunga sottosequenza comune Z di X e Y.

Abbiamo capito che P è un problema di ottimizzazione di massimo, dove:

- $(m, n) \rightarrow$ è la dimensione del problema (lunghezza stringhe)
- Soluzioni possibili \rightarrow tutte le sottosequenze comuni di X e Y
- Funzione obiettivo \rightarrow lunghezza
- $|Z|$ è il valore ottimo del problema
- Z è una soluzione ottimale

1.6 Procedura LCS

Indichiamo con $LCS(A,B)$ la LCS delle sequenze A e B e di conseguenza $|LCS(A,B)|$ la lunghezza della LCS di A e B.

Procediamo con le seguenti fasi:

1. Individuiamo i sottoproblemi
2. Troviamo le equazioni di ricorrenza
3. Applichiamo una strategia bottom-up con memorizzazione dei risultati

Nota Bene Si deve individuare la sottostruttura ottima del problema. La strategia bottom-up trova l'ottimo (lunghezza di LCS) e in seguito si deve ricostruire una soluzione ottimale (una delle LCS).

1.6.1 Definizione dei sottoproblemi

Sottoproblema di dimensione (i,j). Trovare la LCS dei prefissi X_i e $Y_j \rightarrow LCS(X_i, Y_j)$.

$$i \in \{0, 1, \dots, m\}$$

$$j \in \{0, 1, \dots, n\}$$

Numero totale sottoproblemi: $(m+1) \times (n+1)$

Ricordiamo che $LCS(X_m, Y_n)$ è la soluzione del problema principale.

1.6.2 Equazioni di ricorrenza

Casi base

Tutti i sottoproblemi di dimensione (i,j) tale per cui $i = 0$ oppure $j = 0$.

$$\begin{aligned} i = 0 &\implies LCS(X_0, Y_j) = LCS(\epsilon, Y_j) = \epsilon \\ j = 0 &\implies LCS(X_i, Y_0) = LCS(X_i, \epsilon) = \epsilon \\ i = 0, j = 0 &\implies LCS(X_0, Y_0) = LCS(\epsilon, \epsilon) = \epsilon \end{aligned}$$

Passo ricorsivo

Tutti i sottoproblemi di dimensione (i, j) tale per cui $i > 0$ e $j > 0$.
Introduciamo la sottostruttura ottima del problema:

Esempio di sottostruttura ottima

$$X = \langle 3, 5, 4, 3, 10, 12, 8, 30 \rangle \quad m = 8$$

$$Y = \langle 3, 2, 4, 10, 2, 8, 30, 13, 30 \rangle \quad n = 9$$

$LCS(X, Y) \rightarrow Z = \langle z_1, z_2, \dots, z_{k-1}, z_k \rangle = \langle 3, 4, 10, 8, 30 \rangle$.

Sicuramente $z_k = 30$. Come mi comporto per $k-1$? $z_1, z_2, \dots, z_{k-1} \rightarrow LCS(?, ?)$.

Avrò sicuramente che sarà la LCS di $(k-1) + 30$, quindi:

$$LCS(X, Y) = LCS(X_7, Y_8) + \langle 30 \rangle$$

1.6.3 Sottostruttura ottima

Date:

- $X = \langle x_1, x_2, \dots, x_{m-1}, x_m \rangle$
- $Y = \langle y_1, y_2, \dots, y_{n-1}, y_n \rangle$
- $LCS(X, Y) = \langle z_1, z_2, \dots, z_{k-1}, z_k \rangle$

La sottostruttura ottima sarà data da:

$$x_m = y_n \implies LCS(X, Y) = LCS(X_{m-1}, Y_{n-1}) + \langle x_m \rangle$$

$$1. \quad z_k = x_m = y_n$$

$$2. \quad \langle z_1, z_2, \dots, z_{k-1} \rangle = LCS(X_{m-1}, Y_{n-1})$$

$$x_m \neq y_n \implies = \max\{LCS(X_{m-1}, Y_n), LCS(X_m, Y_{n-1})\}$$

$$1. \quad z_k \neq x_m \implies LCS(X, Y) = LCS(X_{m-1}, Y_n)$$

$$2. \quad z_k \neq y_n \implies LCS(X, Y) = LCS(X_m, Y_{n-1})$$

1.6.4 Equazioni di ricorrenza

i=0 \vee j = 0 (CASI BASE)

$$LCS(X_i, Y_j) = \epsilon$$

i > 0 \wedge j > 0 (PASSO RICORSIVO)

$$x_i = y_j \implies LCS(X_i, Y_j) = LCS(X_{i-1}, Y_{j-1}) + \langle x_i \rangle$$

$$x_i \neq y_j \implies LCS(X_i, Y_j) = \max\{LCS(X_{i-1}, Y_j), LCS(X_i, Y_{j-1})\}$$

Applichiamo la funzione "lunghezza" alle equazioni

La funzione lunghezza è definita come segue:

$$c_{i,j} = |LCS(X_i, Y_j)|$$

$i=0 \vee j=0$ (CASI BASE)

$$|LCS(X_i, Y_j)| = |\epsilon| = 0$$

$i > 0 \wedge j > 0$ (PASSO RICORSIVO)

$$x_i = y_j \implies |LCS(X_i, Y_j)| = |LCS(X_{i-1}, Y_{j-1})| + |<x_i>|$$

$$x_i \neq y_j \implies |LCS(X_i, Y_j)| = \max\{|LCS(X_{i-1}, Y_j)|, |LCS(X_i, Y_{j-1})|\}$$

Calcoliamo la lunghezza di ciò che è noto

$i=0 \vee j=0$ (CASI BASE)

$$c_{i,j} = 0$$

Dato che ho $i=0$ e $j=0$.

$i > 0 \wedge j > 0$ (PASSO RICORSIVO)

$$x_i = y_j \implies c_{i,j} = c_{i-1,j-1} + 1 \quad x_i \neq y_j \implies c_{i,j} = \max\{c_{i-1,j}, c_{i,j-1}\}$$

Dato che $<x_i>$ è un singolo carattere, quindi ha lunghezza 1. Le sostituzioni delle funzioni LCS sono la semplice sostituzione della definizione di funzione lunghezza sopra riportata.

Quanti coefficienti/variabili $c_{i,j}$? $(m+1)(n+1)$ coefficienti/variabili.

$c_{m,n} = |LCS(X_m, Y_n)| = |LCS(X, Y)|$, è il **valore ottimo del problema principale**.

Soluzione del problema

- Calcolo del valore ottimo (top-down oppure bottom-up?)
- Ricostruzione di una soluzione ottimale

Perchè io tramite la procedura effettuo il riempimento della matrice, devo capire quindi dove si trova il valore (in questo caso abbiamo visto in $c_{m,n}$) e devo ricostruire la sottosequenza dato che la matrice contiene lunghezze, non stringhe.

```

int ottimo-ricorsivo(i,j)
  if i = 0 || j = 0 then
    return 0
  else
    if  $x_i = y_j$  then
       $c_{i,j} = \text{ottimo-ricorsivo}(i-1, j-1) + 1$ 
      return  $c_{i,j}$ 
    else
       $c_{i-1,j} = \text{ottimo-ricorsivo}(i-1, j)$ 
       $c_{i,j-1} = \text{ottimo-ricorsivo}(i, j-1)$ 
      return  $\max\{c_{i-1,j}, c_{i,j-1}\}$ 

```

Valore ottimo \rightarrow ottimo-ricorsivo(m,n).

Complessità in tempo nel caso migliore? $\Omega(n)$ nel caso migliore in cui $X = Y$ e $|X| = n$.

Complessità in tempo nel caso peggiore? Esponenziale, perchè continua a creare dei rami per testare la stringa diminuendo di 1 prima a sinistra e poi a destra. Quindi non è una buona idea in questo caso applicare questa strategia.

1.6.5 Riassumendo - Per calcolare l'ottimo

1. Definizione dei sottoproblemi
2. Equazioni di ricorrenza
 - Casi base
 - Passo ricorsivo (sottostruttura ottima)
3. Definizione dei coefficienti/variabili (valori ottimi dei sottoproblemi)
4. Individuazione del coefficiente ottimo (valore ottimo dle problema principale)
5. Equazioni di ricorrenza in termini dei coefficienti
6. Calcolo dei coefficienti seguendo una strategia bottom-up
7. Determinazione del valore ottimo

1.6.6 Dimostrazione matematica per assurdo della sottostruttura ottima

Attualmente non ho molta sbatti di trascriverla, quindi rimanderò a domani o dopo questa infausta operazione.

1.6.7 Risoluzione Bottom-Up

- Si calcolano i coefficienti $c_{i,j}$ per dimensione (i, j) crescente a partire dai casi base $(0, j)$ e $(i, 0)$
- Si memorizza $c_{i,j}$ ogni volta che si risolve il sottoproblema (i, j)
- Quando si arriva a calcolare $c_{m,n}$ si ha il valore ottimo

Algoritmo DP (bottom-up)

1. Si costruisce una matrice C di $m+1$ righe e $n+1$ colonne
2. Si indicizzano le righe e le colonne a partire da 0
3. Si riempie C in modo tale che $C[i, j] = c_{i,j}$
4. Valore ottimo si trova in $C[m, n] = c_{m,n}$

Matrice

C		y_1	...	y_j	...	y_n	
	0	0	0	0	0	0	0
x_1	0						1
...	0						...
x_i	0						i
...	0						...
x_m	0					$c_{m,n}$	m
	0	1	...	j	...	n	

IF $x_i = y_j$
 $c_{i,j} = c_{i-1,j-1} + 1$
 ELSE
 $c_{i,j} = \max(c_{i-1,j}, c_{i,j-1})$

Notiamo subito che la prima riga e prima colonna vengono inizializzate a 0, dato che la LCS tra una qualsiasi sequenza e una nulla è 0.

Codice riempimento matrice

```

int ottimo_DP(X,Y)
  for i from 0 to m do
    C[i,0] = 0
  for j from 0 to n do
    C[0,j] = 0
  for i from 1 to m do
    for j from 1 to n do
      if  $x_i = y_j$  then
        C[i,j] = C[i-1,j-1] + 1
      else
        C[i,j] = max(C[i-1,j], C[i,j-1])
  return C[m,n]

```

Complessità in tempo $\Theta(mn)$

Esempio di matrice riempita

C		2	5	12	2	3	12	1	30	
	0	0	0	0	0	0	0	0	0	0
2	0	1	1	1	1	1	1	1	1	1
10	0	1	1	1	1	1	1	1	1	2
5	0	1	2	2	2	2	2	2	2	3
3	0	1	2	2	2	3	3	3	3	4
1	0	1	2	2	2	3	3	4	4	5
12	0	1	2	3	3	3	4	4	4	6
	0	1	2	3	4	5	6	7	8	

$c_{6,8} \rightarrow 4$

$X = \langle 2, 10, 5, 3, 1, 12 \rangle$
 $Y = \langle 2, 5, 12, 2, 3, 12, 1, 30 \rangle$

1.6.8 Ricostruzione della soluzione ottimale

Ora che abbiamo la matrice dei coefficienti abbiamo la LCS, o meglio il valore della LCS (quanto è lunga la più lunga sottosequenza di caratteri in comune fra le 2 sequenze), ma non abbiamo la stringa! È necessario ricostruire tramite la matrice la mia soluzione ottimale.

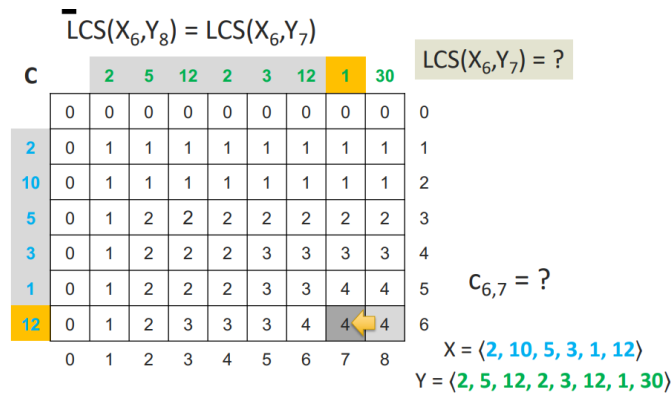
La procedura Partiamo dall'ultima cella in basso a destra (cella del valore ottimo), che nell'esempio riporta il valore di 4:
 Dobbiamo scegliere fra 3 possibili celle dove spostarci:

- La cella sinistra - Se contiene il valore maggiore fra i 3
- La cella in alto - Se contiene il valore maggiore fra i 3
- La cella diagonale - Se le celle hanno lo stesso valore decrementato di 1 rispetto a quello della cella di partenza

Quando mi muovo in diagonale salvo il carattere di riferimento della colonna (o riga, tanto saranno uguali) e proseguo con la procedura.

La procedura consiste nel calcolare la LCS guardando i coefficienti della matrice.

Inizio ricostruzione Partendo da 4 mi chiedo: Quale è la $LCS(X_6, Y_8)$? Guardo i coefficienti della matrice, quale valore fra 4, 4 scelgo? Qua non posso andare in diagonale perchè non ho un decremento del coefficiente di 1, posso andare solo a sinistra o in alto. Scelgo per esempio di andare a sinistra. Non essendoci stato un decremento del coefficiente, cioè non ho valori uguali sugli indici della matrice, infatti $X_6 \neq X_8$, non ho un carattere appartenente alla soluzione ottima, quindi avrò $LCS(X_6, Y_8) = LCS(X_6, Y_7)$, quindi mi sposto e cerco la $LCS(X_6, X_7)$.



Nel caso in cui la cella di partenza e le altre 3 fossero uguali si può scegliere di andare o in alto o a sinistra, si ottiene comunque la stessa soluzione, nel caso in cui si ottengono spostamenti diagonali differenti significa che si avrà un'altra soluzione ottima.

Proseguo e noto che anche qua ho due coefficienti uguali, scelgo di andare in alto, $LCS(X_6, X_7) = LCS(X_5, Y_7)$.

Osservazione Se fossi andato a sinistra al passaggio successivo avrei avuto un passaggio a una diagonale che non percorrero andando in alto, questa è un'altra soluzione ottima.

Ora mi trovo a calcolare $LCS(X_5, X_7)$. Qua ho tutti e 3 i valori uguali decrementati di 1, infatti ho $X_5 = X_7$, posso effettuare uno spostamento diagonale! Si tratta di un carattere della soluzione ottima quindi avrò che:

$$\begin{aligned} c_{5,7} &= c_{4,6} + 1 \\ LCS(X_5, Y_7) &= LCS(X_4, Y_6) + \langle x_5 \rangle \\ LCS(X_6, Y_8) &= LCS(X_5, Y_7) \\ LCS(X_6, Y_8) &= LCS(X_4, Y_6) + \langle x_5 \rangle \\ LCS(X_6, Y_8) &= LCS(X_4, Y_6) + \langle 1 \rangle \end{aligned}$$

Dove 1 è l'ultimo carattere della LCS.

Proseguo a ritrovo fino a quando arriverò in cima alla matrice e avrò il caso base $LCS(X_0, Y_0) = \epsilon$.

Ora ho la sottosequenza ottima!

1.6.9 Algoritmo ricorsivo ricostruzione soluzione ottima

Input Matrice C e indici i e j di una cella di C.

Output Una soluzione ottimale per i prefissi X_i e Y_j .

```

Procedura ricostruisci_LCS(C, i, j)
  if i>0 and j>0 then
    if  $x_i = y_j$  then
      ricostruisci_LCS(C, i-1, j-1)
      print  $x_i$ 
    else
      if  $C[i,j] = C[i-1,j]$  then
        ricostruisci_LCS(C, i-1, j)
      else
        ricostruisci_LCS(C, i, j-1)
```

Algoritmo di ricostruzione LCS iterativo

```
List Ricostruisci_LCS(C, m, n)
  j = m
  j = n
  LCS = empty list
  while i > 0 and j > 0 do
    if  $x_i = y_i$  then
      aggiungi  $x_i$  in testa a LCS
      i = i-1
      j = j-1
    else
      if C[i,j] = C[i-1,j] then
        i = i-1
      else
        j = j-1
  return LCS
```

Capitolo 2

LIS - Longest Increasing Subsequence

Una Increasing Sequence (IS) è definita nel seguente modo:

$$Z = \langle z_1, z_2, \dots, z_k \rangle$$

Tale che $z_i < z_{i+1}$ per ogni indice i da 1 a $k-1$.

Esempi

$\langle 2, 4, 7, 13, 21 \rangle \rightarrow$ è una sequenza crescente

$\langle 2 \rangle \rightarrow$ sequenza crescente

$\langle 2, 4, 13, 13, 21 \rangle \rightarrow$ NON è una sequenza crescente

$\langle 10, 10, 10 \rangle \rightarrow$ NON è una sequenza crescente

LIS di $X = \langle x_1, x_2, \dots, x_m \rangle$ più lunga sottosequenza di X che sia crescente.
 $Z = LIS(X)$.

2.1 Esempio di LIS di X

$$X = \langle 14, 2, 4, 2, 7, 0, 13, 21, 11 \rangle$$

$$LIS(X) = \langle 2, 4, 7, 13, 21 \rangle$$

LIS è la più lunga sottosequenza di X che sia crescente.

2.1.1 Altro esempio

$$X = \langle 5, 4, 3, 2, 1 \rangle$$

$$LIS(X) = \langle 5 \rangle \text{ or } \langle 4 \rangle \text{ or } \langle 3 \rangle \text{ or } \langle 2 \rangle \text{ or } \langle 1 \rangle$$

$$X = \langle 10, 10, 10, 10 \rangle$$

$$LIS(X) = 10$$

2.2 Definizione formale e identificazione tipologia problema

P: Data una sequenza $X = \langle x_1, x_2, \dots, x_m \rangle$ trovare la più lunga sottosequenza crescente $Z = LIS(X)$.

P è un problema di ottimizzazione di massimo, dove:

- $(m) \rightarrow$ dimensione del problema
- Soluzioni possibili \rightarrow tutte le sottosequenze crescenti di X
- Funzione obiettivo \rightarrow lunghezza
- $|Z|$ è il valore ottimo del problema
- Z è una soluzione ottimale

2.3 Soluzione

1. Calcolo dell'ottimo (lunghezza di Z)
2. Ricostruzione di una soluzione ottimale

2.3.1 Definizione dei sottoproblemi - Primo tentativo

Sottoproblema di dimensione (i)

Trovare la LIS del prefisso $X_i \rightarrow LIS(X_i)$.

