Analisi e Progetto di Algoritmi

Fabio Ferrario
@fefabo

Elia Ronchetti
@ulerich

2022

Indice

Introduzione

0.1 Il corso

Il corso è erogato nel primo semestre da Claudio Zandron e Alberto Dennunzio per il turno AL e da Paola Bonizzoni e Raffaella Rizzi per il turno MZ.

Argomenti del corso

1. Programamzione dinamica
   - 1.1 Esempi introduttivi
   - 1.2 Caratteristiche principali
   - 1.3 Implementazione con matrici
2. Algoritmi greedy
   - 2.1 Esempio: Scheduling di attività
   - 2.2 Elementi della strategia greedy
   - 2.3 Algoritmo di Huffman
   - 2.4 Dimostrazione di correttezza
   - 2.5 Greedy vs Dynamic programming: knapsack
   - 2.6 Definizione di matroide: esempi
   - 2.7 il teorema di Rado
3. Algoritmi su grafi
   - 3.1 Rappresentazione in memoria di un grafo
   - 3.2 Visita in ampiezza e in profondità di un grafo
   - 3.3 Ricerca delle componenti connesse di un grafo non orientato

3.4 Ricerca delle componenti (fortemente) connesse in un grafo orientato

3.5 Ricerca di cammini minimi in un grafo - Algorimti di programmazione dinamica

3.6 Costruzione di alberi di copertura minimi

3.7 Problemi di massimo flusso

4. NP completezza

4.1 Problemi trattabili e intrattabili

4.2 Riducibilità polinomiale

Alcuni materiali

https://lonati.di.unimi.it/algopig/2122/?page=materiali

Capitolo 1

Programmazione Dinamica

1.1 Introduzione

La programmazione dinamica è una tecnica di programmazione che viene utilizzata per risolvere problemi più velocemente rispetto ad una soluzione ricorsiva a discapito di un maggiore consumo di memoria.
Viene tipicamente applicata a *problemi di ottimizzazione*. Questi problemi ammettono (in genere) molte soluzioni possibili, ciascuna con un valore, di cui ci interessa trovare quella con il valore ottimo (massimo o minimo) che chiameremo *soluzione ottimale*.

1.1.1 Problemi di ottimizzazione

Per ogni istanza del problema esiste un insieme di soluzioni possibili (feasible solutions).
La funzione obiettivo che associa un valore ad ogni soluzione possibile.

Approccio generale La programmazione dinamica risolve un problema combinando le soluzioni dei suoi sottoproblemi.
l'approccio generale si può riassumere in 4 passi:

1. Caratterizzare la struttura di una soluzione ottimale.
2. Definire ricorsivamente il valore di una soluzione ottimale (e quindi di tutte).
3. Calcolare il valore delle soluzinoi ottimali, tipicamente in maniera bottom-up, memorizzando i loro valori in tabelle.
4. Costruire una soluzione ottimale usando le informazioni già calcolate e memorizzate.

Proprietà necessarie per applicare la programmazione dinamica in modo utile ed efficiente:

1. **Sottostruttura ottima** :Una soluzione ottima è esprimibile in termini di soluzioni ottime di sottoproblemi

2. **Sovrapposizione dei sottoproblemi**: L'insieme dei sottoproblemi distinti ha cardinalità di molto inferiore all'insieme delle soluzioni possibili da cui vogliamo seleziona quella ottima. Quindi un problema deve apparire molte volte come sottoproblema di altri problemi

DP vs D-et-I Come il metodo divide-et-impera, la programmazione dinamica risolve un problema combinando le soluzioni dei suoi sottoproblemi.
La programmazione dinamica è utile quando i sottoproblemi si sovrappongono, ovvero *diversi sottoproblemi contengono gli stessi sottoproblemi*
D-et-I risolverebbe inutilmente i sottoproblemi ogni volta, mentre un algoritmo di programmazione dinamica risolve ogni sottoproblema una sola volta e ne memorizza la soluzione in una tabella, in modo da evitare di dover ripetere ogni volta il calcolo della soluzione di un sottoproblema già risolto.

Esempio: Fibonacci Calcolando il numero di Fibonacci, se utilizziamo la tecnica D-et-I l'albero delle chiamate ricorsive *esplode*. Se invece utilizzando la programmazione dinamica memorizziamo i valori già calcolati possiamo evitare di ripetere inutilmente calcoli già risolti.
DP risulta quindi molto vantaggiosa quando il numero delle chiamate ricorsive distinte è polinomiale.

1.2 Weighted Interval Scheduling

Introduzione Il problema dello scheduling di attività riguarda la gestione di un insime di attività caratterizzate da un tempo di inizio, un tempo di fine e un valore. L'obiettivo è quello di determinare un insieme di attività mutualmente compatibili il cui valore totale è massimo.

Un insieme di attività si dice *mutualmente compatibili* se per ogni attività di tale insieme, nessun'altra attività dell'insieme si sovrappone a questa.

Pseudocodice $p(i)$ è il più grande indice i ¿ j tale che l'intervallo di indice i non si sovrappone all'intervallo di indice j.

```
1 WIS
2     M[0] = 0
```

```
3     for i = 1 to n
4         M[i] = max(v_i + M[p(i)], M[i-1])
```

In pratica, essendo OPT la soluzione ottimale, il valore di M[i] è:

- $v_i = M[p(i)]$ se $i \in OPT$
- $M[i-1]$ se $i \notin OPT$

1.3 Knapsack Problem 0/1

il problema Ci sono n oggetti ai quali sono associati un peso e un valore. Ho uno zaino di capictà in peso C. L'obiettivo è prendere k oggetti tali per cui il valore sia massimo senza superare la capacità dello zaino.
Questo tipo di problema si chiama *0/1* perchè un oggetto *O lo prendo tutto o non lo prendo proprio.* Si utilizza questa terminologia per differenziare lo 0/1 dal Knapsack Frazionario che vedremo in seguito.

Input $I = \{e_1, ..., e_n\} \forall i, e_i$ ha associati due valori: v_i (valore) e w_i (peso). W = Peso totale dello zaino.

Output Trovare un sottoinsieme $S \subset I$ di elementi $S = \{e_{j,1}, ..., e_{j,k}\}, 1 \leq j_l \leq n$ tc. soddisfa 2 vincoli:

- $\max(\Sigma_{e_i \in S} V_i)$;
- $\Sigma_{e_i \in S} W_i \leq W$.

Sottostruttura ottima La sottostruttura ottima del problema non è molto straight-forward. facciamo un esempio:

Se abbiamo un problema (che non necessariamente ci interessa definire) la cui soluzione S è $\{e_2, e_3, e_4\}$.
Se eliminiamo l'elemento e_i, S è comunque la soluzione ottimale dell'input I (senza considerare e_i)? No, perche se nell'input esisteva un altro elemento con lo stesso peso e valore di e_i che non era stato considerato per mancanza di spazio, allora la solzione ottimale è diversa.
Bisogna quindi aggiungere il vincolo che gli elementi sono ordinati e la soluzione è espressa in funzione dell'ordine degli elementi

1.4 LCS

LCS, o Longest Common Susbequence è un problema di programmazione dinamica in cui si richiede di trovare la più grande sottosequenza comune a due sottosequenze.

