Laboratorio di Algoritmi

Damiano Trovato

Primo semestre, anno 2024/25

Indice

Capitolo 1

Introduzione all'analisi degli algoritmi

1.1 Studiare un algoritmo

Analizzare le performance di un algoritmo, significa andare a misurare caratteristiche quali:

- Uso di memoria
- Tempo computazionale
- Larghezza della banda di comunicazione
- Energia consumata

Lo studio che più ci interesserà è quello relativo al tempo computazionale. Un'altro studio particolarmente interessante, è quello relativo alla complessità spaziale.

1.1.1 Modello Random Access Machine

Nel modello RAM, gli algoritmi sono formalizzati come programmi scritti in pseudocodice (in maniera indipendente da specifici linguaggi).

1.1.2 Caratteristiche del modello RAM

- Le operazioni avvengono in maniera sequenziale, supponendo l'esistenza di una sola CPU, senza operazioni simultanee.
- Ogni singola istruzione elementare e ogni accesso alla memoria avviene in tempo costante.

1.1.3 Istruzioni elementari del modello RAM

- Operazioni aritmetiche (+, -, *, /, %, floor, ceiling)
- Spostamento di dati (upload, save, copy)
- Controllo (Branching condizionale, incondizionale, chiamate a subroutine e return)

1.1.4 Formato dei Dati

I dati possono essere di tipo intero, floating point o caratteri. Ogni stringa di dati può essere codificata da un numero limitato di bit.
Inoltre, nel modello RAM non esiste la gerarchia delle memorie.
La modalità di misura dell'input dipende dal problema e dalle strutture dati utilizzate. Ad esempio, nel caso degli array utilizzeremo la dimensione. Con un albero, il numero di nodi, con i grafi il numero di nodi e il numero di archi.

1.2 Calcolo del tempo computazionale

Detto ciò, diremo che il tempo computazionale è il numero di istruzioni elementari e di accessi ai dati eseguiti, in funzione della misura dell'input.
Prendiamo l'algoritmo di ordinamento Insertion Sort. *(In pseudo codice gli array si indicizzano da 1)*

```
1.  for i = 2 to n
2.      key = A[i]
3.      j = i - 1
4.      while j > 0 and A[j] > key
5.          A[j+1] = A[J]
6.          j = j - 1
7.      A[j+1] = key
```

Analizziamo il numero di ripetizioni di ogni costo, in funzione di n.

$$c_1 : n$$

$$c_2 : n-1$$

$$c_3 : n-1$$

$$c_4 : \sum_{i=2}^{n} t_i$$

$$c_5 : \sum_{i=2}^{n} (t_i - 1)$$

$$c_6 : \sum_{i=2}^{n} (t_i - 1)$$

$$c_7 : n-1$$

Studiamo ora i due casi notevoli di questo algoritmo, ovvero il best case e il worst case.

Best Case

Nell'esempio citato in precedenza, nel best case l'input è già ordinato.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1), c_4(n-1), c_5 * 0, c_6 * 0, c_7 * (n-1)$$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_5 + c_6 + c_7)n + (-c_2 - c_3 - c_4 - c_5 - c_6 - c - 7)$$

$$T(n) = an + b$$

Otteniamo quindi un tempo lineare nel best case.

Worst Case

Supponiamo l'input sia inversamente ordinato.
Poniamo $ti = i$ con $i \in \{2, 3, ..., n-1, n\}$.

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} i + c_5 \sum_{i=2}^{n} (i-1) + c_6 \sum_{i=2}^{n} (i-1), c_7(n-1)$$

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(\frac{n(n+1)}{2} - 1) + c_5(\frac{n(n+1)}{2}) + c_6(\frac{n(n+1)}{2}), c_7(n-1)$$

$$T(n) = (\frac{c_4 + c_5 + c_6}{2})n^2 + (c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7)n + (-c_2 - c_3 - c_4 - c_7)$$

$$T(n) = an^2 + bn + c$$

Otteniamo quindi un tempo quadratico.
Un algoritmo si dirà più efficiente di un altro se il runtime sul caso peggiore cresce più lentamente al crescere della dimensione dell'input.

1.2.1 Notazione Asintotica

Per esplicitare la crescita asintotica di un algoritmo, esistono varie notazioni.

- **Big-O notation:**
  Denota un limite superiore (upper bound) al comportamento asintotico di una funzione. Esempio:

  $$f(n) = 7n^3 + 10n^2 - 20n + 6$$

  $f(n)$ non crescerà più velocemente di $8n^3$, che rappresenterà un limite superiore ad $f(n)$, e quindi:

  $$f(n) = O(n^3)$$

  . In generale, diremo che

  $$O(g(n)) = \{f(n) | \exists c \in Q > 0, \exists n_0 \in N : \forall n \geq n_0, 0 \leq f(n) \leq g(n)\}$$

- **Ω notation:**
  Denota un limite inferire (lower bound) al comportamento asintotico di una funzione. Esempio:

  $$f(n) = 7n^3 + 10n^2 - 20n + 6$$

  $f(n)$ non crescerà meno velocemente di $8n^3$, che rappresenterà un limite superiore ad $f(n)$, e quindi:

  $$f(n) = O(n^3)$$

  In generale, diremo che

  $$\Omega(g(n)) = \{f(n) | \exists c \in Q > 0, \exists n_0 \in N : \forall n \geq n_0, 0 \leq c * g(n) \leq f(n)$$

- Θ **notation:**
  Denota limiti (superiori e inferiori, tight bound). Ad esempio:

  $$f(n) = 7n^3 + 10n^2 - 20n + 6$$

  $\Theta(n^3)$ In generale diremo che:

  $\Theta(g(n)) = \{f(n) | \exists c_1, c_2 \in Q > 0, \exists n_0 \in N : \forall n \geq n_0, 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

  Inoltre, se $f(n) = \Omega(g(n)) = O(g(n)) \Rightarrow f(n) = \Theta(g(n))$

Bound non stretti

- **Little-o notation:** Denota un limite superiore (upper bound) non asintoticamente stretto. Definizione:

  $$o(g(n)) = \{f(n) | \forall c \in Q > 0, \exists n_0 \in N : \forall n \geq n_0, 0 <= f(n) < c * g(n)\}$$

  [Nota bene, $g(n)$ non è più maggiore o uguale, ma sempre maggiore.]

- **Little-ω notation:**
  Denota un limite inferiore (lower bound) non
  asintoticamente stretto. Definizione:

  $$\omega(g(n)) = \{f(n) | \forall c \in Q > 0, \exists n_0 \in N : \forall n \geq n_0, 0 <= c * g(n) < f(n)\}$$

Capitolo 2

Equazioni di ricorrenza

2.1 Cos'è il divide et impera?

L'approccio "divide et impera" (o "divide and conquer") è un paradigma di risoluzione ai problemi suddivisibili in sottoproblemi, ed è alla base di molte delle soluzioni ricorsive che studieremo all'interno del corso.
All'interno di un problema ricorsivo, risolto seguendo il divide et impera, effettueremo delle divisioni dei problemi in sottoproblemi più piccoli. Il sottoproblema più piccolo (e atomico) si dirà caso base.

2.2 Introduzione alle equazioni di ricorrenza

Un'equazione di ricorrenza descrive il tempo di esecuzione di un algoritmo. Convenzionalmente, la indichiamo con $T(n)$. Supponiamo che l'algoritmo ricorsivo considerato divida il problema di dimensione n in a sottoproblemi di dimensione $\frac{n}{b}$.
Quindi, a è il numero di sottoproblemi, n/b è la dimensione dei sottoproblemi in cui verrà suddiviso il problema.

$$T(n) = \begin{cases} D(n) + aT(\frac{n}{b}) + C(n) \ \ \forall n \geq n_0 \\ \Theta(1) \ \ \forall n < n_0 \end{cases}$$

In cui: $D(n)$ = complessità divide e $C(n)$ = complessità combine, T la ratio del sottoproblema. In alcuni casi la formula di ricorrenza è una disuguaglianza, che ci permette di individuare upper bound e lower bound.

$$\leq \ \rightarrow \ \textit{upper bound big} - O$$

$$\geq \ \rightarrow \ \text{lower bound } \Omega$$

La complessità del caso base non è sempre esplicitata: in tal caso presumeremo che sia $\Theta(1)$.

2.2.1 Metodi per calcolare le equazioni di ricorrenza

Esistono quattro diversi approcci al calcolo della complessità di un algoritmo ricorsivo.

- Metodo di Sostituzione
- Metodo dell'albero di Ricorsione
- Metodo Master
- Metodo Akra-Bazzi (che non tratteremo)

2.3 Metodo dell'albero di ricorsione

Scomponiamo il costo dei sottoproblemi ad ogni chiamata ricorsiva usando un albero, che chiameremo albero di ricorsione.

2.3.1 Esempio su Merge Sort

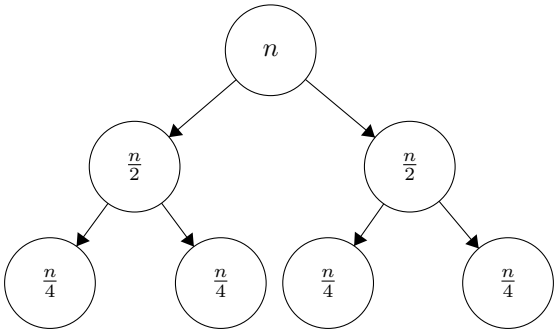
Prendiamo come caso studio il Merge Sort

- Divide:
  Individuo la metà dell'array, complessità $\Theta(1)$
- Conquer:
  Risolve $2 = a$ sottoproblemi di dimensione $\frac{n}{2}$ $(b = 2)$
- Combine:
  Merge, $C(n) = \Theta(n)$

$$T(n) = \Theta(1) + 2T(\frac{n}{2}) + \Theta(n) = 2T(\frac{n}{2}) + \Theta(n)$$

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + \Theta(n) \ \ \forall n \geq 2 \\ \Theta(1) \ \ n = 1 \textit{(Caso base)} \end{cases}$$

Costruiamo un albero che ci permette di denotare la decrescita della dimensione dei problemi.



La complessità sarà data dalla seguente somma:

$$T(n) = n \sum_{k=0}^{log_2 n} 1 \Leftrightarrow T(n) = nlog_2 n$$

Concludiamo che:

$$T(n) = O(nlogn)$$

2.4 Metodo di Sostituzione

- **Guess.**
  Ipotizzare un possibile limite asintotico all'equazione di ricorrrenza.
- **Induzione matematica.**
  Sostituire la presunta soluzione nella ricorrenza e dimostrare il limite asintotico per induzione.

2.4.1 Esempio sulla ricerca binaria

1. La relazione di ricorrenza della ricerca binaria è la seguente:

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \Theta(n)$$

2. Guess. Ipotizziamo che:

$$T(n) = O(n \log n) \Leftrightarrow T(n) \leq c \cdot \log n)$$

Il caso base sarà, per $n < n_0$, $T(n) = \Theta(1)$.
Dobbiamo dimostrare che esistono costanti positive c e n_0 tali che;

$$0 \leq T(n) \leq cn \log n$$

Dimostrazione ipotesi induttiva

Assumiamo che

$$T(\lfloor \frac{n}{2} \rfloor) \leq c \lfloor \frac{n}{2} \rfloor \log(\lfloor \frac{n}{2} \rfloor)$$

per $n \geq n_0$, $\frac{n}{2} \geq n_0$.
Sostituiamo

$$\begin{aligned} T(n) &= 2T(\lfloor \frac{n}{2} \rfloor) + dn \\ &\leq 2(c(\lfloor \frac{n}{2} \rfloor \log \lfloor \frac{n}{2} \rfloor) + dn \\ &\leq \cancel{2} c \frac{n}{\cancel{2}} \log \frac{n}{2} + dn \\ &= cn(\log n - \log 2) + dn \\ &= cn(\log n - 1) + dn \\ &= cn \log n - cn + dn \leq cn \log n \end{aligned} \tag{2.1}$$

$\forall c \geq d$

Caso base

$n_0 = 1 : c \cdot n_0 \log n_0 = 0$, quindi il caso base sarà
$n_0 = 2$.
Base dell'induzione $T(n) \leq cn \log n$

NOTA DA DAM

Work in progress, non credo di aver afferrato appieno i concetti relativi al metodo di sostituzione applicato per la dimostrazione della complessità asintotica delle equazioni di ricorrenza.
Detto ciò, è probabile che approfondirò questa parte appena ne avrò la possibilità, ma osservando lo storico dei compiti d'esame di algoritmi, ho preferito dare priorità ad argomenti che usualmente escono nelle prove. Il metodo master è quello su cui è fondamentale portare la nostra attenzione!

2.5 Metodo Master (o Principale)

2.5.1 Introduzione

Il metodo Master è un metodo applicabile a tutte le ricorrenze che si presentano nella forma Master, ovvero:

$$T(n) = aT(\frac{n}{b}) + f(n);$$

$$con\ a > 0,\ b > 1\ costanti$$

Possiamo considerare la forma Master anche nei casi in cui n non è divisibile per b, senza dover considerare floor e ceiling.

$$a'T(\lfloor\frac{n}{b}\rfloor + a''T(\lceil\frac{n}{b}\rceil),\ \ \exists a', a'' > 0 \Leftrightarrow a' + a'' = a$$

Per comodità andrò a definire un'ulteriore funzione, chiamata *watershed function*, che indicherò con $w(n)$, e che sarà uguale a:

$$w(n) = n^{log_b a}$$

Inoltre, la $f(n)$ è anche chiamata *driving function*.

2.5.2 Enunciato

Siano $a > 0$, $b > 1$ costanti, sia $f(n)$ una funzione definita e non negativa su tutti i numeri reali sufficientemente grandi e sia:

$$T(n) = aT(\frac{n}{b}) + f(n)$$

Allora, potrò procedere in tre modi:

1. Se $w(n)$ è di ordine superiore a $f(n)$, con una separazione polinomiale:

   $$\exists \varepsilon > 0 : f(n) = O(n^{log_b a - \varepsilon}) \Rightarrow T(n) = \Theta(n^{log_b a})$$

2. Se $w(n)$ è di ordine uguale a $f(n)$, o c'è una separazione polilogaritmica : *Formula Generale:*

   $$\exists k \geq 0 : f(n) = \Theta(n^{log_b a} log^k n) \Rightarrow T(n) = \Theta(n^{log_b a} log^{k+1} n)$$

   Formula specifica se non si presenta il logaritmo (k=0):

   $$f(n) = \Theta(n^{log_b a}) \Rightarrow T(n) = \Theta(n^{log_b a} log n)$$

3. Se $w(n)$ è di ordine inferiore a $f(n)$, con una separazione polinomiale:

   $$\exists \varepsilon > 0 : f(n) = \Omega(n^{log_b a + \varepsilon})$$

   e soddisfa la condizione di regolarità: $(\exists c < 1 : af(\frac{n}{b}) \leq cf(n))$

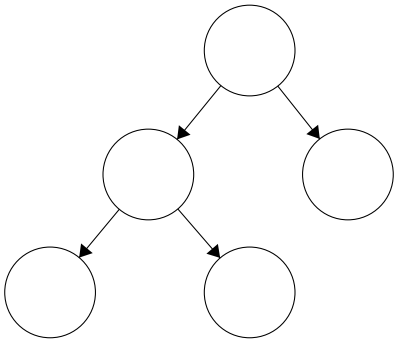
   $$\Rightarrow T(n) = \Theta(f(n))$$

Capitolo 3

Studio dell'heap

3.1 Heapify

Il tempo necessario a correggere la relazione tra A[i], A[i.left], A[i.right] è $\Theta(1)$. L'albero di ricorrenza potrebbe non essere completo, quindi lo consideriamo sempre come caso peggiore.