Numero di sottoproblemi: $m+1$. Mentre per $i = m$ abbiamo il problema principale.

2.3.2 Casi Base

Sottoproblema di dimensione 0.

$$i = 0 \implies LIS(X_0) = \langle \rangle$$

2.3.3 Passo ricorsivo?

→ Tutti i sottoproblemi di dimensione (i) tale per cui $i > 0$.

Qual è la sottostruttura ottima del problema?

Proviamo la seguente sottostruttura:

$$LIS(X_m) = \max\{LIS(X_{m-1}), LIS(X_{m-1}) + \langle x_m \rangle\}$$

$$X = \langle x_1, x_2, \dots, x_{m-1}, x_m \rangle$$

$$LIS(X_m) = LIS(X_{m-1}) + \langle x_m \rangle?$$

- **Sì**, se $LIS(X_{m-1})$ finisce con un elemento $\langle x_m \rangle$
- **No**, se $LIS(X_{m-1})$ finisce con un elemento $\geq x_m$

2.3.4 Test sottostruttura - Primo tentativo

Verifichiamo se la sottostruttura definita poco fa ci permette di trovare le LIS in una stringa.

Esempio $X = \langle 14, 2, 4, 2, 7, 14, 15, 0, 13 \rangle \quad m = 9$

$$LIS(X_m) = LIS(X_{m-1}) + \langle x_m \rangle? \quad x_m = 13$$

$$LIS(X_9) = LIS(X_8) + \langle 13 \rangle? \quad X = \langle 14, 2, 4, 2, 7, 14, 15, 0, 13 \rangle$$

$$LIS(X_9) = LIS(X_8) + \langle 13 \rangle?$$

NO! Perchè risulta uguale a $\langle 2, 4, 7, 14, 15 \rangle + \langle 13 \rangle = \langle 2, 4, 7, 14, 15, 13 \rangle$ e $15 > 13$, questo significa che la sottostruttura non mi permette di avere una LIS.

Questo significa indica che mi manca dell'informazione, è necessario definire un problema ausiliario che ci permette di creare una sottostruttura che trovi una LIS valida.

2.4 Problema Ausiliario

Si tratta di un sottoproblema di dimensione (i) che ha lo scopo di trovare la LIS_v del prefisso $X_i \rightarrow LIS_v(X_i)$, $i \in \{1, 2, \dots, m\}$.

Numero di sottoproblemi: m.

$LIS_v(X_m) \rightarrow$ è la soluzione ottimale del problema ausiliario.

$$LIS(X) \rightarrow \max\{LIS_v X_i \text{ t.c. } 1 \leq i \leq m\}$$

Sostanzialmente cerco la LIS partendo dalla stringa base di 1 carattere, mano a mano incremento di 1 e ricalcolo la LIS, fino ad arrivare a calcolare la LIS di tutta la stringa.

Esempio

$$\begin{aligned}
X &= \langle 14, 2, 4, 2, 7, 14, 15, 0, 13 \rangle \\
LISV(X_1) &= \langle 14 \rangle \\
LISV(X_3) &= \langle 2, 4 \rangle \\
LISV(X_2) &= \langle 2 \rangle \\
LISV(X_5) &= \langle 2, 4, 7 \rangle \\
LISV(X_6) &= \langle 2, 4, 7, 14 \rangle \\
LISV(X_7) &= \langle 2, 4, 7, 14, 15 \rangle \quad LIS(X) \\
LISV(X_8) &= \langle 0 \rangle \\
LISV(X_9) &= \langle 2, 4, 7, 13 \rangle \\
LISV(X_4) &= \langle 2 \rangle
\end{aligned}$$

2.4.1 Casi Base

→ Sottoproblema di dimensione (1).

$$i = 1 \implies LIS_V(X_1) = \langle x_1 \rangle$$

2.4.2 Passo Ricorsivo

→ Tutti i sottoproblemi di dimensione (i) tale per cui $i > 1$.
Qual è la sottostruttura ottima del problema Ausiliario?

Esempio sottostruttura ottima

$$X = \langle 14, 2, 4, 2, 7, 14, 15, 0, 13, 21, 8 \rangle \quad m = 11$$

$$LIS_V(X_{11}) \rightarrow Z = \langle z_1, z_2, \dots, z_{k-1}, z_k \rangle = \langle 2, 4, 7, 8 \rangle$$

$$z_k = x_{11} = 8$$

$$\langle z_1, z_2, \dots, z_{k-1} \rangle \rightarrow \text{sottosequenza di } X_{10} \text{ per cui } z_{k-1} = x_5 = 7, \quad z_{k-1} < z_k$$

$$\text{Ottengo quindi che } \langle z_1, z_2, \dots, z_{k-1} \rangle = LIS_V(X_5) + z_k = \langle 2, 4, 7, 8 \rangle$$

$$LIS_V(X_5) = \max\{LIS_V(X_h) \text{ t.c. } 1 \leq h \leq 11, x_h < x_1\}$$

$$LISV(X_{11}) = \max\{LIS_V(X_h) \text{ t.c. } 1 \leq h < 11, x_h < x_1\} + \langle x_1 \rangle$$

2.4.3 Definizione sottostruttura ottima

Data la sequenza $X = \langle x_1, x_2, \dots, x_m \rangle$ e $LIS_V(X_m) = \langle z_1, z_2, \dots, x_{k-1}, z_k \rangle = Z_{k-1} + \langle z_k \rangle$:

$$\begin{aligned} z_k &= x_m \\ Z_{k-1} &= \max\{LIS_V(X_h) \text{ t.c. } 1 \leq h < m, x_h < x_m\} \\ \text{con } \max\{\emptyset\} &= \langle \rangle \end{aligned}$$

2.4.4 Passo ricorsivo (Problema Ausiliario - PA)

Data la sequenza $X = \langle x_1, x_2, \dots, x_m \rangle$:

$$\begin{aligned} LIS_V(X_m) &= \max\{LIS_V(X_h) \text{ t.c. } 1 \leq h < m, x_h < x_m\} + \langle x_m \rangle \\ \text{con } \max\{\emptyset\} &= \langle \rangle \text{ che implica } LIS_V(X_m) = \langle x_m \rangle \end{aligned}$$

Dato il prefisso $X_i = \langle x_1, x_2, \dots, x_i \rangle$:

$$\begin{aligned} LIS_V(X_i) &= \max\{LIS_V(X_h) \text{ t.c. } 1 \leq h < i, x_h < x_i\} + \langle x_i \rangle \\ \text{con } \max\{\emptyset\} &= \langle \rangle \text{ che implica } LIS_V(X_i) = \langle x_i \rangle \end{aligned}$$

2.4.5 Equazioni di ricorrenza - PA

i=1 (Caso Base) $LIS_V(X_1) = \langle x_1 \rangle$

$i > 1$ (Passo Ricorsivo)

$$\begin{aligned} LIS_V(X_i) &= \max\{LIS_V(X_h) \text{ t.c. } 1 \leq h < i, x_h < x_i\} + \langle x_i \rangle \\ \text{con } \max\{\emptyset\} &= \langle \rangle \text{ che implica } LIS_V(X_i) = \langle x_i \rangle \end{aligned}$$

Definizione dei coefficienti - PA

Coefficiente c_i del sottoproblema (i):

$$\begin{aligned} c_i &= |LIS_V(X_i)| \\ i &\in \{1, 2, \dots, m\} \end{aligned}$$

Numero di coefficienti: m.

$c_m = |LIS_V(X_m)| \rightarrow$ valore ottimo di PA.

2.4.6 Sostituzione coefficienti nelle equazioni di ricorrenza

i=1 (Caso Base) $c_i = 1$

$i > 1$ (Passo Ricorsivo)

$$c_i = \max\{c_h \text{ t.c. } 1 \leq h < i, x_h < x_i\} + 1$$

con $\max\{\emptyset\} = 0$ che implica $c_i = 1$

2.4.7 Calcolo dell'ottimo (PA)

- Si calcolano i coefficienti c_i per dimensione (i) crescente a partire dal caso base per $i = 1$
- Si memorizza c_i ogni volta che si risolve il sottoproblema (i)
- Quando si arriva a calcolare c_m si ha il valore ottimo del problema ausiliario $\rightarrow |LIS_V(X)|$
- Il valore ottimo del problema principale è dato da $\max\{c_i \text{ t.c. } 1 \leq i \leq m\} \rightarrow |LIS(X)|$

2.4.8 Algoritmo DP (Bottom-up)

- Costruzione del vettore $C[1 \dots m]$
- Riempimento di C in modo tale che $C[i] = c_i$
- Valore ottimo \rightarrow massimo in C

2.4.9 Algoritmo DP (bottom-up)

```

int calcolo_ottimo_LIS(X)
    C[1] = 1
    H[1] = 0
    valore_ottimo = C[1]
    for i from 2 to m do
        max = 0
        H[i] = 0
        for h from 1 to i-1 do
            if  $x_h < x_i$  and C[h] > max then

```



```

        max = C[h]
        H[i] = h
    C[i] = 1 + max
    if C[i] > valore_ottimo then
        valore_ottimo = C[i]
    return valore_ottimo

```

2.4.10 Ricostruzione soluzione ottimale

Mi basta leggere la matrice ausiliaria a partire dal valore massimo e vedere quando non ho 0 scrivere quello che ho in corrispondenza della matrice.

$X = \langle 14, \underline{2}, \underline{4}, 2, \underline{7}, 0, \underline{13}, \underline{21}, 20 \rangle$

	14	<u>2</u>	<u>4</u>	2	<u>7</u>	0	<u>13</u>	<u>21</u>	20
C	1	1	2	1	3	1	4	5	5
H	0	0	2	0	3	0	5	7	7
	1	2	3	4	5	6	7	8	9

$LIS(X) = \langle 2, 4, 7, 13, 21 \rangle$

2.4.11 Algoritmo di Ricostruzione - Ricorsivo

```

Procedura ricostruisci_LIS_V(H, i)
    if H[i] != 0 then
        ricostruisci_LIS_V(H, H[i])
    print  $x_i$ 

```

Chiamando la procedura per i_{max} (posizione della cella di C che contiene il valore massimo) si ottiene la stampa di una soluzione ottimale di LIS(X).

Il caso peggiore sarà quando ho una stringa crescente, il tempo sarà di $O(m)$.

Il caso migliore invece è quando la stringa è decrescente, si ferma subito alla prima cella. Avrò infatti T costante.

Capitolo 3

Knapsack Problem 0/1 - Il Problema dello Zaino 0/1

Questione da risolvere: trovare il subset di oggetti di massimo valore complessivo che non superi la capacità C .

Oggetti Ad ogni oggetto viene associato un peso e un valore, quindi il problema consiste nel inserire nello zaino il massimo valore possibile senza superare il peso massimo.

3.1 Istanza del problema

Insieme di n oggetti $\{1, 2, \dots, i, \dots, n\}$:

- $C \rightarrow$ Capacità dello zaino
- $v_n \rightarrow$ Valore dell'oggetto n
- $w_n \rightarrow$ peso/ingombro dell'oggetto n

Esempio

Il problema dello “Zaino 0/1”

1. $v_1=1, w_1=7$
2. $v_2=3, w_2=4$
3. $v_3=1, w_3=5$
4. $v_4=1, w_4=1$
5. $v_5=1, w_5=1$

C = 10

{2, 3, 4}

Peso complessivo → 10
Valore complessivo → 5

{1, 4, 5}

Peso complessivo → 9
Valore complessivo → 3

3.2 Definizione formale

Dato un insieme $X = \{1, 2, \dots, i, \dots, n\}$ di n oggetti, un intero C e due funzioni:

- $V : X \rightarrow N$ tale che $V(i) = v_i$ è il valore dell'oggetto i
- $W : X \rightarrow N$ tale che $W(i) = w_i$ è il peso dell'oggetto i

si vuole trovare un sottoinsieme $S = \{i_1, i_2, \dots, i_k\}$ di X tale per cui:

$$W_S = \sum_{j=1}^k w(i_j) \leq C$$

$$V_S = \sum_{j=1}^k v(i_j)$$

Dove V_S è il massimo tra tutti i valori dei possibili sottoinsiemi di X che sono "compatibili" con lo zaino.

Si tratta dunque di un problema di ottimizzazione di massimo, dove:

- $(n, C) \rightarrow$ dimensione del problema
- Soluzioni possibili \rightarrow tutti i sottoinsiemi S' di X il cui peso totale $W_{S'}$ è al più la capacità C dello zaino

- Funzione obiettivo \rightarrow valore totale $V_{S'}$ della soluzione possibile S' .
- Valore totale di $S \rightarrow$ valore ottimo
- $S \rightarrow$ Soluzione ottimale

3.2.1 Soluzione del problema DP


1. Calcolo del valore ottimo (valore totale di S)
2. Ricostruzione di una soluzione ottimale (un insieme S)

3.3 Sottostruttura Ottima

Consideriamo l'esempio di inizio capitolo:

1. $v_1=1, w_1=7$
2. $v_2=1, w_2=4$
3. $v_3=1, w_3=5$
4. $v_4=1, w_4=1$
5. $v_5=1, w_5=1$

C = 10



Peso complessivo $\rightarrow 10$
 Valore complessivo $\rightarrow 3$

$\{2, 3\}$ è una soluzione ottimale di $X \setminus \{4\}$? **NO**.
 $\{2, 3, 5\}$ è una soluzione ottimale di $X \setminus \{4\}$!

3.3.1 Diamo un ordine agli oggetti

Diamo un ordine agli oggetti all'interno di X , cioè:

1 viene prima di 2 che viene prima di 3, etc. che viene prima dell'ultimo oggetto n .

Data una soluzione ottima S si può verificare

- **CASO 1:** l'oggetto n appartiene a S
- **CASO 2:** l'oggetto n NON appartiene a S

CASO 1 $C \geq w_n$ e l'oggetto n appartiene a S

NB: $S' = S \setminus \{n\}$ non è necessariamente la soluzione ottimale dell'istanza per $X \setminus \{n\}$ e capacità C .

Infatti, se esiste $i \in X \setminus \{n\}$ t.c $i \notin S'$ e $w_i \leq C - W_{S'} \implies S' \cup \{i\}$ ha valore totale maggiore di quello di S' .

Tornando a noi, CASO 1 implica che $\implies S' = S \setminus \{n\}$ è soluzione ottimale per l'istanza data da:

- insieme di oggetti $X \setminus \{n\} = \{1, 2, \dots, n-1\}$
- Zaino di capacità C' pari a $C - w_n$

—

1. $v_1=1, w_1=7$
2. $v_2=1, w_2=4$
3. $v_3=1, w_3=5$
4. $v_4=1, w_4=1$
5. $v_5=1, w_5=1$

$C = 10$

$\{1, 4, 5\}$

Peso complessivo $\rightarrow 8$
Valore complessivo $\rightarrow 2$

$S' = \{1, 4\}$ è soluzione ottimale dell'istanza:

- $X \setminus \{5\} = \{1, 2, 3, 4\}$

- con capacità $C' = C - w_5 = 9$

PROOF

- S' è compatibile con la capacità $C - w_n$
 $W_S \leq C \implies W_S - w_n \leq C - w_n \implies W_{S'} \leq C - w_n$
- S' ha valore totale $V_{S'}$ massimo
 Se esiste S'' tale che $V_{S''} > V_{S'}$ e $W_{S''} \leq C - w_n$
 $\implies S'' \cup \{n\}$ è soluzione ottimale per l'istanza X e C di valore totale maggiore di V_S (contro l'ipotesi)

CASO 2 - L'oggetto n NON appartiene a S

$\implies S$ è soluzione ottimale per l'istanza data da:

- insieme di oggetti $X \setminus \{n\} = \{1, 2, \dots, n-1\}$
- Zaino di capacità pari a C

Non avendo aggiunto l'oggetto allo zaino la capacità totale rimane invariata.

1. $v_1=1, w_1=7$
2. $v_2=1, w_2=4$
3. $v_3=1, w_3=5$
4. $v_4=1, w_4=1$
5. $v_5=1, w_5=8$

$C = 10$



Peso complessivo $\rightarrow 10$
Valore complessivo $\rightarrow 3$

$S = \{2, 3, 4\}$ è soluzione ottimale dell'istanza:

- $X \setminus \{5\} = \{1, 2, 3, 4\}$
- con capacità $C = 10$

PROOF

Se esiste S' che è soluzione ottimale per $X \setminus \{n\}$ e capacità C e con valore $V_{S'} > V_S$ allora S' sarà soluzione ottimale per X e capacità C (contro l'ipotesi).

3.3.2 Sottostruttura ottima

Dato un insieme $X = \{1, 2, \dots, n\}$ di n oggetti, uno zaino di capacità C e una soluzione ottimale S :

$C \geq w_n, n \in S \implies S' = S \setminus \{n\}$ è soluzione ottimale per:

- insieme di oggetti $\{1, 2, \dots, n-1\}$
- capacità $C - w_n$

Problema $(n-1, C - w_n)$

$S = S' \cup \{n\}$ con S' soluzione ottimale per le condizioni viste prima del box.
 $n \notin S \implies S$ è soluzione ottimale per:

- insieme di oggetti $\{1, 2, \dots, n-1\}$
- capacità C

Problema $(n-1, C)$

$S = S''$ con S'' soluzione ottimale per le condizioni viste prima del box.

3.3.3 Passo ricorsivo per (n,C)

Dato un insieme $X = \{1, 2, \dots, n\}$ di n oggetti e uno zaino di capacità C , la soluzione ottimale $S_{n,C}$ è:

Se $C \geq w_n$ allora

$$\bullet S_{n,C} = \max_V \{S' \cup \{n\}, S''\}$$

altrimenti

$$\bullet S_{n,C} = S''$$

S' soluzione ottimale per il problema $(n-1, C-w_n) \rightarrow S_{n-1,C-w_n}$

S'' soluzione ottimale per il problema $(n-1, C) \rightarrow S_{n-1,C}$ Sostituisco nelle equazioni S' e S'' : **Se $C \geq w_n$ allora**

$$\bullet S_{n,C} = \max_V \{S_{n-1,C-w_n} \cup \{n\}, S_{n-1,C}\}$$

altrimenti

$$\bullet S_{n,C} = S_{n-1,C}$$

3.3.4 Definizione dei sottoproblemi

Sottoproblema di dimensione (i,c) Riempire uno zaino di capacità c con oggetti dall'insieme $\{1, 2, \dots, i\} \rightarrow S_{i,c}$

$$i \in \{1, 2, \dots, n\}$$

$$c \in \{0, 1, \dots, C\}$$

Numero di sottoproblemi: $(n+1)(C+1)$

$i = n, c = C \rightarrow$ problema principale

3.4 Equazioni di ricorrenza

CASI BASE \rightarrow Tutti i sottoproblemi di dimensione (i, c) tale per cui $i = 0$, oppure $C = 0$.

$$S_{i,C} = \emptyset$$

PASSO RICORSIVO → Tutti i sottoproblemi di dimensione (i, c) tale per cui $i > 0$ e $c > 0$.

