Algoritmi e Principi dell'Informatica

Linguaggi

Definizioni

`alfabeto`	insieme finito di simboli
`stringa`	sequenza ordinata e finita di elementi dell'alfabeto (ε denota la stringa vuota)
`lunghezza`	numeri di elementi della stringa $\lvert abc\rvert = 3, \lvert\varepsilon\rvert = 0$
`concatenazione`	unione di due o più stringhe in una singola stringa **es.** se $w_1 = abc$ e $w_2 = def$ allora $w_1.w_2 = abcdef$
`riflesso`	inversione degli ordini dei caratteri di una stringa $w = c_1...c_n \iff w^R = c_n...c_1$ con $\lvert c_i\rvert = 1$ **es.** se $w = abc$ allora $w^R = cba$
Σ^*	insieme di tutte le stringe su Σ **es.** $\Sigma = \{0,1\}$, $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, \dots\}$
Σ^+	insieme di tutte le stringe non vuote su Σ **es.** $\Sigma = \{0,1\}$, $\Sigma^+ = \{0, 1, 00, 01, \dots\}$

Operazioni su linguaggi

`unione`	$L_1 \cup L_2 = \{w \mid w \in L_1 \vee w \in L_2\}$
`intersezione`	$L_1 \cap L_2 = \{w \mid w \in L_1 \wedge w \in L_2\}$
`differenza`	$L_1 \setminus L_2 = \{w \mid w \in L_1 \wedge w \notin L_2\}$
`complemento`	$L^c = A^* \setminus L = \{w \mid w \in A^* \wedge w \notin L\}$
`concatenazione`	$L_1.L_2 = \{w_1.w_2 \mid w_1 \in L_1 \wedge w_2 \in L_2\}$
`riflesso`	$L^R = \{w^R \mid w \in L\}$

Modelli di calcolo

Automi a stati finiti (ASF/FSA)

Formalmente definiti come una quintupla (Q, I, δ, q_0, F) dove Q è un insieme finito di stati, I è l'alfabeto di input, $\delta : Q \times I \to Q$ (o, $\delta : Q \times I \to \mathcal{P}(Q)$ se l'automa è non deterministico) è la funzione di transizione che mappa una coppia (stato corrente, input) a uno stato (o stati, se l'automa è non deterministico) di destinazione, q_0 è lo stato di partenza e $F \subseteq Q$ è l'insieme degli stati finali.

FSA riconoscitore

Un linguaggio L su I è riconosciuto dall'FSA se e solo se data una stringa $w \in L$ si ha $\delta^*(q_0, w) \in F$ dove δ^* è un'estensione di δ definita ricorsivamente con caso base $\delta^*(q, \varepsilon) := q$ e passo ricorsivo $\delta^*(q, y.i) := \delta(\delta^*(q, y), i)$ con $i \in I, y \in I^*$

FSA traduttore

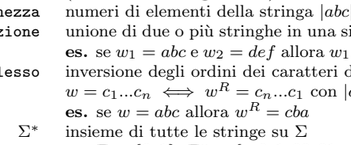
Un FSA traduttore associa un simbolo letto a uno scritto a ogni transizione tramite una funzione di traduzione η.

Operazioni su FSA

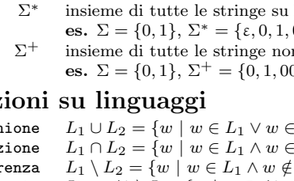
1. Dato un'automa $\mathcal{A}$ che riconosce il linguaggio L, possiamo costruire l'automa $\mathcal{A}'$ che riconosce il linguaggio L^C completando l'automa $\mathcal{A}$ e invertendo stati finali con iniziali.
2. Dati due automi $\mathcal{A}_1, \mathcal{A}_2$ che riconoscono i linguaggi L_1, L_2 rispettivamente, si può costruire algoritmicamente un'automa che riconosce $L_1 \cap L_2, L_1 \cup L_2, L_1 \setminus L_2$ partendo dall'automa prodotto $\mathcal{A}_{12}$.

Esempio 2.

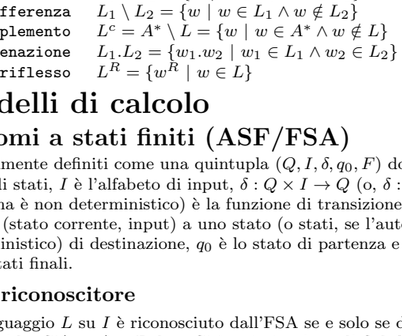
$\mathcal{A}_1$:



$\mathcal{A}_2$:



$\mathcal{A}_{12}$:



Affinché l'automa $\mathcal{A}_{12}$ riconosca $L_1 \cap L_2$, dobbiamo impostare come stati finali gli stati che sono finali in $\mathcal{A}_1$ e $\mathcal{A}_2$ quindi q_2A e q_2B. Affinché riconosca $L_1 \cup L_2$, dobbiamo impostare come stati finali gli stati che sono finali in $\mathcal{A}_1$ o $\mathcal{A}_2$ quindi $q_0A, q_1A, q_2A, q_0B, q_1B, q_2B, q_2A, q_2B, q_2C$ e affinché riconosca $L_1 \setminus L_2$ dobbiamo impostare come stati finali gli stati che sono finali in $\mathcal{A}_1$ ma non $\mathcal{A}_2$, quindi q_0A, q_1A, q_0B, q_1B.

Pumping lemma per FSA

Sia L un linguaggio riconosciuto da un FSA, allora $\exists p \in \mathbb{N}^+$ tale che ogni stringa $w \in L$ con $\lvert w\rvert \geq p$ può essere scritta come $w = xyz$ con:

1. $\lvert xy\rvert \leq p$
2. $y \neq \varepsilon$
3. $\forall i \geq 0, xy^iz \in L$

Esempio Dimostriamo che il linguaggio $L = \{0^n1^n \mid n \geq 0\}$ non è riconosciuto da un FSA usando il pumping lemma.

1. Supponiamo per assurdo che L sia riconosciuto da un FSA e scegliamo $w = 0^p1^p$
2. Scomponiamo w in xyz con:
   - $\lvert xy\rvert \leq p$
   - $\lvert y\rvert \geq 1$

   Scegliamo $x = 0^\alpha, y = 0^\beta, z = 0^{p-\alpha-\beta}1^p$ con $\alpha \geq 0,\ \beta \geq 1,\ \alpha + \beta \leq p$
3. Troviamo $i \geq 0$ tale che $xy^iz \notin L$
   $xy^iz = 0^\alpha 0^{i\beta} 0^{p-\alpha-\beta}1^p = 0^{p+i\beta-\beta}1^p$. Da cui abbiamo $xy^iz \in L \iff p + i\beta - \beta = p \iff i = 1$. Scegliendo quindi $i \neq 1$, ad esempio $i = 0$, abbiamo che $xy^iz \notin L$ che contraddice la nostra supposizione iniziale. □

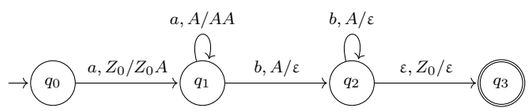
Automi a pila (AP/PDA)

Formalmente definiti come una settupla $(Q, I, \Gamma, \delta, q_0, Z_0, F)$ dove Q è un insieme finito di stati, I è l'alfabeto di input, Γ l'alfabeto di pila, $\delta : Q \times (I \cup \{\varepsilon\}) \times \Gamma \to Q \times \Gamma^*$ è la funzione di transizione, che mappa lo stato corrente, un simbolo $I \in I$ e il simbolo $\gamma \in \Gamma$ in cima alla pila a un nuovo stato e una nuova stringa $\gamma' \in \Gamma^*$ che sostituisce la cima della pila, $q_0 \in Q$ è lo stato di partenza, Z_0 è il simbolo che denota il fondo della pila e $F \subseteq Q$ è l'insieme degli stati finali.

PDA riconoscitore

Un linguaggio L è riconosciuto da un PDA se e solo se $\forall s \in L$, dopo aver scandito la stringa s, si trova in uno stato $q_f \in F$.

Esempio Automa a pila che riconosce il linguaggio $L = \{a^nb^n \mid n > 0\}$



$\varepsilon, Z_0/\varepsilon$ è una ε-mossa, cioè una mossa che non consuma simboli in ingresso.