Definizione del problema Date due sequenze X e Y, trovare *la più lunga sottosequenza comune di caratteri* (non necessariamente consecutivi).
Una sottosequenza di una data sequenza è la sequenza stessa alla quale sono stati tolti zero o più elementi. Una sottosequenza comune di due sequenze è una sottosequenza di entrambe.
Il problema della LCS può essere risolto in modo efficiente applicando la programmazione dinamica.

Sottostruttura ottima di una LCS

LCS gode della proprietà della sottostruttura ottima, infatti:
Data una sequenza $X = \{x_1, x_2, ..., x_m\}$ definiamo $X_i = \{x_1, x_2, ..., x_i\}$ l'i-esimo prefisso di X per $i = 0, ..., m$

Teorema Siano $X = \{x_1, x_2, ..., x_m\}$ e $Y = \{y_1, y_2, ..., y_n\}$ le sequenze; sia $Z = \{z_1, z_2, ..., z_m\}$ una qualsiasi LCS di X e Y.

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

La caratterizzazione di questo teorema dimostra che una LCS di due sequenze contiene al suo interno una LCS di prefissi delle due sequenze. Quindi il problema della più lunga sottosequenza comune gode della proprietà della sottostruttura ottima

Spiegazione: Se sappiamo che *Z è una (qualsiasi) LCS di X e Y*, allora se ne deduce che:
(1) Se gli ultimi elementi di X e Y sono uguali, allora è anche l'ultimo elemento di Z. Inoltre implica che Z_{k-1}, quindi Z se togliamo il suo ultimo elemento, è una LCS di X e Y a cui sono stati tolti l'ultimo elemento ciascuno.
Se gli ultimi elementi di X e Y NON sono uguali invece, qui ci sono due opzioni:
(2) o l'ultimo elemento di Z è uguale all'ultimo elemento di Y, in tal caso Z è

una lcs di Y e X senza l'ultimo elemento, oppure (3) viceversa

La soluzione ricorsiva Definiamo $c[i,j]$ come la lunghezza di una LCS delle sequenze X_i e Y_j

$$c[i,j] = \begin{cases} 0 & \text{se } i = 0 \text{ o } j = 0 \\ c[i-1, j-1] + 1 & \text{se } x_i = y_j \\ max(c[i, j-1], c[i-1, j]) & \text{se } x_i \neq y_j \end{cases}$$

```
1  def LCS-LENGHT(X,Y):
2      m = X.lenght
3      n = Y.lenght
4      b[1..m,1..n] e c[0..m,0..m] -> nuove tabelle
5      for i = 1 to m
6          c[i,0] = 0
7      for j= 0 to n
8          c[0,j] = 0
9      for i = 1 to m
10         for j = 1 to n
11             if x_i == y_i
12                 c[i,j] = c[i-1,j-1] + 1
13                 b[i,j] = ↖
14             elif c[i - 1,j] ≥ c[i,j-1]
15                 c[i,j] = c[i-1,j]
16                 b[i,j] = ↑
17             else
18                 c[i,j] = c[i,j-1]
19                 b[i,j] = ←
20     return c e b
```

Esercizio: LCS $\geq L$

Istanza Date due sequenze:
$X =< x_1, ..., x_m >$ di lunghezza m
$Y =< y_1, ..., y_n >$ di lunghezza n
e dato un $L \geq 0$ stabilire se *La lunghezza di una qualunque LCS di* X *e* Y *è*

$\geq L$.
Quindi TRUE sse $\exists$ una LCS(X,Y) di lunghezza *geqL*

Istanza di un qualunque sottoproblema

X_i (con $0 \leq i \leq m$)

Y_j (con $0 \leq j \leq n$)

l (con $0 \leq l \leq L$)

Quindi ogni sottoproblema è individuato da una n-pla (i, j, l) con $0 \leq i \leq m$, $0 \leq j \leq n$, $0 \leq l \leq L$. La soluzione di ogni sottoproblema sarà un valore *True/False* a se cibda cge esusta i nebi yba LCS di X_i e Y_j di lunghezza *geql*. Pertanto introduciamo una variabile per ogni sottoproblema destinata a contenerne la soluzione.

Definizione delle Variabiili $\forall i \in \{0, ..., m\}, \forall j \in \{0, ...n\}, \forall l \in \{0..., L\}$ C_{ijl} è definita come soluzione del sottoproblema (i,j,l), ossia vale True sse la lunghezza di una qualunque LCS di X_i e Y_j è $\geq l$. Per ogni i,j,l come sopra dobbiamo ora calcoalre il valore di c_{ijl}.

Caso Base

$$c_{ijl} = \begin{cases} \text{True} & \text{se } l = 0 \\ \text{False} & \text{se } l > 0 \wedge (i < l \vee j < l) \end{cases} \tag{1.1}$$

Spiegazione: Se $l = 0$ c.. è truè perchè ovviamente qualunque LCS è maggiore o uguale a 0.
Se invece $l > 0$, ma una delle due stringhe è minore di l allora automaticamente non può esistere una LCS più lunga di l

Passo Ricorsivo Comprende tute le tuple (i,j,l) con $l > 0 \wedge i \geq l \wedge j \geq l$

Caso 1 sse $x_i = y_j$
$c_{ijl} = c_{i-1,j-1,l-1}$

Casi 2 se $x_i \neq y_j$,

2a $\text{LCS}(X_i, Y_j)$ termina con $z_k \neq x_i$
$c_{i-1,j,l}$

2b LCS(X_i, Y_j) termina con $z_k \neq y_j$
$c_{i,j-1,l}$

Risolvendo $\exists$ una LCS(X_i, Y_j) di lunghezza $\geq l$ sse ciò si verifica in almeno uno dei due casi, per cui: $c_{ijl} = c_{i-1,j,l} \vee c_{i,j-1,l}$. Riassumendo, il passo ricorsivo è: Dati $l > 0 \wedge i \geq l \wedge j \geq l$

$$C_{ijl} = \begin{cases} C_{i-1,j-1,l-1} & x_i = y_j \\ C_{i-1,j,l} \vee C_{i,j-1,l} & x_i \neq y_j \end{cases} \tag{1.2}$$

Spiegazione: Siccome si tratta di programmazione dinamica, e siccome tutti i valori che abbiamo calcolato vengono salvati in una matrice (C) prendo il valore di quella matrice, già inizializzata con il caso base, che più si "addice al caso"

Soluzione Con istanza X,Y,L la soluzione del problema è il valore di $c_{m,n,L}$

Pseudocodice LCS $\geq$ L

```
def LCS-MAGGIORE-L(X,Y,L)
  m = lenght[X]
  n = lenght[Y]
  for i=0 to m
    for j=0 to n
      c_{i,j,0} = True
  for l=1 to L
    for i=0 to m
      for j=0 to n
        if i < l ∨ j < l
          c_{i,j,l} = False
        elif x_i = y_j
          c_{i,j,l} = c_{i-1,j-1,l-1}
        elif x_i \neq y_j
          c_{i,j,l} = c_{i-1,j,l} ∨ c_{i,j-1,l}
  return c_{m,n,L}
```

1.5 LIS

LIS, o Longest Increasing Subsequence è un algoritmo che calcola la lunghezza della più lunga sottosequenza crescente di una determinata sequenza. A

differenza di LCS, LIS viene calcolata su una sola sequenza, in questo caso una sottosequenza è un qualunque sequenza di caratteri che fanno parte della nostra sequenza.