Se $C \geq w_n$ allora

- $S_{n,C} = \max_V \{S_{n-1,C-w_n} \cup \{n\}, S_{n-1,C}\}$

altrimenti

- $S_{n,C} = S_{n-1,C}$

Convertito in Equazioni di Ricorrenza:

$i = 0 \vee c = 0$ (**CASI BASE**)

$S_{i,c} = \emptyset$ $i > 0 \wedge c > 0$ (**PASSO RICORSIVO**)

$c \geq w_i \implies S_{i,c} = \max_V \{S_{i-1,C-w_i} \cup \{i\}, S_{i-1,C}\}$

$c < w_i \implies S_{i,c} = S_{i-1,C}$

$d_{i,c} \rightarrow$ valore totale di $S_{i,c}$

3.4.1 Sostituzione coefficienti

Sostituendo quindi $d_{i,c}$ a $S_{i,c}$ otteniamo: $i = 0 \vee c = 0$ (**CASI BASE**)

$d_{i,c} = \emptyset$ $i > 0 \wedge c > 0$ (**PASSO RICORSIVO**)

$c \geq w_i \implies d_{i,c} = \max_V \{d_{i-1,C-w_i} \cup \{i\}, d_{i-1,C}\}$

$c < w_i \implies d_{i,c} = d_{i-1,C}$

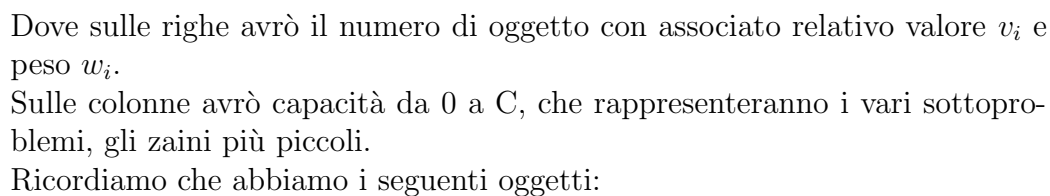
3.4.2 Calcolo dell'ottimo

- Si calcolano i coefficienti $d_{i,c}$ per dimensione (i, c) crescente a partire dai casi base per $i = 0$ e $c = 0$
- Si memorizza $d_{i,c}$ ogni volta che si calcola l'ottimo per il sottoproblema (i, c)
- Quando si arriva a calcolare $d_{n,C}$ si ha il valore ottimo del problema (n, C)

3.5 Algoritmo DP (bottom-up)

1. Costruzione di una matrice $D[0 \dots n, 0 \dots C]$
2. Riempimento di D in modo tale che $D[i, c] = d_{i,c}$

Avrò quindi la seguente matrice:



1. $v_1 = 1, w_1 = 7$
2. $v_2 = 1, w_2 = 4$
3. $v_3 = 1, w_3 = 5$
4. $v_4 = 1, w_4 = 1$
5. $v_5 = 1, w_5 = 1$

Capacità $C = 10$.

3.5.1 Riempimento matrice

Prima di tutto riempio la prima riga e colonna con 0 dato che rappresentano l'oggetto 0 e il sottoproblema con $C = 0$, sono i casi base.
Prendo il primo oggetto:

D

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

0 1 2 3 4 5 6 7 8 9 10

$v_1=1, w_1=7$

$$c < w_i \Rightarrow d_{i,c} = d_{i-1,c}$$

Notiamo che riempiamo di 0 la riga fino a quando non abbiamo $w_1 = c_i = 7$, perchè è inutile provare a inserire un oggetto di peso 7 in sottoproblemi di capacità inferiore a 7.

Qua controllo quale sia il massimo tra $d_{i-1,c-w_i} + v_i$ e $d_{i-1,c}$, cioè controllo se mi conviene riempire lo zaino inserendo l'oggetto che sto considerando sommato con il valore dell'oggetto precedente (considerando $c - w_i$, quindi un peso che non superi la capacità), oppure mi conviene riempire lo zaino con la riga precedente perchè ha valore maggiore rispetto all'aggiunta del mio oggetto.

Proseguo così fino a riempire tutta la matrice, il valore ottimo sarà in fondo alla matrice a destra (come per LCS).

D

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1	1	1	1
0	0	0	0	1	1	1	1	1	2	2	2	2
0	1	1	1	1	2	2	2	2	2	3	3	3
0	1	2	2	2	2	3	3	3	3	3	5	5

0 1 2 3 4 5 6 7 8 9 10

$v_5=1, w_5=1$

$$c \geq w_i \Rightarrow d_{i,c} = \max\{d_{i-1,c-w_i} + v_i, d_{i-1,c}\}$$

3.6 Algoritmo DP - Codice

```
int calcolo_ottimo_zaino(n, C, V, W)
  for i from 0 to n do
    D[i,0] = 0
  for c from 0 to C do
    D[0,c] = 0
```

```

for i from 1 to n
  for c from 1 to C
    D[i,c] = D[i-1,c]
    if c ≥ wi and D[i-1,c-wi] + vi > D[i,c] then
      D[i,c] = D[i-1,c-wi] + vi
return D[n,C]

```

Tempo e Spazio $\Theta(nC)$.

3.6.1 Ricostruzione Soluzione Ottimale

Come al solito devo ragionare in modalità black box, facendo finta di avere già la soluzione dei sottoproblemi.

$S_{0,4} = \emptyset$

D

0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1	1	1	<u>2</u>
0	0	0	0	1	1	1	1	1	2	2	3
0	1	1	1	1	2	2	2	2	2	3	<u>4</u>
0	1	2	2	2	2	3	3	3	3	3	<u>5</u>
0	1	2	3	4	5	6	7	8	9	10	

$$S_{5,10} = S_{0,4} + \{2\} + \{4\} + \{5\}$$

Ricordiamo che abbiamo i seguenti oggetti:

1. $v_1 = 1, w_1 = 7$
2. $v_2 = 1, w_2 = 4$
3. $v_3 = 1, w_3 = 5$
4. $v_4 = 1, w_4 = 1$
5. $v_5 = 1, w_5 = 1$

Capacità $C = 10$. Parto dalla cella in basso a destra della matrice, quella contenente il valore ottimo e mi chiedo: Qual è la soluzione ottimale di $S_{5,10}$? Utilizzo la stessa equazione di prima, ma a ritroso:

$$c \geq w_i \implies d_{i,c} = \max\{d_{i-1,c-w_i} + v_i, d_{i-1,c}\}$$

Controllo il coefficiente maggiore tra la cella superiore $[i-1, c]$ e $[i-1, c-w_i]$ per controllare se in questa casella è stato incrementato il valore perchè è stato aggiunto l'oggetto allo zaino. Sostituendo il massimo trovato a $S_{5,10}$ se (come in questo caso) ho che il massimo è $d_{i-1, c-w_i} + v_i$, ottengo che:

$$S_{5,10} = S_{4,9} + \{5\}$$

Procedo e analizzo $S_{4,9}$, trovo che i valori sono uguali, scelgo di prendere $d_{i-1, c-w_i}$ e quindi di aggiungere $\{4\}$ alla mia soluzione. Quindi $S_{4,9} = S_{3,9} + \{4\}$

$$S_{5,10} = S_{3,8} + \{4\} + \{5\}$$

Proseguo e analizzo $S_{3,8}$. Qua notiamo che dato che ho peso 5 facendo $[i-1, c] = 1$ e $[i-1, c-w_i] = 0$, il massimo sarà nella casella soprastante, per questo avrò che $S_{3,8} = S_{2,8}$, sostituendo ottengo:

$$S_{5,10} = S_{2,8} + \{4\} + \{5\}$$

Proseguo e analizzo $S_{2,8}$. In questo caso abbiamo che i due valori sono pari, scegliendone uno dei due avrò due soluzioni ottime diverse. In questo caso scegliamo $d_{i-1, c-w_i} + v_i = 0 + 1$ e ottengo quindi: $S_{2,8} = S_{1,4} + \{2\}$

$$S_{5,10} = S_{1,4} + \{2\} + \{4\} + \{5\}$$

Ora mi trovo ad analizzare $S_{1,4}$. In questo caso ho che $c = 4$ e $w_i = 7$ quindi $c < w_i \implies d_{i,c} = d_{i-1,c} = S_{0,4}$. Quindi non vengono aggiunti valori.

$$S_{5,10} = S_{0,4} + \{2\} + \{4\} + \{5\}$$

Infine analizzo $S_{0,4}$, avendo $i = 0$, mi ritrovo in un caso base, quindi $S_{0,4} = \emptyset$.

$$S_{5,10} = \{2\} + \{4\} + \{5\}$$

E ottengo così la soluzione ottimale!

3.6.2 Ricostruzione Soluzione Ottimale - Codice

Input Avrò la matrice D e le coordinate (i,c) di una cella.

Output Stampa una Soluzione Ottimale $S_{i,c}$.

```
Procedura ricostruisci_zaino(D, i, c)
    if i > 0 and c > 0
        if c ≥ wi
```

```
    if D[i,c] == D[i-1, c]
        ricostruisci_zaino(D,i-1,c)
    else
        ricostruisci_zaino(D,i-1,c-wi)
        print i
else
    ricostruisci_zaino(D,i-1,c)
```

Chiamando la procedura per $i = n$ e $c = C$ si ottiene la stampa di una soluzione ottimale.

Capitolo 4

Problema dei cammini minimi - Floyd-Warshall

Come al solito diamo qualche definizione per poter lavorare successivamente in maniera agile.

4.1 Definizioni

4.1.1 Grafo

Un Grafo viene definito come $G = (V, E)$ dove:

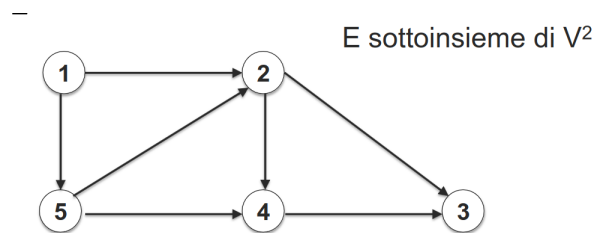
- $V = \{v_1, v_2, v_3, \dots, v_n\}$ insieme di vertici
- $E = \{e_1, e_2, e_3, \dots, e_m\}$ insieme di archi

Dimensione di G $\rightarrow (n, m)$. Arco $e_k \rightarrow$ relazione R tra due vertici v_i e v_j

R può essere

- Simmetrica - Grafo NON Orientato - cioè $v_i R v_j \Leftrightarrow v_j R v_i$
- Asimmetrica - Grafo Orientato (o diretto) - cioè $v_i R v_j \nRightarrow v_j R v_i$

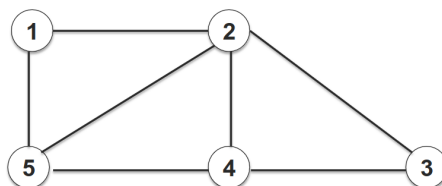
Un grafo orientato è caratterizzato da un verso di percorrenza degli archi unidirezionale. In questo caso E è sottoinsieme di V^2 .



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

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

[Grafo non orientato



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

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

4.1.2 Adiacenza

Un vertice v è adiacente a un vertice u se $(u, v) \in E$.

Per esempio nella rappresentazione del grafo orientato il vertice **1** è adiacente ai vertici **2** e **5**, infatti notiamo che in E è presente $(1, 2), (1, 5)$.

4.1.3 Rappresentazione di un grafo

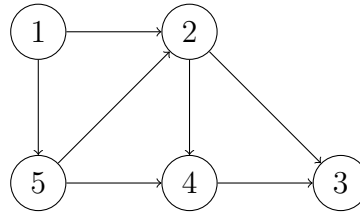
Abbiamo 2 rappresentazioni possibili:

- Liste di adiacenza
- Matrice di adiacenza

1. Le liste di adiacenza utilizzano un vettore L_v di dimensione $|V|$ tale che $V[i]$ è la lista degli adiacenti del vertice v_i . Ogni vertice del grafo avrà un vettore.

2. La matrice di adiacenza è una Matrice M_v di dimensione $n \times n$ tale che $M[i, j] = 1$ se il vertice j è adiacente del vertice i , altrimenti $M[i, j] = 0$.
A differenza delle liste in questo caso ho una sola matrice.

4.1.4 Esempio grafo orientato

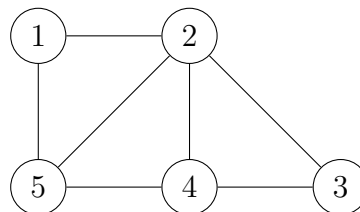


$$M = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 1 & 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 & 1 & 0 \\ 3 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 1 & 0 & 0 & 0 \\ 5 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Dimensione $|V|^2 = n^2$

Numero di celle con 1 $|E|$

4.1.5 Esempio grafo non orientato



$$M = \begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Dimensione $|V|^2 = n^2$

Numero di celle con 1 $2|E|$

4.1.6 Liste VS Matrice (memoria)

Liste di adiacenza Sono ottime dal punto di vista dell'occupazione dello spazio nel caso di Grafi sparsi con $|E|$ molto minore di $|V|^2$.

Matrici di adiacenza Risultano migliori nei grafi densi quindi quando ho $|E|$ che si avvicina a $|V|^2$.

4.1.7 Liste VS Matrice (tempo)

(u,v) Intendiamo se i 2 vertici sono collegati. Come tempo intendiamo il tempo per stabilire se (u,v) appartiene ad E e i tempi sono i seguenti:

- Liste di adiacenza $\rightarrow O(|E|) = O(m)$
- Matrice di adiacenza $\rightarrow O(1)$

4.1.8 Cammino in un grafo orientato

Definizione di cammino Sequenza $P = \langle v_{i_1}, v_{i_2}, \dots, v_{i_{k-1}}, v_{i_k} \rangle$ tale che v_{i_k} appartiene a V per $1 \leq j \leq k$ e $(v_{i_j}, v_{i_{j+1}})$ appartiene ad E per $1 \leq j < k$.

Lunghezza del cammino $k - 1$ (numero di archi)

Ciclo Cammino in cui v_{i_1} coincide con v_{i_k}

Cammino semplice Cammino in cui ogni vertice è presente una volta sola (cioè non contiene cicli)

Predecessore di v_{i_k} in P Vertice di $v_{i_{k-1}}$

4.1.9 Grafo orientato pesato

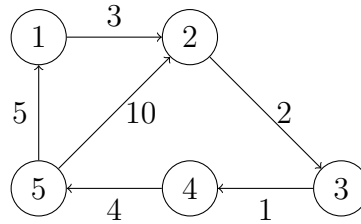
Grafo $G = (V, E, W)$

- $V = \{v_1, v_2, v_3, \dots, v_n\}$ insieme di vertici
- $E = \{e_1, e_2, e_3, \dots, e_m\}$ insieme di archi

- $W : E \rightarrow R$ tale che $W(v_i, v_j) = w_{ij}$ è il peso dell'arco (v_i, v_j)

Peso di un cammino Si tratta della somma dei pesi di tutti gli archi, formalmente: $P = \langle v_{i_1}, v_{i_2}, \dots, v_{i_k} \rangle \rightarrow \sum_{j=1}^{k-1} w(v_{i_j}, v_{i_{j+1}})$

4.1.10 Esempio grafo orientato pesato



4.2 Il problema dei cammini minimi

Input Grafo $G = (V, E, W)$ (senza cappi) orientato e pesato

Output Per ogni coppia di vertici i e j , trovare il cammino di peso minimo (cammino minimo) che parte da i e finisce in j .

Si tratta di un problema di ottimizzazione di minimo, dove

- $(n) \rightarrow$ dimensione del problema
- Soluzioni possibili per una coppia di vertici i e j sono tutti i cammini da i a j
- Funzione obiettivo è il peso del cammino
- Peso del cammino minimo da i a j è il valore ottimo (per i e j)
- Un cammino minimo tra i vertici i e j è la soluzione ottimale

4.2.1 L'input

Funzione peso W $W : E \rightarrow R^+$ tale che $W(i, j) = w_{ij} =$ peso dell'arco (i, j) .

Funzione peso W - Versione estesa $W : V \times V \rightarrow R^+$ tale che $W(i, j) = w_{ij}$ con:

- $w_{ij} = 0$ se $i = j$
- $w_{ij} = \text{peso dell'arco } (i, j)$, se $(i, j) \in E$
- $w_{ij} = \infty$, se $i \neq j$ e $(i, j) \notin E$

Matrice $W = [w_{ij}]$ di n righe e n colonne.

4.2.2 L'output Matrici D e Π

- Matrice $D = [d_{ij}]$ di n righe e n colonne, dove $[d_{ij}]$ è il peso del cammino minimo da i a j
- Matrice $\Pi = [\pi_{ij}]$ di n righe e n colonne, dove π_{ij} è il predecessore di j nel cammino minimo da i a j

Matrice D

- $d_{ij} = 0$ se $i = j$
- $d_{ij} = \text{peso del cammino minimo}$, se esiste un cammino da i a j
- $d_{ij} = \infty$ se non esiste un cammino da i a j

Matrice Π

- $\pi_{ij} = NIL$, se $i = j$
- $\pi_{ij} = u$ appartenente al cammino minimo da i a j , tale che $(u, j) \in E$, se esiste un cammino da i a j
- $\pi_{ij} = NIL$, se non esiste un cammino da i a j

Dopo aver riempito entrambe le matrici mi rendo conto che:

La riga i di Π fornisce l'albero dei predecessori relativo al vertice i .

Albero dei predecessori del vertice i (riga i di Π)

- $\{j \in V \mid \pi_{ij} \neq NIL\} \cup \{i\} \rightarrow$ insieme dei vertici
- $(\pi_{ij}, j) \mid \pi_{ij} \neq NIL$

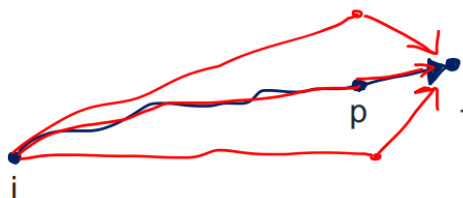
4.3 Sottostruttura ottima (primo tentativo)

Consideriamo come P_{ij} il **Cammino minimo da i a j** e p è il predecessore di j.