PDA traduttore

Un PDA traduttore è dotato di un nastro di uscita su cui può scrivere ad ogni transizione.

Macchine di Turing (MT/TM)

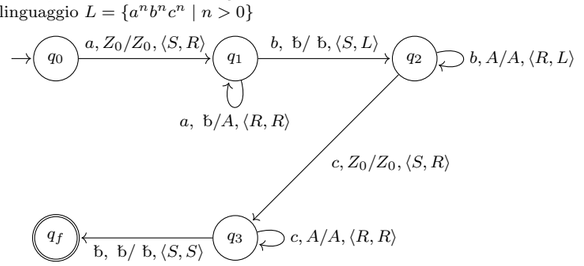
Le macchine di Turing a k nastri di memoria sono macchine dotate di un nastro di input e k nastri di memoria. Sono formalmente definite come una settupla $(Q, I, \Gamma, \mathfrak{b}, \delta, q_0, F)$ dove Q, I, q_0, F sono gli stessi di un PDA/FSA, Γ è l'alfabeto di nastro, $\mathfrak{b} \in \Gamma$ denota una cella di nastro vuota e $\delta : Q \times (I \cup \{\mathfrak{b}\}) \times \Gamma^k \to Q \times \Gamma^k \times \{L, S, R\}^{k+1}$ è la funzione di transizione dove L_i, S_i, R_i denotano il movimento dell testina sull'i-esimo nastro ($k+1$ perché oltre ai k nastri di memoria ci si può muovere anche sul nastro di input). Le macchine di Turing a nastro singolo sono un modello di calcolo equivalente alle macchine di Turing a k nastri dove input e memoria si trovano su un unico nastro.

Def. Una macchina di Turing universale (MTU) è una macchina di Turing in grado di simulare qualsiasi altra macchina di Turing.

MT riconoscitore

Una MT riconosce il linguaggio L se per ogni stringa $w \in L$ si ferma in uno stato di accettazione in un numero finito di transizioni.

Esempio Macchina di Turing a $k = 1$ nastro di memoria che riconosce il linguaggio $L = \{a^nb^nc^n \mid n > 0\}$



L'esempio utilizza la convenzione della MT infinita a destra con il simbolo Z_0 che denota la prima cella del nastro di memoria.

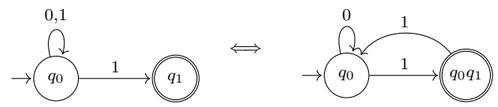
MT traduttore

Una MT trad. è dotata di un addizionale nastro di output su cui può solo scrivere e in cui la testina può muoversi solo a destra (`R`) o stare ferma (`S`).

Non determinismo

Informalmente, un modello di calcolo si dice non deterministico se esiste almeno una configurazione da cui è possibile prendere più di una strada. Gli FSA e le MT non deterministici hanno le stesse capacità espressive della loro versione deterministica. Gli automi a pila non deterministici (NPDA), invece, sono più espressivi di quelli deterministici. Per esempio, possono riconoscere $L = \{a^nb^n\} \cup \{a^nb^{2n}\}$. Dato un FSA non deterministico con insieme di stati Q, è possibile ricavare la sua versione non deterministica con un numero di stati che è al più $2^{\lvert Q\rvert}$ usando la *costruzione per sottoinsiemi* (*powerset construction*).

Esempio



Macchine RAM

Le macchine RAM sono un modello di calcolo equivalente alle macchine di Turing. Hanno un nastro di lettura `In` e uno di scrittura `Out` e sono dotate di memoria infinita con accesso a indirizzamento diretto. Supportano le seguenti istruzioni:

`READ X`	M[X] ← `In`	`WRITE* X`	`Out` ← M[M[X]]
`READ* X`	M[M[X]] ← `In`	`JUMP l`	`PC` ← l
`LOAD X`	M[0] ← M[X]	`JZ l`	se M[0]=0
`LOAD= X`	M[0] ← X		`PC` ← l
`LOAD* X`	M[0] ← M[M[X]]	`JGT l`	se M[0]>0
`STORE X`	M[X] ← M[0]		`PC` ← l
`STORE* X`	M[X] ← M[M[0]]	`JGZ l`	se M[0]≥0
`ADD X`	M[0] ← M[0] + M[X]		`PC` ← l
`SUB X`	M[0] ← M[0] - M[X]	`JLT l`	se M[0]<0
`MUL X`	M[0] ← M[0] × M[X]		`PC` ← l
`DIV X`	M[0] ← M[0] / M[X]	`JLZ l`	se M[0]≤0
`WRITE X`	`Out` ← M[X]		`PC` ← l
`WRITE= X`	`Out` ← X	`HALT`	termina exec.

Logica monadica del I ordine (MFO)

La logica monadica del I ordine è un modello di calcolo in grado di riconoscere linguaggi star-free, ossia linguaggi regolari esprimibili senza l'utilizzo della star di Kleene. La sintassi di una formula φ della logica monadica del I ordine è $\varphi := a(x) \mid x < y \mid \neg\varphi \mid \varphi \wedge \varphi \mid \forall x(\varphi)$
Dove $x, y \in \mathbb{N}$, $a(x)$ è un predicato unario che è vero se e solo se alla posizione x c'è il simbolo a e $<$ è un predicato binario che corrisponde alla relazione di minore. Partendo da questi assiomi si può definire il connettivo $\vee$, il quantificatore esistenziale $\exists$, le relazioni aritmetiche $=, \leq, >, \geq$ e somme e sottrazioni tra variabili e costanti ($y = x \pm k$). Si possono inoltre definire abbreviazioni ausiliarie, come:

`last(x)`	$\neg\exists y(y > x)$
`y=S(x)`	$y = x + 1$

Logica monadica del II ordine (MSO)

La logica monadica del II ordine è un modello di calcolo con le stesse capacità espressive degli automi a stati finiti. Differisce da MFO per il fatto che consente la quantificazione su insiemi di posizioni.
Esempio La seguente formula della logica monadica del II ordine riconosce il linguaggio $L = \{a^{2^n} \mid n \in \mathbb{N}^+\}$.

$\exists P(\forall x(a(x) \wedge \neg P(0) \wedge \forall y(y = x + 1 \implies (\neg P(x) \iff P(y))) \wedge (last(x) \implies P(x))))$

Dove $X(x)$ significa $x \in X$.

Grammatiche

Una grammatica è una quadrupla $G = (T, N, P, S)$ dove T è l'alfabeto terminale, N è l'alfabeto nonterminale, $S \in N$ è il simbolo iniziale e $P \subseteq N^+ \times (N \cup T)^*$ è l'insieme delle produzioni sintattiche. Data una grammatica G si definisce derivazione immediata la relazione binaria $\underset{G}{\Longrightarrow}$ definita come:
$x \underset{G}{\Longrightarrow} y \iff \exists u, v, p, q \in (N \cup T)^* : (x = upv) \wedge (p \to q \in P) \wedge (y = uqv)$
Esempio Prendiamo in considerazione la grammatica $G = \langle\{a, b\}, \{S, R\}, \{S \to aR, R \to bS, S \to \varepsilon\}, S\rangle$, abbiamo che $abaR \underset{G}{\Longrightarrow} ababS$. Il linguaggio $L(G)$ generato dalla grammatica G è l'insieme di tutte e sole le stringhe s di soli caratteri di T tali che $S \overset{*}{\underset{G}{\Longrightarrow}} s$ dove $\overset{*}{\underset{G}{\Longrightarrow}}$ è la chiusura riflessiva e transitiva di $\underset{G}{\Longrightarrow}$. A seconda delle limitazioni imposte sulle regole di produzione $\alpha \to \beta$ si hanno 4 tipi di grammatiche

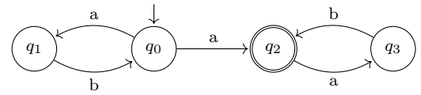
Tipo	Nome	Regole prod.	Equiv.
0	Non ristretta	nessuna	TM
1	Monotone oppure Context sensitive	$\lvert\alpha\rvert \leq \lvert\beta\rvert$ oppure $\alpha = \gamma A\delta,\ \beta = \gamma\chi\delta$ con $\chi \neq \varepsilon$	(LBA)
2	Context free	$\lvert a\rvert = 1$	NPDA
3	Regolare	$A \to \alpha,\ A \to \alpha A$ o $A \to \alpha,\ A \to A\alpha$	FSA

Si può dimostrare che le grammatiche monotone e dipendenti dal contesto hanno la stessa capacità espressiva.