E diremo:

$$T(n) \leq T(\frac{2}{3}n) + \Theta(1)$$

Dove n è il numero di nodi (dimensione del problema).
Nel caso peggiore, in cui l'albero non è completo, il sottoalbero sinistro ha dimensione massima, e sottoalbero destro minima, il sottoalbero sinistro sarà di dimensione $\frac{2}{3}n$, dove n è il numero di nodi dell'intero albero. Essendo questo il caso peggiore, la nostra relazione di riccorrenza sarà una disequazione $\leq$
L'altezza del nodo a cui i sottoalberi sono radicati è $logn$.

$$a = 1,\ b = \frac{3}{2}, f(n) = \Theta(1), w(n) = n^{log_{\frac{3}{2}} 1} = n^0 = 1$$

Quindi $w(n) = f(n)$, siamo nel caso 2, conseguentemente:

$$f(n) = \Theta(1 \cdot log^0 n) \rightarrow_{Caso\ 2} T(n) = O(logn)$$

OSSERVAZIONE:
Eseguire heapify su un sottoalbero di altezza h impiega $O(h) = c \cdot h$.
Non a caso $h = log_2 n$. Concludiamo che l'heapify sarà: $T(n) = O(logn)$, confermando il risultato del teorema master.

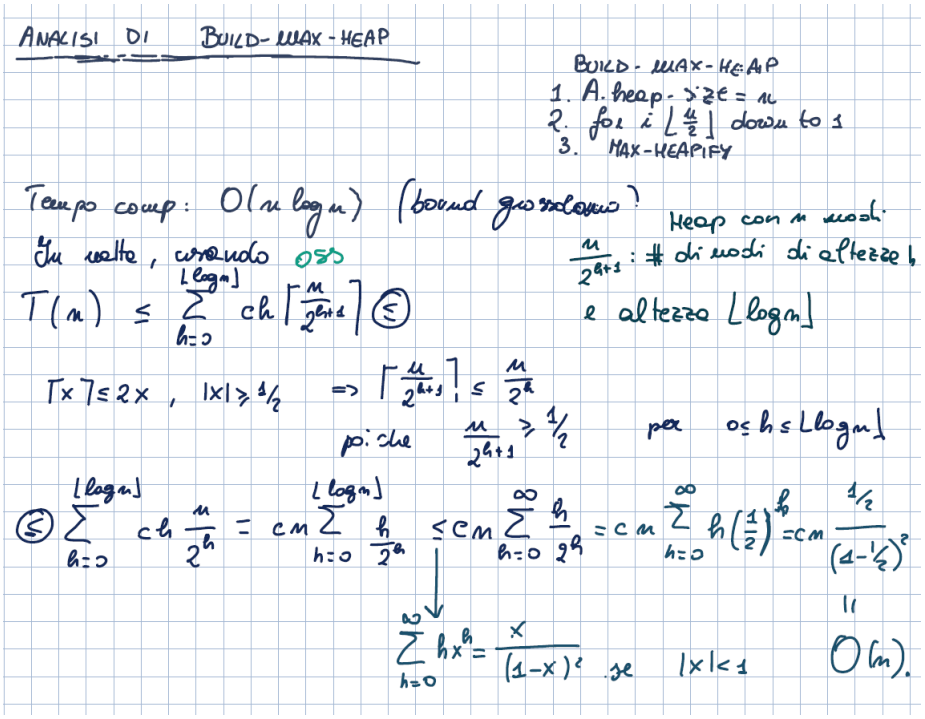
3.1.1 Build-Max-Heap

Stimiamo che abbia come relazione di ricorrenza:

$$T(n) = O(n) \cdot O(logn) \Rightarrow T(n) = O(nlogn)$$

Tuttavia, una stima migliore è la seguente:

$$T(n) \leq \sum_{h=0}^{\lfloor logn \rfloor} c \cdot h\lceil \frac{n}{2^{h+1}} \rceil$$



Capitolo 4

Tabelle Hash

4.1 Introduzione alle tabelle hash

Una tabella hash è una struttura dati efficace nell'implementare dizionari. Sono infatti predisposte per effettuare operazioni di inserimento, ricerca e cancellazione.
Solitamente, viene implementato con l'ausilio di un array di dimensione proporzionale al numero di chiavi da memorizzare, spesso relativamente piccolo rispetto al numero di chiavi possibili $|U|$. L'efficenza delle tabelle hash è legata all'uso di opportune funzioni hash.

4.1.1 Notazione

U: Insieme Universo di chiavi.
$K \subseteq U$: Insieme di chiavi memorizzate, $|K| = n$.
m dimensione della tabella (numero di slot, solitamente minore di $|U|$. La funzione hash mappa l'universo U delle chiavi negli slot di una tabella $T[0, 1, \ldots, m-1]$. Quindi, la funzione hash, è definita così:

$$h : U \to \{0, ..., m-1\}$$

$$h(k)\ hash\ di\ k,\ quindi :$$

$$k \to h(k)$$

4.2 Collisioni risolte con concatenazione

4.2.1 Collisioni

Quando due chiavi k_1 e k_2 hanno $h(k_1) = h(k_2)$, si dice che k_1 e k_2 collidono. Quando $n > m$, non può non avvenire una collisione. In questa sezione parleremo di una tra le due principali soluzioni al problema delle collisioni.

4.2.2 Concatenazione

L'insieme delle chiavi è suddiviso in sottoinsiemi di dimensioni $\frac{n}{m}$. La funzione hash determina a quale sottoinsieme appartiene una chiave. Ogni sottoinsieme è implementato come una lista concatenata.
Una funzione hash **ben costruita** contribuisce a evitare le collisioni.

4.2.3 Funzione hash uniforme e indipendente

Una funzione hash uniforme e indipendente è tale che $\forall k \in U$, $h(k)$ sia scelto in maniera uniforme e indipendente da $\{0, 1, \ldots, m-1\}$.
Formalizzando i due concetti, un hashing si dirà uniforme se:

$$\forall k \in U,\ \forall J \in \{0, \ldots, m-1\}\ ,\ Pr[h(k) = j] = \frac{1}{m};$$

Dove $Pr[evento]$ indica la probabilità che un evento avvenga.
Un hashing si dirà indipendente se:

$$\forall k_1, k_2 \in U, con\ , k_1 \neq k_2, \forall j_1, j_2 \in \{0, \ldots, m-1\}$$

$$Pr[h(k_1) = j_1 | h(k_2) = j_2] = Pr[h(k_1) = j_1]$$

o equivalentemente

$$Pr[h(k_1) = j_1 \wedge h(k_2) = j_2] = Pr[h(k_1) = j_1] \cdot Pr[h(k_2) = j_2]$$

Se sono rispettate queste proprietà, è possibile parlare di una funzione di hash indipendente e uniforme, detta anche *random oracle*.
Per l'analisi degli algoritmi di ricerca nelle tabelle hash assumeremo hashing uniforme e indipendente.

Fattore di carico

È detto fattore di carico α il numero medio di chiavi memorizzate in uno slot (o la lunghezza media delle liste), ed è pari a

$$\alpha = \frac{n}{m}$$

In una tabella hash con concatenazione, il valore di α può essere minore, uguale o maggiore di zero. Ciò non succede nelle tabelle a indirizzamento aperto, dove $\alpha \leq 1$.

4.3 Complessità hash con concatenazione

4.3.1 Ricerca senza Successo

La complessità della ricerca senza successo in una tabella hash con concatenazione, è $\Theta(1+\alpha)$. Per ricerca senza successo, intendiamo i seguenti passaggi:

1. Accesso alla lista di indice $h(k)$, ovvero accesso a $T[h(k)]$
2. Scorrimento dell'intera lista di posizione $h(k)$ e lunghezza media (per ipotesi di hashing uniforme e indipendente) pari a $\alpha = \frac{n}{m}$

Essendo una ricerca senza successo, saranno visitati tutti i nodi della lista.

Valore atteso

Espongo brevemente il concetto di valore atteso in funzione delle prossime dimostrazioni.
X è una variabile aleatoria: assume valori diversi in dipendenza da un determinato fenomeno non-deterministico (casuale).
$x_i \in X$ è un fenomeno di X, l'i-esimo evento.
$Pr[x_i] = p_i$ è la probabilità dell'i-esimo evento.
La formula del valore atteso, è la seguente:

$$\mathbb{E}[X] = \sum_{i=0}^{\infty} x_i p_i$$

È quindi ottenuto dalla somma dei prodotti tra tutti i valori di X per le rispettive probabilità.

Dimostrazione

$Tempo$ sarà uguale al tempo usato calcolare $h(k)$ e accedere a $T[h(k)] + n_{h(k)}$.
$\mathbf{E}[Tempo]$ indica il valore atteso di $Tempo$.

$$\mathbb{E}[Tempo] = \Theta(1) + \mathbb{E}[n_{h(k)}] = \Theta(1) + \sum_{j=0}^{m-1} Pr[h(k) = j] \cdot n_j =$$

$$= \Theta(1) + \sum_{j=0}^{m-1} \frac{1}{m} \cdot n_j = \Theta(1) + \frac{1}{m} \sum_{j=0}^{m-1} n_j = \Theta(1)\frac{n}{m} = \Theta(1+\alpha)$$

4.3.2 Tempo ricerca con successo

Per l'analisi della ricerca con successo, supporremo che l'inserimento degli elementi nelle liste sia sempre in testa.
La complessità della ricerca sarà uguale al tempo usato calcolare $h(k)$ e accedere a $T[h(k)]$ + numero di elementi che hanno lo stesso hash dell'elemento cercato e che sono stati inseriti dopo (ovvero tutti gli elementi da scorrere prima di trovare l'elemento d'interesse).
Chiameremo l'elemento in questione n_i, gli elementi inclusi tra n_1 e n_{i-1} saranno gli elementi inseriti dopo.
La nostra dimostrazione usufruirà di tre variabili aleatorie.

Dimostrazione

Definiamo:
$\forall q \in \{0, \dots, m-1\}, \quad \forall k_i, k_j \in K \quad con \quad k_i \neq k_j$

$$X_{ijq} = \begin{cases} 1 \text{ se stiamo cercando } x_i,\ h(k_i) = q,\ h(k_j) = q \\ 0 \text{ altrimenti} \end{cases}$$

Calcoliamo il valore atteso di X_{ijq}.
Ricordiamo che il valore atteso è dato dalla somma di tutti i casi moltiplicati per le rispettive probabilità.

$$\mathbb{E}[X_{ijq}] = 0 \cdot Pr[X_{ijq} = 0] + 1 \cdot Pr[X_{ijq} = 1] = Pr[X_{ijq} = 1]$$

I tre eventi per cui $X_{ijq} = 1$ sono eventi indipendenti: la probabilità complessiva è data dal prodotto delle probabilità.

$$Pr[X_i jq] = Pr[Stiamo\ cercando\ x_i] Pr[h(k_i) = q] Pr[h(k_j) = q] = \frac{1}{n}\frac{1}{m}\frac{1}{m} = \frac{1}{nm^2}$$

Bene, adessso introduciamo Y_j.

$$Y_j = \begin{cases} 1 \text{ se } x_j \text{ appare nella lista prima dell'elemento cercato} \\ 0 \text{ altrimenti} \end{cases}$$

$$\mathbb{E}[Y_j] = \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} X_{ijq}$$

Ultima variabile, la Z.

$$Z = \sum_{j=1}^{n} Y_j$$

Adesso possiamo procedere a calcolare il valore atteso della ricerca con successo.

$$
\begin{aligned}
\mathbb{E}[Tempo] &= \mathbb{E}[Z+1] \\
&= 1 + \mathbb{E}[Z] \\
&= 1 + \sum_{j=1}^{n} \mathbb{E}[Y_j] \\
&= 1 + \sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} \mathbb{E}[X_{ijq}] \\
&= 1 + \sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} \frac{1}{nm^2} \\
&= 1 + \frac{1}{nm^2} \sum_{j=1}^{n} \sum_{q=0}^{m-1} \sum_{i=1}^{j-1} 1 \\
&= 1 + \frac{1}{nm^2} \sum_{q=0}^{m-1} \sum_{j=1}^{n} j - 1 \\
&= 1 + \frac{1}{nm^2} \sum_{q=0}^{m-1} \sum_{j=0}^{n} j \qquad (4.1) \\
&= 1 + \frac{1}{nm^2} \sum_{q=0}^{m-1} \frac{n(n-1)}{2} \\
&= 1 + \frac{1}{nm^2} \frac{n(n-1)}{2} \sum_{q=0}^{m-1} 1 \\
&= 1 + m \frac{1}{nm^2} \frac{n(n-1)}{2} \\
&= 1 + \frac{1}{m} \frac{(n-1)}{2} \\
&= 1 + \frac{(n-1)}{2m} \\
&= 1 + \frac{1}{2}(\frac{n}{m} - \frac{1}{m}) \\
&= 1 + \frac{1}{2}(\alpha - \frac{\alpha}{n}) = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n}
\end{aligned}
$$

Concluderemo che la complessità della ricerca con successo è $\Theta(1 + \frac{\alpha}{2} - \frac{\alpha}{2n}) = \Theta(1 + \alpha)$.

4.4 Tabelle a indirizzamento aperto

Le tabelle a indirizzamento aperto risolvono il problema delle collisioni in maniera diversa. Ogni slot conterrà un elemento dell'insieme dinamico k o $NULL$.
Una volta riempita una tabella a indirizzamento aperto, essa non potrà più contenenere altri elementi, ottenendo il suddetto fattore di carico minore o uguale a 1.

Come funziona una Tabella a indirizzamento aperto?

