Introduzione

Informatica $\rightarrow$ studio dell'informazione e della sua elaborazione automatica
Cosa è calcolabile? *Teoria della calcolabilità*
Quante risorse chiede una computazione? *Teoria della complessità*

Funzioni

Una funzione è una relazione che associa ad un elemento $a \in A$ al più un elemento $b \in B$:

$$f : A \rightarrow B$$

$$f(a) = b$$

Una funzione opera da un insieme dominio ad un insieme codominio. Il sottoinsieme del codominio degli oggetti in relazione con almeno un elemento del dominio secondo la funzione è detto *immagine* della funzione: $Im_f = \{b \in B : \exists a, f(a) = b\} = \{f(a) : a \in A\}$. Solitamente vale $Im_f \subseteq B$.

Classi di funzioni

Iniettiva

$$f : A \rightarrow B \text{ iniettiva} \iff \forall a_1, a_2 \in A, a_1 \neq a_2 \implies f(a_1) \neq f(a_2)$$

Suriettiva

$$f : A \rightarrow B \text{ suriettiva} \iff \forall b \in B, \exists a \in A : f(a) = b$$

$$f : A \rightarrow B \text{ suriettiva} \iff Im_f = B$$

Biiettiva

$$f : A \rightarrow B \text{ biiettiva} \iff f \text{ iniettiva} \land f \text{ suriettiva} \implies$$

$$\forall b \in B, \exists! a \in A : f(a) = b$$

Composizione di funzioni

Date $f : A \rightarrow B$ e $g : B \rightarrow C$, f *composto* g è la funzione $g \circ f : A \rightarrow C$ definita come $g \circ f(a) = g(f(a))$.
La composizione non è un operatore commutativo ma, quando domini e codomini lo permettono, è *associativo*.

Funzione inversa

Data $f : A \to B$ biiettiva, la sua *inversa* è l'unica funzione $f^{-1} : B \to A$ che soddisfa

$$f^{-1}(b) = a \iff f(a) = b$$

oppure

$$f^{-1} \circ f = i_A \land f \circ f^{-1} = i_A$$

Funzioni parziali e totali

$f(a)\downarrow$ indica che la funzione è definita sull'elemento a, ovvero che la sua applicazione porta ad un valore definito del codominio.
Per contro $f(a)\uparrow$ indica che la funzione non è definita sull'elemento a.
Una funzione definita su tutto A è detta *totale*:
f totale $\iff \forall a \in A, f(a)\downarrow$. È invece detta *parziale* se a qualche elemento di A associo un elemento di B.

Dominio e campo di esistenza

$Dom_f = \{a \in A : f(a)\downarrow\} \subseteq A$

- $Dom_f \subset A \implies f$ parziale
- $Dom_f = A \implies f$ totale

"Totalizzazione" di una funzione parziale

$f : A \to B$ parziale $\to f : A \to B \cup \{\perp\}$ totale

$$f(a) = \begin{cases} f(a) & a \in Dom_f \\ \perp & \text{altrimenti} \end{cases}$$

Scriviamo per abbreviare $B \cup \{\perp\} \to B_\perp$

Funzione di valutazione

Dati A, B e $B_\perp^A$, si definisce la seguente *funzione di valutazione*:

$$w : B_\perp^A \times A \to B$$

$$w(f, a) = f(a)$$

- Tenendo fisso a provo tutte le funzioni su a
- Tenendo fisso f ottengo il grafico di f

Sistema di calcolo

Un sistema di calcolo $\mathbb{C}$ è un'oggetto che presi in ingresso un programma P e dei dati x produce in uscita l'output y o $\perp$ ottenuto

dall'esecuzione di P su x.
Un programma $P \in \text{DATI}_\perp^{\text{DATI}}$ è una sequenza di regole che trasformano un dato di input in un dato di output (o va in loop) $\implies P \in \text{DATI}_\perp^{\text{DATI}}$ è una funzione (in un linguaggio).
$\mathbb{C} : \text{DATI}_\perp^{\text{DATI}} \times \text{DATI} \to \text{DATI}_\perp$ è una funzione di valutazione e $\mathbb{C}(P, x)$ è la funzione calcolata da P, ovvero è la *semantica* di P.