Conversione Automa ↔ Grammatica

Dato un automa $\mathcal{A}$ che riconosce un certo linguaggio $L(\mathcal{A})$, è possibile ricavare la grammatica $\mathcal{G}$ tale che $L(\mathcal{G}) = L(\mathcal{A})$ (o viceversa).
Esempio FSA Linguaggio $L = (ab)^*a(ab)^*$

$\mathcal{A}$:



$\mathcal{G} = (T = \{a, b\}, N = \{q_0, q_1, q_2, q_3\}, P, S = q_0)$
P : $q_0 \to aq_1 \mid aq_2 \mid a$
$q_1 \to bq_0$
$q_2 \to aq_3$
$q_3 \to bq_2 \mid b$

Proprietà di chiusura dei linguaggi

	REG	DCFL	CFL	CSL	RIC	RE
`Unione`	✓	×	✓	✓	✓	✓
`Intersezione`	✓	×	×	✓	✓	✓
`Differenza`	✓	×	×	✓	✓	×
`Complemento`	✓	✓	×	✓	✓	×
`Concatenazione`	✓	×	✓	✓	✓	✓
`Star di Kleene`	✓	×	✓	✓	✓	✓

`REG`	Linguaggi regolari (riconosciuti da una grammatica regolare / esprimibili mediante un'espressione regolare)
`DCFL`	Linguaggi context free deterministici (riconosciuti da una grammatica context free deterministica)
`CFL`	Linguaggi context free (riconosciuti da una grammatica context free)
`CSL`	Linguaggi context sensitive (riconosciuti da una grammatica context sensitive)
`RIC`	Linguaggi ricorsivi (esiste una MT che per ogni stringa appartenente al linguaggio, accetta o rifiuta)
`RE`	Linguaggi ricorsivamente enumerabili (riconosciuti da una grammatica non ristretta)

`FIN` (linguaggi finiti) ⊂ `REG` ⊂ `DCFL` ⊂ `CFL` ⊂ `CSL` ⊂ `RIC` ⊂ `RE`

Teoria della computazione

`Tesi di Church`	tutti i problemi computabili sono descrivibili da una macchina di Turing. *(MT, programma, algoritmo, procedura sono usati come sinonimi)*
`problema`	richiesta di calcolo algoritmico di $f : A \to B$ con $\lvert A\rvert, \lvert B\rvert \leq \lvert\mathbb{N}\rvert$. (In generale f è parziale, cioè $\exists n(f(n) = \perp)$)
`f. computabile`	se esiste un algoritmo che calcola f, f è calcolabile e il problema è computabile. f_i denota l'i-esima funzione computabile. *(calcolabile, computabile e risolvibile sono sinonimi)*
`ins. dec.`	un insieme S è decidibile (o ricorsivo) se esiste una funzione $\mathbb{1}_S$ che restituisce 1 se $n \in S$ e 0 se $n \notin S$.
`ins. semidec.`	un insieme S è semidecidibile (o ricorsivamente enumerabile o r.e.) se esiste una funzione $\mathbb{1}'_S(n)$ che restituisce 1 se $n \in S$.

Se un problema è chiuso, cioè che la sua soluzione non dipende da un valore in ingresso, allora è decidibile.

Teorema Sia S un insieme, allora:
S ric. $\iff$ S r.e. $\wedge$ S^C r.e.

Corollario Se $\neg S$ ric., allora:
(S r.e. $\wedge$ $\neg S^C$ r.e.) $\vee$ ($\neg S$ r.e. $\wedge$ S^C r.e.) $\vee$ ($\neg S$ r.e. $\wedge$ $\neg S^C$ r.e.)

Dimostrazione di semidecidibiltà (Dovetailing)

Per stabilire se $\exists z \mid f_x(z) \neq \perp$ si può usare la tecnica del *dovetailing* che consiste nell'eseguire una sola mossa alla volta per ogni input fin quando non ci si arresta. Se $\exists z \mid f_x(z) \neq \perp$ sicuramente prima o poi viene incontrato e ci si arresta. Non si possono eseguire in sequenza $f_x(0)$ e poi $f_x(1)$ e così via perché se $f_x(0) = \perp$ non si arriverà mai ad eseguire $f_x(1)$.

Teorema di Rice

Sia $F = \{f_i \mid i \in \mathbb{N}\}$ l'insieme di tutte le funzioni computabili, sia $P \subseteq F = \{f_i \mid f_i \text{ soddisfa una certa proprietà}\}$ un sottoinsieme di F di funzioni computabili che soddisfano una certa proprietà e sia $S = \{x \mid f_x \in P\}$ l'insieme degli indici delle funzioni che appartengono a P. Allora S ric. $\iff P = \varnothing \vee P = F$. Più informalmente, il teorema di Rice afferma che per qualsiasi proprietà non banale (cioè vera per almeno una funzione e falsa per almeno una funzione) delle funzioni computabili, non esiste un algoritmo che possa decidere se una data funzione computabile soddisfi o meno quella proprietà.

Riduzione

La riduzione può essere usata per dimostrare che un dato problema è decidibile o indecidibile.

Caso decidibile Sia P_1 un noto problema decidibile e sia P_2 un problema che si sospetta essere decidibile. Si può dimostrare che P_2 è decidibile riducendolo a P_1.

RisolviP$_2(x)$
  $y \leftarrow$ Riduci(x)
  $sol \leftarrow$ RisolviP$_1(y)$
  return sol

Caso indecidibile Sia P_1 un noto problema indecidibile e sia P_2 un problema che si sospetta essere indecidibile. Si può dimostrare che P_2 è indecidibile riducendolo da P_1.

RisolviP$_1(x)$
  $y \leftarrow$ Riduci(x)
  $sol \leftarrow$ RisolviP$_2(y)$
  return sol

In altre parole, se riuscissimo a risolvere P_2, riusciremmo a risolvere anche P_1 ma questo è impossibile in quanto P_1 è un noto problema indecidibile.

Noti problemi indecidibili

Problema dell'arresto (Halting problem)

Data una macchina di Turing $\mathcal{M}$ e un input w, determinare se $\mathcal{M}$ si arresta su w.

$H = \{\langle \mathcal{M}, w\rangle \mid \mathcal{M} \text{ si arresta sull'input } w\}$

Problema dell'equivalenza delle MT

Date due macchine di Turing $\mathcal{M}_1$ e $\mathcal{M}_2$, determinare se calcolano la stessa funzione.

$EQ = \{\langle \mathcal{M}_1, \mathcal{M}_2\rangle \mid \mathcal{M}_1 \text{ e } \mathcal{M}_2 \text{ calcolano la stessa funzione}\}$

Problema della totalità delle MT

Data una macchina di Turing $\mathcal{M}$, determinare se $\mathcal{M}$ si arresta su tutti gli input.

$TOT = \{\langle \mathcal{M}\rangle \mid \mathcal{M} \text{ si arresta su tutti gli input}\}$

Problema del linguaggio vuoto

Data una macchina di Turing $\mathcal{M}$, determinare se il linguaggio riconosciuto da $\mathcal{M}$ è vuoto.

$E = \{\langle \mathcal{M}\rangle \mid L(\mathcal{M}) = \varnothing\}$

Analisi di complessità degli algoritmi

Formule note

$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ $\quad$ $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$ $\quad$ $\sum_{i=1}^{n} i^3 = \frac{n^2(n+1)^2}{4}$

$\sum_{i=1}^{n} x^i = \frac{x^{n+1}-1}{x-1}$ $\quad$ $\sum_{i=1}^{\infty} x^i = \frac{1}{1-x}$ (se $|x| < 1$) $\quad$ $\sum_{i=1}^{n} \log(i) = \log(n!)$

Stirling approx. $n! \sim \sqrt{2\pi n}(\frac{n}{e})^n$ per $n \to \infty$. Da questo si può dimostrare che $\log(n!) \overset{n\to\infty}{\sim} n \log n$.

Definizioni di O-grande, Ω-grande, Θ-grande

O-grande

$O(g(n))$ è l'insieme delle funzioni che sono asintoticamente dominate da $g(n)$ a meno di una costante.