Definizione del Problema Data una sequenza X di n caratteri, si vuole trovare la lunghezza della più lunga sottosequenza di caratteri *strettamente crescenti* (in ordine lessicografico).
Istanza: $X =< x_1, ..., x_n >$, **Soluzione:** La più lunga sotto sequenza di X strettamente crescente.

Sottoproblema

Il sottoproblema di taglia i è la più lunga sottosequenza strettamente crescente che termini con l'*i-esimo* carattere di X.
Nel caso di $i = 1$, la soluzione è ovviamente 1, siccome ogni sottosequenza di dimensione 1 è strettamente crescente (e decrescente). Mentre per i sottoproblemi di taglia maggiore di 1, ipotizzando di aver già risolto i sottoproblemi più piccoli ci basta vedere quale sia la massima lunghezza di una LIS compatibile con li carattere x_i (ovvero che sia abbia l'ultimo carattere minore di x_i) e aggiungerci 1.

Definizione delle variabili Definiaimo $S[i]$ come la soluzione al sottopbroblema di taglia i, ovvero come la lunghezza della più lunga sottosequenza strettamente crescente che *termini* con l'i-esimo carattere di X

Equazioni di ricorrenza

Caso Base con $i = 1$,

$$S[i] = \Big\{1$$

visto che è la lunghezza massima di una sottosequenza di dimensione 1.

Passo Ricorsivo con $1 < i \leq n$,

$$S[i] = \Big\{1 + \max\{S[j]\}|0 \leq j < i|x_j < x_i$$

Ovvero, la soluzione è la più grande tra le soluzioni precedenti in cui l'ultimo carattere è minore del carattere i aumentata di uno.

Soluzione

la soluzione di LIS è il $\max\{S[i]|0 < i \leq n\}$.
A differenza di LCS, in LIS la soluzione non sarà mai allo stesso posto ma alla posizione del carattere in cui termina la sottosequenza più grande.

Pseudocodice

```
1   def LIS(X):
2       S[1] = 1
3       for i = 2 to n
4           max = 0
5           for j = 0 to i
6               if S[j] > max AND X[j] < X[i]
7                   max = S[j]
8           S[i] = max + 1
9       return max(S)
```

Tempo e Spazio

Il tempo dell'algoritmo è pari a $\Theta(n^2)$
Lo spazio è pari a $\Theta(n)$

Per ricostruire la sequenza potremmo ripercorrere l'array S all'indietro controllando, ogni volta che troviamo un valore minore di quello analizzato se il carattere nella posizione di tale valore sia compatibile con quello nella posizione che stiamo analizzando.

1.6 LGCS / LICS

LGCS, o Longest Growing Common Subsequence (LICS - Longest Increasing Common Subsequence), è una variazione di LCS in cui si richiede di trovare la più lunga sottosequenza comune di caratteri *strettamente crescenti* (in ordine alfabetico).

Definizione del Problema Date due sequenze X e Y, rispettivamente di m ed n caratteri, trovare *la lunghezza della più lunga sottosequenza comune di caratteri strettamente crescenti*

Sottoproblema

Il sottoproblema non prende in considerazione l'intero input, ma solo i prefissi di X e Y rispettivamente di dimensione i e j. è così definito:
Date due sequenze X e Y, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti comuni al prefisso X_i e Y_j.
Ad ogni sottoproblema è associata una **variabile** $c_{i,j}$ così definita:
$c_{i,j}$ = lunghezza di una tra le più lunghe sottosequenze crescenti comuni a X_i e Y_j
Il problema così definito però non è risolvibile perchè *manca dell'informazione*, bisogna quindi definire un problema ausiliario nel quale introdurre l'informazione mancante necessaria.

Problema ausiliario

Il problema ausiliario, che ci serve a fornire le informazioni mancanti per risolvere il problema originale, è definito come segue:

Definizione di Problema Ausiliario Date due sequenze X e Y, la soluzione è data dalla lunghezza della più lunga sottosequenza comune crescente *che termini con* $x_n = y_m$.

Spiegazione: Viene aggiunto il vincolo che l'ultimo elemento della Soluzione sia l'ultimo elemento delle stringhe *solo nel caso in cui coincidano*. Se questi non coincidono, la soluzione è la sequenza vuota. Questa richiesta, quando applicata ai sottoproblemi ci permetterà di sapere qual'è l'ultimo elemento di un sottoproblema.

Sottoproblema date due sequenze X e Y , rispettivamente di m ed n caratteri, si determini la lunghezza di una tra le più lunghe sottosequenze crescenti comuni a X_i e a Y_j e che termina con x_i e y_j solo se questi coincidono.
ad ogni sottoproblema è associata la varaible $c_{i,j}$ così definita:
$c_{i,j}$ = lunghezza di una tra le più lunghe sottosequenze crescenti comuni a X_i e Y_j e che termina con x_i e y_j solo se questi coincidono.

Spiegazione: Si noti quindi che per un qualunque sottoproblema di dimensione (s, t), solo imponendo che la soluzione $c_{s,t}$ si riferisca ad una sottosequenza crescente comune a X_s e Y_t che termina con $x_s = y_t$ è possibile stabilire se un altro elemento $x_u = y_v$ con $u > s$ e $v > t$ possa essere accodato a tale

sottosequenza (andando a verificare che $x_s = y_t < x_u = y_v$).

Equazioni di ricorrenza

Nota bene un equazione di ricorrenza è composta da:

- Un caso base che definisce i casi più semplici che possono essere subito risolti senza ricorrere alle soluzioni dei sottoproblemi più piccoli
- Un passo ricorsivo che definisce come risolvere i casi più complessi a partire dalle soluzioni dei sottoproblemi più piccolo (che si assume essre già risolti)

Caso Base Fano parte del caso base:

- tutte le coppie (i, j) con $i = 0 \vee j = 0$
- tutte le coppie (i, j) tale che $x_i \neq y_j$.