Potenza computazionale di un sistema di calcolo

La *potenza computazionale* di un sistema di calcolo indica cosa il sistema di calcolo è in grado di calcolare: ci chiediamo se la semantica del programma P sia un sottoinsieme proprio o improprio delle funzioni che dai dati vanno nei dati.

$$F(\mathbb{C}) = \{\mathbb{C}(P, \_) : P \in \text{PROG}\} \subseteq \text{DATI}_\perp^{\text{DATI}}$$

Stabilire cosa può fare l'informatica significa studiare il carattere dell'inclusione: se è impropria, ovvero se la semantica coincide con l'insieme delle funzioni da DATI a DATI, allora l'informatica può tutto, altrimenti se è propria alcuni problemi non sono automatizzabili.

Cardinalità di insiemi infiniti

Relazione

Relazione binaria su A: $R \subseteq A \times A$. Gli elementi $a, b \in A$ stanno in relazione R se e solo se $(a, b) \in R$ e scriviamo aRb o $a \not R b$.

Relazione di equivalenza

$R \subseteq A^2$ è una relazione di *equivalenza* se e solo se è:

1. riflessiva: $\forall a, aRa$;
2. simmetrica: $\forall a, b, aRb \iff bRa$;
3. transitiva: $\forall a, b, c, aRb \land bRc \implies aRc$.

Definiamo classe di equivalenza $[a]_R$ per una relazione d'equivalenza R l'insieme degli elementi che sono in relazione con a:
$[a]_R = \{b \in A : aRb\}$.
Ogni classe di equivalenza induce una *partizione* sull'insieme dominio, ovvero individua $A_1, \dots, A_n \subseteq A$ tali che

1. $A_i \neq \emptyset$;
2. $i \neq j \implies A_i \cap A_j = 0$;

3. $\bigcup_{i\geq 1} A_i = A$.

Si dimostra facilmente che:

1. Non esistono classi di equivalenza vuote;
2. Dati $a, b \in A$, vale $[a]_R \cap [b]_R = \emptyset$ oppure $[a]_R = [b]_R$;
3. $\bigcup_{a\in A} [a]_R = A$.

L'insieme A partizionato in tal modo è detto *insieme quoziente* di A rispetto ad R ed è denotato da A/R.

Isomorfismo

Due insiemi sono *isomorfi* (o equinumerosi) se esiste una biiezione tra essi. Formalmente scriviamo $A \sim B$. L'isomorfismo è una relazione d'equivalenza nella classe di tutti gli insiemi.

Dato $n \in \mathbb{N}$ definiamo la famiglia di insiemi J_n:

$$J_n = \begin{cases} \emptyset & n = 0 \\ \{1, \dots, n\} & n > 0 \end{cases}$$

Un insieme A ha cardinalità *finita* se e solo se $J_n \sim A$ per qualche $n \in N$.
Un insieme che non ha cardinalità finita ha cardinalità *infinita*.

Insiemi numerabili e non numerabili

Un insieme A è detto numerabile se e solo se $\mathbb{N} \sim A$ (ovvero $A \in [\mathbb{N}]_\sim$). Per gli insiemi numerabili esiste quindi una biiezione f che consente di elencare l'insieme A senza perdere alcun elemento: $A = \{f(0), \dots, f(n), \dots\}$.
Se un insieme non è listabile su $\mathbb{N}$ non è numerabile, perché è "più fitto" e perderemmo qualche elemento nell'elenco.

$\mathbb{R}$ non è numerabile

La dimostrazione segue 3 passi:

1. dimostrazione che $\mathbb{R} \sim (0, 1)$ (intervallo aperto);
2. dimostrazione che $\mathbb{N} \nsim (0, 1)$;
3. $\mathbb{N} \nsim (0, 1) \sim \mathbb{R} \implies \mathbb{N} \nsim \mathbb{R}$.

Il primo punto si dimostra graficamente: una semicirconferenza rappresenta l'insieme $(0, 1)$ in modo che la proiezione dell'intersezione del raggio con la circonferenza sul diametro identifichi l'elemento dell'insieme.

Il raggio viene poi prolungato come una retta fino ad intersecare la retta dei reali, parallela al diametro.
Dato che quindi ad ogni punto di $(0,1)$ posso associare un punto di $\mathbb{R}$ (iniezione) e viceversa posso risalire ad ogni punto del segmento partendo dalla retta dei reali (suriezione), la funzione è biiettiva e quindi un isomorfismo.