$O(g(n)) = \{f(n) \mid \exists c > 0, n_0 > 0 \text{ tali che } \forall n \geq n_0, f(n) \leq c \cdot g(n)\}$

Ω-grande

$\Omega(g(n))$ è l'insieme delle funzioni che dominano asintoticamente $g(n)$ a meno di una costante.

$\Omega(g(n)) = \{f(n) \mid \exists c > 0, n_0 > 0 \text{ tali che } \forall n \geq n_0, f(n) \geq c \cdot g(n)\}$

Θ-grande

$\Theta(g(n))$ è l'insieme delle funzioni che hanno approssimativamente lo stesso comportamento astintotico di $g(n)$.

$\Theta(g(n)) = \{f(n) \mid \exists c_1 > 0, c_2 > 0, n_0 > 0 \text{ tali che } \forall n \geq n_0,\ c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$

N.B. Spesso si utilizza la notazione $f(n) = O(g(n))$ invece di $f(n) \in O(g(n))$ per indicare che la funzione $f(n)$ appartiene all'insieme di funzioni $O(g(n))$ anche se non è una notazione formalmente corretta.

Criterio di costo costante e logaritmico

Nella valutazione della complessità algoritmica con criterio di costo costante, ogni istruzione ha costo $\Theta(1)$. Con il criterio di costo logaritmico, invece, leggere (`READ`), copiare (`LOAD`), spostare (`STORE`), scrivere (`WRITE`), eseguire somme (`ADD`) e sottrazioni (`SUB`) ha costo $\Theta(\log(n))$, eseguire moltiplicazioni (`MUL`) e divisioni (`DIV`) ha costo $\Theta(\log^2(n))$ e il resto (`JUMP/HALT`) ha costo $\Theta(1)$.

Esempio Calcolo di 2^{2^n} con macchina RAM con criterio di costo logaritmico.

1	`READ 2`	$\log(n)$
2	`LOAD= 2`	$\log(2) = k$
3	`STORE 1`	$\log(2) = k$
4	`loop: LOAD 1`	$\log(2^{2^{n-1}}) = 2^{n-1}$
5	`MUL 1`	$(\log(2^{2^{n-1}}))^2 = (2^{n-1})^2 = 2^{2n-2}$
6	`STORE 1`	$\log(2^{2n}) = 2n$
7	`LOAD 2`	$\log(n)$
8	`SUB= 1`	$\log(n)$
9	`STORE 2`	$\log(n-1)$
10	`JGT loop`	1
11	`WRITE 1`	$\log(2^{2^n}) = 2^n$
12	`HALT`	1

$T(n) = \log(n) + n(2^{n-1} + 2^{2n-2} + 2^n + 3\log(n)) + 2^n = \Theta(n2^{2n-2})$

Ricorrenze

Master theorem

Data l'equazione di ricorrenza $T(n) = aT(\frac{n}{b}) + f(n)$ con $a \geq 1, b > 1$.

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{se } f(n) = O(n^c) \text{ con } c < \log_b a \\ \Theta(n^{\log_b a} \log n) & \text{se } f(n) = \Theta(n^{\log_b a}) \\ \Theta(f(n)) & \text{se } f(n) = \Omega(n^c) \text{ con } c > \log_b a \text{ e } af(\frac{n}{b}) \leq kf(n) \end{cases}$$

Per qualche $k < 1$.

Generalizzazione del caso 2

Se $f(n) = \Theta(n^{\log_b a} \log^k n)$ per qualche $k \in \mathbb{R}$, allora:

$$T(n) = \begin{cases} \Theta(n^{\log_b a} \log^{k+1} n) & \text{se } k > -1 \\ \Theta(n^{\log_b a} \log\log n) & \text{se } k = -1 \\ \Theta(n^{\log_b a}) & \text{se } k < -1 \end{cases}$$

Notiamo che se $k = 0$ ci troviamo nel caso semplice visto sopra.

Sostituzione

Consiste nell'intuire una soluzione, sostituirla nell'equazione di ricorrenze $T(n)$ e verificare che esistano c e n_0 che rispettino la definizione di O-grande.

Esempio Consideriamo $T(n) = 3T(\log(n)) + 2^n$. Intuiamo che $T(n) = O(2^n)$. Verifichiamo che esistano $c > 0, n_0 > 0$ tale che $\forall n \geq n_0,\ T(n) \leq c2^n$ da cui segue che $T(\log(n)) \leq c2^{\log n} = cn$

$T(n) = 3T(\log(n)) + 2^n \leq 3cn + 2^n \leq \frac{c}{2}2^n + 2^n = (\frac{c}{2} + 1)2^n \overset{?}{\leq} c2^n$. Sì, basta prendere $c \geq 2$. $\square$

L'intuizione per la O-grande si può ottenere espandendo le chiamate ricorsive e individuando il termine dominante o trovando due funzioni tra cui $T(n)$ è compresa. In generale, per dimostrare che $T(n) \in \Theta(f(n))$ bisogna dimostrare che $T(n) \in O(f(n))$ e che $T(n) \in \Omega(f(n))$ in modo analogo a quanto visto nell'esempio.

Strutture dati

Vettori

I vettori sono strutture dati che consentono l'accesso diretto ad ogni elemento data la sua posizione.

Operazione	Non ord.	Ord.
`Search`	$O(n)$	$\Theta(\log(n))$
`Minimum`	$O(n)$	$\Theta(1)$
`Maximum`	$O(n)$	$\Theta(1)$
`Successor`	$O(n)$	$\Theta(\log(n))$
`Insert`	$O(n)$	$O(n)$
`Delete`	$O(n)$ (oppure $O(1)$ usando dei simboli di cella vuota)	$O(n)$

L'inserimento in un vettore pieno può essere rifiutato $O(1)$ o può causare una riallocazione $O(n)$.

Liste

Le liste semplici sono strutture dati che memorizzano elementi sparsi in memoria dove ogni elemento contiene un riferimento al successivo.

Operazione	Non ord.	Ord.
`Search`	$O(n)$	$O(n)$
`Minimum`	$O(n)$	$\Theta(1)$ o $\Theta(n)$
`Maximum`	$O(n)$	$\Theta(n)$ o $\Theta(1)$
`Successor`	$O(n)$	$O(n)$
`Insert`	$O(1)$	$O(n)$
`Delete`	$O(n)$	$O(n)$

A seconda di come è ordinata la lista, uno tra `Minimum` e `Maximum` è $\Theta(1)$ e l'altro è $\Theta(n)$.

Pile (Stack)

Le pile sono strutture dati con le seguenti operazioni:

`Push(S,e)`	aggiunge l'elemento `e` in cima alla pila `S`
`Pop(S)`	restituisce e cancella l'elemento in cima alla pila
`Empty(S)`	restituisce `true` se la pila è vuota

Implementazione con vettori

Possono essere implementate con dei vettori memorizzando l'indice della cima della pila (Top of Stack, ToS).

`Push(S,e)`	se c'è spazio incrementa ToS e salva `e` in `A[ToS]` $O(1)$ altrimenti rifiuta $O(1)$ o rialloca $O(n)$
`Pop(S)`	restituisce `A[ToS]` e decrementa `ToS` $O(1)$
`Empty(S)`	restituisce `true` se `ToS=0` $O(1)$

Implementazione con liste

Possono essere implementate con delle liste dove la testa della lista è la cima della pila.

`Push(S,e)`	inserisce `e` in testa alla lista $O(1)$
`Pop(S)`	restituisce e cancella l'elemento in testa alla lista $O(1)$
`Empty(S)`	restituisce `true` se il successore della testa è `NIL` $O(1)$

Code (Queues)

Le code sono strutture dati con le seguenti operazioni:

`Enqueue(Q,e)`	aggiunge l'elemento `e` alla fine della coda `Q`
`Dequeue(Q)`	restituisce e cancella l'elemento all'inizio della coda
`Empty(Q)`	restituisce `true` se la coda è vuota

Implementazione con vettori

Possono essere implementate con dei vettori tenendo traccia della posizione dove va inserito un nuovo elemento e di quella dell'elemento più vecchio con due indici `tail` e `head` e del numero di elementi contenuti n. Sia A il vettore e $l := A.length$

`Enqueue(Q,e)`	se $n < l$ salva `e` in `A[tail]` e incrementa n e `tail` $O(1)$ altrimenti rifiuta $O(1)$ o rialloca $\Theta(n)$
`Dequeue(Q)`	restituisce `A[head]`, decrementa n e incrementa `head` $O(1)$
`Empty(Q)`	restituisce `true` se $n = 0$ $O(1)$

Implementazione con liste

Possono essere implementate con delle liste tenendo traccia sia della testa (`head`) che della coda (`tail`) della lista.