Quando un elemento deve essere inserito nella tabella, esso viene allocato nella posizione "prima scelta", $h(k, 1)$. Se essa risulta occupata, si passa alla posizione $h(k, 2), h(k, 3), \ldots, h(k, m)$. Se non si trova uno slot vuoto, si suppone la tabella sia piena. Questo perché la funzione di hashing $h(k, 1), h(k, 2), \ldots, h(k, m)$, andrà a generare, uno ad uno, tutti gli indici della tabella, e quindi una permutazione di tutti i numeri da 1 a m.
Sappiamo già che il numero di permutazioni di numeri che vanno da 1 a m è pari a $m!$. Indichiamo una permutazione con σ, e quindi:

$$h : K \to \{\sigma_1, \ldots, \sigma_{m!}\}.$$

4.4.1 Hashing di permutazione uniforme e indipendente

Un hashing di permutazione uniforme e indipendente è tale che $\forall k \in U$, $h(k)$ sia scelto in maniera uniforme e indipendente dall'insieme $\{\sigma_1, \ldots, \sigma_{m!}\}$.
Un hashing di permutazione si dirà uniforme se:

$$\forall k \in U, \forall \sigma \in \{\sigma_1, \ldots, \sigma_{m!}\} \ , \ Pr[h(k) = \sigma] = \frac{1}{m!};$$

Un hashing di permutazione si dirà indipendente se:

$$\forall k_1, k_2 \in U \ , \ k_1 \neq k_2 \ , \ \forall \sigma \in \{\sigma_1, \ldots, \sigma_{m!}\}$$

$$Pr[h(k_1) = \sigma_1 | h(k_2) = \sigma_2] = Pr[h(k_1) = \sigma_1]$$

o equivalentemente

$$Pr[h(k_1) = \sigma_1 \wedge h(k_2) = \sigma_2] = Pr[h(k_1) = \sigma_1] \cdot Pr[h(k_2) = \sigma_2]$$

4.5 Complessità hash indirizzamento aperto

In una tabella hash a indirizzamento aperto, ogni slot della tabella può contenere un elemento dell'insieme U, o essere vuoto.

4.5.1 Fattore di carico

Il fattore di carico sarà sempre minore o uguale a 1.

$$n \leq m \Rightarrow \alpha = \frac{n}{m} \leq 1$$

Tuttavia, nelle seguenti analisi, presumeremo che $n < m \Rightarrow \alpha < 1$, e che quindi almeno uno slot sia vuoto.

4.5.2 Analisi ricerca senza successo

Dimostriamo che la complessità della ricerca senza successo è $O(\frac{1}{1-\alpha})$, ovvero al più il numero di prove degli slot.

Dimostrazione

Ogni prova incontra uno slot occupato, eccetto l'ultima.

$$T(n) = \Theta(1) + \mathbb{E}[X] = \mathbb{E}[X]$$

Dove X è il numero di prove in una ricerca senza successo $\Rightarrow \mathbb{E}[X]$ numero attes di prove.
Allora

$$\mathbb{E}[X] = \sum_{i=1}^{\infty} Pr[X \geq i]$$

Definiamo un evento A_i: "L'i-esima prova è avvenuta e ha incontrato uno slot occupato".

$$[X \geq i] = A_1 \cap A_2 \cap \cdots \cap A_{i-1}$$

$$\begin{aligned} Pr[X \geq i] =& Pr[A_1 \cap \cdots \cap A_{i-1}] \\ =& Pr[A_1] \cdot Pr[A_2|A_1] \cdot Pr[A_3|A_1 \cap A_2] \cdot \ldots \\ & \ldots \cdot Pr[A_{i-1}|A_1 \cap \cdots \cap A_{i-2}] \end{aligned} \tag{4.2}$$

Osserviamo che la probabilità di incontrare il primo slot occupato è:

$$Pr[A_1] = \frac{n}{m} = \alpha$$

Secondo slot (e il precedente) occupato

$$Pr[A_2|A_1] = \frac{n-1}{m-1} < \frac{n}{m} = \alpha$$

i-esimo slot occupato (e i precedenti)

$$Pr[A_{i-1}|A_1 \cap \cdots \cap A_{i-2}] = \frac{n-(i-2)}{m-(i-2)} < \frac{n}{m} = \alpha$$

Maggiorando tutte le probabilità col valore α, il prodotto tra tutte queste probabilità sarà uguale a α^{i-1}. In termini matematici:

$$\begin{aligned} Pr[X \geq i] &= \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-(i-2)}{m-(i-2)} \\ &\leq (\frac{n}{m})^{i-1} \\ &= \alpha^{i-1} \end{aligned} \tag{4.3}$$

Quando $i \leq n+1$.
Chiaramente, per $i > n+1$, la probabilità è uguale a 0. Questa osservazione apparentemente scontata ci servirà proprio nel calcolo del valore atteso.
Ritorniamo proprio su quest'ultimo.

$$\begin{aligned} \mathbb{E}[X] &= \sum_{i=1}^{\infty} Pr[X \geq i] \\ &= \sum_{i=1}^{n+1} Pr[X \geq i] + \sum_{n+2}^{\infty} Pr[X \geq i] \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha} \end{aligned} \tag{4.4}$$

Essendo $a < 1$, la serie convergerà a questo risultato.
Riporto per comodità la "formula" della serie geometrica convergente.

$$\sum_{n=0}^{\infty} q^n = \frac{1}{1-q}, \quad \forall q \in]-1, 1[$$

4.5.3 Corollario sull'inserimento

L'inserimento di un elemento in una tabella hash a indirizzamento aperto con un fattore di carico α, richiede in media non più di $\frac{1}{1-\alpha}$ ispezioni, nell'ipotesi di hashing uniforme.

Dimostrazione

L'inserimento viene effettuato solo se la tabella ha almeno uno slot libero, quindi $\alpha < 1$.
L'inserimento di una richiede una ricerca senza successo, seguita dalla sistemazione della chiave nella prima cella vuota che viene trovata.
Di conseguenza, il numero di ispezioni massimo è $\frac{1}{1-\alpha}$, e l'inserimento avrà valore atteso del tempo $O(\frac{1}{1-\alpha})$.

4.5.4 Analisi della ricerca con successo

Supponendo hashing indipendente e uniforme, e che nessuna cancellazione sia avvenuta, il numero atteso di prove in una tabella hash a indirizzamento aperto con un fattore di carico $\alpha < 1$, è al più $\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$, e quindi

$$\mathbb{E}[Time] = O(\frac{1}{\alpha} \ln \frac{1}{1-\alpha})$$

Dimostrazione

La ricerca in una chiave k produce la stessa sequenza di tentativi del suo inserimento.
Assumiamo che k sia la $(i+1)$-esima chiave inserita per qualche $i \in \{1, 2, \ldots, n\}$.
Quando k è stato inserito, il fattore di carico era $\tilde{\alpha} = \frac{i}{m} \Rightarrow$ il numero atteso di tentativi per l'inserimento di k è

$$\frac{1}{1-\tilde{\alpha}} = \frac{1}{1-\frac{i}{m}} = \frac{m}{m-i}$$

Detto ciò, il numero atteso di ispezioni durante una ricerca, è dato dalla probabilità che stiamo cercando la chiave k $(i+1)$-esima, per il valore atteso di

ispezioni, secondo il corollario precedente, e che abbiamo appena calcolato.

$$\begin{aligned}
\frac{1}{n}\sum_{i=0}^{n-1}\frac{m}{m-i} &= \frac{m}{n}\sum_{i=0}^{n-1}\frac{1}{m-i} \\
&= \frac{1}{\alpha}\sum_{k=m-n+1}^{m}\frac{1}{k} \\
&\leq \frac{1}{\alpha}\int_{m-n}^{m}\frac{1}{x}dx \\
&= \frac{1}{\alpha}(\ln m - \ln(m-n)) \\
&= \frac{1}{\alpha}\ln\frac{m}{m-n} \\
&= \frac{1}{\alpha}\ln\frac{1}{1-\alpha}
\end{aligned} \tag{4.5}$$

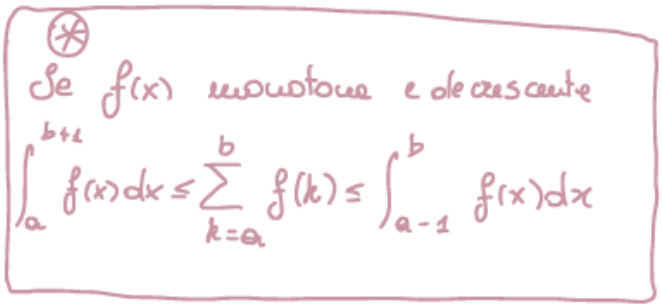


Figure 4.1: Approsssimazione integrale usata nella dimostrazione

Capitolo 5

Alberi rosso-neri

5.1 Analisi degli alberi rosso-neri

Avendo precedentemente introdotto questa struttura dati (appunti del professore Faro), andremo semplicemente a ricapitolarne le cinque proprietà.

5.1.1 Le cinque proprietà

1. Ogni nodo è rosso oppure nero
2. La radice è nera
3. Le foglie (NULL) sono nere
4. Se un nodo è rosso, i suoi figli sono neri
5. Per ogni nodo, ogni cammino semplice fino alle foglie ha un numero fisso di nodi neri (denotiamo questo numero con $bh(x)$, ovvero altezza nera di x)

5.2 Dimostrazioni

5.2.1 Claim - Nodi interni

Useremo il seguente claim per dimostrare il prossimo lemma.
Il sottoalbero radicato a un nodo x contiene almeno $2^{bh(x)} - 1$ nodi interni.

Dimostrazione del claim

Usiamo l'induzione sull'altezza $h(x)$ per dimostrare il claim.

- **Base dell'induzione.**
  $h(x) = 0 \Rightarrow x$ *è una foglia* $\Rightarrow$ *il sottoalbero radicato a* x *ha* 0 *nodi.*
  Inoltre $h(x) = 0 \Rightarrow bh(x) = 0$

- **Ipotesi induttiva.**
  Assumiamo il claim come vero per ogni nodo di altezza $< h(x)$ *con* $h(x) > 0$

- **Passo induttivo.**
  Siccome $h(x) > 0 \Rightarrow x$ ha dei figli.
  Se $x.child$ è

  $$\begin{cases} nero \Rightarrow bh(x) = bh(x.child) + 1 \\ rosso \Rightarrow bh(x) = bh(x.child) \end{cases} \Rightarrow bh(x.child) \geq bh(x) - 1$$

  Di conseguenza, $bh(x.child) \geq bh(x) - 1$.
  Inoltre, $h(x.child) < h(x)$, quindi possiamo applicare l'ipotesi induttiva a $x.child$.
  Il sottoalbero radicato a x contiene i sottoalberi radicati ai suoi figli, e quindi il numero di nodi interni è dato dalla somma dei nodi interni dei sottoalberi radicati al figlio sinistro di x e il figlio destro di $x + 1$.

  $$\begin{aligned} &\geq (2^{bh(x.left)} - 1) + (2^{bh(x.right)} - 1) + 1 \\ &\geq 2((2^{bh(x.child)} - 1) + 1 \\ &\geq 2((2^{bh(x)-1} - 1) + 1 \\ &= 2^{bh(x)-1+1} - 2 + 1 = 2^{bh(x)} - 1 \end{aligned} \tag{5.1}$$

5.2.2 Lemma 1

Un albero rosso-nero con n nodi interni ha altezza al più $2log(n+1)$, cioè $O(logn)$.

Dimostrazione

Sia h^* l'altezza dell'intero albero.
Vogliamo dimostrare che $h^* \leq 2\log(n+1)$, con n numero di nodi interni.
$h^* = h(root)$, quindi possiamo usare il claim con $x = root$.

1. $x \geq 2^{bh(root)} - 1$
2. Inoltre,
   $bh(root) \geq \frac{h^*}{2}$ (per la proprietà 4 degli alberi rosso neri).

$$\begin{aligned} &\Rightarrow n \geq 2^{\frac{h^*}{2}} - 1 \\ &\Rightarrow 2^{\frac{h^*}{2}} \leq n+1 \\ &\Rightarrow \frac{h^*}{2} \leq \log(n+1) \\ &\Rightarrow h^* \leq 2\log(n+1) \end{aligned} \tag{5.2}$$

Conseguenza immediata di questo lemma?
Le seguenti operazioni impegheranno tempo $O(\log n)$

- Ricerca
- Minimo nel tree
- Massimo nel tree
- Successore di un nodo
- Predecessore di un nodo

Capitolo 6

Problemi di ottimizzazione e programmazione dinamica

6.1 Cos'è un problema di ottimizzazione?

Un problema di ottimizzazione è un problema computazionale sulle cui variabili sono definite da una funzione o più funzioni di costo. L'obiettivo è trovare una soluzione che:

- Soddisfi i requisiti del problema
- Minimizzi/Massimizzi una funzione obiettivo, che è data dalla combinazione di funzioni di costo

6.2 Programmazione dinamica

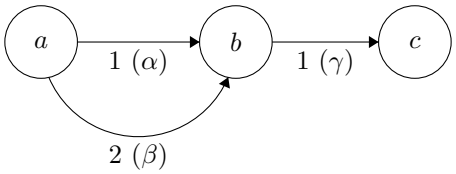
La programmazione dinamica è un paradigma di risoluzione ai problemi di ottimizzazione.
Per applicare la programmazione dinamica ad un problema di ottimizzazione, due requisiti devono essere soddisfsatti:

- **Proprietà di sottostruttura ottima.**
  Un problema di ottimizzazione ha questa proprietà se le restrizioni di una soluzione ottima ai sottoproblemi, sono soluzioni ottime per i sottoproblemi.
- **Overlapping subproblems.**
  Ovvero un numero polinomiale di sottoproblemi che si ripetono.

Come dimostrare la sottostruttura ottima?

Prima di andare ad applicare soluzioni di programmazione dinamica, sarà fondamentale andare a dimostrare la proprietà di sottostruttura ottima.

Dimostreremo questa proprietà, cercando di dimostrare un assurdo:
una soluzione ottima può "contenere" una soluzione non ottima ad un sotto-problema. Applichiamo questo ragionamento sul seguente problema.



Dobbiamo trovare il percorso di peso minore da a a c.
Supponiamo che $S^* = (\beta, \gamma)$ sia una soluzione ottima del problema. γ è evidentemente una soluzione ottima alla restrizione (b, c) del problema. Ciò non può essere detto invece della soluzione β su (a, b).
Se β non è ottima, esiste una soluzione α su (a, b) migliore di β. Esisterà anche una soluzione (α, γ) su (a, c) migliore di S^*, in quanto $\alpha < \beta$.
Incappiamo così in un assurdo! Siamo infatti partiti dall'ipotesi che S^* fosse una soluzione ottima.
L'assurdo deriva però dall'aver incluso al suo interno la soluzione non ottima β.
Concludiamo quindi affermando che una soluzione ottima è costituita solo da soluzioni ottime alle sue restrizioni.

6.3 Rod-Cutting problem

In questa sezione andremo ad aggiungere qualche informazione in più riguardo il Rod-Cutting problem, andando a esplicitare la complessità delle soluzioni e a dimostrarne la sottostruttura ottima.

6.3.1 Introduzione al problema

Abbiamo una barra di lunghezza n e una tabella di prezzi p_i con $i \in 1, 2, ..., n$ per i pezzi di lunghezza i.
L'obiettivo è quello di massimizzare il guadagno delle vendite sfruttando la migliore combinazione di tagli.

$$r_n = \max_{i_1,\dots,i_k} (p_{i1}, p_{i2}, \dots, p_{ik})$$

con $i_1 + \cdots + i_k = n$.