Per il secondo punto ipotiziamo per assurdo di poter elencare esaustivamente i numeri tra 0 e 1, e li scriviamo nella seguente forma:

$$\begin{array}{llllll} 0. & a_{00} & a_{01} & a_{02} & \dots \\ 0. & a_{10} & a_{11} & a_{12} & \dots \\ 0. & a_{20} & a_{21} & a_{22} & \dots \\ 0. & a_{30} & a_{31} & a_{32} & \dots \\ \vdots & & & & \end{array}$$

Costruisco il numero c scegliendo gli elementi della diagonale, in modo che la i-esima cifra di c sia costruita così:

$$c_i = \begin{cases} a_{ii} + 1 & a_{ii} < 9 \\ 0 & a_{ii} = 9 \end{cases}$$

Questo numero differisce da ogni numero della lista nella cifra diagonale.
Siamo quindi arrivati ad un assurdo per cui la lista esaustiva non elenca ogni numero possibile, segue che $\mathbb{N} \nsim (0,1)$.

Per il terzo punto ci basta considerare la transitività dell'isomorfismo per concludere che $\mathbb{R}$ non è isomorfo a $\mathbb{N}$ e quindi l'insieme dei numeri reali non è numerabile.

Insiemi continui

Insiemi che hanno la stessa cardinalità dei numeri reali sono detti *continui*.
Ad esempio, l'insieme delle parti di $\mathbb{N}$ non è numerabile e lo si dimostra con un simile procedimento diagonale: per assurdo si ipotizza di poter elencare tutti gli insiemi della partizione per mezzo del loro vettore caratteristico (per ogni $n \in N$ la n-esima cifra del vettore caratteristico di A è 0 se n non è in A, 1 viceversa) e si costruisce un nuovo vettore caratteristico negando gli elementi della diagonale. Si è quindi ottenuto un nuovo vettore che non era stato prima elencato, arrivando ancora una volta ad un assurdo.

Un importante insieme non numerabile è $\mathbb{N}^{\mathbb{N}}$, ovvero l'insieme delle funzioni dai naturali ai naturali.
Ancora una volta si dimostra questa proprietà per assurdo e per diagonalizzazione: si costruisce una matrice in cui ogni riga è l'applicazione di una funzione su ogni elemento di $\mathbb{N}$:

$$\begin{array}{cccccc} & 0 & 1 & 2 & \dots \\ f_0 & f_0(0) & f_0(1) & f_0(2) & \dots \\ f_1 & f_1(0) & f_1(1) & f_1(2) & \dots \\ f_2 & f_2(0) & f_2(1) & f_2(2) & \dots \\ \vdots & & & & \end{array}$$

A questo punto si costruisce la funzione $\varphi : \mathbb{N} \to \mathbb{N}$ in modo che $\varphi(n) = f_n(n) + 1$.
Questa funzione è ben definita ma non compare nell'enumerazione precedente, quindi ecco ancora l'assurdo che porta a dire che $\mathbb{N}^{\mathbb{N}} \nsim \mathbb{N}$.

Potenza computazionale, ancora

Due considerazioni sono da fare:

- $\mathrm{PROG} \sim \mathbb{N}$
- $\mathrm{DATI} \sim \mathbb{N}$

Detto ciò possiamo dire che $F(\mathbb{C}) \sim \mathrm{PROG} \sim \mathbb{N}$, ovvero la potenza computazionale, che è il numero di funzioni che possiamo calcolare, è isomorfo ai naturali e quindi numerabile.
Siccome i dati sono anch'essi numeri, l'insieme delle funzioni che vanno dai dati ai dati ha una cardinalità pari a, ed è quindi isomorfo a, $\mathbb{N}_{\perp}^{\mathbb{N}}$.
Tutto questo significa che ci sono pochi programmi ($\mathbb{N}$) e troppe funzioni ($\mathbb{N}_{\perp}^{\mathbb{N}}$), ovvero esistono funzioni non calcolabili:

$$F(\mathbb{C}) \sim \mathrm{PROG} \sim \mathbb{N} \nsim \mathbb{N}_{\perp}^{\mathbb{N}} \sim \mathrm{DATI}_{\perp}^{\mathrm{DATI}}$$

Tale risultato non dipende dalla tecnologia, e rimane quindi vero a prescindere dal modello di calcolo considerato.