`Enqueue(Q,e)`	inserisce l'elemento `e` in coda alla lista $O(1)$
`Dequeue(Q)`	restituisce e cancella l'elemento in testa alla lista $O(1)$
`Empty(Q)`	restituisce `true` se `head=tail` $O(1)$

Code doppie (Double-ended queues o Deques)

Le code doppie sono strutture dati in cui è possibile inserire sia in testa che in coda. Sono dotate delle seguenti operazioni:

`PushFront(Q,e)`	inserisce l'elemento `e` in testa
`PushBack(Q,e)`	inserisce l'elemento `e` in coda
`PopFront(Q)`	restituisce e cancella l'elemento in testa
`PopBack(Q)`	restituisce e cancella l'elemento in coda
`Empty(Q)`	restituisce `true` se la deque è vuota

Implementazione con vettori

`PushBack`, `PopFront` e `Empty` sono analoghi a `Enqueue` e `Dequeue` della coda realizzata con vettore.

`PushFront(Q,e)`	se $n < l$ decr. `head`, ins. `e` in `A[head]` e incr. n $O(1)$ altrimenti segnala l'errore $O(1)$
`PopBack(Q)`	se $n > 0$, restituisce `A[tail]` e decr. n e `tail` $O(1)$

Implementazione con liste *doppiamente concatenate*

`PushBack` e `PopFront` sono analoghi a `Enqueue` e `Dequeue` della coda realizzata con lista. Una lista vuota è rappresentata con `head` e `tail` che puntano l'uno all'altro.

`PushFront(Q,e)`	ins. `e` in testa aggiornando `head` e il suo succ. $O(1)$
`PopBack(Q)`	restituisce e cancella `tail.prev` $O(1)$
`Empty(Q)`	se $n > 0$, restituisce `true` se `head.next=tail` $O(1)$

Dizionari

I dizionari sono strutture dati che contengono elementi accessibili direttamente data la loro chiave. Supportano le operazioni: `Insert`, `Delete`, `Search` e sono implementati come vettori di puntatori dove le chiavi vengono usate come indici del vettore.

`Insert(D,e)`	`D[e.key]` $\leftarrow$ `e`	$\Theta(1)$
`Delete(D,e)`	`D[e.key]` $\leftarrow$ `NIL`	$\Theta(1)$
`Search(D,e.key)`	**return** `D[e.key]`	$\Theta(1)$

La complessità spaziale è $O(|\mathbf{D}|)$ con $\mathbf{D}$ il dominio delle chiavi.

Tabelle di hash

Una tabella di hash è un caso speciale di dizionario con una complessità spaziale pari al numero di chiavi m per cui è effettivamente presente un valore. L'indice associato a una certa chiave è dato dal risultato di una funzione $h(\cdot) : \mathbf{D} \to \{0, \ldots, m-1\}$ detta funzione di hash.

Def. Il fattore di carico α è definito come $\frac{n}{m}$ dove n è il numero di elementi contenuti nella tabella e m è il numero di righe totali.

Collisioni

Date due chiavi k_1, k_2 con $k_1 \neq k_2$ abbiamo una collisione se $h(k_1) = h(k_2)$. Le collisioni si risolvono principalmente con due metodi.

Indirizzamento chiuso (closed addressing/open hashing/chaining)

Ogni riga della tabella (bucket) contiene la testa di una lista. Nel caso di collisione, l'elemento viene inserito in testa alla lista $\Theta(1)$. Nel caso peggiore tutti gli elementi collidono dando origine a una lista lunga m elementi, quindi `Insert/Delete/Search` in $O(m)$.

Def. Ipotesi di Hashing Uniforme Semplice (IHUS) è un'oppurtuna funzione di hash dove ogni chiave ha probabilità $\frac{1}{m}$ di finire in una qualsiasi delle m celle.

In caso di IHUS, la lunghezza media di una listà è α e il tempo medio per cercare una chiave è $\Theta(1 + \alpha)$. Se α non è eccessivo, tutte le operazioni sono $O(1)$ in media. Per mantenere l'efficienza delle operazioni, si sceglie un α oltre il quale viene fatta una riallocazione in una tabella più grande (**rehashing**). (i solito $\alpha_{max} \simeq 0.75$)

Indirizzamento aperto (open addressing/closed hashing)

`Insert`: Consiste nel selezionare deterministicamente l'indirizzo di un altro bucket in caso di collisione (ispezione). Se non ci sono bucket liberi, l'inserimento fallisce $\Theta(m)$ e viene fatto il rehashing. Si modifica `Search` affinché effettui la stessa ispezione in caso l'elemento non sia presente nel suo bucket. `Delete` si effettua inserendo un opportuno valore (tombstone) che non corrisponde ad alcuna chiave.

Complessità Sotto IHUS, cercare un valore non presente costa $\Theta(\frac{1}{1-\alpha})$, mentre cercare un valore presente costa $\Theta(\frac{1}{\alpha}\log(\frac{1}{1-\alpha}))$.

Ispezione lineare (linear probing): Si sceglie $h(k, i) = a + c \cdot i \mod m$ come candidato per l'i-esimo inserimento.
Se ci sono molte collisioni in un bucket, aumenta la probabilità di collisione nelle vicinanze (**clustering primario**).

Ispezione quadratica (quadratic probing): Si sceglie $h(k, i) = a + c_1 i + c_2 i^2 \mod m$ come candidato per l'i-esimo inserimento. Rimuove il clustering primario ma non garantisce che tutte le celle vengano toccate dalla sequenza di ispezione.

Doppio hashing (passo di ispezione dipende dalla chiave) $h(k, i) = h_1(k) + h_2(k)i$. Per garantire che tutte le celle vengano toccate dalla sequenza di ispezione $h_2(k)$ deve essere coprimo con m. In particolare:

- $m = 2^x \leftrightarrow h_2$ genera solo num. dispari
- m primo $\leftrightarrow h_2$ genera numero $< m$
- h_2 non deve mai restituire 0 altrimenti la sequenza di ispezione degenera

Esempio Sia T una tabella di hash con 31 celle e $\alpha = 3/4$. Vogliamo reindirizzare i dati in una tabella T' di modo che $\alpha' \leq 1/2$. Qual è il numero minimo m' di celle che è opportuno che T' abbia? Si assuma che $h_{T'}(k, i) = (k + i(2k + 1)) \mod m'$.

Sol. Essendo $h_2(k)$ sempre dispari, scegliamo m' come potenza di 2 che soddisfi la restrizione imposta su α':

$\alpha' = \frac{\#\text{ celle occupate}}{\#\text{ celle disponibili}} = \frac{3/4*31}{m'} \leq \frac{1}{2} \iff m' \geq \frac{3}{2} * 31 = 46.5$

Scegliamo quindi $m' = 64$.

Aritmetica modulare

$\frac{a}{n}$ stesso resto di $\frac{r}{n} \iff a \equiv r \mod n \iff n \mid (a - r) \iff a = qn + r$.

Proprietà

$(a \pm b) \mod n = [(a \mod n) \pm (b \mod n)] \mod n$

$(a \times b) \mod n = [(a \mod n) \times (b \mod n)] \mod n$

Alberi

Un albero è una struttura dati rappresentata da una coppia $(\mathbf{V},\mathbf{E})$ con $\mathbf{V}$ insieme di nodi e $\mathbf{E}$ insieme di archi dove ogni nodo può apparire un'unica volta come destinazione di un arco.

Definizioni

`radice`	l'unico nodo privo di un arco entrante
`foglia`	nodo senza archi uscenti
`padre` di n	nodo da cui l'arco entrante in n ha origine
`figlio` di n	nodo in cui uno degli archi uscenti da n termina
`livello`	distanza di un nodo dalla radice
`albero completo`	albero in cui tutti i livelli hanno tutti i nodi tranne l'ultimo che può non avere tutti i nodi ma i nodi devono essere più a sinistra possibile

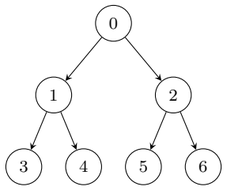
Alberi binari

Un albero binario è un albero in cui ogni nodo ha al più due figli. Dato un nodo `A`:

`A.left`	ref. a figlio sinistro	`A.right`	ref. a figlio destro
`A.p`	ref. al padre	`A.root`	ref. alla radice

`A.p` è `NIL` solo per la radice.