Quindi: il caso base base si ha per quei sottoproblemi di dimensione (i, j) con $i < 0 \vee j < 0 \vee x_i \neq y_j$. è scrivibile come:

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

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

Spiegazione: Per la definizione del problema, noi sappiamo solamente che se x_i e y_j non coincidono, oppure una delle due stringe (X o Y) è vuota, allora anche la soluzione per quel sottoproblema è vuota, perchè non abbiamo sottosequenze comuni crescenti.

Passo ricorsivo Il passo ricorsivo lo si ha per un qualunque sottoproblema di dimensione (i, j) tale che $x_i = y_j$, ossia quando i prefissi X_i e Y_j terminano con lo stesso elemento.
In questo caso, la lunghezza della LGCS fra X_i e Y_j è uguale alla *lunghezza della più lunga LGCS fra X e Y calcolata per un sottoproblema minore di dimensione (h,k)* che termina con *un carattere minore di $x_i = y_j$ aumentata di uno.* Ovvero:

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

$$\text{se } i > 0 \wedge j > 0 \wedge x_i = y_j$$

Dato che può accadere che l'insieme $\{c_{h,k} | 1 \leq h < i, 1 \leq k < j, x_h < x_i\}$ sia vuoto, assumiamo per definizione che $\max\{\emptyset\} = 0$ così che il cporrispondente valore di $c_{i,j}$ risulti uguale a 1.

Soluzione del problema Una volta calcolati tutti i valori di c fino a $c_{m,n}$ si hanno a disposizione tutte sottosequenze LGCS fra qualsiasi prefisso di X e di Y. Quindi la soluzione del problema iniziale è:

$$max\{c_{i,j} | 1 \leq i \leq m \wedge 1 \leq j \leq n\}$$

Pseudocodice di LGCS con implementazione Bottom-Up

```
1  def LGCS(X,Y)
2      max = 0
3      for i = 1 to m
4          for j = 1 to n
5              if x_i ≠ y_j
6                  c[i,j] = 0 #caso base
7              else
8                  temp = 0
9                  for h = 1 to i-1
10                     for k=1 to j-1
11                         if (x_h < x_i) ∧ (c[h,k] > temp)
12                             temp = c[h,k]
13                 c[i,j] = 1 + temp
14             if c[i,j] > max
15                 max = c[i,j]
16     return max
```

Capitolo 2

Algoritmi Greedy

2.1 Definizione

A differenza degli algoritmi di programmazione dinamica, gli algoritmi **greedy** non si basano su risultati precedenti già calcolati, ma fanno una serie di scelte *basandosi su quella che risulta essere la scelta migliore ad ogni istante*, ovvero fa una scelta *localmente ottima* sperando che tale scelta porterà ad una soluzione *globalmente ottima*.
Spesso la difficoltà degli algoritmi greedy non sta tanto nello scrivere l'algoritmo in se, ma nel determinare *se* l'algoritmo funziona.

Applicazioni Il metodo Greedy è molto potente e può essere applicato a una vasta gamma di problemi, ad esempio viene spesso usato per algoritmi per *alberi di connessione minimi*, o Dijkstra per il problema dei *cammini minimi da sorgente unica*.

2.2 Algoritmo: Scheduling di Attività

Introduzione

Il primo esempio è il problema della programmazione di più attività in competizione che richiedono l'uso *esclusivo* di una risorsa comune, con l'obiettivo di selezionare il più grande insieme di attività mutualmente compatibili.

Mutualmente compatibili significa che date due attività i e j esse non si devono sovrapporre, quindi i deve iniziare dopo (o durante) la fine di j oppure viceversa.

Sottostruttura ottima

Possiamo facilmente verificare che il problema della selzione di attività presenta una **sottostruttura ottima** (quindi una sua soluzione ottima è esprimibile in termini di soluzioni ottime di sottoproblemi).
Sia $A \subseteq S$ una *soluzione ottima* (S è insieme delle attività) e sia $a_i \in A$. a_i induce i due sottoproblemi:

- $S_i^- = \{k \in S : f_k \leq S_i\}$
- $S_i^+ = \{k \in S : f_i \leq S_k\}$

è immediato verificare che:

- $A \cap S_i^-$ è una soluzione ottima per S_i^-
- $A \cap S_i^+$ è una soluzione ottima per S_i^+

Da qui si può dedurre una soluzione risolvibile tramite programmazione dinamica, però noi vogliamo fare una scelta Greedy

La scelta Greedy

Se risolvessimo il problema tramite la programmazione lineare, dovremmo considerare ogni volta tutte le scelte previste dalla sua ricorrenza, noi però possiamo considerare una sola scelta: quella *golosa*. Tra tutte le attività che possiamo scegliere ce ne deve essere una che finisce per prima. Se noi scegliamo quell'attività la risorsa risulterebbe disponibile per il maggior numero possibile di attività successo (questo succede perchè le attività sono ordinate per tempo di fine). Ma questa intuizione è corretta? Se scegliamo la prima attività, sappiamo che non esistono altre attività che finiscono prima di a_1, quindi non ne troveremmo prima, inoltre bisogna considerare il seguente teorema:

teorema Consideriamo un sottoproblema non vuoto S_k e sia a_m l'attività in S_k che ha il primo tempo di fine; allora l'attività a_m p inclusa in qualche sottoinsieme massimo di attività mutualmente compatibili di S_k.

2.3 Sistemi di Indipendenza

Definizione Data la coppia $< E, F >$ dove E è un insime finito e F è una famiglia di sottoinsiemi di E, definiamo tale coppia *sistema di indipendenza*

Se vale la seguente proprietà:

$$\forall A \in F \wedge B \subseteq A \implies B \in F$$

Ovvero: dato un insieme di sottoinsiemi di E chiamato F e preso un qualsiasi elemento A (che è un insieme) da F, in F abbiamo anche tutti i sottoinsiemi di A

Osservazione Un grafo può essere visto come un sistema di indipendenza in cui solo lati e vertici costituiscono gli insiemi indipendenti.

Capitolo 3

Algoritmi su Grafi

Questo capitolo tratta di molti algoritmi (e rispettive varianti) che operano sui *grafi*.

3.1 Riassunto sui grafi

Innanzitutto, è bene fare un piccolo ripasso su cosa sono i grafi e di che tipo sono.

Definizione di Grafo Un grafo è un *insieme di elementi detti nodi* (o vertici) che possono essere *collegati fra loro da linee chiamate archi* (o lati o spigoli). Più **formalmente**, si dice grafo una coppia ordinata $G = (V, E)$ di insiemi, con V insieme dei nodi ed E insieme degli archi, tali che gli elementi di E siano coppie di elementi di V.