Definizione ricorsiva

$$r_n = \max\{p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{\lceil \frac{n}{2} \rceil} + r_{\lfloor \frac{n}{2} \rfloor}\}$$

Dove p_n è il prezzo di vendità dell'asta di lunghezza n senza tagli, mentre gli altri termini sono la somma del prezzo di vendita dei due pezzi di lunghezza i e $n - i$.
Questa definizione è ricorsiva perché ogni r_i e r_{n-1} è ottenuto dalla definizione di r_n, e quindi r_i e r_{n-1} saranno i prezzi di vendità massimi che possono essere ottenuti tagliando in maniera ottimale le due barre.
Si fa più chiara la sottostruttura ottima del nostro problema.

Formulazione ricorsiva migliore

$$r_n = \max\{p_i + r_{n-1} \mid 1 \leq i \leq n\},\ r_0 = 0$$

In questo modo, una soluzione ottima incorporerà solo la soluzione ottima di un sottoproblema, anziché due.

6.3.2 Sottostruttura ottima

$$r_n = \max\{p_i + r_{n-1} \mid 1 \leq i \leq n\},\ r_0 = 0$$

Fissiamo una certa i e sia r_n^* ottima $\Rightarrow r_n^* \geq r_n \forall$ soluzione r_n.
Sia $r_n^* = p_i + r_{n-i}^*$.
Supponiamo che r_{n-i}^* non sia ottima $\Rightarrow$ ne esiste una migliore $r'_{n-i} > r_{n-i}^*$.
Ma allora posso definire $r'_n = p_i + r'_{n-1} > p_i + r_{n-1}^* = r_n^*$, e ciò è un assurdo.
r_{n-1}^* deve essere ottima $\Rightarrow$ vale la proprietà di sottostruttura ottima.

Capitolo 7

Elementi di programmazione dinamica

7.1 Matrix Chain Multiplication

7.1.1 Descrizione

Input

una sequenza $\langle A_1, ..., A_n \rangle$ di matrici di dimensioni tali che il numero di colonne della matrice A_i deve essere uguale al numero di righe di A_{i+1} $\forall i = 1, ..., n-1$.

Output

Calcolare $A_1, ..., A_n$ minimizzando il numero di moltiplicazioni scalari.

Forma di una soluzione al problema

La soluzione è una parentesizzazione. La funzione di costo è la seguente:

$$costo(P) = \textit{ moltiplicazioni scalari della parentesizzazione}$$

7.1.2 Dimostrazione sottostruttura ottima

Proposizione

Il problema Matrix-Chain Multiplication ha la proprietà di sottostruttura ottima.

Dimostrazione

[N.B. Usiamo la lettera P per parlare di una parantesizzazione]
Sia P_{opt} una parantesizzazione ottima, ovvero tale che $costo(P_{opt} \leq costo(P)$ $\forall P$ *di* $A_i, ..., A_n$
Partiamo da una soluzione ottima.

$$P_{opt}(A_1, ..., A_n) = P_1(A_1, ..., A_{i-1})P_2(A_i, ..., A_n) \;\; \exists P_1, P_2, \; \exists i \in \{2, ..., n\}$$

Dove il costo di P_{opt} è definito come:

$$costo(P_{opt}) = costo(P_1) + costo(P_2) + h$$

Con h: numero di moltiplicazioni scalari per moltiplicare la matrice risultante da $A_1, ...A_{i-1}$ per quella risultate da $A_i, ..., A_n$.
Il valore di h non dipende da $P_!$ o da P_2, ma solo da i, che è fissato.
Restringiamo P_{opt} ai sottoproblemi $(A_1, ..., A_{i-1}) \rightarrow P_1$ e $(A_i, ..., A_n) \rightarrow P_2$.
Per mostrare che matrix chain multiplication ha la proprietà di sottostruttura ottima dobbiamo mostrare che:

- P_1 è una parentesizzazione ottima di $A_1, ..., A_{i-1}$
- P_2 è una parentesizzazione ottima di $A_i, ..., A_n$

Per dimostrare ciò, supponiamo per assurdo che P_1 non sia una parentesizzazione ottima di $A_1, ...A_{i-1}$, questo vuol dire che $\exists$ una parentesizzazione P_1' di $A_1, ...A_{i-1}$ che ha costo inferiore a quello di P_1.
Definisco allora la parentesizzazione $P*$

$$P*(A_1, ..., A_n) = P_1'(A_1, ..., A_{i-1})P_2(A_i, ..., A_n)$$

Cui costo sarà:

$$costo(P*) = costo(P_1') + cost(P_2) + h$$

$$costo(P_1') + cost(P_2) + h < costo(P_1) + costo(P_2) + h$$

Incappando in un assurdo!

$$costo(P*) < costo(P_{opt})$$

Infatti, nessuna P di $A_1, ..., A_n$ può avere costo inferiore di P_{opt}.

7.2 Due problemi sui grafi a confronto

In questa sezione andremo a mettere a confronto due problemi sui grafi apparentemente simili, per vedere quando è (e non è definita) la sottostruttura ottima.

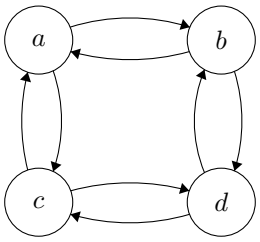
7.2.1 Dimostrare sottostruttura ottima

Consideriamo i seguenti due problemi aventi lo stesso input, ovvero un grafo $G = \{V, E\}$ e due vertici $u, v \in V$.

1. **Unweighted shortest path:**
   Trovare un cammino da u a v che abbia lunghezza minima:

2. **Longest simple path:**
   Trovare un ciclo semplice da u a v che abbia lunghezza massima.

La lunghezza di un cammino $p = \langle a_0, a_1, ..., a_n \rangle$ è $l(p) = n$, dove n il numero di archi da contare (se il grafo non è pesato).



Proviamo a verificare se la proprietà di sottostruttura ottima vale per entrambi i problemi.

Unweighted shortest path

Supponiamo che il problema sia trovare il cammino più breve da a a d nel grafo appena mostrato. Una soluzione ottima è $(a, b), (b, d)$.
Proviamo adesso a trovare le soluzioni ottime alle restrizioni:

1. Restrizione (a, b), soluzione $a \to b$

2. Restrizione (b, d), soluzione $b \to d$

Queste sono soluzioni ottime alle restrizioni, e sono all'interno della soluzione al problema globale.
È facile verificare la proprietà di sottostruttura ottima:

Esercizio: sottostruttura ottima di unweighted shortest path

Supponiamo che C^* sia un cammino minimo da u a v, ed escludiamo il caso banale $u = v$.
C^* può essere suddiviso in sottocammini compatibili [1].
Supponiamo per assurdo che uno dei due sottocammini non sia ottimo, e che quindi $C* = \langle c_1, c_2^* \rangle$. Questo implica l'esistenza di un cammino ottimo migliore di c_1 compatibile con c_2, che chiameremo c_1'. Se $c_1' < c_1*$, allora esisterà un cammino C' da u a v tale che:

$$C' = c_1' + c_2^* < c_1^* + c_2^* = C^*$$

E quindi esisterà un cammino minimo più breve di C^*, che abbiamo però imposto per ipotesi come soluzione ottima. Incappiamo in un assurdo, quindi nel problema non può non valere la sottostruttura ottima.

Longest simple path

Supponiamo che il problema sia trovare il cammino più lungo da a a d nel grafo appena mostrato. Una soluzione ottima è $a \to b \to d$.
Proviamo adesso a trovare le soluzioni ottime alle restrizioni:

1. Restrizione (a, b), soluzione $a \to c \to d \to b$
2. Restrizione (b, d), soluzione $b \to a \to c \to d$

Esse non sono contenute nella soluzione ottima globale. Il problema Longest simple path è un problema NP-hard!

Dimostrazione della sottostruttura ottima Longest Simple Path?

Non è richiesta, basta un controesempio per dimostrare che un problema non gode di una determinata proprietà.

[1] Per compatibili, intendo che se l'ultimo arco del sottocammino c_n avrà come nodo raggiunto il nodo a, c_{n+1} sarà compatibile con c_n se il primo arco di c_{n+1} partirà da a

7.3 Longest Common Subsequence

7.3.1 Introduzione al problema

Input

Due sequenze:
$X = \langle x_1, \ldots, x_m \rangle$
$Y = \langle y_1, \ldots, y_n \rangle$

Output e goal

Trovare una sottosequenza Z comune a X e Y di lunghezza massima.
La funzione obiettivo è la funzione $length(Z)$, e deve essere massimizzata.

Esempio:

$X = \langle ABCBDAB \rangle, \quad m = 7$
$Y = \langle BDCABA \rangle, \quad n = 6$
$Z = LCS(X, Y) = \langle BCBA \rangle$, $length(Z) = 4$ E non esistono sequenze comuni di lunghezza > 4.

Definizione di sottosequenza

Data una sequenza $X = \langle x_1, \ldots, x_m \rangle$ e una sequenza $Z = \langle z_1, \ldots, z_k \rangle$, diciamo che Z è una sottosequenza di X se $\exists$ una sequenza crescente di indici $\{i_1, \ldots, i_k\}$ di X tale che $x_{ij} = z_j$.

7.3.2 Approccio brute-force?

Enumerare tutte le sottosequenze di X e controllare per ognuna di esse se è una sottosequenza di Y, tenendo traccia della sottosequenza comune più lunga, ha complessità 2^m, dove m è il numero di indici di X, ovvero la lunghezza della sequenza di X.

7.3.3 Sottostruttura ottima di LCS

Siano $X = \langle x_1, \ldots, x_m \rangle$ e $= \langle y_1, \ldots, y_n \rangle$ sequenze e sia $Z = \langle z_1, \ldots, z_k \rangle$ 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 $x_m \neq y_n$ e $z_k \neq x_m$ allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X_{m-1} e Y
3. Se $x_m \neq y_n$ e $z_k \neq x_m$ allora $z_k = x_m = y_n$ e Z_{k-1} è una LCS di X e $Yn-1$

Dimostrazione

1. Se $z_k \neq x_m$, allora potremmo concatenare $x_m = y_n$ a Z ottenendo una sottosequenza comune di lunghezza $k+1$, contraddicendo l'ipotesi che Z sia una LCS comune di X e Y. Quindi $z_k = x_m = y_n$.
   Ora, il prefisso Z_{k-1} è una sottosequenza comune di X_{m-1} e Y_{n-1} di lunghezza $k-1$.
   Supponiamo per assurdo che esista una sottosequenza comune W di X_{m-1} e Y_{n-1} di lunghezza $> k-1$, allora concatenando a W $x_m = y_n$ si ottiene una sequenza più lunga di Z, avendo lunghezza maggiore di k, contraddicendo l'ipotesi che Z sia un LCS.
2. Se $z_k \neq x_m$, allora Z è una sottosequenza comune di X_{m-1} e Y. Se esistesse W comune a X_{m-1} e Y di lunghezza maggiore a Z, allora W sarebbe comune a X e Y e più lunga di Z, contraddicendo l'ipotesi.
3. Se $z_k \neq y_n$, allora Z è una sottosequenza comune di Y_{m-1} e X. Se esistesse W comune a Y_{m-1} e X di lunghezza maggiore a Z, allora W sarebbe comune a X e Y e più lunga di Z, contraddicendo l'ipotesi.

Le ultime due dimostrazioni sono chiaramente simmetriche.

7.3.4 Definizione ricorsiva della lunghezza di una LCS

Definiamo $c[i,j]$ come la lunghezza della LCS per
$X_i = \langle x_1, \ldots, x_i \rangle$ e $Y_i = \langle y_1, \ldots, y_i \rangle$.

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

Siano $X = \langle x_1, \ldots, x_m \rangle$ e $Y = \langle y_1, \ldots, y_n \rangle$ sequenze e sia $Z = \langle z_1, \ldots, z_k \rangle = LCS(X, Y)$.

- Se $x_m = y_n \Rightarrow z_k = x_m = y_n$ e ottengo LCS per X_{m-1} e Y_{n-1}
- Se $x_m \neq y_n$, o studio LCS per X_{m-1} e Y, o studio LCS per X e Y_{n-1}

Basandoci sulla definizione ricorsiva di $c[i,j]$, possiamo scrivere un algoritmo ricorsivo.
Inoltre, lo spazio dei sottoproblemi ha dimensione polinomiale (proprietà dei sottoproblemi ripetuti, o *overlapping subproblems*).
Il numero di sottoproblemi distinti è infatti uguale al numero di $c[i,j]$ distinti, $0 \leq i \leq m,\ 0 \leq j \leq n \Rightarrow \Theta(mn)$.
Avendo verificato sottostruttura ottima e overlapping subproblems, possiamo usare un approccio basato sulla programmazione dinamica!

7.3.5 Programmazione dinamica su LCS

Descrizione

La procedura LCS LENGTH prende in input due sequenze $X = \langle x_1, \ldots, x_m \rangle$ e $Y = \langle y_1, \ldots, y_n \rangle$ sequenze, assieme alle loro lunghezze, memorizza i valori di $c[i, j]$ in una tabella $c[0 : m, 0 : n]$ e calcola gli elementi in ordine di riga crescente da sinistra a destra.
Mantiene inoltre una tabella $b[1 : w, 1 : n]$ di "frecce". Il tempo di questa procedura è $\Theta(nm)$, il calcolo di ogni elemento è calcolato in tempo $\Theta(1)$.

Pseudocodice

```
1. LCS-LENGTH(X,Y,m,n)
2.  Inizializzamo c[0:m,0:n] e b[1:m,1:n]
3.  FOR i = 0 TO m
4.      c[i,0]=0
5.  FOR j = 1 TO n
6.      c[0,j]=0
7.  FOR i = 1 TO m
8.      FOR j = 1 TO n
9.          IF x_i == y_j
10.             c[i,j] = c[i-1,j-1]+1
11.             b[i,j] = "Top-Left"
12.         ELSE IF c[i-1,j] >= c[i,j-1]
13.             c[i,j] = c[i-1,j]
14.             b[i,j] = "Top"
15.         ELSE
16.             c[i,j-1]
17.             b[i,j] = "Left"
18. RETURN c AND b
```

Troveremo a $c[m, n]$ la lunghezza della/e LCS!
Una volta costruita la tabella b, possiamo risalire a una LCS di X, Y. Iniziamo da $b[m, n]$ e seguiamo le indicazioni contenute al suo interno.

```
1. PRINT-LCS(b,X,i,j)
2.  IF i == 0 OR j == 0
3.      RETURN // la lunghezza di una LCS è zero / caso base
4.  IF b[i,j] == "Top-Left"
5.      PRINT-LCS(b,X,i-1,j-1)
6.      PRINT(x_i)
7.  IF b[i,j] == "Top"
8.      PRINT-LCS(b,X,i-1,j)
9.  ELSE
10.     PRINT-LCS(b,X,i,j-1)
```

7.3.6 Migliorare le procedure di LCS?

È possibile migliorare il codice eliminando la tabella b.
Il valore di $c[i,j]$ dipende infatti esclusivamente da $c[i,j-1]$, $c[i-1,j]$ e $c[i-1,j-1]$, e può essere stabilito in tempo $\Theta(1)$.
In questo modo, si può costruire una LCS in tempo $\Theta(m+n)$ usando una procedura simile a PRINT-LCS risparmiano spazio $\Theta(m \cdot n)$. Tuttavia, lo spazio usato non diminuisce asintoticamente perché la tabella c richiede spazio $\Theta(m \cdot n)$.
Si può ridurre asintoticamente lo spazio usato per LENGTH-LCS conservando ogni volta solo due righe, in quanto è da due righe (corrente e precedente) che è possibile stabilire la lunghezza della LCS. Si implementa con una sorta di meccanismo di sliding window.