Dimostriamo rigorosamente l'isomorfismo di programmi e dati ai naturali.

$\mathrm{DATI} \sim \mathbb{N}$

Cerchiamo una legge che:

- associ biunivocamente dati e numeri;

- consenta di operare direttamente sui numeri per operare sui corrispondenti dati;
- consenta di dire, senza perdita di generalità, che i nostri programmi lavorano su numeri.

Useremo questa catena di isomorfismi: $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}^+ \to \mathbb{N} \times \mathbb{N} \sim \mathbb{N}$.

Coppia di Cantor

Funzione coppia di Cantor: $<,>: \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+, < x, y >= n$. Costruiamo questa funzione che sia suriettiva e iniettiva ed in modo che esistano due funzioni per risalire ad x e y: $sin : \mathbb{N}^+ \to \mathbb{N}, sin(n) = x$ e $des : \mathbb{N}^+ \to \mathbb{N}, des(n) = y$.
Graficamente:

$$\begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 & \cdots \\ 0 & 1 & 3 & 6 & 10 & 15 & \cdots \\ 1 & 2 & 5 & 9 & 14 & \cdots \\ 2 & 4 & 8 & 13 & \cdots \\ 3 & 7 & 12 & \cdots \\ 4 & 11 & \cdots \\ 5 & \cdots \\ \vdots \end{array}$$

Si vede chiaramente che, contando sulle diagonali, la funzione permette di "contare" $\mathbb{N} \times \mathbb{N}$ e che è sia iniettiva che suriettiva. Proviamo a determinare la forma analitica: notiamo innanzitutto che il valore di $< x, y >$ si trova sulla diagonale $x + y$, quindi $< x, y >=< x + y, 0 > +y$; la 0-esima colonna è data dalla somma degli indici incrementata di uno, ovvero $< z, 0 >= \sum_{i=0}^{z} i = 1 + z(z+1)/2$; unendo i due risultati abbiamo:

$$< x, y >=< x + y, 0 > +y = \frac{(x+y)(x+y+1)}{2} + y + 1$$

Cerchiamo analiticamente la forma inversa. Il primo passo è definire il valore intermedio γ tale che il numero cercato giaccia sulla diagonale $< \gamma, 0 >$:

$$\gamma = \left\lfloor \frac{-1 + \sqrt{8n - 7}}{2} \right\rfloor$$

quindi so che $y = n - < \gamma, 0 >$ e $x = \gamma - y$.

Per ora ho dimostrato solo che $\mathbb{N} \times \mathbb{N}$ è isomorfo a $\mathbb{N}^+$, ma per dimostrare che è isomorfo anche al semplice $\mathbb{N}$ basta definire la funzione $[x, y] =< x, y > -1$ che in pratica è una coppia di Cantor in cui

si inizia a contare da 0. Questo mostra sia che $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$ sia che, dato che i razionali non sono altro che coppie di naturali, $\mathbb{Q} \sim \mathbb{N}$.

Codifica di dati in numeri

Ogni tipo di dato può essere rappresentato, in particolar modo grazie alla funzione coppia di Cantor, come un intero naturale, ad esempio codificando una lista di numeri con una applicazione ripetuta della funzione, un suono con una lista di campionature, un grafo con la matrice di adiacenza che è una lista di liste, e così via.
Possiamo quindi lavorare su numeri invece che su dati, e trasporre le nostre funzioni $f : \mathrm{DATI} \to \mathrm{DATI}_\perp$ in funzioni $f : \mathbb{N} \to \mathbb{N}_\perp$.
L'universo dei problemi per i quali cerchiamo una soluzione automatica è rappresentato dall'insieme $\mathbb{N}_\perp^\mathbb{N}$.

$\mathrm{PROG} \sim \mathbb{N}$

Per mostrare anche questo isomorfismo useremo due sistemi di calcolo: il sistema di calcolo RAM e il sistema di calcolo WHILE.
Il sistema di calcolo RAM è sostanzialmente un assembly molto semplificato che permette di definire in maniera rigorosa l'isomorfismo $\mathrm{PROG} \sim \mathbb{N}$, la semantica dei programmi e una prima proposta di potenza computazionale di questo sistema.
Per evitare di valutare un solo sistema di calcolo che potrebbe essere troppo semplicistico, lo confronteremo con il sistema di calcolo più sofisticato WHILE. Dal loro confronto potremo renderci conto se la potenza dipende dallo strumento o, se hanno le stesse capacità, non dipenda dalla piattaforma ma sia una caratteristica teorica, intrinseca nei problemi.