Tipi di grafi Esistono vari tipi di grafi, che si differenziano per struttura e/o funzione. Tra questi abbiamo:

- **Grafo Semplice**: Grafo non orientato che non comprende cappi e archi multipli.
- **Grafo Completo**: è un grafo semplice nel quale ogni vertice è collegato a tutti gli altri vertici, quindi $n_{archi} = n_{vertici} * (n_{vertici} - 1)/2$.
- **Albero**: è un grafo non orientato G connesso tc:
  G è *aciclico* $\vee$ $|E| = |V| - 1$.
  Se ogni nodo di un Albero ha $\{0, 1, 2\}$ figli, allora è detto **Albero Binario**.

- **DAG**: Grafo diretto (orientato) senza cicli, quindi aciclico. Un grafo diretto può dirsi aciclico se una visita in profondità NON presenta archi all'indietro.

Altre definizioni

Chiusura Transitiva Dato un grafo $G = (N, A)$ (diretto o non) si dice la *chiusura transitiva* quel grafo $G* = (N, A*)$ in cui esiste un arco tra i nodi i e j se esiste un cammino tra i e j.

Spiegazione: La chiusura transitiva di un grafo è un'altro grafo con gli stessi vertici che per ogni nodo i e j ha un arco se e solo se nel grafo originale esisteva un cammino tra di essi.

3.2 Algoritmi di Visita

Esistono due tipi di algoritmi di visita dei grafi, in ampiezza (BFS) e in profondità (DFS)

Colore dei nodi $col[u]$ Sia BFS che DFS per funzionare assegnano un colore ad ogni vertice per capire se è già stato scoperto o no

- White: Vertice non ancora scoperto
- Gray: Vertice scoperto ma la cui lista di adiacenza non è ancora stata scandita del tutto
- Black: Vertice scoperto e di cui ho scandito per intero la lista di adiacenza

Predecessore del nodo $\pi[u]$ Il predecessore di u è il nodo che mi ha permesso di scoprirlo, quindi quello da cui sono "passato" nella mia ricerca per scoprire u

Tempo di scoperta del nodo $d[u]$ Il tempo di scoperta di u indica quanti "passaggi" (quindi quanti nodi ho scoperto prima) mi ci sono voluti per arrivare a u

3.2.1 Algoritmo BFS

BFS (Breadth-First search) è un algoritmo di visita di un grafo in ampiezza. L'algoritmo scopre tutti i vertici raggiungibili partendo da un vertice sorgente s, quindi soltanto della sua componente connessa.
La scoperta avviene in ampiezza, ovvero parte da tutti i vertici a distanza 1 dalla sorgente, poi 2 e così via.
Alla fine dell'esecuzione, tutti i vertici della componente connessa di s avranno colore NERO.

Pseudocodice Algoritmo BFS

```
1  def BFS(G, s)
2      for ogni v ∈ V\{s}
3          col[v] = White
4          d[v] = ∞
5          π[v] = NIL
6      col[s] = Gray
7      d[s] = 0
8      π[s] = NIL
9      ENQUEUE(Q,s)
10     while Q≠ ∅
11         u = DEQUEUE(Q)
12         for ogni v ∈ adj[u]
13             if col[v]==White
14                 col[v] = Grigio
15                 d[v] = d[u] +1
16                 π[v] = u
17                 ENQUEUE(Q,v)
18         col[u] = Nero
```

Spiegazione Codice BFS inizializza tutti i nodi del grafo (tranne s) in modo da renderli "elaborabili", quindi li mette bianchi, con distanza infinita e con padre NULL. In seguito crea una coda Q che andrà a contenere tutti i nodi grigi, quindi quelli di cui va ancora completata la lista di adiacenza, dove inserisce s, che è il primo nodo Grigio.
Nel ciclo while l'algoritmo prende il primo nodo da Q e va a scoprire tutti i nodi bianchi (quindi non ancora elaborati) nella sua lista di adiacenza, inserendoli di volta in volta in Q. Quando la lista di adiacenza è stata scandita per intero, vuol dire che il nodo è stato "scoperto del tutto" e quindi diventa Nero. Il ciclo ricomincia finchè Q non sarà vuota.

Tempo di esecuzione Il tempo totale di esecuzione di BFS è $O(V + E)$. Di cui $O(V)$ è il tempo delle operazioni con la coda, e $O(E)$ è il tempo per l'ispezione di ADJ.

Sottografo dei predecessori La visita in ampiezza costruisce un albero BF *(Albero di ricerca in ampiezza)*, che ha alla radice il vertice sorgente s. Quando durante l'ispezione della lista di adiacenza di un vertice u viene scoperto un vertice *bianco* v, il vertice v e l'arco (u, v) che lo collega a u vengono aggiunti all'albero. Il vertice u viene detto *padre* di v.

RIASSUMENDO BFS

L'algoritmo BFS effettua una visita in ampiezza di un grafo partendo da un nodo sorgente s fornito. Utilizza una coda Q, che andrà a contenere tutti i nodi grigi, quindi quelli di cui ha iniziato la scannerizzazione di ADJ. Quindi parte dal primo nodo in Q (che sarà s) e ne scannerizza la lista di adiacenza, inserendo ogni nodo bianco (trasformato in grigio) all'interno di Q. Una volta finito il nodo diventa nero. Grazie all'assegnazione di un valore π ai nodi BFS è in grado di costruire un albero BFS.

3.2.2 Algoritmo DFS

DFS (Depth-First search) è un algoritmo di visita di un grafo in profondità. Alla fine dell'esecuzione dell'algoritmo siamo in grado di determinare quante sono le componenti connesse del grafo ed a quale componente connessa appartiene ogni nodo.

Classificazione degli archi Dato un grafo orientato, DFS differenzia ogni arco (u, v) in 4 modi diversi

- Arco dell'albero: $col[v] = BIANCO$, quindi è la prima volta che visitamo v
- Arco all'indietro: $col[v] = GRIGIO$, quindi v è antenato di u
- Arco in avanti: $col[v] = NERO \wedge d[u] < d[v]$, quindi v è stato scoperto dopo u
- Arco di attraversamento: $col[v] = NERO \wedge d[u] > d[v]$, quindi v è stato scoperto prima di u

In un grafo NON ORIENTATO esistono soltanto gli archi dell'albero e gli archi all'indietro.

Pseudocodice Algoritmo DFS

```
1  def DFS(G)
2      for ogni u ∈ V
3          col[u] = White
4          π[u] = NIL
5      time = 0
6      for ogni u ∈ V
7          if col[u] == White
8              DFS-VISIT(u)
```

```
1      def DFS-VISIT(G,u)
2          col[u] = Gray
3          time++
4          d[u] = time
5          for ogni w ∈ Adj[u]
6              if color[w] == White
7                  π[w] = u
8                  DFS-VISIT(G,w)
9          col[u] = Black
10         time ++
11         f[u] = time
```

Spiegazione pseudocodice DFS comincia inizializzando il grafo come BFS ma senza impostare d[]. Poi fa un ciclo su ogni nodo del grafo, chiamato DFS-VISIT su tutti i nodi bianchi che trova. DFS-VISIT(G,u) visita tutti i nodi adiacenti a u, operando ricorsivamente su ogni nodo bianco che trova. Alla fine dell'esecuzione DFS-VISIT mette nero il nodo in esame e procede con il primo nodo bianco che trova nell'albero.