Sicuramente $P_{ij} = P_{ip} + \langle j \rangle$, con P_{ip} cammino minimo da i a p.



Con P_{ip} cammino minimo da i a p. Come potrei trovare P_{ij} ?



1. Considero tutti i vertici p' tali che $(p', j) \in E$
2. Per ogni vertice p' determino il cammino dato da: $P_{ip'} + \langle j \rangle$
3. Seleziono il cammino di peso minimo

Attenzione! Non è sicuro che quando si calcola P_{ij} si abbiano già a disposizione i cammini $P_{ip'}$. Si deve parametrizzare rispetto alla lunghezza l del cammino:

1. Prima calcolo tutti i cammini minimi a lunghezza 0 $\rightarrow P_{ij}^0$
 $P_{ij}^0 = \langle i \rangle$ se $i = j$, altrimenti $P_{ij}^0 = \infty$
2. Poi calcolo tutti i cammini minimi a lunghezza 1 $\rightarrow P_{ij}^1$
 $P_{ij}^1 = \langle i, j \rangle$ se $i \neq j$ e $(i, j) \in E$, altrimenti $P_{ij}^1 = \infty$
3. Poi calcolo tutti i cammini minimi a lunghezza 2 $\rightarrow P_{ij}^2$
4. Poi calcolo tutti i cammini minimi a lunghezza 3 $\rightarrow P_{ij}^3$
5. ...
6. Ci si ferma per $l = |E| = m$ (l è lunghezza)
7. Per ogni coppia i e j scelgo tra i cammini $P_{ij}^0, P_{ij}^1, \dots, P_{ij}^m$, quello di peso minimo

Friendly Reminder $\langle i, j \rangle$ Significa perorso con i vertici i e j.

Domande Come calcolo P_{ij}^l con $l \geq 2$?

E qual è il tempo nel caso peggiore dell'algoritmo DP che sfrutta questa struttura ottima?

4.3.1 Diamo un ordine ai vertici del grafo

1 viene prima di 2 che viene prima di 3 etc. che viene prima dell'ultimo vertice n.

Parametrizziamo rispetto ai vertici intermedi del cammino:

- Trovo $P_{ij}^0 \rightarrow$ cammino minimo senza vertici intermedi
- Trovo $P_{ij}^1 \rightarrow$ cammino minimo con vertici intermedi $\in \{1\}$
- Trovo $P_{ij}^2 \rightarrow$ cammino minimo con vertici intermedi $\in \{1, 2\}$
- Trovo $P_{ij}^3 \rightarrow$ cammino minimo con vertici intermedi $\in \{1, 2, 3\}$
- Trovo $P_{ij}^n \rightarrow$ cammino minimo con vertici intermedi $\in \{1, 2, \dots, n\}$

Analizziamo nel dettaglio i cammini minimi intermedi

$P_{ij}^0 \rightarrow$ cammino minimo senza vertici intermedi.

$$\begin{aligned} P_{ij}^0 &= \langle i \rangle & \text{se } i = j \\ P_{ij}^0 &= \langle i, j \rangle & \text{se } i \neq j \text{ e } (i, j) \in E \\ P_{ij}^0 &= NIL & \text{se } i \neq j \text{ e } (i, j) \notin E \end{aligned}$$

Per $k > 0$, $P_{ij}^k \rightarrow$ cammino minimo con vertici intermedi $\in \{1, 2, \dots, k\}$.

Per $k = n$, $P_{ij}^n \rightarrow$ cammino minimo con vertici intermedi $\in \{1, 2, \dots, n\}$.

Quindi $P_{ij}^n \rightarrow$ cammino minimo P_{ik} .

Sottoproblema di dimensione k Per ogni coppia (i,j), trovare il cammino minimo P_{ij}^k dal vertice i al vertice j che ha vertici intermedi $\in \{1, 2, \dots, k\}$ se $k > 0$, oppure non ha vertici intermedi se $k = 0$.

$$\begin{aligned} k &\in \{0, 1, \dots, n\} \\ i &\in \{1, \dots, n\} \\ j &\in \{1, \dots, n\} \end{aligned}$$

Numero di sottoproblemi $n \times n \times (n + 1)$.

$k = n \rightarrow P_{ij}^n = P_{ij}$.

4.3.2 Equazioni di ricorrenza

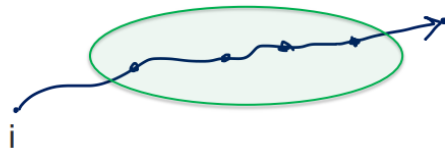
Casi base Sottoproblema di dimensione (0)

$$\begin{aligned} P_{ij}^0 &= \langle i \rangle && \text{se } i = j \\ P_{ij}^0 &= \langle i, j \rangle && \text{se } i \neq j \text{ e } (i, j) \in E \\ P_{ij}^0 &= NIL && \text{se } i \neq j \text{ e } (i, j) \notin E \end{aligned}$$

Passo ricorsivo Tutti i sottoproblemi di dimensione (k) tale che $k > 0$

$$P_{ij}^k = ?$$

Ricerchiamo la sottostruttura ottima.



Data una soluzione ottimale $P_{ij} = P_{ij}^n$ **si possono verificare due casi:**

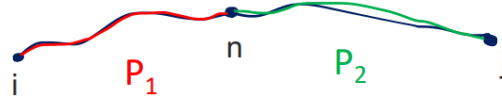
1. Il vertice n NON è uno dei vertici intermedi
2. Il vertice n è uno dei vertici intermedi

Caso 1 - Il vertice n NON è uno dei vertici intermedi

- P_{ij}^n coincide con P_{ij}^{n-1}
- Predecessore di j in P_{ij}^n coincide con predecessore di j in P_{ij}^{n-1}

Caso 2 - Il vertice n è uno dei vertici intermedi

$$P_{ij}^n = P_1 + P_2$$



$$P_1 = P_{in}^{n-1} \rightarrow P_{ij}^n = P_1 + P_2 = P_{in}^{n-1} + P_2$$

Mentre per quanto riguarda P_2 avrò che

$$P_2 = P_{nj}^{n-1}$$

Quindi sostituendo P_2 all'interno dell'equazione avrò che:

$$P_2 = P_{nj}^{n-1} \rightarrow P_{ij}^n = P_1 + P_2 = P_{in}^{n-1} + P_{nj}^{n-1}$$

Abbiamo quindi che il predecessore di j in P_{ij}^n coincide con il predecessore di j in P_{nj}^{n-1} .

Passo ricorsivo per P_{ij}^n

La soluzione ottimale $P_{ij}^n = P_{ij}$ è data da:

$$P_{ij}^n = \min_p \{P_{ij}^{n-1}, P_{in}^{n-1} + P_{nj}^{n-1}\}$$

$$i = n \text{ oppure } j = n \rightarrow P_{ij}^n = P_{ij}^{n-1}.$$

Passo ricorsivo per P_{ij}^k

La soluzione ottimale $P_{ij}^k (k > 0)$ è data da:

$$P_{ij}^k = \min_p \{P_{ij}^{k-1}, P_{ik}^{k-1} + P_{kj}^{k-1}\}$$

$$i = k \text{ oppure } j = k \rightarrow P_{ij}^k = P_{ij}^{k-1}.$$

4.3.3 Equazioni di ricorrenza

Riassumendo abbiamo le seguenti equazioni di ricorrenza:

k=0 (CASI BASE)

$$\begin{aligned} P_{ij}^0 &= \langle i \rangle & \text{se } i = j \\ P_{ij}^0 &= \langle i, j \rangle & \text{se } i \neq j \text{ e } (i, j) \in E \\ P_{ij}^0 &= NIL & \text{se } i \neq j \text{ e } (i, j) \notin E \end{aligned}$$

$k > 0$ (**PASSO RICORSIVO**)

$$P_{ij}^k = \min_p P_{ij}^{k-1}, P^{k-1}ik + P_{kj}^{k-1}$$

Definizione dei coefficienti

Coefficienti d_{ij}^k dei sottoproblemi.

$d_{ij}^k \rightarrow$ peso del cammino P_{ij}^k

$$k \in \{0, 1, \dots, n\}$$

$$i \in \{1, \dots, n\}$$

$$j \in \{1, \dots, n\}$$

Quindi abbiamo **Numero di coefficienti** uguale a $n \times n \times (n + 1)$.

$k = n \rightarrow d_{ij}^n$ è il peso d_{ij} di P_{ij} .

Ricordiamo che la funzione obiettivo è trovare il peso del cammino, definiamo quindi i coefficienti nella seguente maniera:

k=0 (CASI BASE)

$$d_{ij}^0 = 0 \quad \text{se } i = j$$

$$d_{ij}^0 = w_{ij} \quad \text{se } i \neq j \text{ e } (i, j) \in E$$

$$d_{ij}^0 = \infty \quad \text{se } i \neq j \text{ e } (i, j) \notin E$$

$k > 0$ (**PASSO RICORSIVO**)

$$d_{ij}^k = \min_p d_{ij}^{k-1}, d^{k-1}ik + d_{kj}^{k-1}$$

Predecessori π_{ij}^k

$\pi_{ij}^k \rightarrow$ predecessore del vertice j in P_{ij}^k

$$k \in \{0, 1, \dots, n\}$$

$$i \in \{1, \dots, n\}$$

$$j \in \{1, \dots, n\}$$

Numero di predecessori:: $n \times n \times (n + 1)$.

$\pi_{ij}^n \rightarrow$ predecessore π_{ij} di j in P_{ij} .

Aggiungiamo alle equazioni di ricorrenza anche i predecessori.

k=0 (CASI BASE)

$$\begin{aligned}
d_{ij}^0 &= 0 \quad \pi_{ij}^0 = NIL \quad \text{se } i = j \\
d_{ij}^0 &= w_{ij} \quad \pi_{ij}^0 = i \quad \text{se } i \neq j \text{ e } (i, j) \in E \\
d_{ij}^0 &= \infty \quad \pi_{ij}^0 = NIL \quad \text{se } i \neq j \text{ e } (i, j) \notin E
\end{aligned}$$

k > 0 (PASSO RICORSIVO)

$$\begin{aligned}
d_{ij}^k &= \min_p d_{ij}^{k-1}, d^{k-1}ik + d_{kj}^{k-1} \\
\pi_{ij}^k &= \pi_{ij}^{k-1} \text{ se } d_{ij}^k = d_{ij}^{k-1} \text{ altrimenti } \pi_{ij}^k = \pi_{kj}^{k-1}
\end{aligned}$$

4.4 Algoritmo bottom-up

Per ogni valore di k da 0 a n, vengono calcolate due matrici ($n \times n$):

$$\begin{aligned}
D^k &= [d_{ij}^k] \\
\Pi^k &= [\pi_{ij}^k]
\end{aligned}$$

Il numero totale di matrici è $2(n+1)$.

Caso Base Ho che:

$$\begin{aligned}
D^0 &= [d_{ij}^0] = W \text{ matrice dei pesi input} \\
\Pi^0 &= [\pi_{ij}^0]
\end{aligned}$$

Passo Ricorsivo In questo caso ho:

$$D^k = [d_{ij}^k] \text{ e } \Pi^k = [\pi_{ij}^k]$$

Sono calcolate usando le matrici $D^{k-1} = [d_{ij}^{k-1}]$ e $\Pi^{k-1} = [\pi_{ij}^{k-1}]$

Avrò quindi che le matrici $D^n = [d_{ij}^n]$ e $\Pi^n = [\pi_{ij}^n]$ sono le matrici di output!

4.4.1 Algoritmo bottom-up - Codice

```

Procedura calcola_valori_ottimi_FW(V,E,W)
  D0 = W
  Π0 = (n x n) matrix of NIL values
  for i = 1 to n do
    for j = 1 to n do

```

```

    if i != j and wij != ∞ then
        Π0[i, j] = i
for k = 1 to n do
    for i = 1 to n do
        for j = 1 to n do
            Dk[i, j] = Dk-1[i, j]
            Πk[i, j] = Πk-1[i, j]
            if i != k and j != k then
                if Dk[i, j] > Dk-1[i, k] + Dk-1[k, j] then
                    Dk[i, j] = Dk-1[i, k] + Dk-1[k, j]
                    Πk[i, j] = Πk-1[k, j]

```

Tempo $\Theta(n^3)$.

Rappresentazione esecuzione algoritmo

Link al video Youtube.

Capitolo 5

Algoritmi Greedy

Si tratta di una tecnica che si applica sempre ai problemi di ottimizzazione, ma rispetto alla programmazione dinamica ha un approccio diverso, dato che il calcolo della soluzione ottima (in questo caso ne calcola una sola) avviene attraverso una sequenza di scelte **localmente** ottime.

Caratteristiche degli algoritmi Greedy

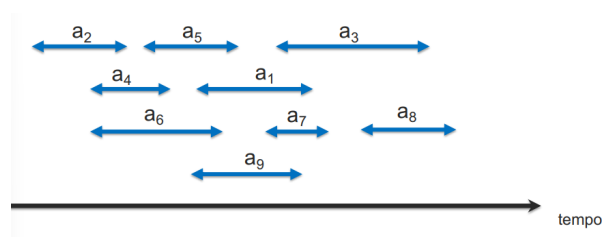
- Semplici da scrivere
- Efficienti

Questioni

- Dimostrare la correttezza di un algoritmo greedy
- Capire quali problemi sono affrontabili con una strategia greedy

5.1 Problema - Selezione attività

INPUT Dato un insieme $A = \{a_1, a_2, \dots, a_n\}$ di n attività, tale che $a_i = [s_i, e_i)$ per $1 \leq i \leq n$, dove s_i è il tempo di inizio ed e_i è il tempo di fine.



$a_i = [s_i, e_i)$ e $a_j = [s_j, e_j)$ sono **compatibili** se $s_i \geq e_j$ oppure $s_j \geq e_i$. In poche parole se un'attività inizia nello stesso momento della fine dell'altra, oppure dopo. Le attività non devono accavallarsi, cioè eseguirsi nello stesso tempo di un'altra. Facciamo qualche esempio che sicuramente è più semplice. Per esempio $a_5 = [s_5, e_5)$ e $a_8 = [s_8, e_8)$ sono compatibili, mentre $a_5 = [s_5, e_5)$ e $a_1 = [s_1, e_1)$ NON sono compatibili, infatti l'inizio di a_1 è minore della fine di a_5 .

OUTPUT il sottoinsieme X di cardinalità massima composto di attività mutuamente compatibili. In questo esempio l'OUTPUT desiderato è $X = \{a_2, a_5, a_7, a_8\}$.

5.1.1 Soluzione con DP

$A = \langle a_1, a_2, \dots, a_n \rangle$ tale che $e_1 \leq e_2 \leq \dots \leq e_n$.

$A = A \cup \{a_0, a_{n+1}\} = \langle a_0, a_1, \dots, a_n, a_{n+1} \rangle$ tale che $e_0 \leq s_1$ e $s_{n+1} \geq e_n$.

Sottoproblema (i,j) per $0 \leq i < j \leq n+1$ Trovare il sottoinsieme X_{ij} di attività mutuamente compatibili di cardinalità massima per $A_{ij} = \langle a_{i+1}, a_{i+2}, \dots, a_j \rangle$.

Sottoproblema (0,n+1) Trovare il sottoinsieme $X_{0,n+1} = X$ di attività mutuamente compatibili di cardinalità massima per $A_{0,n+1} = \langle a_1, a_2, \dots, a_n \rangle = A$.

Numero totale di sottoproblemi $\rightarrow (n+1) + n + (n-1) + (n-2) + \dots + 1$.

CASI BASE per $j = i+1 (A_{ij} = \emptyset)$ $X_{ij} = \emptyset$.

PASSO RICORSIVO per $j > i+1 (A_{ij} \neq \emptyset)$ **Sottostruttura ottima**

a_k appartiene a $X_{ij} \implies X_{ij} = X_{ik} \cup \{a_k\} \cup X_{kj}$

X_{ik} soluzione ottima di A_{ik}

X_{kj} soluzione ottima di A_{kj}

$X_{ij} = \max\{X_{ik} \cup \{a_k\} \cup X_{kj} \text{ per } i < k < j\}$.

Valore ottimo - Sostituzione coefficiente all'equazione

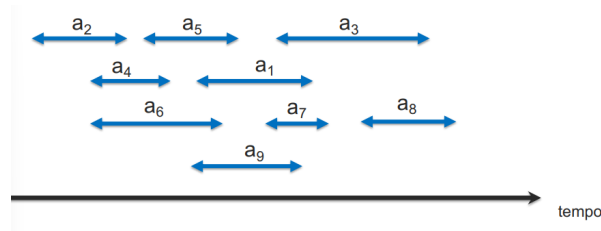
CASI BASE per $j = i+1 (A_{ij} = \emptyset)$ $c_{ij} = 0$ (valore ottimo)

PASSO RICORSIVO per $j > i+1 (A_{ij} \neq \emptyset)$ $c_{ij} = \max\{c_{ik} + 1 + c_{kj} \text{ t.c. } i < k < j\}$ (valore ottimo).

5.1.2 Svantaggi della Soluzione tramite DP

1. Tutti i sottoproblemi devono essere risolti per arrivare a calcolare il valore ottimo
2. Si deve in seguito ricostruire la soluzione ottima (soluzione ottimale) perchè io ho solo i coefficienti, non ho la sequenza richiesta in OUTPUT

5.1.3 Approccio greedy

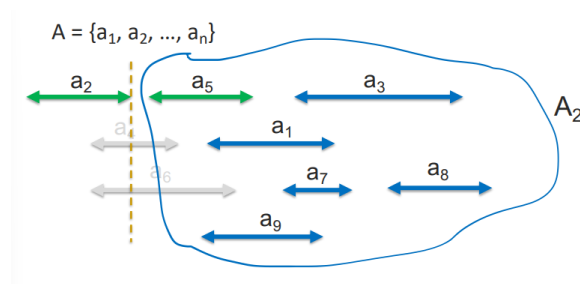


In questo caso dobbiamo scegliere un parametro, per esempio \rightarrow Attività con il tempo di fine più basso, che in questo caso è a_2 .

IPOTESI: a_2 appartiene alla soluzione ottima $X \implies X = \{a_2\} \cup X_2$.

X_2 è la soluzione ottima per $A_2 = \{a = [s, e) \in A \mid \text{a compatibile con } a_2\}$

Compatibile con a_2 significa che $s \geq e_2$. Graficamente devo avere che la fine di a_2 non si intersechi nessuna attività. Quindi devo cercare l'attività con il tempo di fine più basso in $A_2 \rightarrow a_5$.