Estratti dalla lezione della prof

```
LCS-LENGTH(X, Y, m, n)
1.  inizializziamo c[0:m, 0:n] e b[1:m, 1:n]
2.  for i=0 to m
3.      c[i,0]=0
4.  for j=1 to n
5.      c[0,j]=0
6.  for i=1 to m
7.      for j=1 to n
8.          if xi == yj
9.              c[i,j] = c[i-1,j-1]+1
10.             b[i,j] = "↖"
11.         else if c[i-1,j] ≥ c[i,j-1]
12.                 c[i,j] = c[i-1,j]
13.                 b[i,j] = "↑"
14.         else    c[i,j] = c[i,j-1]
15.                 b[i,j] = "←"
16. return c and b
```

Esempio: $X = \langle A\ B\ C\ B\ D\ A\ B \rangle$, $m = 7$
$Y = \langle B\ D\ C\ A\ B\ A \rangle$, $n = 6$

		j	0	1	2	3	4	5	6
i			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	0	0	0	1	1	1
2	B		0	1	1	1	1	2	2
3	C		0	1	1	2	2	2	2
4	B		0	1	1	2	2	3	3
5	D		0	1	2	2	2	3	3
6	A		0	1	2	2	3	3	4
7	B		0	1	2	2	3	4	4

lunghezze di una LCS (X, Y)

```
PRINTLCS(b, X, i, j)
1. if i==0 or j==0
2.   return // lunghezze di una LCS è zero
3. if b[i,j] == "↖"
4.   PRINTLCS(b, X, i-1, j-1)
5.   print "x_i"
6. else if b[i,j] = "↑"
7.   PRINTLCS(b, X, i-1, j)
8. else PRINTLCS(b, X, i, j-1)
```

Capitolo 8

Problemi dello Zaino

8.1 Zaino 0-1 vs Frazionario

In questo capitolo andremo a mettere a confronto due problemi dello zaino: lo zaino intero (0-1) e lo zaino frazionario. Questi problemi sono entrambi problemi di ottimizzazione, ma solo uno dei due può essere risolto con una strategia greedy.

8.1.1 Introduzione al problema dello zaino 0-1

Input

- n oggetti rappresentati da n variabili $x_1, \ldots, x_n$ tali che l'oggetto i ha valore v_i e peso w_i, $\forall i \in \{1, \ldots, n\}$
- Un limite di peso W.

Goal

Scegliere un sottoinsieme $J \subseteq \{1, \ldots, n\}$ di oggetti di valore complessimo massimo e peso complessivo limitato superiormente da W.

Goal equivalente

Trovare un assegnamento per le variabili $\{x_1, \ldots, x_n\}$ tale che
Funzione di costo:

$$\max \sum_{i=1}^{n} v_i x_i$$

Requisiti stretti:

$$\sum_{i=1}^{n} w_i x_i \leq W;\ x_i \in \{0, 1\}, \forall i \in \{1, \ldots, n\}$$

8.1.2 Introduzione al problema dello zaino frazionario

Input

- n oggetti rappresentati da n variabili $x_1, \ldots, x_n$ tali che l'oggetto i ha valore v_i e peso w_i, $\forall i \in \{1, \ldots, n\}$
- Un limite di peso W.

Goal

Trovare un assegnamento per le variabili $\{x_1, \ldots, x_n\}$ tale che
Funzione di costo:

$$\max \sum_{i=1}^{n} v_i x_i$$

Requisiti stretti:

$$\sum_{i=1}^{n} w_i x_i \leq W;\ x_i \in \mathbb{Q},\ 0 \leq x_i \leq 1, \forall i \in \{1, \ldots, n\}$$

8.1.3 Sottostruttura ottima Knapsack intero e frazionario

Knapsack ha sottostruttura ottima

Proof

Sia $\{x_1*, x_2*, \ldots, x_n*\}$ una soluzione ottima al problema.
Consideriamo un sottoproblema:

- di variabili $\{x_1, \ldots, x_{j-1}, x_{j+1}, \ldots, x_n\}$
- limite di peso $W - x_j w_j$

Supponiamo che la restrizione $\{x_1*, \ldots, x_{j-1}*, x_{j+1}*, \ldots, x_n*\}$ non sia ottima per il suo sottoproblema (per assurdo).
Ciò implica l'esistenza di un'altra soluzione $\{x'_1, \ldots, x'_{j-1}, x'_{j+1}, \ldots, x'_n\}$ ottima di peso è minore della soluzione $\{x_1*, \ldots, x_{j-1}*, x_{j+1}*, \ldots, x_n*\}$, ovvero

$$\sum_{i=1, i\neq j}^{n} v_i x'_i > \sum_{i=1, i\neq j}^{n} v_i x^*_i,\ \sum_{i=1, i\neq j}^{n} w_i x'_i \leq W - x^*_j w_j$$

Allora $\{x'_1, x'_2, \ldots, x'_{j-1}, x^*_j, x'_{j+1}, \ldots, x'_n\}$ è una soluzione ottima, al problema originario, migliore di $\{x^*_1, \ldots, x_j, \ldots, x^*_n\}$. Infatti:

$$\sum_{i=1, i\neq j}^{n} v_i x'_i + v_j x^*_j > \sum_{i=1, i\neq j}^{n} v_i x^*_i + v_j x^*_j = \sum_{i=1}^{n} v_i x^*_i$$

$$\sum_{i=1, i\neq j}^{n} w_i x'_i + w_j x^*_j \leq W - w_j x^*_j + w_j x^*_j = W$$

Osserviamo che:

- Nel caso dello zaino (0-1), $x^*_i \in \{0, 1\}$.
- Nel caso dello zaino frazionario, $x^*_i \in [0, 1]$

Ora che sappiamo di poter applicare la programmazione dinamica (avendo verificato , verifichiamo se è possibile usare l'approccio greedy.

8.1.4 Scelta greedy?

Proveremo a usare la seguente euristica: il rapporto valore-peso. Ordiniamo gli oggetti secondo il rapporto k_i, ovvero il rapporto valore-peso.

$$k_i = \frac{v_i}{w_i} \quad \forall i \in \{1, \ldots, n\}$$

Otterremo un algoritmo che impiega $O(nlogn)$, dove $O(nlogn)$ è il tempo richiesto per ordinare gli elementi.

8.1.5 Proprietà di scelta greedy su zaino frazionario

È verificata!

Dimostrazione

Gli elementi sono ordinati in maniera decrescente rispetto al rapporto valore-peso k.

$$k_1 \geq \cdots \geq k_n$$

Sia

$$\sum_{i=1}^{n} x_i v_i = \sum_{i=1}^{n} q_i k_i$$

Sia $\{q_1, \ldots, q_n\}$ una soluzione ottima che non contiene k_1 (la scelta greedy), ovvero $q_1 = 0$.
Allora, supponiamo che $q_2 \neq 0$. Allora possiamo sostituire il peso dell'oggetto 2 con lo stesso peso dell'oggeto 1, ottenendo

$$\{q'_1 = q_2, q'_2 = 0, q'_3 = q^*_3, \ldots, q'_n = q^*_n\}$$

Allora:

- **Il peso**

$$\begin{aligned} \sum_{i=1}^{u} q'_i &= q'_1 + q'_2 + \sum_{i=3}^{n} q_i \\ &= q_2 + 0 + \sum_{i=3}^{n} q_i \\ &= \sum_{i=1}^{n} q_i \leq W \end{aligned} \tag{8.1}$$

- **Il valore**

$$\begin{aligned}\sum_{i=1}^{u} q'_i k_i &= q'_1 k_i + q'_2 k_2 + \sum_{i=3}^{n} q_i k_i \\ &= q_2 k_1 + 0 \cdot k_2 + \sum_{i=3}^{n} q_i k_i \\ &\geq q_2 k_2 + \sum_{i=3}^{n} q_i k_i = \sum_{i=1}^{n} q_i k_i \\ &\Rightarrow \sum_{i=1}^{n} q'_i k_i \text{ è massimo}\end{aligned} \tag{8.2}$$

  Quindi, posso sempre usare la scelta greedy per trovare una soluzione ottima, nel problema dello zaino frazionario.

8.1.6 Proprietà di scelta greedy su zaino intero

Per dimostrare che lo zaino intero con gode della proprietà di scelta greedy, basterà portare un controesempio che la scelta greedy.

Controesempio

$W = 50kg$

i	w_i (kg)	v_i (\$)	k_i $(\$/kg)$
1	10	60	6
2	20	100	5
3	30	120	4

L'algoritmo greedy sceglierà il primo e il secondo oggetto, in quanto quelli dal miglior rapporto valore-dollaro con un guadagno di 160 dollari (e 30 kg su 50 trasportati).
È chiaro che questa non sia una soluzione ottima. La soluzione ottima al problema, sarà infatti 2 e 3, per un valore totale di 220 dollari.

Capitolo 9

Elementi di strategia Greedy

9.1 Cos'è un algoritmo greedy?

Un algoritmo greedy ottiene una soluzione ottima ad un problema di ottimizzazione attraverso una successione di scelte. Ad ogni "bivio", l'algoritmo fa la scelta che sembra la migliore localmente.
È una strategia che non sempre può produrre una soluzione ottima, ed è per questo che è meno versatile della DP.

9.1.1 Come sviluppare un algoritmo greedy

1. Formulare il problema da risolvere come un problema su cui effettuare una scelta. A ogni scelta, si dovrà rimanere con un sottoproblema.

2. Dimostrare che esiste una soluzione ottima che contiene la scelta greedy a ogni iterazione, per dimostrare che sia una scelta sicura (una "safe choice").

3. Dimostrare la sottostruttura ottima del problema.

9.1.2 Verificare la scelta greedy

La scelta greedy deve essere una scelta localmente ottima capace di costruire una scelta globalmente ottima.
Per dimostrare ciò, possiamo iniziare esaminando una soluzione globalmente ottima, e mostrare che è possibile sostituire le soluzioni ai sottoproblemi con la scelta greedy.

9.2 Activity Selection Problem

9.2.1 Introduzione al problema

Siete dei turisti, potete scegliere tot attività ma non possono sovrapporsi nel tempo. Ciascuna attività ha un tempo di inizio e un tempo di fine. Una soluzione ottima sarà un insieme di attività cui orari non si sovrappongono.

Input

Un insieme $S = \{a_1, a_2, ..., a_n\}$ di attività.
Per ogni attività sono specificati tempo di inizio (s_i) e di terminazione (f_i), con $0 \leq s_i < f_i < +\infty$, e $a_i = [s_i, f_i)$.

Goal

Selezionare un sottoinsieme $A \subseteq S$ di attività compatibili di cardinalità massima, tale che

$$\forall a_i, a_j \;\; [s_i, f_i) \cap [s_j, f_j) = \emptyset$$

9.2.2 Soluzione greedy

Possiamo ordinare le attività per tempo di terminazione crescente

$$f_1 \leq f_2 \leq \cdots \leq f_n$$

e scegliere ogni volta l'attività con tempo di terminazione minore compatibile con le attività già scelte. L'algoritmo avrà complessità $O(n \log n)$

Scelta greedy

Attività con il più piccolo tempo di terminazione $a*$.

Sottoproblema

Elimino da S tutte le attività che si sovrappongono ad $a*$

THM (scelta greedy activity selection)

Sia S_k un sottoproblema non vuoto e sia a_m l'attività di S_k con il più piccolo tempo di terminazione. Allora a_m è inclusa in qualche soluzione ottima. ovvero in qualche sottoinsieme di attività compatibili di S_k di cardinalità massima.

Dimostrazione

Sia A_k una soluzione **ottima** di S_k, ovvero un sottoinsieme di cardinalità massima di attività compatibili di S_k. Sia a_j l'attività che termina per prima di A_k.

- Se $a_j = a_m \Rightarrow a_m \in A_k$, la scelta greedy è contenuta.
- Se $a_j \neq a_m \Rightarrow a_m \notin A_k$
  Definiamo $A' = (A_k \backslash \{a_j\}) \cup \{a_m\}$, ovvero la soluzione ottima A_k, in cui andremo a sostituire a_j con a_m.
  Le attività in A'_k sono mutualmente compatibiliti, perché lo sono quelle in A_k. Inoltre, essendo $a_j \neq a_m$, e $a_j < a_m$, allora tutte le attività saranno compatibili tra loro. In più, $|A_k| = |A'_k|$, quindi A'_k è ottima.

9.3 Algoritmo di Huffman - Scelta greedy

9.3.1 Definizione algoritmo di Huffman

L'algoritmo di Huffman costruire un albero di codifica binaria prefix-free cui foglie sono i caratteri dell'alfabeto del testo da codificare.
La codifica di un carattere coinciderà con il cammino (semplice) dalla radice alla foglia contenente il carattere.

- Figlio sinistro: 0;
- Figlio destro: 1;

Il numero di foglie coincide col numero di caratteri C. Di conseguenza, l'albero avrà $|C| - 1$ nodi interni

Notazione

Sia T un albero di codifica prefix-free.
$c.freq$: frequenza del carattere $c \in C, \forall c \in C$.
$d_t(c)$: profondità di c nell'albero T = lunghezza della codifica di c
La nostra funzione di costo sarà $B(T)$, e coincide con la somma del prodotto frequenza per lunghezza della codifica di tutti i caratteri.

$$B(T) = \sum_{c \in C} c.freq \cdot d_t(c)$$

Vogliamo trovare T con $B(T)$ minimo.

9.3.2 Scelta greedy Huffman

La nostra euristica per la scelta greedy sarà il carattere più frequente.
Esiste un albero con $B(T)$ minimo che contiene la scelta greedy? Andiamo a verificare.

Proposizione di scelta greedy per codici prefix-free

Sia C alfabeto.
$c \in C$ caratteri e $c.freq$ frequenza del carattere c.
Siano $x, y \in C$ i caratteri con le frequenze minime. Allora, esiste una codifica prefix-free ottima in cui le codifiche di x e y hanno la stessa lunghezza massima e differiscono per l'ultimo bit, trovandosi a profondità massima.