Tempo di calcolo DFS ha un tempo di esecuzione di $\theta(V + E)$. DFS-VISIT è chiamata una volta per ogni vertice (quando è bianco, quindi $\theta(V)$) e il ciclo in DFS-VISIT è chiamato una volta per ogni arco (ogni volta che c'è una adiacenza, quindi $\theta(E)$)

Sottografo dei predecessori Anche DFS genera un *sottografo dei predecessori* $G_\pi < V, E_\pi >$, ma a differenza di BFS G_π forma effettivamente una *foresta di alberi*, in cui ogni albero rappresenta una componente connessa del grafo. Nota che, a differenza del sottografo dei predecessori di BFS, G_π contiene tutti i vertici di G.
Formalmente, il sottografo dei predecessori (o foresta DF) è così definito:

$G\pi = (V, E\pi)$, dove: $E\pi = (\pi[v], v) : v \in V \wedge \pi[v] \neq NIL$
in pratica, G_π è formato da tutti i vertici di G e tutti gli archi che vanno dal "padre" di un nodo v a v.

RIASSUMENDO DFS

DFS è un algoritmo di visita (in profondità) che visita tutte le componenti connesse di un grafo G. DFS inizia mettendo bianco ogni vertice di G, poi chiama DFS-VISIT per un nodo u e visita l'intera lista di adiacenza di quel nodo usando delle chiamate ricorsive su ogni nodo bianco. Quando il controllo ritorna a DFS significa che ha controllato l'intera componente connessa di u. L'algoritmo cerca il prossimo nodo bianco e il ciclo ricomincia. Ogni volta che DFS trova un nodo bianco significa che ha trovato una nuova compoente connessa. DFS ha un tempo di $\theta(V + E)$, in particolare chiama $DFS - VISIT$ una volta per vertice, con il ciclo al suo interno chiamato una volta per arco.

Ordinamento Topologico

Questo paragrafo spiega come utilizzare la visita in profondità per eseguire l'ordinamento topologico di un grafo *orientato aciclico* o DAG (Directed acyclic graph).
Un ordinamento topologico di un DAG $G < V, E >$ è un ordinamento lineare di tutti i suoi vertici tale che , se G contiene un arco (u, v) allora u appare prima di v nell'ordinamento.
In pratica, un ordinamento topologico può essere visto come un ordinamento dei suo vertici lungo una linea orizzontale in modo che tutti gli archi orientati siano diretti da sinistra a destra.
l'algoritmo TOPOLOGICAL-SORT(G) per ottenere un ordinamento topologico chiama DFS per calcolare i tempi di completamento $v.f$ e poi completata l'ispezione inserisce il vertice in una lista concatenata che poi ritorna.

3.3 Cammini minimi

Alcuni algoritmi sui grafi molto importanti sono quelli che calcolano i *cammini minimi* per grafi orientati **pesati**, quindi grafi (orientati) che hanno un peso assegnato su ogni arco.

definizioni Sia $G = (V, E)$ un grafo orientato con costi w sugli archi, il costo di un cammino $p =< v_1, v_2, ...v_k >$ è dato dalla somma del peso di tutti

i vertici di quel cammino.
Il **Cammino minimo** tra una coppia di vertici x e y è *un cammino di costo minore o uguale a quello di ogni altro cammino tra gli stessi vertici.*
Sottostruttura ottima: ogni sottocammino di un cammino minimo è anch'esso minimo.
Albero dei cammini minimi: I cammini minimi da un vertice s a tutti gli altri vertici del grafo possono essere rappresentati tramite un albero radicato in s, detto albero dei cammini minimi

Gli algoritmi che studieremo sono di due tipi: da sorgente unica, come Dijkstra e Bellman-Ford, oppure quelli che calcolano i cammini minimi per ogni coppia di vertici, come Floyd-Warshall.

3.3.1 Algoritmo di Floyd-Warshall

L'algoritmo di Floyd-Warshall calcola il *peso* del cammino minimo da i a j *per ogni coppia di vertici* (i, j) del grafo (pesato e orientato) su cui viene eseguito.

Funzionamento L'idea alla base di questo algoritmo è un processo iterativo che, scorrendo tutti i nodi, ad ogni passo h si ha (data una matrice D) nella posizione $[i, j]$ la *distanza di peso minimo* dal nodo di indice i a quello j attraversando solo nodi di indice minore o uguale a h.
Quindi D^h equivale alla matrice che contiene i cammini minimi utilizzando come nodi intermedi al massimo i nodi di indice h.
Se non vi è collegamento tra due nodi allora nella cella corrispondente c'è infinito. Ovviamente alla fine (con h = numero di nodi) leggendo la matrice si ricava la distanza minima fra i vari nodi del grafo. L'algoritmo di Floyd-Warshall è un algoritmo di programmazione dinamica bottom-up.

Equazione di Ricorrenza

$$d_{i,j}^{(h)} = \begin{cases} W_{ij}, & \text{if } h = 0 \\ min\{d_{ij}^{(h-1)}, d_{ih}^{(h-1)} + d_{jh}^{(h-1)}\} & \text{if } h > 0 \end{cases}$$

Spiegazione: La variabile $d_{i,j}^{(h)}$, che contiene il peso del cammino minimo da i a j con al più h vertici intermedi e vale: il peso del cammino (i, j) se non usiamo vertici intermedi (se non c'è il collegamento questo vale infinito), oppure il minore tra i pesi dei cammini che utilizzano un cammino intermedio,

quindi tra il cammino che non usa h come vertice intermedio e la somme dei due cammini da i a h e da h a j.

Struttura di un cammino minimo L'algoritmo considera i vertici "intermedi" di un cammino minimo, dove un vertice intermedio di un cammino semplice $p =< v_1, .., v_l >$ è un vertice qualsiasi di p diverso da v_1 e v_l ovvero un vertice qualsiasi dell'insieme $\{v_2, ..., v_{l-1}\}$.