Macchina RAM

La macchina RAM altro non è che un processore che esegue sequenzialmente una serie di istruzioni agendo su una memoria che è una lista infinita di registri R: in particolare il registro R_0 conterrà l'output del programma, e il registro R_1 conterrà all'avvio l'input del programma; oltre a ciò esiste L, ovvero il *Program Counter* che tiene traccia dell'istruzione da eseguire.
$Istr_x$ indica l'istruzione x-esima del programma P.
Le istruzioni disponibili sono di 3 tipi:

1. $R_k \leftarrow R_k + 1$: incremento del valore contenuto in R_k;

2. $R_k \leftarrow R_k \dot{-} 1$: decremento del valore contenuto in R_k in maniera sicura, ovvero senza mai scendere sotto 0;
3. $\text{IF } R_k = 0 \text{ THEN GOTO } m$: salto condizionato all'istruzione m se il registro R_k contiene il valore 0.

L'inizializzazione consiste nell'impostare L a 1, tutti i registri a 0, e il registro R_1 all'input n. Per convenzione $L = 0 \implies \text{HALT}$, perciò un programma termina quanto il contatore è 0 e potrebbe non terminare mai nel caso di loop. Il contenuto di R_0 al termine (senza loop) è l'output dell'esecuzione. La semantica del programma P è quindi $\varphi_P : \mathbb{N} \to \mathbb{N}_\perp$.

Definizione formale

Il programma induce una sequenza possibilmente infinita di stati partendo dallo stato iniziale. Ogni stato riporta il contenuto del contatore e di ogni registro della macchina RAM, ed ogni istruzione del programma fa passare da uno stato al successivo.
Semantica di P:

$$\varphi_P : \mathbb{N} \to \mathbb{N}_\perp \implies \varphi_P(x) = \begin{cases} y \\ \perp \end{cases}$$

Definizione formale di stato

Stato: foto di tutte le componenti della macchina.
$S : \{L, R_i\} \to \mathbb{N}$
$S(R_j)$ restituisce il contenuto del registro R_j ponendo la macchina nello stato S.
Stati $= \mathbb{N}^{\{L, R_i\}} = \{\text{possibili stati della macchina}\}$.
Lo stato finale è qualunque stato per cui valga $S(L) = 0$, cioè in HALT.
L'inizializzazione $in(n) : \text{DATI} \to \text{STATI}$ è una semplice funzione dai dati agli stati che sostanzialmente genera lo stato iniziale in cui tutti i registri sono 0 tranne il registro di input e il contatore.

Funzione stato prossimo

L'esecuzione di un programma è la dinamica del programma, definita da una funzione detta *funzione stato prossimo*:

$$\delta : \text{STATI} \times \text{PROG} \to \text{STATI}_\perp$$

$$\delta(S, P) = S'$$

Questa funzione restituisce lo stato successivo S' prendendo in input lo stato attuale S e il programma P.

Se il contatore è 0, lo stato successivo non è definito; se invece il contatore è maggiore di $Istr_{|P|}$ lo stato successivo ha tutti i registri uguali e il contatore posto a 0; nel caso in cui il contatore fosse un valore valido, lo stato successivo ad una operazione di incremento/decremento lascia tutti i valori dei registri inalterati tranne quello toccato dall'operazione e il contatore che avanza di 1, mentre lo stato successivo ad un salto condizionale lascia tutti i valori inalterati tranne il contatore che o avanza di uno o viene impostato al valore richiesto dal salto.

Esecuzione

L'esecuzione è una sequenza eventualmente infinita di stati $S_0, \dots, S_i, S_{i+1}, \dots$ partendo da $S_0 = in(n)$ in cui da ogni stato si passa al successivo secondo la legge $\delta(S_i, P) = S_{i+1}$.
La terminazione avviene se si raggiunge uno stato S_m tale che $S_m(L) = 0$ (ovvero tale che δ non è definita) ed in quel caso l'output y è il contenuto del registro R_0, altrimenti l'output è indefinito ($\perp$).