Implementazione con vettore



0	1	2	3	4	5	6

- La radice ha indice 0.
- Dato un nodo con indice i, il suo figlio sinistro ha indice $2i+1$ e il suo figlio destro ha indice $2i+2$
- Il padre del nodo con indice i ha indice $\lfloor \frac{i-1}{2} \rfloor$

Visita di un albero

Le principali procedure di visita di un albero sono InOrder, PreOrder e PostOrder.

InOrder(T)
```
1 if T ≠ NIL
2   InOrder(T.left)
3   Print(T.key)
4   InOrder(T.right)
5 return
```
3, 1, 4, 0, 5, 2, 6

PreOrder(T)
```
1 if T ≠ NIL
2   Print(T.key)
3   PreOrder(T.left)
4   PreOrder(T.right)
5 return
```
0, 1, 3, 4, 2, 5, 6

PostOrder(T)
```
1 if T ≠ NIL
2   PostOrder(T.left)
3   PostOrder(T.right)
4   Print(T.key)
5 return
```
3, 4, 1, 5, 6, 2, 0

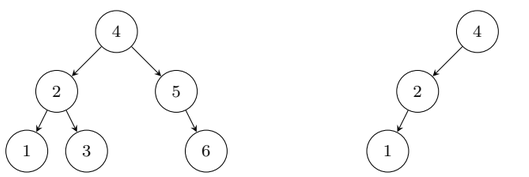
Tutte queste procedure hanno complessità temporale $\Theta(n)$ in quanto toccano ogni nodo dell'albero esattamente una volta.

Alberi binari di ricerca (BST)

Un albero binario di ricerca è un albero binario che, per ogni nodo x, soddisfa le sequenti condizioni:

- Se y è contenuto nel sottoalbero sinistro di x, allora $y.key \leq x.key$
- Se y è contenuto nel sottoalbero destro di x, allora $y.key \geq x.key$

Esempi



Operazioni su BST

Search(T, x) : $O(h)$
```
1 if T = NIL or T.key = x.key
2   return T
3 if T.key < x.key
4   return Search(T.right, x)
5 else return Search(T.left, x)
```

Min(T) : $O(h)$
```
1 cur ← T
2 while cur.left ≠ NIL
3   cur ← cur.left
4 return cur
```

Max(T) : $O(h)$
```
1 cur ← T
2 while cur.right ≠ NIL
3   cur ← cur.right
4 return cur
```

Successor(x) : $O(h)$
```
1 if x.right ≠ NIL
2   return Min(x.right)
3 y ← x.p
4 while y ≠ NIL and y.right = x
5   x ← y
6   y ← y.p
7 return y
```

Insert(T, x) : $O(h)$
```
1  pre ← NIL
2  cur ← T.root
3  while cur ≠ NIL
4    pre ← cur
5    if x.key < cur.key
6      cur ← cur.left
7    cur ← cur.right
8  x.p ← pre
9  if pre = NIL
10   T.root ← x
11 elseif x.key < pre.key
12   pre.left ← x
13 else pre.right ← x
```

Delete(T,x) : $O(h)$
```
1  if x.left = NIL or x.right = NIL
2    del ← x
3  else del ← Successor(x)
4  if del.left ≠ NIL
5    subtree ← del.left
6  else subtree ← del.right
7  if subtree ≠ NIL
8    subtree.p ← del.p
9  if del.p = NIL
10   T.root ← subtree
11 elseif del = del.p.left
12   del.p.left ← subtree
13 else del.p.right ← subtree
14 if del ≠ x
15   else x.key ← del.key
16 Free(del)
```

La complessità di tutte le operazioni è $O(h)$ con h altezza dell'albero. Nel caso ottimo (albero bilanciato) è $O(\log(n))$, nel caso pessimo (albero degenere in una lista) è $O(n)$.

Alberi rosso-neri (Red-black trees / RB trees)

Gli alberi rosso-neri sono BST i cui nodi hanno un colore $\in$ {rosso, nero} e che soddisfano le seguenti proprietà:

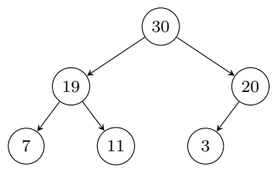
1. Ogni nodo è rosso o nero
2. La radice è nera
3. Le foglie sono nere
4. I figli di un nodo rosso sono entrambi neri
5. Per ogni nodo dell'albero, tutti i cammini dai suoi discendenti alle foglie contenute nei suoi sottoalberi hanno lo stesso numero di nodi neri

Le operazioni che non modificano la struttura dell'albero sono le stesse dei BST (Search, Min, Max, Successor). Le procedure di inserimento (Insert) e cancellazione (Delete) sono modificate per mantenere l'albero bilanciato assicurando una complessità temporale di $O(\log(n))$ per tutte le operazioni.

Heap binari

Gli heap binari sono alberi binari quasi-completi dove la chiave del nodo padre è sempre maggiore o minore di quella dei figli. Nel primo caso si parla di max-heap nel secondo di min-heap.

Esempio (max-heap)



Operazioni su max-heap

MaxHeapify(A, n) riceve un array e una posizione in esso: assume che i due sottoalberi con radice in Left(n) $= 2n$ e Right(n) $= 2n+1$ siano dei max-heap e modifica A in modo che l'albero radicato in n sia un max-heap.

MaxHeapify(A, n) : $O(\log(n))$
```
1  l ← Left(n)
2  r ← Right(n)
3  if l ≤ A.heapsize and A[l] > A[n]
4    posmax ← l
5  else posmax ← n
6  if r ≤ A.heapsize and A[r] > A[posmax]
7    posmax ← r
8  if posmax ≠ n
9    Swap(A[n], A[posmax])
10   MaxHeapify(A, posmax)
```

BuildMaxHeap(A) : $O(n)$
```
1 A.heapsize ← A.length
2 for i ← ⌊A.length/2⌋ downto 1
3   MaxHeapify(A, i)
```

Max(A) : $O(1)$
```
1 return A[1]
```

ExtractMax(A) : $O(\log(n))$
```
1  if A.heapsize < 1
2    return ⊥
3  max ← A[i]
4  A[1] ← A[A.heapsize]
5  A.heapsize ← A.heapsize − 1
6  MaxHeapify(A, 1)
7  return max
```

Insert(A, key) : $O(\log(n))$
```
1 A.heapsize ← A.heapsize + 1
2 A[A.heapsize] ← key
3 i ← A.heapsize
4 while i > 1 and A[Parent(i)] < A[i]
5   Swap(A[Parent(i)], A[i])
6   i ← Parent(i)
```

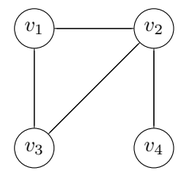
Grafi

Un grafo è una coppia $\mathcal{G} = (\mathbf{V}, \mathbf{E})$ con $\mathbf{V}$ un insieme di nodi (detti anche vertici) ed $\mathbf{E}$ un insieme di archi (detti anche lati).

Def.

- Un grafo con $|\mathbf{V}|$ nodi ha al più $|\mathbf{V}|^2$ archi
- Due nodi collegati da un arco si dicono *adiacenti*
- Un grafo è detto *orientato* (directed) se gli archi (arcs) che collegano i nodi sono orientati
- Un grafo è detto *connesso* se esiste un percorso per ogni coppia di nodi
- Un grafo è detto *completo* (o *completamente connesso*) se esiste un arco per ogni coppia di nodi.
- Un percorso è un *ciclo* se il nodo di inizio e fine coincidono. Un grafo privo di cicli si dice *aciclico*
- Un *cammino* tra due nodi v_1, v_2 è un insieme di archi di cui il primo ha origine in v_1, l'ultimo termina in v_2 e ogni nodo compare almeno una volta sia come destinazione di un arco che come sorgente

Esempio Cammino $v_3 \to v_4 : (v_3, v_2), (v_2, v_4)$



$\mathbf{V} = \{v_1, v_2, v_3, v_4\}$

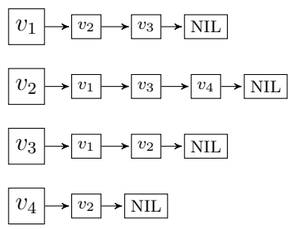
$\mathbf{E} = \{(v_1, v_2), (v_1, v_3), (v_2, v_3), (v_2, v_4), (v_2, v_1), (v_3, v_1), (v_3, v_2), (v_4, v_2)\}$

Rappresentazione in memoria

I grafi vengono rappresentati in memoria con *liste di adiacenza* o *matrice di adiacenza*.