Pseudocodice Algoritmo di Floyd-Warshall

```
1 def FLOYD-WARSHALL (G, W)
2     D^(0) = W
3     for h = 1 to n
4         for i = 1 to n
5             for j = 1 to n
6                 d_ij^(h) min{d_ij^(h-1), d_ih^(h-1) + d_jh^(h-1)}
```

Spiegazione Codice L'algoritmo inizia mettendo nella matrice D il peso di ogni arco senza nodi intermedi. Procede poi con tre cicli for, il cui più esterno è h e il più interno j, in cui confronta i cammini minimi di ogni nodo con ogni altro nodo aumentando di volta in volta il (possibile) numero di nodi intermedi (h)

Tempo di esecuzione

$$O(|V|^3)$$

Variante di FW: Cammini minimi $\leq L$

Dato un grafo orientato e senza cappi (V, E, W) e dato un itnero L>0 calcolare $\forall (i, j) \in V^2$ il peso di un cammino minimo da i a j di lunghezza $\leq L$

Variabili introdotte $D^{(k,l)} = (d_{ij}^{(k,l)})$
dove $d_{ij}^{(k,l)}$ è il peso del cammino minimo da i a j con vertici intermedi $\in \{1, ..., k\}$ di lunghezza $\leq l$

Caso base (k,l) con k=0

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

Spiegazione caso base Se i = j allora la distanza è 0, siccome il cammino deve essere di lunghezza ≤l un cammino di lunghezza 0 è accettato
Se $i \neq j \wedge (i,j) \in E$ allora è w_{ij}, perchè siccome cè un solo cammino sarà sempre di lunghezza 1, che è minore o uguale ad ogni l
Infinito altrimenti, perchè non c'è un cammino che collega i a j

Passo ricorsivo (k,l) con $k > 0$

$$d_{(ij)}^{(k,l)} = \begin{cases} min\{d_{ij}^{(k-1,l)}, d_{(ik)}^{(k-1,l_1)} + d_{(kj)}^{(k-1,l_2)}\} & \text{if } l > 1 \text{ e } l_1, l_2 \in \{1,...,l\}, l_1 + l_2 \leq l \\ min\{d_{ij}^{(k-1,l)}, \infty\} & \text{if } l = 1 \end{cases}$$

Spiegazione Passo ricorsivo Se $l > 1$ vuol dire che ci può essere un vertice intermedio k tra i e j, quindi bisogna trovare il minimo tra la distanza a k-1 e la somma di due percorsi di lunghezza minore di l tra due percorsi (ovviamente minori di k) che utilizzando k come intermedio e la cui somma non superi l.

Invece se k non fa parte del cammino minimo, si sceglie il minore tra un eventuale percorso tra i e j che non include k oppure infinito.

```
1  CAMMINIMINIMI_MINORE_DI_L(V,E,W,L)
2  for l=1 to L //Caso base puro (k=0)
3      for i=1 to n
4          for j=1 to n
5              if i==j
6                  d_ij^(0,l) = 0
7              elif (i,j) ∈ E
8                  d_ij^(0,l) = w_ij
9              else
10                 d_ij^(0,l) = ∞
11 for k=1 to n //Caso passo
12     for l=1 to L
13         for ogni i
```

```
14          for ogni j
15              if l==1
16                  d_ij^(k,l) = min{d_ij^(k-1,l), ∞}
17              else
18                  for l_1 = 0 to L
19                      for l_2 = 0 to L
20                          if l_1 + l_2 ≤ L
21                              d_ij^(k,l) = min{d_ij^(k-1,l), d_ik^(k-1,l_1) + d_jk^(k-1,l_2)
                                    }
```

3.4 Cammini minimi da sorgente unica

Gli algoritmi di Dijkstra e di Bellman-Ford risolvono il problema dei cammini minimi da sorgente unica. Quindi vengono usati quando vogliamo trovare un cammino minimo che va da un dato vertice sorgente $s \in V$ a ciascun vertice $v \in V$ in un grafo orientato pesato $G = (V, E)$

Differenze Dijkstra funziona soltanto se tutti i pesi degli archi sono NON NEGATIVI, mentre Bellman-Ford non ha bisogno di questa premessa.

Funzionamento comune Come tutti gli algoritmi per cammini minimi entrambi si basano sulla proprietà della Sottostruttura ottima di un cammino. In questi algoritmi vengono assegnati due attributi per ogni vertice del grafo:

- $\pi(v)$ Che indica il predecessore di v nel cammino minimo
- $d(v)$ Che indica la distanza di v dal nodo sorgente s

Inoltre questi algoritmi hanno bisogno di due funzioni d'appoggio, INITIALIZE e RELAX

Inizializzazione del grafo Per gli algoritmi di questo tipo viene spesso utilizzata una funzione INITIALIZE, che imposta le distanze e i "padri" di ogni nodo rendendo s la nostra sorgente (quindi a distanza zero)

```
1
2  INITIALIZE(G, s)
3      for ogni v ∈ V
```

```
4         v.d = ∞
5         v.π = NIL
6     s.d = 0
```

Tecnica del rilassamento La tecnica del rilassamento di un $arco(u, v)$ consiste nel verificare se, passando per u, è possibile migliorare il cammino minimo per v precedentemente trovato. Quindi partendo da stime per eccesso delle distanze le decrementiamo progressivamente fino a renderle esatte

```
1
2 RELAX(u, v, w)
3     if v.d > u.d + w(u, v)
4         v.d = u.d + w(u, v)
5         v.π = u
```

In sostanza, se la distanza del vertice v è maggiore della distanza di u più il peso dell'arco che va da u a v, allora sostituisci la distanza di v con $u.d + w(u, v)$ e imposta u come padre di v

3.4.1 Algoritmo di Dijkstra

L'algoritmo di Dijkstra ritorna in output l'insieme S contenente tutti i cammini minimi per ogni nodo del grafo. In Dijkstra il rilassamento viene eseguito esattamente *una volta per arco.*

Attenzione Dijkstra funziona solo se w $\geq 0 \; \forall$ w $\in$ W

Pseudocodice Algoritmo di Dijkstra

```
1 DIJKSTRA(G, w, s)
2     INITIALIZE(G,s)
3     S = ϕ
4     Q = G.V
5     While Q ≠ ϕ
6         u = extract-min(Q)
7         S = S ∪ u
8         for ogni vertice v ∈ Adj[u]
9             RELAX(u, v, w)
```

Spiegazione codice Q è una coda che contiene tutti i vertici del grafo ed S è l'insieme delle soluzioni. Viene estratto il vertice con distanza minore dalla sorgente (al primo giro sarà sempre s dato che è a distanza 0) e viene aggiunto all'insieme delle soluzioni. Viene poi fatto il rilassamento per ogni vertice adiacente a quello aggiunto alla soluzione, aggiornandone la stima della distanza dalla sorgente e il predecessore. viene ripetuto per ogni nodo rimanente nella coda Q.

Greedy Dijkstra segue l'approccio Greedy, quindi sceglie sempre il vertice più "leggero" o "vicino" in $V \setminus S$ da aggiungere all'insieme S. Vi è un'analogia con il criterio per l'ordinamento negli algoritmi Greedy

Tempo di esecuzione

- $O(E + |V| * log|V|)$ Se utiliziamo lo Heap di Fibonacci per estrarre velocemente il nodo a distanza minore nella coda
- $O(|V|^2)$ Altrimenti (implementazione naive)

3.4.2 Algoritmo di Bellman-Ford

Attenzione Bellman-Ford funziona anche se i pesi degli archi sono negativi

Funzionamento Bellman-Ford ritorna in Output un valore booleano che indica se esiste oppure no un ciclo di peso negativo che è raggiungibile dalla sorgente. Se tale ciclo non esiste, l'algoritmo fornisce i cammini minimi e i loro pesi. L'algoritmo restituisce true se e solo se il grafo non contiene cicli di peso negativo che sono raggiungibili dalla sorgente.
In Bellman-Ford il rilassamento viene eseguito $|V| - 1$ volte per arco indipendentemente dalla morfologia del grafo. Bellman-Ford NON segue un approccio Greedy.