Scelgo quindi a_5 dato che ha il tempo di fine più basso rispetto a tutte le attività compatibili con a_2 .

IPOTESI a_5 appartiene alla soluzione ottima $X_2 \implies X = \{a_2\} \cup \{a_5\} \cup X_2$.

X_5 è soluzione ottima per $A_5 = \{a = [s, e) \in A \mid s \geq e_5\}$.

Ancora una volta cerco le attività compatibili con a_5 , che abbiano quindi un inizio che sia maggiore o uguale rispetto alla fine di a_5 , e scelgo quella con tempo di fine minore.

Determino quindi che $A_5 \rightarrow a_7$.

IPOTESI: a_7 appartiene alla soluzione ottima $X_5 \implies X = \{a_2\} \cup \{a_5\} \cup$

$\{a_7\} \cup X_7$.

Cerco nuovamente l'attività con tempo di fine più basso rispetto a quelle compatibili con a_7 e trovo che l'attività in questione è a_8 .

IPOTESI: a_8 appartiene alla soluzione ottima X_7 .

Quindi questo implica che $X = \{a_2\} \cup \{a_5\} \cup \{a_7\} \cup \{a_8\} \cup X_8$.

X_8 è soluzione ottima per $A_8 = \{a = [s, e) \in A \mid t.c \ s \geq e_8\}$.

Ma dato che non ho più eventi compatibili con a_8 (perchè sono finiti, ma valeva anche il caso che c'erano altri eventi che NON compatibili), $A_8 = \emptyset$. Significa che sono arrivato ad avere la soluzione e guardandola notiamo che è una soluzione ottimale.

$$X = \{a_2, a_5, a_7, a_8\}$$

5.1.4 Osservazioni sulla risoluzione Greedy

Notiamo che ad ogni passo la scelta localmente ottima minimizza il tempo di fine. Sono state effettuate 4 scelte localmente ottime, sono quindi stati risolti 4 sottoproblemi.

In sintesi Ad ogni passo:

1. effettuo una scelta localmente ottima
2. risolvo il sottoproblema generato dalla scelta appena effettuata
3. la scelta non dipende dalle scelte successive (Greedy è anche detto algoritmo miope)
4. la scelta riduceo il sottoproblema da risolvere (approccio Top-Down)

5.1.5 Codice Greedy

```

Procedura greedy_scheduling(A)
  n = |A|
  a_s = attivita' di A con il minore tempo di fine
  X = {a_s}
  while a_s ≠ NIL do
    a_s = a=[s,t) t.c s ≥ e_s e minore tempo di fine
  return X

```

Tempo di esecuzione $O(n)^2$, determinato dal While, perchè le altre operazioni o sono costanti o nel caso della ricerca dell'evento compatibile con tempo di fine minore impiegano tempo $O(n)$.

Miglioramento Posso migliorare l'algoritmo ordinando le attività di A per tempo di fine non decrescente, prima di eseguire le operazioni di ricerca evento minimo, inserisco quindi un tempo di $O(n \log n)$ determinato dall'algoritmo di ordinamento (es. MergeSort o QuickSort).

1. Ordine le attività di A per tempo di fine non decrescente
2. Aggiungo a_1 alla soluzione ottima X
3. Aggiungo a X la prima attività dopo a_1 che ha tempo di inizio $\geq e_1$
4. Aggiungo a X la prima attività dopo la precedente che ha tempo di inizio \geq al suo tempo di fine
5. Aggiungo a X la prima attività dopo la precedente che ha tempo di inizio \geq al suo tempo di fine
6. Etc.
7. Mi fermo quando ho considerato tutte le attività

```

Procedura greedy_scheduling(A)
begin
  n ← |A|
  ordina A per tempo di fine non decrescente  $O(n \log n)$ 
  X ← {a1}
  k ← 1
  for i from 2 to n do
    if si ≥ ek then
      X ← X ∪ {ai}
      k ← i
  return X
end

```

$O(n)$

5.2 Greedy VS DP

- Soluzione ottima **VS** Valore ottimo + Soluzione ottima
- Top-down **VS** Bottom-up
- Pochi sottoproblemi da risolvere **VS** Tanti sottoproblemi da risolvere

- Efficiente e semplice da scrivere **VS** Meno efficiente e più complicato da scrivere
- Applicabilità inferiore (te pareva) **VS** Applicabilità maggiore

5.2.1 Due ingredienti chiave di un algoritmo Greedy

- Proprietà della sottostruttura ottima (tipica anche della Programmazione Dinamica)
- Proprietà della **scelta greedy**

Per la **Correttezza di un algoritmo Greedy** si deve DIMOSTRARE che la sequenza di scelte localmente ottima conduce a una soluzione globalmente ottima.

Proprietà della scelta greedy: la scelta che compio ad ogni passo appartiene a una soluzione ottima del sottoproblema che sto risolvendo in quel momento.

5.3 Correttezza di un algoritmo Greedy

Analizziamo la sottostruttura del problema della selezione di attività.

Sottoproblema al passo generico: Trovare il sottoinsieme di cardinalità massima di attività compatibili, per l'insieme di attività che iniziano dopo la fine dell'attività aggiunta al passo precedente.

Sottoproblema 1: sottoinsieme di cardinalità massima di attività compatibili, per l'insieme A

Sottoproblema 2: sottoinsieme di cardinalità massima di attività compatibili, per l'insieme A_2 delle attività che hanno tempo di inizio \geq al tempo di fine della scelta fatta al passo 1.

...

Sottoproblema k: sottoinsieme di cardinalità massima di attività compatibili, per l'insieme A_k delle attività che hanno tempo di inizio \geq al tempo di fine della scelta fatta al passo $k - 1$.

Scelta localmente ottima per il sottoproblema $k \rightarrow$ Attività a_s con il minor tempo di fine A_k .

Proprietà della scelta greedy da dimostrare Ad ogni passo l'attività scelta è inclusa in una soluzione ottima del sottoproblema che sto risolvendo. Basta dimostrare che a_1 è inclusa in una soluzione ottima per l'insieme A.

Dimostrazione

Suppongo X soluzione ottima per A. Suppongo a'_1 attività con il minore tempo di fine X.

Se a'_1 uguale ad a_1 la dimostrazione è finita, altrimenti sostituisco in X l'attività a'_1 con a_1 .

Le attività in X sono ancora disgiunte. La dimensione di X non è cambiata.

Conclusione: il nuovo X è un sottoinsieme massimo di attività compatibili di A che include ora a_1 .

5.3.1 Cosa succede se sbaglio parametro?

Consideriamo per esempio la durata massima come parametro, cosa succede alle soluzioni?

Prendendo l'esempio precedente notiamo subito che gli eventi della soluzione sarebbero i più lunghi quindi a_3 e a_6 e questa non è una soluzione ottima.

Se scegliamo la durata minima otteniamo che in ordine di scelta a_7, a_4, a_8 , anche in questo caso questa non è una soluzione ottima.

Ci accorgiamo inoltre che i valori scelti inizialmente non sono soluzione ottima del sottoproblema, mi rendo conto abbastanza velocemente quindi che sto sbagliando, ma analizziamo nel dettaglio la questione considerando il seguente problema.

5.4 Il problema dello Zaino Frazionario

Dati n oggetti $\langle 1, 2, \dots, n \rangle$, un intero C e due funzioni.

- $V : X \rightarrow N$ $V(i) = v_i$, valore dell'oggetto i
- $W : X \rightarrow N$ $W(i) = w_i$, peso dell'oggetto i

Trovare una sequenza $P = \langle p_1, p_2, \dots, p_n \rangle$ con p_i in $[0, 1]$ per $1 \leq i \leq n$, tale per cui:

$$\sum_{i=1}^n p_i w_i \leq C$$

$$\sum_{i=1}^n p_i v_i \rightarrow \text{massimo valore}$$

In questo caso, a differenza del problema dello zaino 0/1, posso scegliere di frazionare gli oggetti, quindi non devo per forza prendere tutto l'oggetto, ma posso prenderne anche solo una parte.

5.5 Possibile strategia Greedy

- Ordino gli oggetti per valore non crescente
- Prendo la maggiore quantità del primo oggetto compatibile con la capacità dello zaino
- Prendo la maggiore quantità del secondo oggetto compatibile con la capacità residua dello zaino
- Prendo la maggiore quantità del terzo oggetto compatibile con la capacità residua dello zaino
- etc.
- Mi fermo non appena lo zaino è completamente pieno

Vediamo se questa strategia funziona:

Istanza Problema Oggetti $\rightarrow \langle 1, 2, 3 \rangle$

1. $v_1 = 10, w_1 = 20$
2. $v_2 = 9, w_2 = 8$
3. $v_3 = 8, w_3 = 5$

La capacità totale $C = 20$, mentre c è la capacità residua.

Proviamo inserendo quindi la maggiore quantità di oggetto con valore maggiore.

Inserisco quindi l'oggetto 1, avendo capacità 20 e peso 20 posso inserirlo tutto.

Avendo peso 20 ho saturato completamente la capacità dello zaino e ho un valore di 10, ottengo quindi che la soluzione $P = \langle 1, 0, 0 \rangle$ con appunto Valore $\rightarrow 10$.

È facile osservare che NON si tratta della soluzione ottima, questo significa che ho sbagliato a scegliere il parametro per la strategia Greedy.

5.5.1 Cambio parametro

Se scegliessi iniziassi a riempire lo zaino considerando il rapporto tra peso e valore, otterrei i seguenti valori:

1. $v_1 = 10, w_1 = 20, \frac{v_1}{w_1} = 0.5$
2. $v_2 = 9, w_2 = 8, \frac{v_2}{w_2} = 1.125$
3. $v_3 = 8, w_3 = 5, \frac{v_3}{w_3} = 1.6$

Risulta quindi evidente che conviene inserire prima l'oggetto 3 perchè posso inserire più parti di oggetto e avere più valori rispetto a inserire lo stesso peso degli altri.

Inserisco quindi tutto il terzo oggetto, occupando capacità 5, inserisco quindi il secondo oggetto, che ha rapporto peso valore maggiore rispetto al primo. Dopo aver inserito il secondo oggetto, noto di avere ancora dello spazio disponibile, quindi procedo a inserire il primo oggetto che però non è possibile inserire tutto, ne inserisco quindi peso 7, cioè la capacità residua.

Otengo quindi $P = \langle 0.35, 1, 1 \rangle$ e Valore $\rightarrow 20.5$ che oltre a essere un valore maggiore del primo tentativo è soluzione ottima. Questo è quindi il parametro corretto.

5.5.2 Strategia Greedy Attuata

- Calcolo per ogni oggetto i il valore specifico v_i/w_i
- Ordino gli oggetti per valore specifico non crescente
- Prendo la maggiore quantità del primo oggetto compatibile con la capacità dello zaino
- Prendo la maggiore quantità del secondo oggetto compatibile con la capacità residua dello zaino
- ...
- Mi fermo appena lo zaino è completamente pieno

Generico passo i Scelta localmente ottima \rightarrow percentuale di oggetto i-esimo da prendere $p_i = \min(c/w_i, 1)$.

c è C diminuita del peso totale degli oggetti da 1 a $i-1$.

Sottoproblema lasciato dalla scelta di p_i :

- Oggetti da $i+1$ a n
- Capacità C diminuita del peso totale degli oggetti da 1 a i

5.5.3 Codice

Inserire Codice

5.5.4 Proprietà della scelta Greedy (da dimostrare)

La percentuale p_i scelta per l'oggetto i è inclusa in una soluzione ottima relativa al sottoproblema:

- oggetti da i a n
- Capacità residua dello zaino pari a C diminuita del peso totale degli oggetti aggiunti da 1 a $i-1$

Basta dimostrare la proprietà per la percentuale p_1 e sottoproblema:

- Oggetti da 1 a n
- Capacità C dello zaino

5.6 Algoritmi Greedy in Generale

5.6.1 Codice generico

```

Procedura general_greedy( $S = \{s_1, s_2, \dots, s_n\}$ )
    Calcola per ogni elemento un certo parametro
    Ordina  $S$  sulla base del parametro calcolato
     $X = \emptyset$ 
    for  $i$  from 1 to  $n$  do
        if  $s_i$  is la scelta localmente ottima then
             $X = \{s_i\} \cup X$ 
    return  $X$ 

```

5.7 Non tutti i problemi ammettono un algoritmi Greedy

Il problema zaino 0/1 non ammette un algoritmo di tipo greedy come soluzione. Nelle slide vengono fatti degli esempi che mostrano che è impo-dibile identificare una strategia greedy valida.

Questo perchè non tutti i problemi ammettono un algoritmo Greedy, ma la vera questione è: **Come capire, in linea di principio, se un problema**

ammette un algoritmo Greedy?

Per capire questo aspetto dobbiamo introdurre una struttura denominata Matroide.

Matroide: struttura combinatoria a cui è associato un algoritmo greedy.

5.7.1 Sistema di Indipendenza

Una coppia (S, F)

- S , insieme finito $\{s_1, s_2, \dots, s_n\}$ di elementi
- F , una famiglia di sottoinsiemi di S , cioè un sottoinsieme dell'insieme $P(S)$ delle parti di S

Esempio

$$S = \{1, 2, 3\}$$

$$P(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

$$F = \{\{1\}, \{3\}, \{1, 3\}\}$$

Una coppia (S, F) così definita è un **Sistema di Indipendenza** se vale la seguente proprietà:

preso $A \in F$ **allora un qualsiasi** $B \subseteq A$ **appartiene a** F .

Elementi di $F \rightarrow$ sottoinsiemi indipendenti.

Esempio

- S , insieme $\{1, 2, \dots, n\}$ di n oggetti a cui è associato un peso
- F , famiglia dei sottoinsiemi di S che hanno peso totale $\leq C$

è un Sistema di Indipendenza, cioè:

$$A \in F, B \subseteq A \implies B \in F$$

PROOF: Se $A \in F$, allora il suo peso totale è $\leq C$.

Un qualsiasi $B \subseteq A$, avrà peso totale $\leq C$ e quindi anche $B \in F$.

5.7.2 Proprietà di scambio

Un sistema di Indipendenza (S, F) è un **matroide** se vale la seguente **proprietà di scambio**:

Per qualsiasi $A, B \in F$ tali che $|B| = |A| + 1$, allora esiste almeno un elemento $b \in B - A$ tale che $\{b\} \cup A \in F$.

NB in un matroide l'insieme vuoto è un insieme indipendente.

5.7.3 Esempio 1

Coppia (S, F) :

- S , insieme finito di oggetti
- F , famiglia dei sottoinsiemi di S di cardinalità $\leq k$

(S, F) è un Sistema di Indipendenza, cioè:

$$A \in F, B \subseteq A \implies B \in F$$

PROOF: Se $A \in F$, allora la sua cardinalità è $\leq k$.

Un qualsiasi $B \subseteq A$ avrà cardinalità $\leq k$ e quindi anche $B \in F$.

Proprietà Scambio Per (S, F) vale la proprietà dello scambio, cioè:

$$\forall A, b \in F \text{ t.c. } |B| = |A| + 1 \implies \exists b \in B - A \text{ t.c. } \{b\} \cup A \in F$$

PROOF Se A e $B \in F$, allora $|A| \leq k$ e $|B| \leq k$, A ha un elemento in meno rispetto a B quindi $|A| < k$.

Un qualsiasi $b \in B - A$ che viene aggiunto ad A produce un insieme di cardinalità $\leq k$, che quindi appartiene a F .

5.7.4 Esempio 2

Coppia S, F :

- S , insieme E degli archi di un grafo non orientato
- F , famiglia dei sottoinsiemi di S composti da archi che hanno un vertice in comune

(S, F) è un Sistema di Indipendenza, cioè:

$$A \in F, B \subseteq A \implies B \in F$$

PROOF Se $A \in F$, allora A è composto da archi incidenti in un vertice v . Un qualsiasi $B \subseteq A$ sarà composto di archi incidenti in v e quindi anche $B \in F$.

Proprietà dello scambio Per (S, F) NON vale la proprietà dello scambio cioè:

$$\forall A, B \in F \text{ t.c. } |B| = |A| + 1 \nrightarrow \exists b \in B - A \text{ t.c. } \{b\} \cup A \in F$$

PROOF A e B potrebbero riferirsi a due vertici diversi.

CONTROESEMPIO: $A = \{(5, 6), (6, 7)\}$ e $B = \{(2, 3), (3, 4), (3, 5)\}$.

Esempio 3 Nelle slide c'è un terzo esempio riguardo l'insieme finito di vettori di uno spazio vettoriale (S) e la famiglia dei sottoinsiemi di S composti da vettori linearmente indipendenti (F).

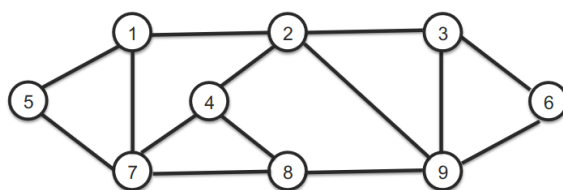
5.8 Matroide Grafico M_G

Dato un grafo $G = (V, E)$ non orientato e connesso $M_G = (S, F)$ con:

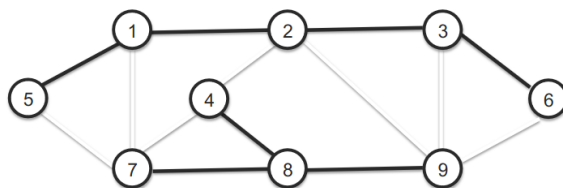
- S , insieme E degli archi
- F , tutti i sottoinsiemi di S che sono aciclici

è il **matroide grafico** relativo a G .

$A \in F \implies G_A = (V, A)$ è una **foresta**.



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



$A = \{(1, 2), (2, 3), (4, 8), (3, 6), (1, 5), (7, 8), (8, 9)\} \in F$
 $G_A = (V, A)$ è una foresta di $|V| - |A| = 2$ alberi.

M_G è un matroide

1. $A \in F, B \subseteq A \implies B \in F$
 Se $A \in F$, allora anche $B \subseteq A$ sarà aciclico e appartenente a F
2. $\forall A, B \in F$ t.c. $|B| = |A| + 1 \implies \exists b \in B - A$ t.c. $\{b\} \cup A \in F$

5.8.1 Dimostrazione della proprietà dello scambio

Siano $A, B \in F$ tali che $|B| = |A| + 1$.