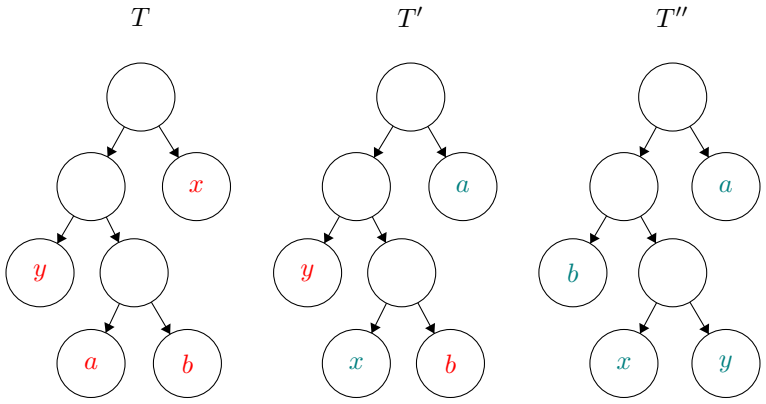
Dimostrazione

Sia T un albero che rappresenta un codice prefix-free ottimo. Siano a e b i due caratteri che corrispondono alle foglie di massima profondità di T.
Non perdendo generalità, assumiamo che $freq.a \leq freq.b$ e $x.freq \leq y.freq$.
Ne consegue che avremo:

$$x.freq \leq a.freq$$

$$y.freq \leq b.freq$$

Potrebbe accadere che $x.freq = a.freq$ oppure che $y.freq = b.freq$.
Inoltre, se $x.freq = b.freq \implies x.freq = a.freq = b.freq = y.freq \Rightarrow$ la proposizione sarebbe immediatamente vera.
Perciò possiamo assumere che $x.freq < b.freq$, ovvero che $x \neq b$.
Nell'albero T scambiamo le posizioni di a e di x, ottenendo T' scambiando le posizioni di b e di y ottenendo T'' in cui x e y hanno profondità massime.



Osserviamo inoltre che se $x = b$ ma $y \neq a$, T'' non ha x e y come foglie di profondità massima! Ora, facciamo vedere che il costo di T' non è maggiore di quello di T, $B(T') \leq B(T) \Rightarrow B(T) - B(T') \geq 0$

$$
\begin{aligned}
&B(T) - B(T') \\
&\quad = \sum_{c \in C} e.freq \cdot d_T(c) - \sum_{c \in C} e.freq \cdot d_{T'}(c) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_{T'}(x) - a.freq \cdot d_{T'}(a) \\
&\quad = x.freq \cdot d_T(x) + a.freq \cdot d_T(a) - x.freq \cdot d_T(a) - a.freq \cdot d_T(x) \\
&\quad = (a.freq - x.freq)(d_T(a) - d_T(x)) \\
&\quad \geq 0
\end{aligned}
\tag{9.1}
$$

Come facciamo a sapere che $(a.freq - x.freq)(d_T(a) - d_T(x))$ è positivo?
$(a.freq - x.freq) \geq 0$ in quanto x ha frequenza minima ($\Rightarrow$ a ha frequenza superiore o uguale).
$(d_T(a) - d_T(x)) \geq 0$ in quanto a è una foglia con profondità massima in T ($\Rightarrow$ x ha profondità minore o uguale).
Allora, se $B(T') \leq B(T)$, e analogamente, $B(T'') \leq B(T') \leq B(T)$

Capitolo 10

Grafi

10.1 Introduzione ai grafi

Sono molti gli algoritmi relativi ai grafi che andremo a studiare in questo capitolo. Proprio per questo, andremo in primis a fornire tutte le definizioni utile relative ai grafi.

10.1.1 Definizione di un grafo

Un grafo è una coppia di insiemi:

$$G = (V, E)$$

Con V insieme dei vertici e $E \subseteq V^2$.
Un grafo si dirà completo quando $E = V^2$.
$V = \{v_1, \ldots, v_n\}$.
Definiamo inoltre l'arco (v, v) come cappio.

10.1.2 Grafo non orientato vs orientato

Un grafo $G = \{V, E\}$ è detto orientato se:

$$(v_1, v_2) \neq (v_2, v_1)$$

E quindi saranno entrambe coppie ordinate.
Sarà invece detto non orientato se:

$$(v_1, v_2) = (v_2, v_1)$$

E quindi $(v_1, v_2) = (v_2, v_1) = \{v_1, v_2\}$, ovvero un insieme e non una coppia ordinata.

10.2 Ricerca in profondità (DFS)

Sia $G = (V, E)$ un grafo (orientato).
La ricerca DFS esplora gli archi che dipartono dal vertice v scoperto più recentemente e che ancora sono inesplorati.
Una volta che tutti gli archi uscenti da v sono stati esplorati, la ricerca fa retromarcia per esplorare gli archi uscenti dal vertice u da cui aveva raggiunto v.
Il processo continua finché non tutti i vertici raggiungibili dal vertice sorgente sono stati esplorati.
Se nel grafo sono ancora presenti dei vertici non esplorati, generalmente [1] , la ricerca DFS seleziona un nuovo vertice sorgente e ripete la procedura.

10.2.1 Grafo dei predecessori

È una foresta che comprende uno o più alberi dei predecessori, e rappresenta l'esplorazione fatta.

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

Dove $v.\pi$ è predecessore di v.

[1] Tratto dal libro: *Potrebbe sembrare arbitrario il fatto che una visita in ampiezza sia limitata a una sola sorgente, mentre una visita in profondità può cercare da più sorgenti. Anche se una visita in ampiezza potrebbe concettualmente partire da più sorgenti, e una visita in profondità potrebbe essere limitata ad una sorgente, il nostro approccio riflette il modo in cui i risultati di queste visite vengono comunemente usati.*

10.3 Topological Sort

10.3.1 Definizione Ordinamento topologico

Sia $G = \{V, E\}$ un grafo.
Un ordinamento topologico di G è un ordinamento lineare crescente dei suoi vertici tale che

$$(u, v) \in E \Rightarrow u < v$$

10.3.2 Topological Sort

L'algoritmo "Topological Sort" prende in input un DAG[1] $G = \{V, E\}$, e restituisce in output un ordinamento topologico di G.

Ossservazione

L'algoritmo in questione è definito solo su DAG.
Un ciclo all'interno del grafo non permette di individuare un ordinamento topologico relativo ai nodi contenuti nel ciclo.
È inoltre evidente che non c'è modo di orientare topologicamente un grafo non direzionato.

Topological Sort passo-passo

1. Invoca DFS su G per calcolare il finish-time $v.f \forall v \in V$
2. Al termine dell'esplorazione di ogni vertice, inseriscilo in una lista concatenata L
3. Ritorna la lista L

Tempo computazionale

Il tempo computazionale di Topological Sort è $\Theta(|V| + |E|)$, dato che DFS impiega tempo $\Theta(|V| + |E|)$ e l'inserimento di un vertice nella lista impiega $O(1)$[2].

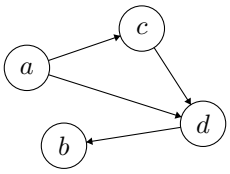
[1] *Directed acyclic graph*, o Grafo diretto aciclico
[2] Supponendo che l'inserimento nelle liste di adiacenza sia sempre in testa

10.3.3 Dimostrazione funzionamento del Topological Sort

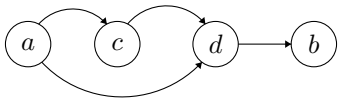
Iniziamo dal seguente Lemma.

Lemma 1:

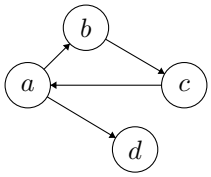
Un grafo diretto G è aciclico se e solo se una ricerca DFS di G non presenta archi all'indietro.
Esempio:



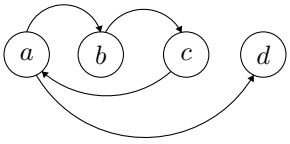
Partendo da a avremo:



Non si presenta alcun arco all'indietro, quindi G è un DAG.
Esempio 2:



Partendo da a avremo:

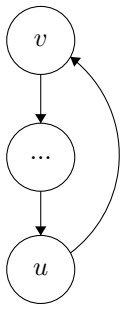


Si presenta un arco all'indietro (c, a), non è un DAG.

Dimostrazione

Andiamo a suddividere in due parti la dimostrazione della doppia implicazione (...se e solo se...).

1. $G = (V, E)$ è un DAG $\Rightarrow DFS(G)$ non genera archi all'indietro.
   Supponiamo che una DFS su G produca un arco all'indietro (u, v). Allora v è un predecessore di u nella foresta DF G_π.

   Quindi, G contiene un ciclo, ma questo è un assurdo, perché per ipotesi, G è un DAG.

2. $G = (V, E)$ è un DAG $\Leftarrow DFS(G)$ non genera archi all'indietro.
   Supponiamo che G contenga un ciclo γ e facciamo vedere che, in tal caso, una ricerca DFS genererebbe un arco all'indietro.
   Sia v il primo nodo ad essere scoperto dalla DFS. Al momento $v.d$ (discovery time di v) i vertici di γ formano un cammino bianco da $v$al predecessore di v in γ, che chiameremo u.
   Allora, come conseguenza del White Path Theorem, u è discendente di v nella foresta $G_\pi \Rightarrow (u, v)$ è un arco all'indietro, il che va contro la nostra ipotesi!

Dimostrazione correttezza Topological Sort

Sarà sufficiente mostrare che se $(u, v) \in E$ è un arco, $u < v$.
Sappiamo inoltre che $u < v \Rightarrow v.f < u.f$, ovvero che se il tempo di fine visita di v è minore di quello di u.
Per mostrare ciò, osserviamo che quando DFS esplora u, il vertice v non può essere grigio, altrimenti v sarebbe predecessore di u, rendendo (u, v) un arco all'indietro, cosa proibita in un DAG dal Lemma 1. Rimangono solo due casi:

- v è bianco: v è un discendente di $u \Rightarrow v.f < u.f$
- v è nero: v è già stato scoperto, $v.f$ già determinato, e quindi $v.f < u.f$

Capitolo 11

Single-Source Shortest Path

11.1 Algoritmi Single-Source Shortest Path

In questo capitolo andremo a introdurre molteplici algoritmi che risolvono il "problema" del *Single-Source Shortest Path*.

11.1.1 Introduzione al SSSP

Input

- Un grafo $G = (V, E)$
- Un vertice $s \in V$, chiamato sorgente (source)
- Una funzione di peso sugli archi $w : E \to R$.

Goal

Trovare un cammino da s a v, $\forall v \in V$ ovvero $p = \langle u_0, u_1, \ldots, u_n \rangle$ (p = path) tale che $(u_{i-1}, u_i) \in E$ e inoltre vogliamo che

$$w(p) = \sum_{i=1}^{n} w(u_{i-1}, u_i)$$

sia minimo.
È chiaro che il problema del single-source shortest path ricada nella categoria dei problemi di ottimizzazione.

Notazione

Il peso di un cammino minimo $\delta(s, v)$ da s a v è definito in questo modo:

$$\delta(s, v) = \begin{cases} \min(w(p) : s \rightsquigarrow^p v) \text{ se esiste un cammino da } s \text{ a } v \\ \infty \text{ se non esiste alcun cammino da } s \text{ a } v \end{cases}$$

Pesi negativi: osservazioni

All'interno del nostro grafo potrebbero essere contenuti dei cammini di peso negativo, o addirittura dei cicli.

- Quando non sono presenti cicli di peso complessivo negativo raggiungibili, è sempre possibile individuare $\delta(s, v)$ ben definiti $\forall v$.
- Quando sono presenti cicli di peso negativo raggiungibili, non è possibile trovare un valore ben definito di $\delta(s, v)$, in quanto sarà sempre possibile diminuirlo effettuando un numero indefinito di passi all'interno del ciclo. In tal caso, diremo che $\delta(s, v) = -\infty$

11.2 Algoritmo Bellman-Ford

11.2.1 Introduzione all'algoritmo Bellman-Ford

Risolve Single-Source Shortest Path nel caso generale in cui i pesi degli archi possono essere negativi ($w : E \Rightarrow R$).
Dato un grafo $G = (V, E)$ orientato, $s \in V$ vertice sorgente e $w : E \to R$, l'algoritmo Bellman-Ford restituisce un valore booleano $True$ o $False$.
Se ritorna $False$ significa che esiste un ciclo di peso negativo raggiungibile dalla sorgente.
Se ritorna $True$, allora $\forall v \in V,\ \exists \delta(s, v)$, e sono valori ben definiti.

11.2.2 Inizializzazione e rilassamento

Tutti gli algoritmi che andremo ad affrontare in questo capitolo, usufruiscono di due metodi:

1. **Initialize**
   Inizializza il grafo impostando le distanze di tutti i nodi dal nodo sorgente a $v.d = +\infty$ e il predecessore di ogni nodo $v.\pi = NULL$. Alla fine, impostiamo la distanza della sorgente da se stessa a zero, ovvero $s.d = 0$.
   Prende in input il grafo G e il nodo sorgente s.

   ```
   1. INITIALIZE-SINGLE-SOURCE(G,s)
   2.  FOR EACH v IN G.V
   3.       v.d = INFTY
   4.       v.pi = NULL
   5. s.d = 0
   ```

2. **Relax**
   Il rilassamento è i una procedura che prende in input due nodi u e v, e una funzione peso w sui cammini, e verifica se è possibile migliorare la distanza $v.d$ passando da u. Se è così, u diventa predecessore di v e $v.d = u.d + w(u, v)$

   ```
   1. RELAX(u,v,w)
   2.  IF v.d > u.d + w(u,v)
   3.       v.d = u.d + w(u,v)
   4.       v.pi = u
   ```

   È chiamata "rilassamento" perché "rilassa" l'upper bound della distanza minima da $v.d$ a ogni passaggio.

11.2.3 Pseudo-codice procedura Bellman-Ford

```
1. BELLMAN-FORD(G,s,w)
2.  INITIALIZE-SINGLE-SOURCE(G,s)   // Inizializza il problema
3.   FOR i=1 TO |G.V|-1             // Ciclo iterativo per V-1 volte
4.       FOR EACH (u,v) in G.E      // Ciclo iterativo su
5.           RELAX(u,v,w)           // tutti gli archi
6.   FOR EACH (u,v) in G.E
7.           IF v.d > u.d + w(u,v)  // Verifica dei cicli di
8.              RETURN False        // peso negativo
9.  RETURN True
```

L'algoritmo di Bellman-Ford impiega tempo $O(|V|^2 + |V||E|)$, quando il grafo è rappresentato dalle sue liste di adiacenza. La procedura Initialize impiega tempo $\Theta(|V|)$, mentre il codice Bellman-Ford, a riga 4-5, impiega tempo $\Theta(|V| + |E|)$. In realtà, spesso, sono sufficienti meno di $|V| - 1$ iterazioni del ciclo a righe 4-5.

11.2.4 Proprietà del rilassamento e dei cammini minimi

Le useremo per dimostrare la correttezza dell'algoritmo di Bellman-Ford.

Proprietà di disuguaglianza triangolare dei cammini minimi

$\forall (u, v) \in E$, il cammino minimo $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

Upper-Bound property

Per tutto il tempo della computazione, abbiamo che $v.d \geq \delta(s, v) \quad \forall v \in V$, e quando $v.d = \delta(s.v)$, non cambia più.