Pseudocodice algoritmo di Bellman-Ford

```
1 BELLMAN-FORD(G, w, s)
2     INITIALIZE(G, s)
3     for i = 1 to |V|-1
4         for ogni (u,v) ∈ E
5             RELAX(u, v, w)
6     for ogni (u,v) ∈ E
7         if v.d > u.d + w(u, v)
```

```
8             return FALSE
9     RETURN TRUE
```

Spiegazione codice Dopo aver inizializzato il grafo, Bellman-Ford procede rilassando ogni arco $|V| - 1$ volte, Se il grafo ha N nodi è certo che dopo N-1 giri tutti i nodi hanno a loro assegnato il costo minimo per essere raggiunti dal nodo sorgente. L'ultimo ciclo controlla se ci sono cicli di peso negativo, in tal caso ritorna FALSE

Tempo di esecuzione

$$O(|V| * |E|)$$

3.5 Strutture dati per insiemi disgiunti

Alcune applicazioni richiedono di raggruppare n elementi distinti in una collezione di insiemi disgiunti. Queste applicazioni spesso richiedono l'esecuzione di due particolari operazioni: trovare l'unico insieme che contiene un determinato elemento e unire due insiemi.

Definizione Una struttura dati per insiemi disgiunti serve a *mantenere una collezione* $S = \{S_1, S_2, ..., S_k\}$ *di insiemi disgiunti.* Ogni insieme S_i è individuato da un rappresentante, che è un elemento dell'insieme.

Operazioni Ogni elemento di un insieme è rappresentato da un oggetto. Quindi, indicando con x un oggetto, supportiamo le segenti operazioni:

- **Make-set(x)**, che crea un nuovo insieme (e lo aggiunge alla struttura dati) il cui unico elemento e rappresentante è x, che non può trovarsi in un altro insieme.
- **Union(x,y)** unisce gli insiemi dinamici che contengono x e y in un unico insieme.
- **Find-set(x)** restituisce un puntatore al rappresentenate dell'insieme (unico) che contiene x.

Applicazioni delle strutture dati per insiemi disgiunti

3.6 Alberi di Copertura Minimi

Alberi di Copertura Sia $G = (V, E)$ un grafo non orientato e connesso. Si definisce albero ricoprente di G un sottografo $T \subseteq G$ Tale che:

- T è un albero;
- T contiene tutti i vertici di G

Ovvero Un albero di copertura di un grafo è un albero con gli stessi vertici di G, quindi rimane la possibilità di avere un cammino che porta da ogni nodo a ogni nodo ma vengono tolti gli archi "ridondanti".

Peso di un albero di copertura Se G è pesato, il peso del suo albero di copertura è dato dalla somma dei costi degli archi contenuti nell'albero.
Un albero di copertura minimo è un albero di copertura il cui peso totale è il minimo

Capitolo 4

NP Completezza

4.1 Cenni Teorici

Lo studio della NP-Completezza inizia formalizzando il concetto dei problemi risolvibili in tempo polinomiale. In primo luogo, sebbene sia ragionevole considerare intrattabile un problema che richiede $\theta(n^{100})$, ci sono pochissimi problemi nella pratica che richiedono un tempo polinomiale così alto. Una volta che viene scoperto un algoritmo con un tempo polinomiale, spesso seguono altri alogritmi più efficienti. In secondo luogo, per molti modelli di calcolo, un problema che può essere risolto in tempo polinomiale in un modello può essere riolsto sempre in tempo polinomiale in un altro modello. In terzo luogo, la classe dei problemi risolvibili in tempo polinomiale ha delle interessanti proprietà in chiusura, in quanto i polinomi sono chiusi rispetto all'addizione, alla moltiplicazione e alla composizione.

Problema di decisione Un problema di decisione è un problema tale per cui la risposta è *si o no*.

Classe di complessità Nella teroia della complessità computazionale, una classe di complessità è un insieme di problemi di una certa complessità. Un esempio tipico di definizione di classe di complessità a la forma: L'insieme di problemi che, se esistem la soluzione, possono essere risolti d una macchina astratta M usando $O(f(n))$ della risorsa R, con n dimensione dell'input

4.2 P

P è una classe di complessità che rappresenta l'insieme di tutti i problemi di decisione che possono essere risolti in tempo polinomiale, ovvero quei pro-

blemi tali che dato un input, riescono a dare un output si o no in tempo polinomiale. I problemi P sono i problemi che possono essere risolti da una macchina di Turing deterministica in tempo polinomiale.

4.3 NP

NP è una classe di complessità che rappresenta l'insieme di tutti i problemi di decisione per i quali le istanze che danno si come risposta possono essere verificate in tempo polinomiale. Ovvero: se la Sibilla Cumana mi dice che, dato un input x, la risposta è si, posso verificare la correttezza dell'affermazione in tempo polinomiale. I problemi NP sono i problemi che possono essere risolti da una macchina di Turing non-deterministica in tempo polinomiale.

4.4 NP-Completo

NP-completo è una classe di complessità che rappresenta l'insieme di tutti i problemi $x \in NP$ tali che è possibile una riduzione da un qualsiasi altro problema $y \in NP$ a x in tempo polinomiale. Intuitivamente, possiamo risolvere y velocemente se sappiamo risolvere x velocemente.

4.5 NP-Hard

Un problema x è NP-Hard se e solo se esiste un problema y NP-completo tale che y è riducibile a x in tempo polinomiale. Intuitivamente questi sono i problemi che sono almeno difficili quanto i problemi NP-completi. I problemi NP-hard non necessariamente sono in NP e non necessariamente sono problemi di decisione. I problemi NP-hard e NP sono i problemi NP-completi. Non tutti i problemi NP-hard sono NP-completi: affinchè un problema NP-hard x sia NP-complete, x deve essere in NP. è importante notare che poichè un qualsiasi problema NP-completo può essere ridottoa a qualsiasi altro problema NP-completo in tempo polinomiale e tutti i problemi in NP sono riducibili in tempo polinomiale a problemi NP-completi. Tutti i problemi NP possono essere ridotti a un qualsiasi problema NP-hard in tempo polinomiale. La conseguenza di queste osservaizoni è che se esiste una soluzione in tempo polinomiale a un qualsiasi problema NP-hard, allora esiste una soluzione a tutti i problemi NP in tempo polinomiale!