$G_A = (V, A) \rightarrow$ foresta di $|V| - |A|$ alberi.

$G_B = (V, B) \rightarrow$ foresta di $|V| - |B|$ alberi.

G_B ha un albero in meno di G_A .

\implies in G_B esiste un arco (u, v) che connette due vertici u e v che stanno in G_A stanno su due alberi diversi.

\implies in G_B esiste un arco (u, v) che connette due vertici u e v che in G_A stanno in due alberi diversi.

$\implies \{(u, v)\} \cup A \in F$.

5.9 Insieme massimale di un matroide

Sia $M = (S, F)$ un matroide

Un elemento $s \in S$ è estensione di $A \in F$ se $A \cup \{s\} \in F$.

$A \in F$ è **massimale** se non ha estensioni ovvero non è contenuto in un insieme indipendente più grande.

Osservazione: Tutti gli insieme indipendenti massimali di un matroide hanno la stessa dimensione

.

5.9.1 Insieme massimale di un matroide grafico

Sia $M_G = (S, F)$ un matroide grafico.

Insieme massimale $A \in F$ (aciclico) che non ha estensioni

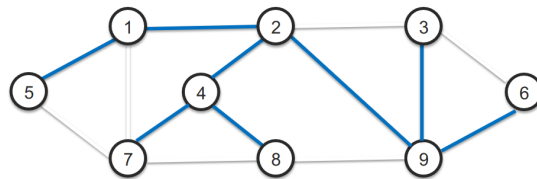
$\rightarrow \nexists s \in S$ t.c. $\{s\} \cup A \in F$

$G_A = (V, A)$, se A è massimale, è una foresta composta da un solo albero di $|V| - 1$ archi che connette tutti i vertici del grafo.

G_A è un albero di connessione per G (Spanning Tree, ST).

Esempio Grafico

Prendiamo l'esempio di prima M_G e evidenziamo gli archi che compongono lo Spanning Tree, cioè un albero aciclico che connette tutti i nodi (un albero di connessione).



$A = \{(1, 2), (1, 5), (2, 4), (2, 9), (4, 7), (4, 8), (3, 9), (6, 9)\} \in F$

$\rightarrow A$ è massimale

$\rightarrow G_A = (V, A)$ è uno Spanning Tree (ST)

5.9.2 Matroide Pesato

Un *Matroide Pesato* è un matroide $M = (S, F)$ a cui viene associata una **Funzione peso W** :

$W : S \rightarrow R^+$

$W(s)$, $s \in S \rightarrow$ peso dell'elemento s

Funzione peso estesa a F : $W(A)$, $A \in F \rightarrow$ peso di A definito come somma dei pesi degli elementi di S che appartengono ad A .

DEFINIZIONE

Sottoinsieme ottimo \rightarrow sottoinsieme indipendente di peso massimo (sicuramente massimale)

5.10 Algoritmo Greedy Standard

→ Algoritmo greedy che trova, per un matroide $M = (S, F)$ il sottoinsieme ottimo.

```

Procedura standard_greedy_algorithm( $S = \{s_1, s_2, \dots, s_n\}$ )
     $X = \emptyset$ 
    Ordine  $S$  per peso  $W$  decrescente (non crescente)

    for  $i$  from 1 to  $n$  do
        if  $\{s_i\} \cup X \in F$  then
             $X = \{s_i\} \cup X$ 
    return  $X$ 

```

Osservazione: L'insieme restituito è indipendente

Complessità temporale algoritmo

- L'ordinamento peso $O(n \log n)$
- Il ciclo for $O(n)$
- L'if $O(f(n))$

In totale avrò che la complessità è di $O(n \log n + nf(n))$

5.11 Teorema di Rado

DEFINIZIONE

La coppia (S, F) è un matroide se e solo se per ogni funzione peso W , l'algoritmo greedy standard fornisce la soluzione ottima (sottoinsieme indipendente di peso massimo)

Problema P → Matroide pesato

- Esiste un algoritmo greedy che lo risolve per qualsiasi funzione
- In caso contrario → non esiste un algoritmo greedy per qualsiasi funzione peso

5.11.1 Proof

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

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

Si può dimostrare che un Sistema di Indipendenza (S, F) è un matroide se e solo se per ogni $C \subseteq S$ gli insiemi massimali di (C, F_C) hanno la stessa cardinalità.

L'algoritmo greedy standard fornisce sempre (per costruzione) insiemi massimali.

La Dimostrazione è divisa in due parti:

- La prima dimostra che se (S, F) è un matroide allora il problema può essere risolto con la tecnica Greedy
- La seconda dimostra il contrario, cioè che se (S, F) non è un matroide allora il problema non può essere risolto tramite la tecnica Greedy

5.11.2 Parte 1 - (S, F) è un matroide

IPOTESI : (S, F) è un matroide.

Sia W la funzione peso e $X = \{s_1, s_2, \dots, s_p\}$ la soluzione fornita dall'algoritmo greedy standard.

TESI : per qualsiasi X' massimale, si ha $W(X) \geq W(X')$.

NB X' è un sottoinsieme massimale NON fornito dall'algoritmo greedy, ma creato ad hoc per la dimostrazione.

Per costruzione, X è massimale e $W(s_1) \geq W(s_2) \geq \dots \geq W(s_p)$.

Sia $X' = \{s'_1, s'_2, \dots, s'_p\}$ un insieme massimale diverso da X (stessa cardinalità di X) tale che $W(s'_1) \geq W(s'_2) \geq \dots \geq W(s'_p)$.

Ho quindi i seguenti insiemi:

$$\begin{aligned} X &= \{s_1, s_2, s_3, \dots, s_{k-1}, s_k, \dots, s_p\} \\ W &= \{\geq, \geq, \geq, \dots, \geq, \geq, \dots, \geq\} \\ X' &= \{s'_1, s'_2, s'_3, \dots, s'_{k-1}, s'_k, \dots, s'_p\} \end{aligned}$$

W serve a indicare la relazione fra i pesi di X e X' .

Caso 1 : $W(s_i) \geq W(s'_i)$, $1 \leq i \leq p \implies W(X) \geq W(X')$. Cioè se per tutti il peso di ogni oggetto s_i è maggiore o uguale del peso di ogni oggetti s'_i , per ogni oggetto da 1 a p (per i compreso tra 1 e p), allora necessariamente il peso totale di $W(X)$ sarà maggiore o uguale di $W(X')$.

Caso 2 : $W(s_i) < W(s'_i)$ per qualche i. Cioè qualche oggetto s_i avrà peso minore di qualche oggetto di s'_i .

Sia k il più piccolo indice tale che $W(s_k) < W(s'_k)$. Cioè da S_x a D_x cerco il primo elemento per cui $W(s_k) < W(s'_k)$.

$\implies W(s_k) < W(s'_k) \leq W(s'_k - 1) \leq W(s_{k-1})$.

$$X = \{s_1, s_2, s_3, \dots, s_{k-1}, s_k, \dots, s_p\}$$

$$W = \{\geq, \geq, \geq, \dots, \geq, <, \dots\}$$

$$X' = \{s'_1, s'_2, s'_3, \dots, s'_{k-1}, s'_k, \dots, s'_p\}$$

Sia $C = \{s \in S \text{ t.c } W(s) \geq W(s'_k)\} \implies (C, F_C)$ Sottosistema Indotto da C
 $\implies s'_k \in C, \quad X' \cap C = \{s_1, s_2, \dots, s_{k-1}, s'_k\}$.

$\implies s_k \notin C, \quad X \cap C = \{s_1, s_2, s_3, \dots, s_{k-1}\}$. Questo perchè $W(s_k) < W(s'_k)$, cioè non appartiene al Sottosistema Indotto da C perchè non rispetta questa condizione.

$\implies |X' \cap C| > |X \cap C|$, ed è in **contraddizione** con il fatto che X e X' sono massimali, per (S,F).

implies **NON è vero** che esiste un k tale per cui $W(s_k) < W(s'_k)$ e quindi $W(X) \geq W(X')$.

5.11.3 Parte 2 - (S,F) non è un matroide

Esiste un insieme $C \subseteq S$, ed esistono due insiemi A e B massimali in (C, F_C) con $|B| = p$ e $|A| > |B|$.

Essendo $|A| > |B|$ avrò che $|A| \geq p + 1$ Le cardinalità sono diverse perchè abbiamo ipotizzato che (S,F) non è un matroide, quindi non vale la proprietà di scambio.

Si consideri il seguente peso W:

$$W(s) = p + 2 \quad \text{se } s \in b$$

$$W(s) = p + 1 \quad \text{se } s \in A - B$$

$$W(s) = 1 \quad \text{altrimenti}$$

Sostanzialmente sto associando un peso agli oggetti di B di $p+2$, mentre agli elementi che appartengono solo ad A associo un peso di $p+1$, 1 altrimenti. L'algoritmo greedy restituisce B, perchè controlla gli oggetti di peso massimo

e quindi estrae B.

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

Ho $p+1$, perchè in A ho almeno $p+1$ oggetti dato che la cardinalità di A è maggiore di B.

$\Rightarrow W(A) > W(B)$, ma Greedy restituiva $W(B)$ come maggiore!

Conclusione l'algoritmo greedy standard non restituisce la soluzione ottima per qualsiasi peso W.

Osservazione: Il teorema di Rado afferma che se il problema non è riconducibile a un matroide non è possibile trovare un algoritmo Greedy che restituisca una soluzione ottima per qualsiasi peso W. Ma NON dice che se un problema non può essere risolto con Greedy, questo teorema si riferisce ai problemi che hanno dei pesi e per cui ha senso considerare dei valori come pesi, se prendiamo il problema visto inizialmente degli eventi non si tratta di un problema con i pesi, infatti non si tratta di un matroide, ma è comunque possibile risolverlo tramite Greedy perchè è possibile scegliere un parametro (che ovviamente non è il peso), che mi permette di calcolare la soluzione ottima tramite greedy. Resta comunque da verificare che il parametro Greedy sia corretto.

5.11.4 Esempio 1 Applicazione Rado

Problema Dato un insieme V di vettori pesati da una funzione W, determinare il sottoinsieme di vettori linearmente indipendenti che ha peso massimo.

Matroide $M = (S, F)$

- S, insieme finito V di vettori
- F, famiglia dei sottoinsiemi di S composti da vettori linearmente indipendenti

Funzione peso W: $S \rightarrow R^+$

Sottoinsieme ottimo di M \rightarrow Soluzione del problema

```
Procedura greedy_ind_vectors(V = {v1, v2, ..., vn})
  X = ∅
  Ordine V per peso W decrescente (non crescente)

  for i from 1 to n do
```

```

        if  $\{v_i\} \cup X$  contiene vettori lin. indep. then
             $X = \{v_i\} \cup X$ 
    return  $X$ 

```

5.11.5 Esempio 2 - Peso massimo archi

Problema dato un grafo connesso non orientato con archi pesati da una funzione W , trovare l'insieme di peso massimo composto da archi che incidono nello stesso vertice.

Coppia (S,F)

- S , insieme degli archi del grafo
- F , famiglia dei sottoinsiemi di S composti da archi che incidono in uno stesso vertice

Questo NON è un matroide.

Applico l'algoritmo greedy standard per trovare l'insieme di peso massimo in F .

$X \leftarrow$ insieme vuoto

Ordino gli archi in S per peso decrescente (non crescente)

Aggiungo a X l'arco più pesante (v_1, v_2) .

Proseguo a considerare gli altri archi fino a quando trovo un arco incidente in v_1 oppure in v_2 e lo aggiungo.

Da questo momento aggiungo un arco solo se incide in v_1 (se prima ho aggiunto un arco incidente in v_1) oppure solo se incide in v_2 (se prima ho aggiunto un arco incidente in v_2).

Se (v_1, v_2) è isolato non aggiungo nient'altro...

Voglio trovare il sottoinsieme indipendente massimale di peso minimo.

$W' : S \rightarrow R^+$ tale che $W'(s) = W_0 - W(s)$ per $s \in S$.

$W_0 \rightarrow$ massimo valore della funzione W .

```

Procedura standard_greedy_algorithm( $S = \{s_1, s_2, \dots, s_n\}$ )
     $X = \emptyset$ 
    Ordine  $S$  per peso  $W'$  decrescente (non crescente)

    for i from 1 to n do
        if  $\{s_i\} \cup X \in F$  then
             $X = \{s_i\} \cup X$ 
    return  $X$ 

```

```
Procedura standard_greedy_algorithm( $S = \{s_1, s_2, \dots, s_n\}$ )  
   $X = \emptyset$   
  Ordine  $S$  per peso  $W$  crescente (non decrescente)  
  
  for  $i$  from 1 to  $n$  do  
    if  $\{s_i\} \cup X \in F$  then  
       $X = \{s_i\} \cup X$   
  return  $X$ 
```

Capitolo 6

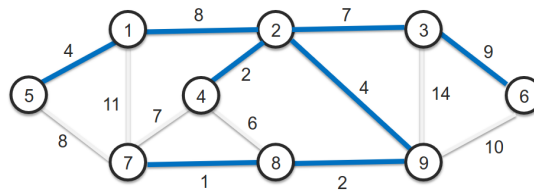
Minimum Spanning Tree - MST

INPUT : Grafo connesso non orientato pesato $G = (V, E)$ con:
 $W : E \rightarrow R^+$ tale che $W(u, v)$ è il peso dell'arco (u, v) .

OUTPUT : $T \subseteq E$ aciclico tale che:

1. $\forall v \in V, \exists (u, v) \in T$
2. $W(T) = \sum_{(u,v) \in T} W(u, v)$ è minimo

$G_T = (V, T) \rightarrow$ Minimum Spanning Tree (MST).



$$W(T) = 37$$

6.1 Soluzione Generica

Algoritmo Generico

1. Inizializza un insieme A vuoto
2. Aggiunge ad ogni passo un arco (u, v) in modo tale che $A \cup \{(u, v)\}$ è sottoinsieme dell'insieme T degli archi MST

3. L'algoritmo termina non appena $A = T$ (cioè, $G_A = (V, T)$ è MST)

$(u, v) \rightarrow$ arco sicuro per A (cioè appartiene a MST)

```

Procedura Generic_MST(G, W)
  A =  $\emptyset$ 
  while  $G_A = (V, A) \neq \text{MST}$  do
    trova arco  $(u, v)$  sicuro per A
    A = A  $\cup \{(u, v)\}$ 
  return A

```

6.2 Definizioni Principali

6.2.1 Taglio

DEFINIZIONE

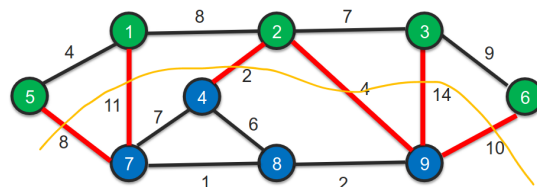
Definizione di Taglio: Partizione di V in due insiemi V' e $V - V'$

6.2.2 Arco che Attraversa il Taglio

DEFINIZIONE

Definizione di Arco che Attraversa il Taglio: arco $(u, v) \in E$ tale che u appartiene a V' e v appartiene a $V - V'$

Esempio di Taglio



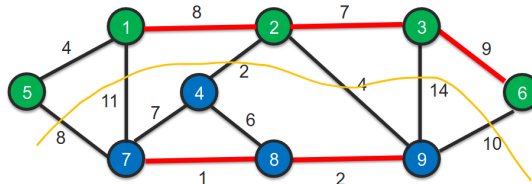
$$V' = \{1, 2, 3, 5, 6\}$$

$$V - V' = \{4, 7, 8, 9\}$$

6.2.3 Taglio che rispetta un insieme

DEFINIZIONE

Un taglio che rispetta un insieme $A \subseteq E$, se nessun arco di A attraversa il taglio

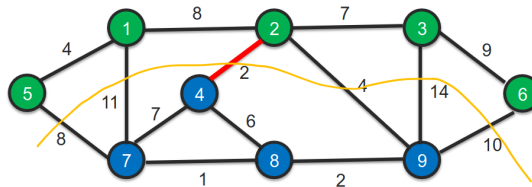


Il taglio rispetta $A = \{(1, 2), (2, 3), (3, 6), (7, 8), (8, 9)\}$.

6.2.4 Arco Leggero

DEFINIZIONE

Arco di peso minimo che attraversa il taglio



$(4,2) \rightarrow$ arco leggero

6.3 Teorema dell'arco sicuro

Dati un grafo connesso non orientato e pesato $G = (V, E)$, un sottoinsieme A dell'insieme T di archi di un Minimum Spanning Tree (MST) e un qualsiasi taglio che rispetti A , allora un arco leggero (u,v) del taglio è sicuro per A , cioè $A \cup \{(u, v)\} \subseteq T$.

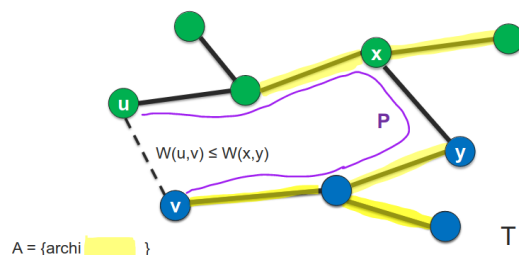
6.3.1 Proof

Considero $T \rightarrow$ insieme di archi di un MST.

$A \subseteq T \rightarrow$ sottoinsieme di T .

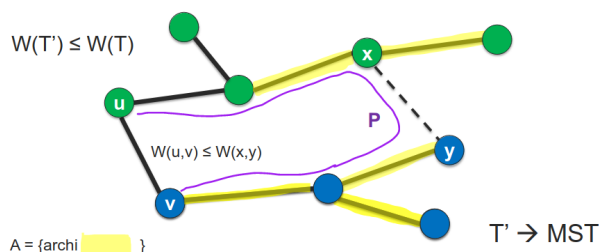
Taglio che rispetta $A = (\bullet, \bullet)$.

$(u, v) \rightarrow$ arco leggero del taglio.



Il peso $W(u, v)$ è sicuramente \leq rispetto al peso $W(x, y)$.

Sostituisco (x, y) , con (u, v) e inserisco quindi l'arco (u, v) all'interno del grafo e noto che il grafo resta comunque un albero, vedo che resta un MST dato che non ci sono cicli, ottengo quindi T' con $W(T') \leq W(T)$.



Per ipotesi, $A \subseteq T$ non contiene (u, v) e (x, y) e per costruzione T' non contiene (x, y) .