Liste di adiacenza

Vettore di liste lungo $|\mathbf{V}|$, indicizzato dai nomi dei nodi dove ogni lista contiene i nodi adiacenti all'indice della sua testa.



Matrice di adiacenza

Matrice $|\mathbf{V}| \times |\mathbf{V}|$ di valori booleani con righe e colonne indicizzate dai nomi dei nodi dove la cella alla riga i e colonna j contiene 1 se l'arco $(v_i, v_j) \in \mathbf{E}$ e 0 altrimenti.

$$\begin{array}{c} \\ v_1 \\ v_2 \\ v_3 \\ v_4 \end{array} \begin{array}{c} \begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \end{array} \\ \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \end{array}$$

Confronto

	Liste	Matrice
Complessità spaziale	$\Theta(\lvert\mathbf{V}\rvert + \lvert\mathbf{E}\rvert)$	$\Theta(\lvert\mathbf{V}\rvert^2)$
Determinare se $(v_1, v_2) \in \mathbf{E}$	$O(\lvert\mathbf{V}\rvert)$	$O(1)$
Num. di archi o_e uscenti da un nodo	$\Theta(o_e)$	$O(\lvert\mathbf{V}\rvert)$

Se il grafo è denso, cioè $|\mathbf{E}| \approx |\mathbf{V}|^2$, è conveniente usare la matrice di adiacenza. Se il grafo è sparso, cioè $|\mathbf{E}| \ll |\mathbf{V}|^2$, è conveniente usare le liste di adiacenza.

Operazioni su grafi

Visita in ampiezza (BFS o Breadth-first search)

La strategia di visita in ampiezza visita tutti i nodi di un grafo $\mathcal{G}$ a partire da un nodo sorgente s. I nodi vengono visitati in ordine di distanza, dove i nodi più vicini al nodo sorgente vengono visitati per primi.

VisitaAmpiezza(G, s) : $O(|\mathbf{V}| + |\mathbf{E}|)$
```
1  for each n ∈ V \ {s}
2    n.color ← white
3    n.dist ← ∞
4  s.color ← grey
5  s.dist ← 0
6  Q ← ∅
7  Enqueue(Q, s)
8  while ¬IsEmpty(Q)
9    cur ← Dequeue(Q)
10   for each v ∈ cur.adiacenti
11     if v.color = white
12       v.color ← grey
13       v.dist ← cur.dist + 1
14       Enqueue(Q, v)
15   cur.color ← black
```

Visita in profondità (DFS o Depth-first search)

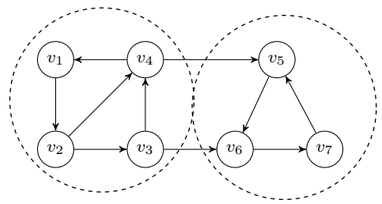
La strategia di visita in ampiezza visita tutti i nodi di un grafo $\mathcal{G}$ a partire da un nodo sorgente s. I nodi vengono visitati in profondità, cioè che si visita un nodo fino a quando non si raggiungono i nodi più lontani (in profondità) rispetto al nodo sorgente. Il codice è uguale a quello per il BFS usando una pila invece che una coda.

VisitaProfondità(G, s) : $O(|\mathbf{V}| + |\mathbf{E}|)$
```
1  for each n ∈ V \ {s}
2    n.color ← white
3    n.dist ← ∞
4  s.color ← grey
5  s.dist ← 0
6  S ← ∅
7  Push(S, s)
8  while ¬IsEmpty(S)
9    cur ← Pop(S)
10   for each v ∈ cur.adiacenti
11     if v.color = white
12       v.color ← grey
13       v.dist ← cur.dist + 1
14       Push(S, v)
15   cur.color ← black
```

Componenti connesse

Una *componente connessa* di un grafo è un insieme $\mathbf{S}$ di nodi tali per cui esiste un cammino tra ogni coppia di essi e nessuno di essi è connesso a nodi $\notin \mathbf{S}$.

Esempio Componenti connesse $S_1 = \{v_1, v_2, v_3, v_4\}, S_2 = \{v_5, v_6, v_7\}$



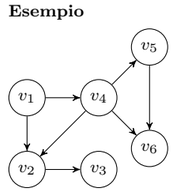
ComponentiConnesse(G) : $O(|\mathbf{V}| + |\mathbf{E}|)$
```
1 for each v ∈ V
2   v.etichetta ← −1
3 eti ← 1
4 for each v ∈ V
5   if v.etichetta = −1
6     VisitaEdEtichetta(G, v, eti)
7     eti ← eti + 1
```

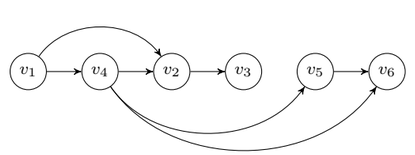
VisitaEdEtichetta funziona come VisitaAmpiezza o VisitaProfondità ma imposta a *eti* il campo *etichetta* del nodo visitato.

Ordinamento topologico

L'*ordinamento topologico* è una sequenza di nodi di un grafo *orientato aciclico* tale per cui nessun nodo compare prima di un suo predecessore.

Esempio





OrdinamentoTopologico(G)
```
1 for each v ∈ V
2   v.color ← white
3 for each v ∈ V
4   if v.color = white
5     VisitaProfOT(G, v, l)
6 return L
```

VisitaProfOT(G, s, L)
```
1 s.color ← grey
2 for each v ∈ cur.adiacenti
3   if v.color = white
4     VisitaProfOT(G, v, L)
5 s.color ← black
6 PushFront(L, s)
```

Percorso più breve

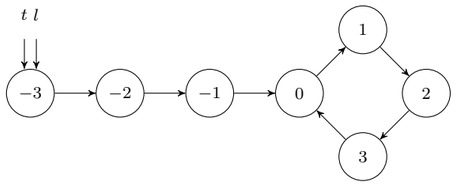
Dato un grafo e un suo nodo s, individua i percorsi più brevi da s a qualunque altro nodo.

DijkstraQueue(G, s) : $O((|\mathbf{V}| + |\mathbf{E}|)\log(|\mathbf{V}|))$
```
1  Q ← ∅
2  s.dist ← 0
3  for each v ∈ |V|
4    if v ≠ s
5      v.dist ← ∞
6      v.pred ← NIL
7    AccodaPri(Q, v, v.dist)
8  while Q ≠ ∅
9    u ← CancellaMin(Q)
10   for each v ∈ u.succ
11     ndis ← u.dist + peso(u, v)
12     if v.dist > ndis
13       v.dist ← ndis
14       u.prev ← u
15       DecrementaPri(Q, v, ndis)
```

Rilevamento cicli (Floyd lepre e tartaruga)

Dato un grafo orientato, per cui ogni nodo ha un solo successore determina, dato un nodo di partenza s, se il cammino che parte da s ha cicli.

Grafo di esempio



L'idea è quella di usare due riferimenti t e l che partono dal nodo iniziale (-3 nell'esempio) che vengono spostati a ogni passo. Nello specifico, t viene spostato dal nodo a cui punta al successore e l viene spostato dal nodo a cui punta al successore del successore. È garantito che se c'è un ciclo, i due riferimenti si incontreranno.