No-path property

Se non ci sono cammini da s a v, allora $\delta(s, v) = \infty$

Convergence property

Se $s \rightsquigarrow^p u \rightarrow v$ è un cammino minimo su G per qualche $u, v \in V$ e $u.d = \delta(s, u)$ a qualunque istante precedente, $RELAX(u, v, w) \Rightarrow v.d = \delta(s, v)$ a tutti gli istanti successivi

Predecessor-Subgraph Property

Se $v.d = \delta(s, v) \quad \forall v \in V$, il grafo dei predecessori radicato ad s, è anche un albero dei cammini minimi radicato ad s.

Path-Relaxation Property

Se $p = \langle v_0, v_1, \ldots, v_k \rangle$ da $s = v_0$ e se gli archi di p vengono rilassati secondo l'ordine $((v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k))$ allora $v_k.d = \delta(s, v_k)$ a prescindere dagli altri passi di rilassamento che posso occorrere, anche se inframezzate da rilasssamenti degli archi di p.

11.2.5 Dimostrazione della correttezza di Bellman-Ford, Lemma e Corollario

Ignoriamo i vertici non raggiungibili dalla sorgente. Dimostriamo prima il seguente lemma e enunciamo un corollario.

Lemma

Sia $G = (V, E)$ un grafo orientato, $s \in V$ un vertice sorgente, e $w : E \to R$ funzione peso.
Assumiamo non ci siano cicli di peso negativo raggiungibili da s. Allora, dopo $|V| - 1$ iterazioni del ciclo for (righe 2-4) dell'algoritmo di Bellman-Ford

$$v.d = \delta(s, v), \quad \forall v \textit{ raggiungibile da } s$$

Proof

Utilizziamo la **path-relaxation property**.
Si consideri v un qualunque vertice raggiungibile da s e sia $p = \langle u_0, u_1, \ldots, u_k \rangle$ un cammino minimo $u_0 = s$ e $u_k = v$ da s a v.
Poichè i cammini minimi sono semplici, p ha al più $|V| - 1$ archi, pertanto $k \leq |V| - 1$. Ognuna delle $|V| - 1$ iterazioni del ciclo for(linee 2-4) rilassa tutti gli archi di E.
Tra gli archi rilassati alla i-esima iterazione c'è sicuramente (u_{i-1}, u_i) e questo vale $\forall i \in \{1, \ldots, k\}$.
Per la path-relaxation property, allora $v.d = v_k.d = \delta(s, v_k) = \delta(s, v)$.

Corollario

Sia $G = (V, E)$ un grafo orientato con $s \in V$ sorgente e funzione peso $w : E \to R$. Allora

$$\forall v \in V, \exists (s, v) \Leftrightarrow \; BELLMAN - FORD(G, w, s) \text{ termina con } v.d < \infty$$

Ovvero, esiste un cammino tra s e tutti gli altri nodi, se tutte le distanze sono minori di ∞.

11.2.6 Teorema di correttezza di Bellman-Ford

Supponiamo di eseguire Bellman-Ford su (G, s, w), dove $G = (V, E)$ grafo orientato, $s \in V$ sorgente, $w : E \to R$ funzione peso.
Se non esistono cicli di peso negativo raggiungibili da s, allora:

1. L'algoritmo restituisce $TRUE$
2. $v.d = \delta(s, v) \quad \forall v \in V$
3. Il sottografo dei predecessori G_π è un albero dei cammini minimi radicato ad s

Se esiste un ciclo di peso negativo raggiungibile da s, allora l'algoritmo restituisce $FALSE$.

Dimostrazione

Dimostriamo prima il caso in cui il grafo **non contiene** cicli di peso negativo raggiungibili da s.

- **Affermazione 2**
  Claim: al termine dell'algoritmo, $v.d = \delta(s, v) \forall \quad v \in V$.
  Infatti, se v è raggiungibile da s, il lemma afferma che $v.d = \delta(s, v)$.
  Se v non è raggiungibile da s, non esiste un cammino da s a v, e quindi per il corollario $v.d = \infty$, e per la no-path property $\delta(s, v) = \infty$.
  Quindi, $v.d = \delta(s, v)$.

- **Affermazione 3**
  L'affermazione 2 e la Predecessor-Subgraph Property dimostrano l'affermazione 3.

- **Affermazione 1**
  Ora, osserviamo che al termine dell'eseguzione, $\forall (u, v) \in E$

$$\begin{aligned} v.d &= \delta(s, v) \\ &\leq \delta(s, u) + w(u, v) \\ &= u.d + w(u, v) \end{aligned} \tag{11.1}$$

  Per la disuguaglianza triangolare dei cammini minimi.
  Allora, la verifica a riga 7 del codice ritorna sempre $TRUE$.

Passiamo al caso in cui il G **contiente** un ciclo di peso negativo raggiungibile da s, che chiameremo c.

$$c = \langle v_0, v_1, \ldots, v_k \rangle, \quad \text{con } v_0 = v_k$$

e con

$$w(c) = \sum_{i=1}^{k} w(v_{i.1}, v_i) < 0$$

(peso negativo).

Adesso, supponiamo per assurdo che l'algoritmo restituisca $TRUE$, questo vuol dire che il test $v.d > u.d + w(u,v)$ (linea 7) non è mai soddisfatto.
In particolare, per i vertici del ciclo, $\forall i = \{1, \ldots, k\}$

$$v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$$

Sommando le disuguaglianze lungo il ciclo, si ottiene

$$\begin{aligned} \sum_{i=1}^{k} v_i.d &\leq \sum_{i=1}^{k} (v_{i-1}.d + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} v_{i-1}.d + \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned} \tag{11.2}$$

Poiché $v_k = v_0$. ogni vertice di c è contato esattamente una volta nelle due somme, ovvero

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} (v_{i-1}.d)$$

Poiché $v_k = v_0$, ogni vertice di c è contato esattamente una volta nelle due somme, ovvero

$$\sum_{i=1}^{k} v_i.d = \sum_{i=1}^{k} v_{i-1}.d$$

$$\overbrace{v_1.d + v_2.d + \cdots + v_k.d}^{\sum_{i=1}^{k} v_i.d} \qquad \overbrace{v_0.d + v_1.d + \cdots + v_{k-1}.d}^{\sum_{i=1}^{k} v_{i-1}.d} .$$

$v_k.d$

Inoltre, per il corollario

$$v_i.d \neq \infty \Rightarrow \sum_{i=1}^{k} v_i.d \neq \infty$$

Se il ciclo c è raggiungibile da s, lo saranno tutti i suoi nodi. Verificato ciò, torniamo sulla disequazione.

$$\begin{aligned} \cancel{\sum_{i=1}^{k} v_i.d} &\leq \cancel{\sum_{i=1}^{k} v_{i-1}.d} + \sum_{i=1}^{k} w(v_{i-1}, v_i) \\ 0 &\leq \sum_{i=1}^{k} w(v_{i-1}, v_i) \end{aligned} \tag{11.3}$$

Incontrando una contraddizione rispetto all'ipotesi, che affermava:

$$w(c) = \sum_{i=1}^{k} w(v_{i.1}, v_i) < 0$$

Un grafo che contiene cicli negativi, non può non restituire $FALSE$.

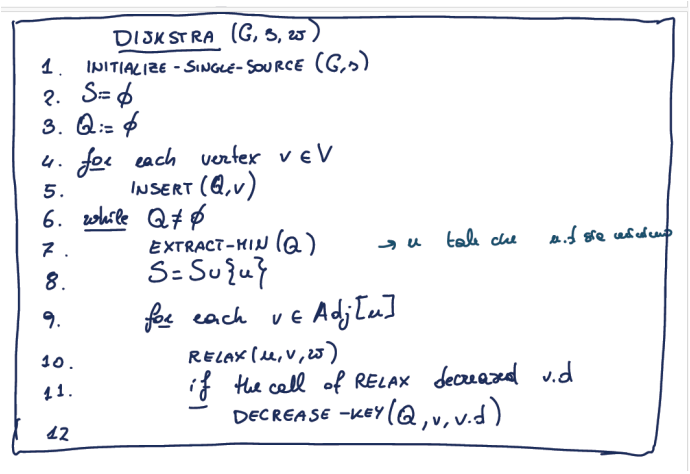
11.3 L'algoritmo di Dijkstra

Risolve il Single-Source Shortest Path problem nel caso in cui $w : E \to \mathbb{R}^+$, ovvero nel caso in cui gli archi sono sempre di peso non-negativo $\forall (u, v) \in E,\ w(u, v) \geq 0$.
Con una buona implementazione, il tempo computazionale dell'algoritmo di Dijkstra ha complessità inferiore di quello di Bellman-Ford.
Possiamo pensare all'algoritmo Dijkstra come una generalizzazione della ricerca BFS ai grafi pesati.

11.3.1 Introduzione a Dijkstra

L'implementazione sfrutta un insieme S e una coda con priorità Q di vertici u, e che userà come chiave la distanza $u.d$ da s.

1. Ogni nodo $v \in V$ viene inserito nella coda.
2. Se Q non è vuoto, estraiamo il minimo da Q, ossia la chiave con distanza da s minore.
3. Inseriamo u nell'insieme S e usiamo $RELAX(u, v, w)$ per ogni nodo $v \in Adj[u]$.
4. Per ogni nodo $v \in Adj[u]$ cui $v.d$ è stata ridotta, richiamiamo la funzione $DECREASE - KEY(Q, v, v.d)$ su Q.
5. Se Q non è vuoto, ritorna allo step 2



11.3.2 Teorema della correttezza di Dijkstra

L'algoritmo di Dijkstra eseguito su un grafo orientato $G = (V, E)$ con una funzione di peso $w : E \to R_+^0$ a valori non negativi e vertice sorgente s, termina con $u.d = \delta(s.u) \ \ \forall u \in V$

Dimostrazione

Dimostriamo che all'inizio di ogni iterazione del ciclo while (linee 6-12) $v.d = \delta(s, v) \ \forall v \in S$.
L'algoritmo, infatti, termina quando $S = V$, così che $v.d = \delta(s, v) \ \forall v \in V$.
Usiamo l'induzione matematica per dimostrare ciò.
La dimostrazione avviene per induzione sul numero di iterazioni già avvenute del ciclo while, che è sempre uguale alla cardinalità di $|S|$ all'inizio d ogni iterazione.

- Caso base:
  $|S| = 0 \Rightarrow S = \emptyset$, claim banalmente vero, in quanto implica che il grafo non abbia nodi.
  $|S| = 1 \Rightarrow S = \{s\} \Rightarrow s.d = 0 = \delta(s, s)$ in quanto nodo sorgente.

- Ipotesi induttiva:
  $v.d = \delta(s, v) \ \forall v \in S, \ |S| > 1$.
  Passo induttivo: viene estratto u da $V \backslash S = Q$ e viene aggiunto a S.
  Dobbiamo mostrare che $u.d = \delta(s, u)$ a quest'istante.

  - Se non ci sono cammini da s a u, allora per la NO-PATH property, $u.d = \infty = \delta(s, u)$.
  - Se esiste un cammino da s a u, allora sia y il prio vertice su un cammino minimo da s a u, che non sia in S e sia x il suo predecessore (potrebbe accadere che y coincida con u o x coincida con s). Siccome y sicuramente appare non successivamente ad u e i pesi sono non negativi, allora $\delta(s, y) \leq \delta(s, u)$ (1).
    Inoltre, EXTRACT-MIN (linea 7) ha restituito u perchè $u.d$ ha stima minore, quindi abbiamo che $u.d \leq y.d$ (2).
    In più, abbiamo che $x \in S$, e quindi, per l'ipotesi induttiva, $x.d = \delta(s, x)$, quando x è stato aggiunto a S, (x, y) è stato rilassato. Quindi, per la convergence property, $y.d = \delta(s, y)$ (3).
    Infine, per la upperbound property, abbiamo che $\delta(s, u) \leq u.d$.

  Allora

  $$\delta(s, y) \leq \delta(s, u) \leq u.d \leq y.d = \delta(s, y) \Rightarrow u.d = \delta(s, u)$$

  e per la UPPER BOUND property, sappiamo che questo valore non cambia più.

11.3.3 Corollario

Dopo aver eseguito Djikstra su un grafo orientato e pesato G con pesi non negativi e sorgente s, il sottografo dei predecessori G_π è un albero dei cammini minimi radicato ad s.

Dimostrazione

Conseguenza immediata della dimostrazione precedente e della PREDECESSOR SUBGRAPH property.

11.3.4 Analisi complessità Djikstra

L'algoritmo mantiene la coda di prioritò Q invocando INSERT (riga 5), EXTRACT-MIN (riga 7) e DECREASE-KEY (riga 12).
Invoca INSERT e EXTRACT-MIN una volta per ogni vertice.
Per ogni vertice u aggiuntonto ad S, ogni arco in $Adj[u]$ è esaminato nel ciclo for (riga 9-12) esattamente una volta durante il corso dell'intero algoritmo.

$$\#\text{totale liste di adiacenza} = \sum_{u \in V} |adj[u]| = |E| \Rightarrow \text{ il ciclo viene iterato } |E| \text{ volte}$$

Il tempo di esecuzione di Dijkstra è

$$O(|V|(Tempo(\text{EXTRACT-MIN}) + |E|)$$

Questo tempo dipende dalla specifica implementazione della coda di priorità. Se usiamo un array lineare e quindi enumeriamo i vertici $1, \ldots, |V|$, così che v si trovi alla v-esima posizione dell'array, il tempo di EXTRACT-MIN è $O(|V|)$ e quindi globalmente abbiamo

$$O(|V|^2 + |E|) = O(|V|^2)$$

Se implementiamo Q con un min-heap, allora il tempo computazionale è $O((|V| + |E|) \log V)$ che è uguale a $O(E \log V)$ se $|E| = \Omega(V)$

Capitolo 12

All-Pairs Shortest Path

12.1 Introduzione al problema

Estendiamo il problema del single-source shortest path, che pone come obiettivo il trovare tutti i cammini minimi da singola sorgente, con l'all-pairs shortest path, che estende il problema a tutte le possibili sorgenti.
Tra le applicazioni di questi algoritmi, abbiamo il calcolo del diametro di una rete.

12.1.1 Input e output

Input

$G = (V, E)$ grafo orientato e pesato sugli archi, con funzione di peso $w : E \to \mathbb{R}$

Goal

Trovare un cammino minimo da u a v per tutte le coppie $u, v \in V$
L'output di un algoritmo che risolve questo problema è una tabella (matrice) in cui lo slot di coordinate $[u][v] = \delta(u, v)$. La matrice conterrà quindi il peso del cammino minimo da u a v nello slot della u-esima riga e della v-esima colonna.

12.1.2 Riutilizzare algorimi SSSP

È possibile riutilizzare gli algoritmi Single-Source Shortest Path su tutti i nodi per risolvere il problema All-Pairs Shortest Path.

- Se il grafo non presenta cammini negativi, possiamo usare Djikstra. Usando Djikstra con la Q implementata con un array, otterremo run-time $O(|V|^3 + |V||E|) = O(|V|^3)$
- Djikstra con un min-heap per la Q avrà tempo $O(|V|(|V| + |E| \log |V|)$, che con $|E| = \Omega(|V|)$ diventa $O(|V||E| \log |V|)$, inferiore a $O(|V|^3)$ se il grafo è sparso.