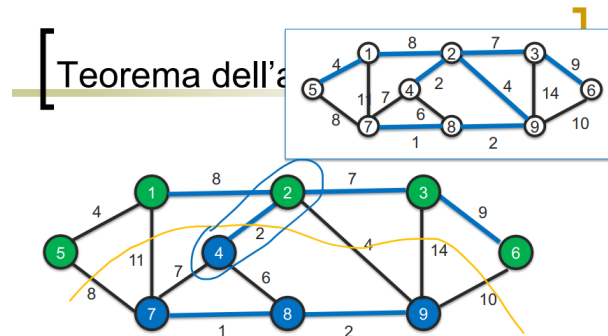
$\Rightarrow A \subseteq T'$

$A \cup \{(u, v)\} \subseteq T'$

6.3.2 Esempio Teorema

Dato il seguente sottoinsieme di T denominato $A = \{archiblu\} \subseteq T$, prendiamo

l'arco $(2, 4) \rightarrow$ che è arco leggero (dato che ha peso 2), possiamo dire che è un arco sicuro, infatti se guardiamo T notiamo che $A \cup \{(2, 4)\} \subseteq T$, infatti l'arco appartiene all'MST T .



6.3.3 Soluzione generica

```

Procedura Generic_MST( $G, W$ )
   $A = \emptyset$ 
  while  $|V| - |A| > 1$  do
    trova arco  $(u, v)$  sicuro per  $A$ 
     $A = A \cup \{(u, v)\}$ 
  return  $A$ 

```

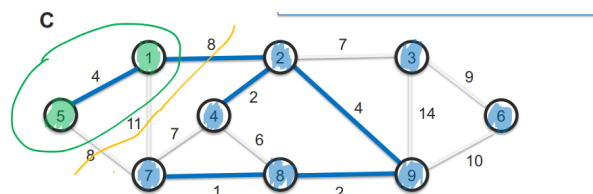
1. A rimane aciclico durante le iterazioni
2. $G_A = (V, A)$ (ad ogni iterazione) è una foresta di $|V| - |A|$ alberi
3. All'inizio G_A contiene $|V|$ alberi (i singoli vertici)
4. Ogni iterazione riduce di 1 il numero di alberi e l'arco sicuro (u, v) collega sempre componenti distinte di G_A
5. Quando si arriva a un solo albero l'algoritmo termina
6. Il numero di iterazioni è pari a $|V| - 1$

6.3.4 COROLLARIO

$A \subseteq T$ è tale che $G_A = (V, A)$ è una foresta di $|V| - |A|$ alberi.
 Sia $C = (V_C, A_C)$, $V_C \subseteq V$ e $A_C \subseteq A$, una componente connessa di G_A
 $\Rightarrow (V_C, V - V_C)$ è sicuramente un taglio che rispetta A
 \Rightarrow un arco leggero di $(V_C, V - V_C)$ è arco sicuro per A .

QUINDI Ad ogni iterazione, per aggiungere ad A un arco sicuro (arco di MST) basta:

1. Considerare una delle componenti C della foresta $G_A = (V, A)$
2. Trovare l'arco di peso minimo che collega un vertice in C con un vertice non in C



$A = \{(7,8), (2,4), (8,9), (2,9), (1,5), (1,2)\}$
 $V_C = \{1,5\} \rightarrow \text{TAGLIO} = (V_C, V - V_C)$
 $(1,2) \rightarrow$ arco leggero = arco sicuro da aggiungere ad A

Corollario Dati un grafo connesso non orientato e pesato $G = (V, E)$, un sottoinsieme A degli archi di MST, una componente connessa $C = (V_C, E_C)$ di $G_A = (V, A)$, allora un arco leggero (u, v) del taglio $(V_C, V - V_C)$ è un arco sicuro per A .

6.3.5 Implementazione GENERIC-MST

Ci sono principalmente due implementazioni di GENERIC-MST e sono 2 algoritmi:

- Algoritmo di Kruskal
- Algoritmo di Prim

6.4 Algoritmo di Kruskal

MST = Sottoinsieme ottimo di un matroide grafico M_G . $M_G = (S, F)$ per $G = (V, E, W)$ non orientato e connesso:

- S , insieme E degli archi di G
- F , tutti i sottoinsiemi di S che sono aciclici

pesato con:

$W : S \rightarrow R^+$ tale che $W(s)$ è il peso dell'arco s .

Sottoinsieme ottimo di M_G pesato con W

→ Archi dello Spanning Tree (ST) di peso massimo.

$W' : S \rightarrow R^+$ tale che $W'(s) = W_0 - W(s)$

W_0 è il massimo valore del peso degli archi (massimo per W)

Sottoinsieme ottimo di M_G pesato con W'

→ Sottoinsieme masimale di peso massimo (per W') e di peso minimo per W che equivale a dire:

→ Archi dello Spanning Tree (ST) di peso minimo, cioè l'MST.

6.4.1 Algoritmo Greedy Standard per il matroide grafico

```

Procedura Generic_matroid_grafic(E = {e1, e2, ..., en})
  X = ∅
  Ordino E per peso W crescente (non decrescente)
  for i form 1 to n do
    if ei arco sicuro then
      A = {ei} ∪ A
  return A

```

6.4.2 Algoritmo Kruskal

```

Procedura KRUSKAL_MST(G = (V, E), W)
  A = ∅
  E = ⟨e1, e2, ..., en⟩ ordinati per peso crescente (non decr.)
  for i form 1 to n do
    if {ei} ∪ A ⊆ T then
      A = {ei} ∪ A
  return A

```

Devo tradurre in codice l'if che determina se e_i è un arco sicuro e per fare questo **applico il corollario**:

$e_i = (u, v)$ è arco sicuro se è un arco di peso minimo che connette un vertice di una componente C con un vertice che non è in C . Quindi e_i è arco sicuro se connette due componenti diverse di $G_A = (V, A)$. Quindi basta assegnare l'arco che sto analizzando e_i a (u, v) e se non appartengono alla stessa componente di $G_A(V, A)$, allora posso aggiungere l'arco alla soluzione A .

```

Procedura KRUSKAL_MST(G = (V, E), W)
  A = ∅
  E = ⟨e1, e2, ..., en⟩ ordinati per peso crescente (non decr.)

```

```

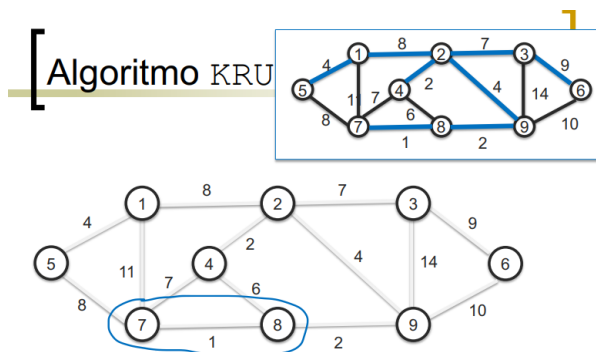
for i form 1 to n do
    (u,v) = ei
    if u e v ∉ stessa componente di GA(V,A) then
        A = {(u,v)} ∪ A
return A

```

6.4.3 Esempio di Esecuzione Algoritmo Kruskal

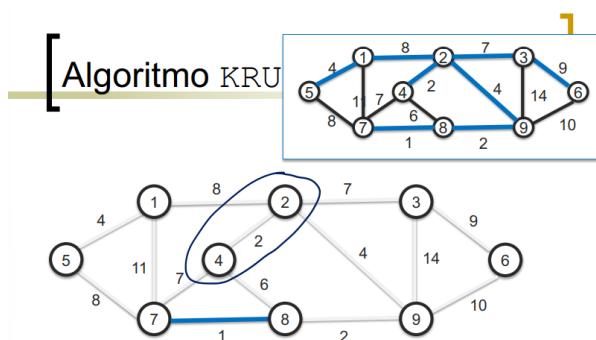
Passo 1 Ordina E per peso non decrescente $\rightarrow E = \langle e_1, e_2, \dots, e_m \rangle$, in modo tale da avere nell'insieme E prima gli archi più leggeri e mano a mano quelli più pesanti, in questo modo posso applicare un approccio Greedy. Inizializza $A = \emptyset$.

Inizio a considerare gli archi Considero $e_1 = (7, 8)$, connette due componenti di G_A ?



Sì, quindi $\implies A = \{(7, 8)\}$.

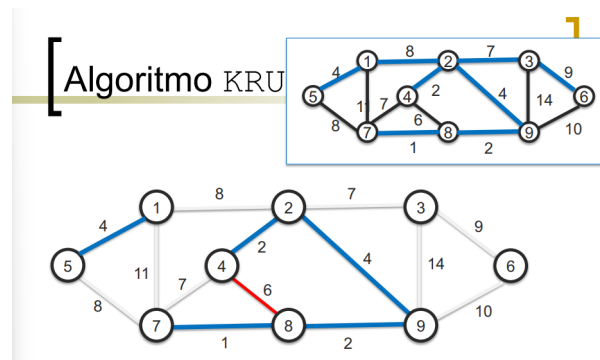
Considero ora il secondo arco $e_2 = (2, 4)$, connette due componenti di G_A ?



Sì, quindi $\Rightarrow A = \{(7, 8), (2, 4)\}$.

Saltiamo qualche passo dove aggiungiamo sempre archi perchè troviamo sempre che gli archi considerati connettono due componenti.

Consideriamo l'arco $e_6 = (4, 6)$



Notiamo che in questo caso l'arco non connette due componenti di G_A (le componenti sono già connesse tramite altri archi, aggiungere questo arco creerebbe un ciclo), per questo non considero questo arco e procedo a considerare il successivo.

Continuo in questo modo fino a quando non ho considerato tutti gli archi. Alla fine avrò ottenuto l'MST!

6.4.4 Codice Kruskal

Abbiamo già scritto in precedenza il codice di Kruskal, ma come possiamo tradurre le istruzioni matematiche in codice? Riscriviamo le parti riguardante il controllo di appartenenza al grafo.

```

KRUSKAL - MST (G=(V, E), W)
A = ∅
E = ⟨e1, e2, ..., en⟩ ordinati per peso crescente (non decr.)
foreach v ∈ V do
    MAKE_SET(V)
for i from 1 to n do
    (u, v) = ei
    if FIND_SET(u) ≠ FIND_SET(v) then
        A = {(u, v)} ∪ A
return A

```

Tramite la strutture dati FIND SET andiamo ad analizzare se aggiungendo l'arco (u, v) , otteniamo un nuovo albero (quindi connettiamo due componenti) o se abbiamo sempre lo stesso albero (quindi non stiamo connettendo le

componenti, ma stiamo generando un ciclo). se le 2 FIND SET sono diverse allora possiamo aggiungere l'arco alla soluzione finale tramite la UNION, altrimenti lo scarto e passo a quello successivo.

Complessità Algoritmo

- L'ordinamento degli archi avrà complessità $O(|E| \log |E|)$ (Merge Sort)
- il MAKE SET avrà complessità $O|V|$
- Mentre tutto il blocco for fino a UNION ha complessità $O(|E|\alpha)$, dove α è una funzione che cresce molto lentamente che è il tempo di aggiunta degli elementi tramite UNION

Avendo G connesso $\rightarrow |E| \geq |V| - 1$ che possiamo approssimare a $|E|$, quindi $(|V| + |E|\alpha)$ sarà $(2|E|\alpha)$, ma essendo il 2 un coefficiente costante nell'utilizzo dell'O è trascurabile.

In totale avrò $O(|E| \log |E| + |E|\alpha)$.

Sicuramente $\alpha \leq \log |V|$, perchè α cresce molto lentamente $|V| = |E|$, per questo motivo avrò:

Complessità $O(E \log |E| + |E| \log |E|)$ quindi $O(|E| \log |E|)$.

6.5 Algoritmo di Prim

Altro algoritmo utilizzato per trovare l'albero di copertura minimo in un grafo che in questo caso si basa sull'effettuare tagli e scegliere l'arco leggero (quindi sicuro).

Si basa sull'assegnazione di un peso ai vertici e questo è un meccanismo utilizzato anche da un altro algoritmo che vedremo più avanti denominato Dijkstra.

6.5.1 Idea dell'algoritmo

1. Sceglie un vertice arbitrario r (componente C iniziale composta dal solo vertice r)
2. Trova l'arco di peso minimo che connette r a un altro vertice v (il vertice v entra a far parte di C)

3. Trova l'arco di peso minimo che connette un vertice in C a un vertice v non in C , (il vertice v entra a far parte di C)
4. Etc.
5. L'algoritmo termina quando la componente C comprende tutti i vertici del grafo e coincide quindi con il Minimum Spanning Tree

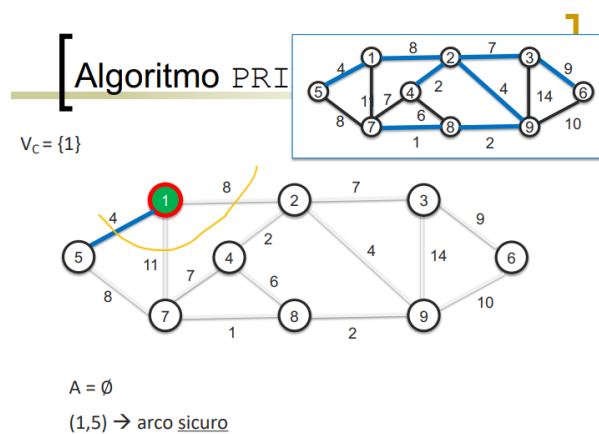
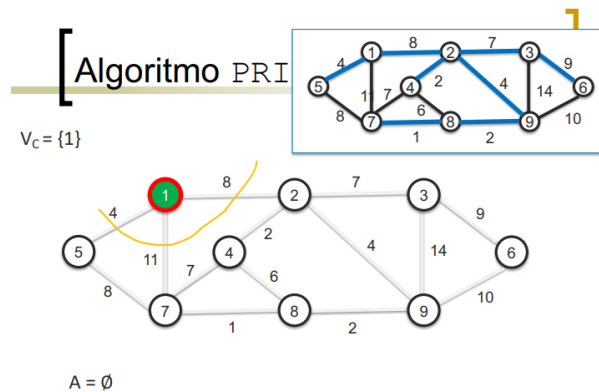
6.5.2 Proprietà dell'algoritmo

Ad ogni passo:

1. Il sottoinsieme A degli archi di MST aggiunti, stanno tutti nella componente C . La foresta $G_A = (V, A)$ è composta da:
 - $C = (V_C, A)$
 - $|V - V_C|$ componenti di vertici singoli
2. Il taglio $(V_C, V - V_C)$ rispetta l'insieme A
3. l'arco sicuro è l'arco di peso minimo che connette un vertice in C con un vertice non in C

L'idea è quindi quella di effettuare dei tagli tali per cui $(V_C, V - V_C)$ rispetta l'insieme A , quindi effettuo dei tagli tra l'insieme A gli archi già aggiunti all'MST e gli altri archi e considero l'arco leggero. Per il teorema dell'arco sicuro, l'arco sicuro è l'arco di peso minimo che connette un vertice in C con un vertice non in C .

Esempio Qui di seguito notiamo che effettuando il taglio e scegliendo l'arco leggero preleviamo automaticamente l'arco sicuro.



Continuando ad effettuare tagli validi e a selezione l'arco leggero (quindi sicuro) otterremo l'MST.

6.5.3 Implementazione Prim

Come possiamo implementare in maniera efficiente l'esecuzione del taglio e la conseguente ricerca dell'arco di peso minimo?

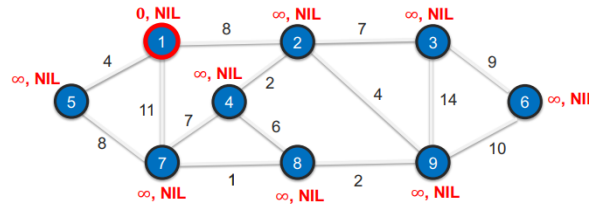
Assegneremo dei pesi ai vertici e un valore contenente il suo predecessore per ogni v in V e utilizzeremo come struttura dati una **coda Q di min-priority**. La **min priority queue** è coda particolare che con l'operazione di Dequeue permette di estrarre l'elemento di priorità minima, in questo caso ci servirà per ottenere sempre l'elemento di peso minimo.

Inizializziamo i pesi a infinito e i valori del predecessore a NIL:

- $v.\text{key} = \infty$
- $v.\pi = \text{NIL}$

Scegliamo un **vertice** r arbitrario, che sarà il nodo d'origine dell'algoritmo. Nell'esempio scegliamo $r = 1$ e peso $1.key = 0$.

Inseriamo tutti i vertici in una coda Q di min priority che permette l'estrazione del vertice con il minimo valore del campo key .



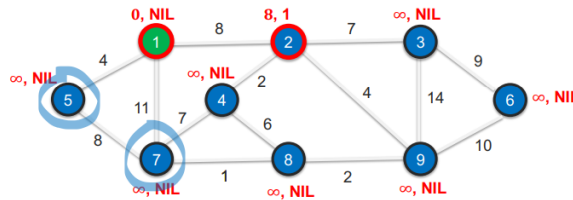
Prima estrazione coda Estraiamo ora da Q il vertice con il minimo valore di $key \rightarrow$ vertice 1.

NB L'estrazione implica la rimozione dell'oggetto dalla coda!

Procediamo quindi ad esaminare gli adiacenti di $1 \rightarrow 2, 7, 5$.

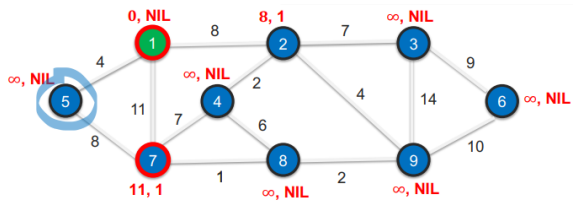
2 appartiene a Q and $W(1, 2) < 2.key$

$\Rightarrow 2.key = (1, 2); \quad 2.\pi = 1$.



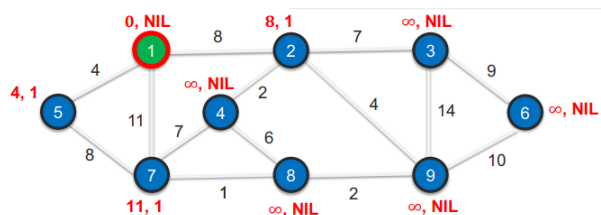
Considero il secondo adiacente, 7. Esso appartiene a Q and $W(1, 7) < 7.key$

$\Rightarrow 7.key = W(1, 7); \quad 7.\pi = 1$.