Sia C la lunghezza del ciclo (4 nell'esempio) e T la lunghezza della "coda" che lo precede (3 nell'esempio), allora l'algoritmo è il seguente:

FloydLT(G, x) : $\Theta(2(T + C) - r)$
```
1  t ← x.succ
2  l ← x.succ.succ
3  while l ≠ t
4    t ← t.succ
5    l ← l.succ.succ
6  T ← 0
7  t ← x
8  while l ≠ t
9    t ← t.succ
10   l ← l.succ
11   T ← T + 1
12 l ← t
13 C ← 0
14 while l ≠ t
15   l ← l.succ
16   C ← C + 1
17 return T, C
```

Algoritmi di ordinamento

Bubble sort

Bubble Sort funziona passando ripetutamente attraverso l'array da ordinare, confrontando gli elementi adiacenti e scambiandoli se sono nell'ordine sbagliato. Prosegue attraverso l'array più volte fino a quando non sono più necessari scambi, indicando che l'array è ordinato.

BubbleSort(A) : $O(n^2)$
```
1 for i ← 1 to A.length − 1
2   for j ← 1 to A.length − i
3     if A[j] > A[j + 1]
4       Swap(A[j], A[j + 1])
```

Selection sort

Selection Sort seleziona ripetutamente l'elemento più piccolo dall'array non ordinato e lo scambia con l'elemento nella posizione corrente. Continua questo processo, aumentando progressivamente la porzione ordinata dell'array fino a quando l'intero array è ordinato.

SelectionSort(A) : $O(n^2)$
```
1 for i ← 1 to A.length − 1
2   min_index ← i
3   for j ← i + 1 to A.length
4     if A[j] < A[min_index]
5       min_index ← j
6   Swap(A[i], A[min_index])
```

Insertion sort

Insertion Sort costruisce l'array ordinato uno alla volta. Prende ciascun elemento dall'input e lo inserisce nella sua posizione corretta rispetto agli elementi già ordinati. Questo processo continua fino a quando l'intero array è ordinato.

InsertionSort(A) : $O(n^2)$
```
1 for i ← 2 to A.length
2   tmp ← A[i]
3   j ← i − 1
4   while j ≥ 1 and A[j] > tmp
5     A[j + 1] ← A[j]
6     j ← j − 1
7   A[j + 1] ← tmp
```

Merge sort

Merge Sort è un algoritmo di divide et impera. Divide l'array di input in due metà, ordina ricorsivamente ciascuna metà e poi fonde le due metà ordinate in un singolo array ordinato.

MergeSort(A, p, r) : $\Theta(n \log n)$
```
1 if p < r − 1
2   q ← ⌊(p+r)/2⌋
3   MergeSort(A, p, q)
4   MergeSort(A, q + 1, r)
5   Merge(A, p, q, r)
6   else if A[p] > A[r]
7     tmp ← A[r]
8     A[r] ← A[p]
9     A[p] ← tmp
```

Merge(A, p, q, r) : $\Theta(n)$
```
1  len1 ← q − p + 1
2  len2 ← r − q
3  Alloca(L[1..len1 + 1])
4  Alloca(R[1..len2 + 1])
5  for i ← 1 to len1
6    L[i] ← A[p + i − 1]
7  for i ← 1 to len2
8    R[i] ← A[q + i]
9  L[len1 + 1] ← ∞
10 R[len2 + 1] ← ∞
11 i ← 1
12 j ← 1
13 for k ← p to r
14   if L[i] ≤ R[j]
15     A[k] ← L[i]
16     i ← i + 1
17   else
18     A[k] ← R[j]
19     j ← j + 1
```

Quick sort

Quick Sort è anch'esso un algoritmo di divide et impera. Sceglie un elemento pivot e suddivide l'array intorno a esso, in modo che gli elementi più piccoli del pivot siano a sinistra e quelli più grandi a destra. Ordina ricorsivamente le sotto-liste formate dalla partizione fino a quando l'intero array è ordinato.

Quick sort con schema di partizione di Lomuto

Lo schema di partizione di Lomuto seleziona l'ultimo elemento dell'array come pivot. Successivamente, scorre l'array e posiziona gli elementi più piccoli del pivot a sinistra e quelli più grandi a destra. Infine, sposta il pivot nella sua posizione corretta tra i due gruppi.

Quicksort(A, lo, hi)
```
1 se lo < hi
2   p ← PartLomuto(A, lo, hi)
3   Quicksort(A, lo, p − 1)
4   Quicksort(A, p + 1, hi)
```

PartLomuto(A, lo, hi)
```
1  pivot ← A[hi]
2  i ← lo − 1
3  for j ← lo a hi − 1
4    if A[j] ≤ pivot
5      i ← i + 1
6      Swap(A[i], A[j])
7  Swap(A[i + 1], A[hi])
8  return i + 1
```

Quick sort con schema di partizione di Hoare

Lo schema di partizione di Hoare seleziona il primo elemento dell'array come pivot. Utilizza due indici, uno che procede dall'inizio dell'array verso destra e uno che procede dalla fine dell'array verso sinistra, scambiando gli elementi fuori posizione fino a quando i due indici si incontrano. Alla fine, il pivot viene inserito tra i due gruppi e l'indice di partizione è determinato.

Quicksort(A, lo, hi)
```
1 se lo < hi
2   p ← PartHoare(A, lo, hi)
3   Quicksort(A, lo, p)
4   Quicksort(A, p + 1, hi)
```

PartHoare(A, lo, hi)
```
1  pivot ← A[lo]
2  i ← lo − 1
3  j ← hi + 1
4  while true
5    repeat
6      j ← j − 1
7    until A[j] ≤ pivot
8    repeat
9      i ← i + 1
10   until A[i] ≥ pivot
11   if i < j
12     Swap(A[i], A[j])
13   else return j
```

Lo schema di partizione di Hoare effettua in media $\frac{1}{3}$ degli scambi effettuati dallo schema di partizione di Lomuto. (Entrambi hanno complessità $\Theta(n)$)

Heap sort

Heap Sort ordina l'array costruendo un max heap. Successivamente, estrae ripetutamente l'elemento massimo dall'heap e lo posiziona alla fine dell'array. Questo processo continua fino a quando l'intero array è ordinato.

HeapSort(A): $\Theta(n \log n)$
```
1 BuildMaxHeap(A)
2 for i ← A.length to 2
3   swap (A[1], A[i])
4   A.heapsize ← A.heapsize − 1
5   MaxHeapify(A, 1)
```

Counting sort

Counting Sort è un algoritmo usato per ordinare un array di interi quando si conosce in anticipo il range dei valori possibili. Calcola un istogramma delle occorrenze di ciascun valore, quindi costruisce l'array ordinato posizionando ogni elemento nella sua posizione corretta basata sul conteggio accumulato degli elementi precedenti.

Non stabile

CountingSort(A) : $O(n + k)$
```
1  Ist[0..k] ← 0
2  for i ← 0 to A.length − 1
3    Ist[A[i]] ← Ist[A[i]] + 1
4  idxA ← 0
5  for i ← 0 to k
6    while Ist[i] > 0
7      A[idxA] ← i
8      idxA ← idxA + 1
9      Ist[i] ← Ist[i] − 1
```

Stabile

CountingSort(A) : $O(n + k)$
```
1  B[0..A.length − 1] ← 0
2  Ist[0..k] ← 0
3  for i ← 0 to A.length − 1
4    Ist[A[i]] ← Ist[A[i]] + 1
5  sum ← 0
6  for i ← 0 to k
7    sum ← sum + Ist[i]
8    Ist[i] ← sum
9  for i ← A.length − 1 to 0
10   idx ← Ist[A[i]]
11   B[idx − 1] ← A[i]
12   Ist[A[i]] ← Ist[A[i]] − 1
13 return B
```

Confronto

Confronto delle complessità temporali e spaziali degli algoritmi di ordinamento:

Algoritmo	Stabile?	$T(n)$ (pessimo)	$T(n)$ (ottimo)	$S(n)$
Bubble	✓	$\Theta(n^2)$ (***inv.***)	$\Theta(n)$ (***ord.***)	$O(1)$
Selection	×	$\Theta(n^2)$	$\Theta(n^2)$	$O(1)$
Insertion	✓	$\Theta(n^2)$ (***inv.***)	$\Theta(n)$ (***ord.***)	$O(1)$
Merge	✓	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n)$
Quick	×	$\Theta(n^2)$ (***ord.***)	$\Omega(n \log n)$	$O(1)$
Heap	×	$\Theta(n \log n)$	$\Theta(n \log n)$	$O(1)$
Counting	✓	$O(n + k)$	$O(n + k)$	$O(n+k)$

N.B. Il quick sort nel caso medio ha complessità $\Theta(n \log n)$