- Su grafi con pesi di qualsiasi tipo, usando Bellman-Ford, otteniamo $O(|V^2||E|)$ che diventa $O(|V|^4)$ se il grafo è denso.

12.1.3 Notazione e Rappresentazione

Enumerazione e vertici di G, $V = \{1, 2, \ldots, n\}$.
L'input al problema (ovvero il grafo pesato) è rappresentato dalla matrice $W = (w_{ij})$, in cui

$$w_{ij} = \begin{cases} 0 \text{ se } i = j \\ w(i,j) \text{ se } i \neq j \text{ e } (i,j) \in E \\ \infty \text{ se } i \neq j \text{ e } (i,j) \notin E \end{cases}$$

Nel grafo possono essere presenti archi di peso negativo, ma assumeremo che non ci siano cicli di peso negativo.

Matrice dei predecessori

Una soluzione quanto più completa al problema fornisce non solo il peso dei cammini minimi, ma anche una matrice $\Pi = (\pi_{ij})$, chiamata matrice dei predecessori, con $\pi_{ij} = NULL$ se $i = j$ o se non ci sono cammini da i a j (ovvero $\delta(i,j) = \infty$), altrimenti π_{ij} sarà uguale al predecessore di j in un cammino minimo da i.
La i-esima riga di Π induce un albero di cammino minimo radicato a i.

```
1. PRINT-ALL-PAIRS-SHORTEST-PATH(Pi,i,j)
2.  IF i == j
3.      PRINT i
4.  ELIF Pi[i][j] == NULL
5.      PRINT "Non esistono cammini da i a j"
6.  ELSE PRINT-ALL-PAIRS-SHORTEST-PATH(Pi,i,Pi[i][j])
7.      PRINT j
```

12.2 Programmazione dinamica

Osserviamo che ogni sottocammino di un cammino minimo è un sottocammino minimo nella sua restrizione. Tutti i problemi di cammini minimi hanno la proprietà di sottostruttura ottima (se non esistono cicli di peso negativo). Per usare un approccio basato sulla programmazione dinamica, andiamo a formulare il problema in forma ricorsiva.

12.2.1 Formulazione ricorsiva del problema

Sia $l_{ij}^{(r)}$ il peso di un cammino minimo da i a j che contiente al più r archi.

$$l_{ij}^{(0)} = \begin{cases} 0 \text{ se } i = j \\ \infty \text{ se } i \neq j \end{cases}$$

Per $r \geq 1$, avrò

$$l_{ij}^{(r)} = \min_{1 \leq k \leq n} \{l_{ik}^{(r-1)} + w_{kj}\}$$

Se il grafo $G = (V, E)$ non contiene cicli di peso negativo, allora i cammini minimi (se esistono) sono semplici, ovvero non contengono nodi ripetuti[1].
$L^{(r)} = (l_{ij}^{(r)}) = (l_{ij}^{(r-1)} + w_{kj})$. I cammini minimi sono sempre semplici, e pertanto, hanno al più $n - 1$ archi.

$(l_{ij}^{(n-1)}) = L^{(n-1)} = L^{(n)} = L^{(n+1)} = \ldots$ Completiamo la formulazione ricorsiva in questo modo:

$$l_{ij}^{(r)} = \min_{1 \leq k \leq n} \{l_{ik}^{(r-1)} + w_{kj}\}$$

12.3 Algoritmo per APSP basato su DP e prodotto tra matrici

Data una matrice $W = (w_{ij})$, calcoliamo una serie di matrici $L^{(0)}, L^{(1)}, L^{(2)}, \ldots, L^{(n-1)}$ con $L^{(r)} = (l_{ij}^{(r)})$.

$$L^{(0)} = l_{ij}^{(0)} \text{ con } l_{ij} = \begin{cases} 0 \text{ se } i = j \\ \infty \text{ se } i \neq j \end{cases}$$

e, con $r \geq 1$

$$l_{ij}^{(r)} = \min_{1 \leq k \leq n} \{l_{ik}^{(r-1)} + w_{kj}\}$$

Ribadiamo che la matrice $L^{(n-1)}$ è la matrice contenente la lunghezza definitiva dei cammini minimi in un grafo, ovvero:

$$L^{(n-1)} = (l_{ij}^{(n-1)}) = (\delta(i, j))$$

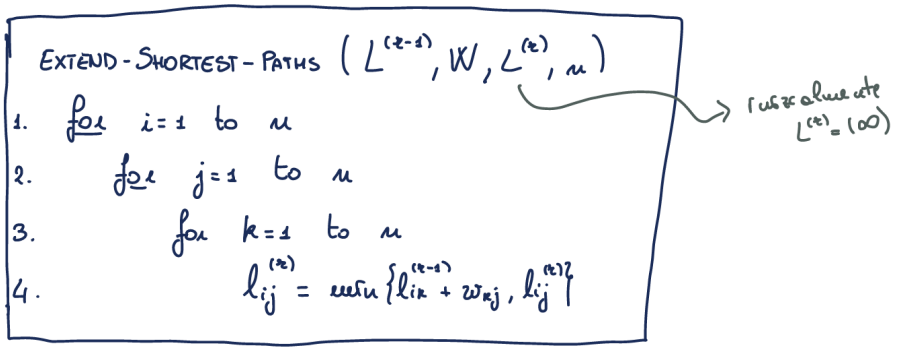
Osserviamo che esiste una relazione tra la formula per i cammini minimi e quella del prodotto tra matrici.

$$l_{ij}^{(r)} = \min_{1 \leq k \leq n} \{l_{ik}^{(r-1)} + w_{kj}\} \quad \text{e} \quad c_{ij} = \sum_{k=1}^{n} a_{ik \cdot b_{kj}}$$

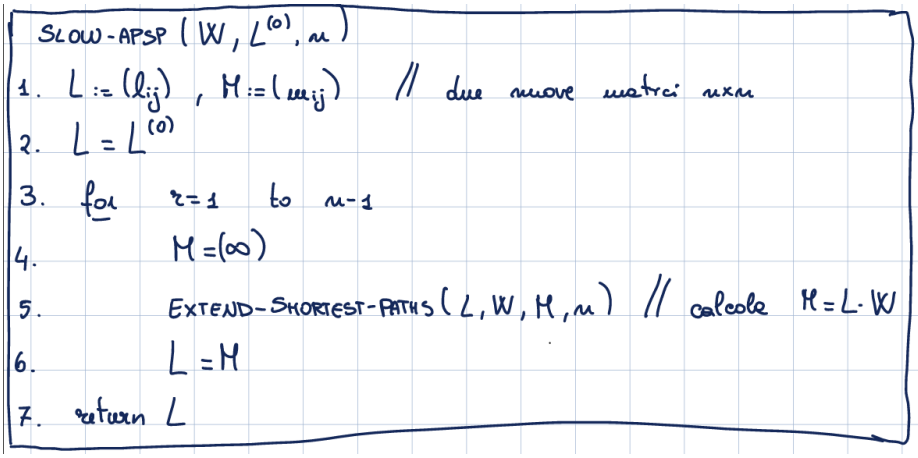
$l_{ij}^{(r)} \to c_{ij}$, $l_{ik}^{(r-1)} \to a_{ik}$,
$w_{kj} \to b_{kj}$, $+ \to \cdot$,
$\min \to +$.
E quindi, ottenere l'algoritmo EXTEND-SHORTEST-PATHS è possibile tramite la sostituzione di alcuni elementi in un algoritmo di moltiplicazioni tra matrici.

[1] È banale il fatto che un cammino minimo, in un grafo senza cicli di peso negativo, non effettuerà mai cicli, e quindi, non essendoci nodi ripetuti nei cammini in questione, ogni cammino minimo sarà semplice, contenendo al più $|V| - 1$ vertici

È quindi possibile ottenere tutti i cammini minimi usando questa insolita definizione di moltiplicazione tra matrici[2] su tutte le coppie di nodi per un numero $n-1$ di volte. La matrice $L^{(n-1)}$ conterrà tutti i cammini minimi del grafo.



Questo algoritmo, contenente la procedura del "prodotto" tra matrici avrà complessità $\Theta(|V|^4)$.



[2]Il prodotto tra matrici è un semi-anello su cui sono definite operazioni di somma e prodotto, entrambe associative, somma commutativa e prodotto distributivo. Questo vale sia per il semianello su cui somma e prodotto sono $+$ e $\cdot$, sia sul semianello tropicale su cui somma e prodotto sono $\bigoplus$ e $\bigotimes$, in cui il primo corrisponde con l'operazione min, e il secondo con l'addizione.

12.3.1 Ridurre la complessità dell'algoritmo

La complessità dell'algoritmo SLOW-ASP è data dall'operazione EXTEND-SHORTEST-PATH di complessità $\Theta(|V|^3)$ ripetuto per $|V|-1$ volte, ottenendo una complessità pari a $\Theta(|V|^4)$. Tuttavia, per ottenere la matrice $L^{(n-1)}$ possiamo ridurre il numero di prodotti tra matrici usando la tecnica dei quadrati ripetuti, per cui:

$$L^{(1)} = W,\ L^{(2)} = W \cdot W,\ L^{(4)} = W^2 \cdot W^2$$

Riducendo il numero di prodotti tra matrici pari a $\lceil \log(|V|-1) \rceil$.

```
1. FASTER-APSP(W,n)
2.  L,M NEW MATRIX[n][n]
3.  L = W
4.  r = 1
5.  WHILE r < n-1
6.      M = infinito
7.      EXTEND-SHORTEST-PATHS (L,L,M,n)
8.      r = 2r
9.      L = M
10. RETURN L
```

E permettendoci di ottenere tempo $\Theta(n^3 \log n)$.

12.4 L'algoritmo di Floyd-Warshall

12.4.1 Introduzione

L'algoritmo di Floyd-Warshall risolve il problema ALL-PAIRS-SHORTEST-PATH $\Theta(V^3)$.

Definizione

Un vertice intermedio di un cammino semplice $p = \langle v_1, v_2, \ldots, v_l \rangle$ è un qualunque vertice di p diverso da v_1 e da v_l, e appartiene quindi a $\{v_2, \ldots, v_{l-1}\}$. Enumeriamo i vertici di G, $V = \{1, \ldots, n\}$ e consideriamo un sottoinsieme $\{1, 2, \ldots, k\}$ di vertici per qualche $1 \leq k \leq n$.

Osservazione

Per ogni coppia di vertici (i, j) consideriamo tutti i cammini da i a j i cui vertici intermedi appartengono a $\{1, \ldots, k\}$ e sia p un cammino di peso minimo tra quest (p è un cammino semplice).

- Se k non è un vertice intermedio di p, allora tutti i vertici intermedi di p appartengono a $\{1, \ldots, k-1\}$;
- Altrimenti, se k è un vertice intermedio di p, allora possiamo decomporre p in $i \rightsquigarrow_{p_1} k \rightsquigarrow_{p_2} j$, e k non è un vertice intermedio né di p_1, né di p_2 e quindi i vertici intermedi di p_1 e p_2 appartengono a $\{1, \ldots, k-1\}$.

Quest'osservazione suggerisce una formulazione ricorsiva di ALL-PAIRS-SHORTEST-PATHS diversa da quella vista in precedenza.

12.4.2 Formulazione ricorsiva

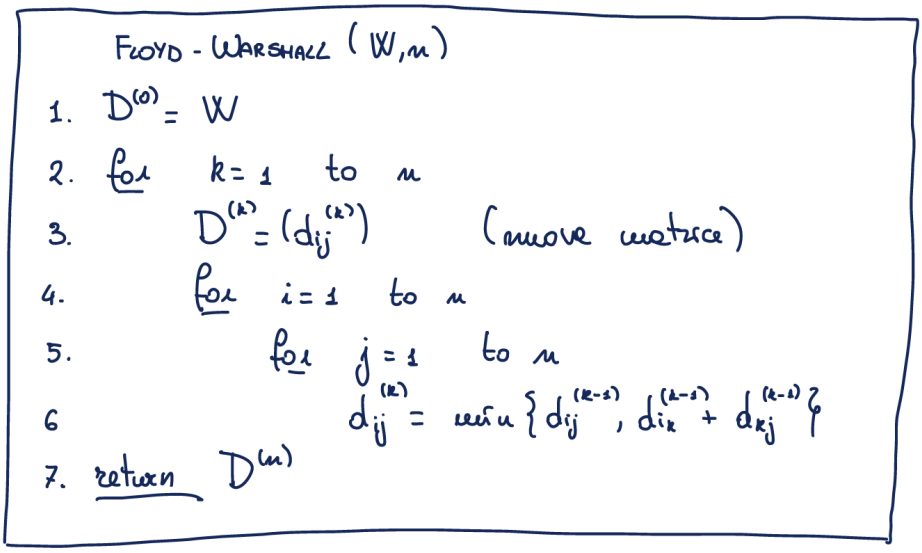
Sia $d_{ij}^{(k)}$ il peso di un cammino minimo da i a j i cui vertici intermedi appartengono a $\{1, \ldots, k\}$.

$$d_{ij}^{(k)} = \begin{cases} w_{ij} \text{ se } k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \text{ se } k \geq 1 \end{cases}$$

La matrice $D^{(u)} = (d_{ij}^{(u)})$ darà la soluzione al problema dato che i vertici di un cammino minimo qualunque appartengono a $\{1, \ldots, n\}$ quindi $d_{ij}^{(u)} = \delta(i, j)$ $\forall i, j \in V$.

12.4.3 Algoritmo

Floyd-Wrashall è un algoritmo bottom-uè che calcola i valori $d_{ij}^{(k)}$, e quindi le matrici $D^{(k)}$ in ordine di k crescente.



Il tempo computazionale dell'algoritmo è determinato dai cicli for.
L'algoritmo impiegherà tempo $\Theta(n^3)$.

12.4.4 Costruire lo shortest path

Esistono vari modi per costruire un cammino minimo usando Floyd-Warshall. Uno è calcolare la matrice dei predecessori Π e poi usare la procedura PRINT-ALL-PAIRS-SHORTEST-PATHS.
Alternativamente, la matrice Π può essere calcolata nel corso dell'algoritmo calcolando $\Pi^{(0)}, \Pi^{(1)}, \ldots, \Pi^{(n)}$ contestualmente a $D^{(0)}, D^{(1)}, \ldots, D^{(n)}$.
$\Pi = \Pi^{(n)}$, e $\pi_{ij}^{(k)}$ è definito il predecessore di j in un cammino minimo da i a j i cui vertici intermedi sono in $\{1, \ldots, k\}$. Formalmente

$$\pi_{ij}^{(0)} = \begin{cases} NULL & \text{se } i = j \text{ o } w_{ij} = \infty \\ i & \text{se } i \neq j \text{ o } w_{ij} < \infty \end{cases}$$

Con $k > 0$, invece

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{kj}^{(k-1)} & \text{se } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{ij}^{(k-1)} & \text{se } d_{ij}^{(k-1)} \leq d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$