Considero il terzo vertice adiacente, 5. Esso appartiene a Q and $W(1, 5) < 5.key$

$\Rightarrow 5.key = W(1, 5); \quad 5.\pi = 1$.



Seconda estrazione coda Ora che ho esaurito tutti gli adiacenti procedo nuovamente a estrarre da Q il vertice con il minimo valore di key \rightarrow vertice 1, in questo passo si tratta del vertice 5 che ha $key = 4$.

Esamino gli adiacenti di 5 \rightarrow 1, 7.

1 non è più in Q, quindi lo ignoro ed esamino 7.

7 appartiene a Q and $W(5, 7) < 7.key$

$\Rightarrow 7.key = W(5, 7); \quad 7.\pi = 5$.

Terza estrazione coda Non ho più adiacenti quindi riprocedo a estrarre dalla coda Q il vertice di peso minimo, ora si tratta del vertice 2.

Gli adiacenti di 2 sono 1, 4, 9, 3.

1 non è più nella coda, lo ignoro.

4 appartiene a Q and $W(2, 4) < 4.key$, quindi aggiorno il suo valore di key

$\Rightarrow 4.key = W(2, 4); \quad 4.\pi = 2$.

Saltiamo qualche passaggio Arriviamo fino al passaggio dove estraiamo dalla coda il vertice 9. I suoi adiacenti sono 8, 2, 3, 6.

Questo caso è interessante perchè vediamo come effettivamente l'algoritmo trova un arco che collega il vertice 8 di peso inferiore rispetto al precedente trovato e lo aggiorna.

Verifichiamo che 8 appartiene ancora a Q e inoltre $W(9, 8) < 8.key$ dato che $2 < 6$

$\Rightarrow 8.key = W(9, 8); \quad 8.\pi = 9$. Viene aggiornato il predecessore con quello più che percorre l'arco più leggero in questo momento.

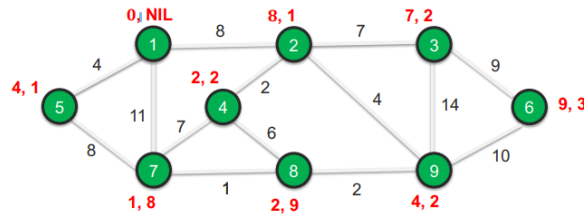
Il vertice 2 non è più nella coda quindi non ci interessa.

Altro caso interessante è il vertice 3, procediamo ad analizzarlo.

3 appartiene a Q AND $W(9, 3)$ non è $< 3.key$, perchè $14 > 7$, cioè sto percorrendo un arco peggiore in termini di peso rispetto a quello percorso in precedenza, quindi non aggiorno il valore key di 3 e lascio il suo predecessore invariato.

...

Procedo fino a quando la coda Q è vuota, in quel momento l'algoritmo termina.



In questo momento ho esplorato tutti i vertici, ma come faccio a determinare l'MST?

6.5.4 Determinare l'MST

Considero gli archi per cui: $T = \{(v.\pi, v) \text{ t.c. } v \in V, v \neq r\} \rightarrow$ archi di MST. Cioè percorro i predecessori degli archi, tranne nel caso in cui il sono nel vertice d'origine dove $r.\pi = NIL$.

In questo modo ricostruisco tutto l'MST.

6.5.5 Riassunto PRIM-MST

In questo algoritmo viene mantenuta una coda Q di min-priority che all'inizio contiene tutti i vertici del grafo e ad ogni passo Q:

- contiene i vertici che non appartengono ancora alla componente C
- permette di estrarre il vertice v tale che (u,v) è l'arco di peso minimo che collega un vertice u in C con un vertice non ancora in C

Ogni vertice v del grafo ha due campi:

- $v.key$ (quando v viene stratto da Q) minimo valore del peso degli archi (u,v) , incidenti in v tali che u sia nella componente C
- $v.\pi$, vertice u tale che (u,v) è l'arco di peso $v.key$

All'inizio $v.key = \infty$ e $v.\pi = NIL$ per ogni vertice del grafo.
 $r.key = 0$.

Ad ogni passo

1. Viene estratto Q il vertice u con il minor valore del campo key (dato che abbiamo una min priority queue)
 - l'arco $(u.\pi, u)$ è un nuovo arco di MST
 - u è un nuovo vertice che si aggiunge alla componente C

6.6. ALGORITMO DI DIJKSTRA - CAMMINI MINIMI DA SORGENTE UNICA⁸⁷

2. Per ogni vertice v adiacente a u , se v è in Q e $v.key$ è inferiore al peso $W(u, v)$ (ho un arco migliore del precedente), allora aggiorni i valori

- $v.key$ al valore $W(u, v)$
- $v.\pi$ al valore u

L'algoritmo termina quando Q è vuota.

$T = \{(v.\pi, v) \mid v \in V, v \neq r\} \rightarrow$ **archi di MST.**

6.5.6 Codice PRIM-MST

```
Procedura PRIM_MST( $G, W, r$ )
  foreach  $v \in V$  do
     $v.key = \infty$ 
     $v.\pi = \text{NIL}$ 
   $r.key = 0$ 
  Aggiungi tutti i vertici di  $V$  alla coda  $Q$ 
  while  $Q \neq \emptyset$ 
     $u =$  estrai vertice da  $Q$ 
    foreach  $v \in \text{adj}(u)$  do
      if  $v \in Q$  and  $W(u, v) < v.key$  then
         $v.key = W(u, v)$ 
         $v.\pi = u$ 
```

Complessità La fase di inizializzazione impiega $O(|V|)$, lo stesso per il while dato che deve scorrere tutti i vertici ($O(|V|)$). Mentre l'estrazione richiede $O(\log |V|)$. La parte relativa agli adiacenti dipende dal numero di archi infatti impiegherà $O(|E|)$. La parte relativa all'aggiornamento della valore key $O(\log |V|)$.

La complessità alla fine sarà di $O(|E| \log |V|)$.

6.6 Algoritmo di Dijkstra - Cammini minimi da sorgente unica

L'algoritmo di Dijkstra nasce per calcolare la distanza più breve da un nodo di partenza a tutti gli altri nodi del grafo. L'algoritmo è applicabile solo a grafi con pesi non negativi sugli archi.

INPUT Grafo $G = (V, E, W)$, orientato e pesato:

- $W : E \rightarrow R^+$ tale che $W(i, j) = w_{ij}$ = peso dell'arco (i, j)
- Un vertice $s \in V$ (vertice sorgente)

OUTPUT Per ogni $v \in V$ diverso da s , trovare il cammino di peso minimo che inizia in s e termina in v .

6.6.1 Sottostruttura Ottima di un cammino minimo in generale

Se il cammino $P = \langle v_1, v_2, \dots, v_{k-1}, v_k \rangle$ è minimo, allora sono minimi anche tutti i sottocammini $P_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ per $1 \leq i < j \leq k$.

PROOF : se esistesse un cammino da v_i a v_j di costo inferiore a quello di P_{ij} , allora sostituendo in P il sottocammino P_{ij} con tale cammino si otterrebbe un cammino da v_1 a v_k di costo inferiore a quello di P .

In particolare è minimo il sottocammino $\langle v_1, v_2, \dots, v_{k-1}, v_k \rangle$ dove v_{k-1} è il predecessore di v_k nel cammino minimo P .

Scomposizione del peso del cammino minimo $\delta(s, v)$ dalla sorgente s al vertice v

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

u **predecessore** di v nel cammino minimo da s a v .

$\delta(s, v)$, peso del cammino minimo da s a v .

$\delta(s, u)$, peso del cammino minimo da s a u .

$W(u, v)$, peso dell'arco (u, v) .

Limite superiore del peso del cammino minimo $\delta(s, v)$ dalla sorgente s al vertice v

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

u **predecessore** di v nel cammino minimo da s a v .

$\delta(s, v)$, peso del cammino minimo da s a v .

$\delta(s, u)$, peso del cammino minimo da s a u .

$W(u, v)$, peso dell'arco (u, v) .

6.6.2 Tecnica del Rilassamento

Si aggiungono ad ogni vertice v i due attributi $\mathbf{v.d.}$ e $\mathbf{v.\pi}$ che durante tutta l'esecuzione dell'algoritmo sono tali che:

- $\mathbf{v.d.}$ è un limite superiore per $\delta(s, v)$
- $\mathbf{v.\pi}$ è un vertice u tale che $(u, v) \in E$

Prima dell'esecuzione:

- $\mathbf{v.d.} = \infty$ per ogni vertice v diverso dalla sorgente s
- $\mathbf{v.\pi} = \text{NIL}$ per ogni vertice v
- $\mathbf{s.d.} = 0$ per la sorgente s

Alla fine dell'esecuzione:

- $\mathbf{v.d.} = \delta(s, v)$, peso del cammino minimo da s a v
- $\mathbf{v.\pi} = u$, predecessore di v nel cammino minimo da s a v

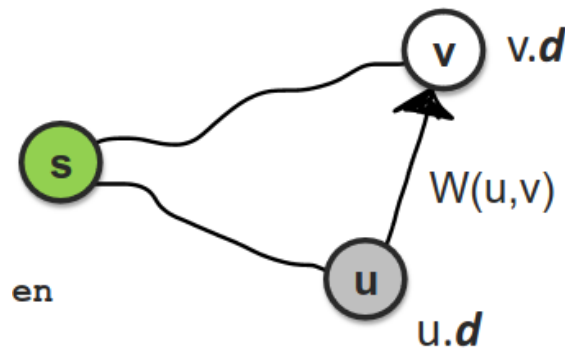
6.6.3 Codice Tecnica del Rilassamento

```
Initialize-Single-Source( $G, s$ )
  foreach  $v \in V$  do
     $v.d = \infty$ 
     $v.\pi = \text{NIL}$ 
   $s.d = 0$ 
```

PASSO DEL RILASSAMENTO DELL'ARCO (u, v) Se $v.d > u.d + W(u, v)$, allora si pone:

1. $v.d = u.d + W(u, v)$
2. $v.\pi = u$

```
Relax( $u, v, W$ )
  if  $v.d > u.d + W(u, v)$  then
     $v.d = u.d + W(u, v)$ 
     $v.\pi = u$ 
```



A partire dai valori:

- $v.d = \infty$ per ogni vertice v diverso dalla sorgente s
- $v.\pi = NIL$ per ogni vertice v
- $s.d = 0$ per la sorgente s

Viene eseguito per ogni arco (u,v) del grafo un solo passo di rilassamento per ottenere alla fine dell'esecuzione:

- $v.d = \delta(s, v)$, peso del cammino minimo da s a v
- $v.\pi = u$, predecessore di v nel cammino minimo da s a v

6.6.4 Esecuzione Dijkstra

Una **Coda Q di min-priority** contiene all'inizio tutti i vertici del grafo.

Ad ogni passo :

- Q contiene i vertici v per i quali il campo $v.d$ non ha raggiunto il valore $\delta(s, v)$
- Nel momento in cui il vertice u viene estratto da Q, il suo campo $u.d$ è uguale a $\delta(s, u)$
- Viene eseguito il rilassamento di ogni arco (u,v) uscente da u
- L'algoritmo termina quando Q è vuota

Alla fine tramite i valori del campo dei predecessori si ricostruisce il cammino minimo della sorgente s a un determinato vertice v .

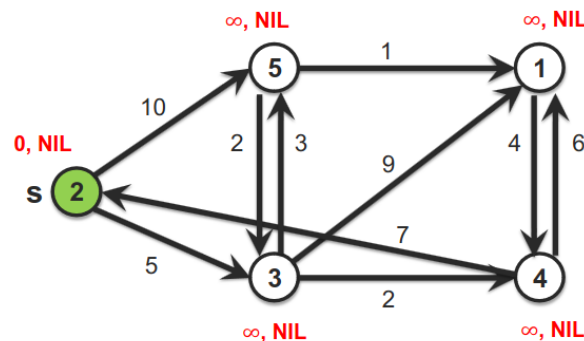
6.6.5 Esecuzione Grafica

Inizializzazione e Prima estrazione Per ogni v in V :

- $v.d = \infty$
- $v.\pi = NIL$

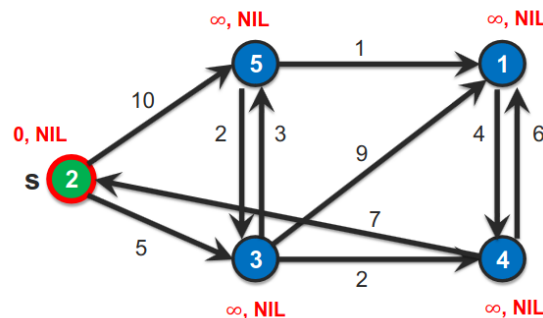
$s.d = 0$.

Inizializza un insieme vuoto S .



Inserisce tutti i vertici in una coda Q di min-priority ed estrae da Q il vertice con il minimo valore di $d \rightarrow$ vertice $s = 2$.

Aggiunge il vertice 2 all'insieme $S \rightarrow S = \{2\}$.



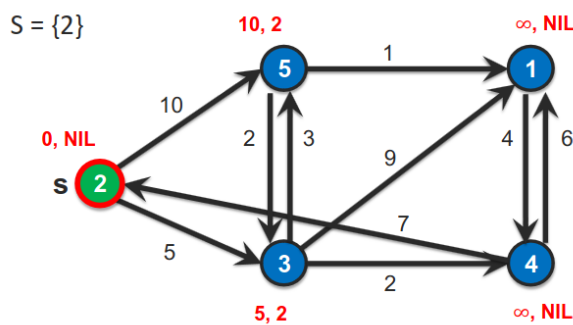
Esamina gli adiacenti del vertice 2 $\rightarrow 3, 5$.

Esegue il rilassamento dell'arco $(2, 5)$, quindi verifica $5.d > 2.d + W(2, 5)$, e se si verifica la condizione $\rightarrow 5.d = 2.d + W(2, 5) = 10$. Aggiorno quindi peso e predecessore del vertice 5, il predecessore sarà chiaramente il vertice 2, chiaramente più avanti potrà cambiare, e vedremo che sarà così, perchè mano a mano che scopro il grafo Dijkstra aggiorna i pesi e i predecessori in base al cammino minimo.

Esegue il rilassamento per il prossimo adiacente, il vertice 3 e quindi l'arco

(2,3).

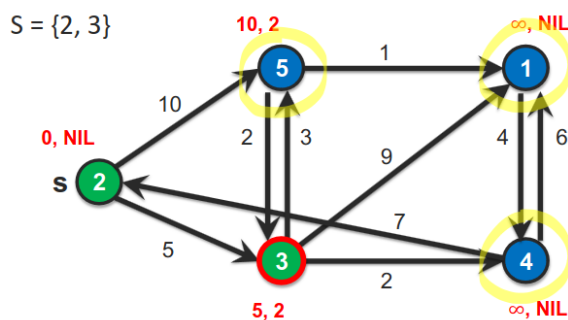
$$3.d > 2.d + W(3,5) \rightarrow 3.d = 2.d + W(3,5) = 5.$$



Seconda estrazione Estrae da Q il vertice con il minimo valore di $d \rightarrow$ vertice 3.

Aggiunge il vertice 3 all'insieme $S \rightarrow S = \{2, 3\}$.

Esamina gli adiacenti del vertice 3 $\rightarrow 1, 4, 5$.



Esegue il rilassamento dell'arco (3,5).

$$5.d > 3.d + W(3,5)? \text{ Sì } \rightarrow 5.d = 3.d + W(3,5) = 8.$$

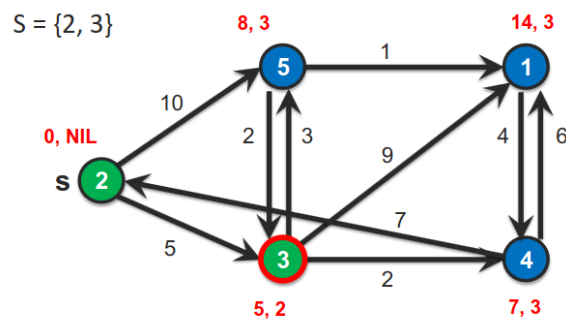
Esegue il rilassamento dell'arco (3,1).

$$1.d > 3.d + W(3,1) \rightarrow 1.d = 3.d + W(3,1) = 14.$$

Esegue il rilassamento dell'arco (3,4).

$$4.d > 3.d + W(3,4) \rightarrow 4.d = 3.d + W(3,4) = 7.$$

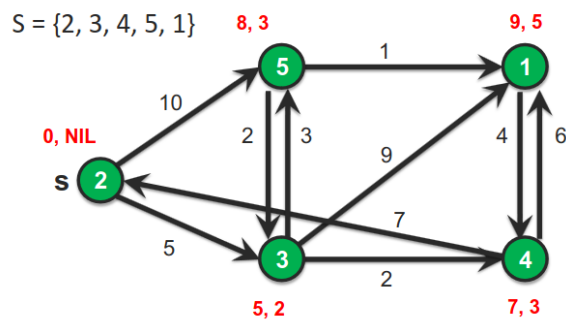
6.6. ALGORITMO DI DIJKSTRA - CAMMINI MINIMI DA SORGENTE UNICA93



Continuo a estrarre fino a quando la coda non è vuota Può capitare di avere come adiacente un vertice che non è più nella coda, lo ignoro e passo al successivo adiacente.

Può capitare che la condizione di rilassamento non sia soddisfatta, in quel caso ignoro il vertice e procedo al prossimo adiacente.

A fine esecuzione il grafo si presenterà nel seguente modo:



Il cammino minimo da s a un vertice v può essere ricostruito utilizzando il campo π dei predecessori.

Codice Dijkstra

```

Dijkstra( $G, W, s$ )
    Initialize-Single-Source( $G, s$ )
     $S = \emptyset$ 
    Aggiungi tutti i vertici di  $V$  alla coda  $Q$ 
    while  $Q \neq \emptyset$  do
         $u$  = estrai vertice da  $Q$ 
         $S = S \cup \{u\}$ 
        foreach  $v \in \text{adj}(u)$  do
            Relax( $u, v, W$ )
    
```

6.6.6 Prova di correttezza

TEOREMA : sia $\langle v_1 = s, v_2, \dots, v_n \rangle$ la sequenza di vertici estratti da Q in un'esecuzione. Quando il vertice v_k viene estratto, allora $v_k.d = \delta(s, v_k)$.

Lemma : se $v.d = \delta(s, v)$ a qualche passo dell'esecuzione, allora sicuramente v.d rimarrà uguale a $\delta(s, v)$ per il resto dell'esecuzione.