Potenza computazionale

$$F(RAM) = \{f \in \mathbb{N}_\perp^\mathbb{N} : \exists P \in \text{PROG}, \varphi_P = f\} = \{\varphi_P : P \in \text{PROG}\} \subset \mathbb{N}_\perp^\mathbb{N}$$

Questo è l'insieme delle funzioni calcolabili da una macchina RAM.

$\text{PROG} \sim \mathbb{N}$

Goedelizzazione

Aritmetizzare o goedelizzare una struttura significa associarle in modo biunivoco un numero naturale. Se riuscissimo a definire una relazione Ar di aritmetizzaione che permettesse di aritmetizzare ogni istruzione, otterremmo una lista di numeri (le istruzioni del programma) che saremmo poi tranquillamente in grado di tradurre in un unico numero naturale grazie ai metodi visti, utilizzando ad esempio la funzione coppia di Cantor.
Vogliamo quindi trovare $Ar : Istr \to \mathbb{N}$ e $Ar^{-1} : \mathbb{N} \to Istr$ tali che $Ar(Istr) = n \iff Ar^{-1}(n) = Istr$.
Per costruirla definiamo così 3 casi:

1. $Ar(R_k \leftarrow R_k + 1) = 3k$;
2. $Ar(R_k \leftarrow R_k \dot{-} 1) = 3k + 1$;
3. $Ar(\text{IF } R_k = 0 \text{ THEN GOTO } m) = 3 < k, m > -1$.

Per l'inverso, basta vedere se $mod\,3$ il resto è 0, 1 o 2 per sapere quale delle 3 istruzioni è stata codificata, e dividere per 3 per avere l'indice del registro codificato; nel terzo caso è necessario un passaggio in più, ovvero applicare sin e des per estrarre rispettivamente registro e istruzione per il salto.
Codificare un intero programma significa semplicemente applicare la funzione coppia di Cantor ad una lista di istruzioni codificate in questo modo, mentre decodificarlo significa applicare la funzione sin per ottenere l'istruzione di testa, des per avere il resto della lista.

Osserviamo quindi che $n \leftrightarrow P$, ovvero i numeri sono programmi:
$F(RAM) = \{\varphi_P : P \in \mathrm{PROG}\} = \{\varphi_i\}_{i \in \mathbb{N}}$.
Da questo si ha rigorosamente che $F(RAM) \sim \mathbb{N} \nsim \mathbb{N}^{\mathbb{N}}$, quindi alcuni problemi non sono risolvibili da una macchina RAM.

Macchine WHILE

Il sistema di calcolo while è un linguaggio strutturato, quindi le istruzioni vengono eseguite una dopo l'altra senza bisogno di un contatore. Possiede 21 registri, il primo usato come registro di output e il secondo come registro di input. La sua sintassi è definita induttivamente, ovvero partendo da istruzioni semplici si compongono comandi più complessi.

Comandi

Assegnamento (comandi base):

1. $x_k := 0$;
2. $x_k := x_j + 1$;
3. $x_k := x_j \dot{-} 1$;

Istruzione while (comando complesso):
$\mathrm{WHILE}\ x_k \neq 0\ \mathrm{DO} : \mathbb{C}$
Nel corpo dell'istruzione while può comparire uno qualunque tra i comandi base, complessi o composti.

Comando composto:
$begin\, C_1; \ldots, C_m\, end$
È un comando che racchiude una serie di comandi di qualsiasi tipo tra un *begin* e un *end*.

Indichiamo con $\mathrm{W\text{-}PROG}$ l'insieme dei programmi while.

Semantica

$\Psi_w(n) = cont(x_0)/\perp$
La semantica di un programma while su dato n è l'output contenuto nel registro x_0 in caso di terminazione oppure è indefinito.

A differenza della RAM, che aveva infiniti stati, non abbiamo bisogno di definire una funzione di stato: qui basta una lista di 21 elementi contenente il valore dei registri. Lo spazio degli stati è quindi $\mathbb{N}^{21}$, e uno stato $\underline{x}$ è $\underline{x} \in \mathbb{N}^{21}$ e l'inizializzazione semplicemente imposta a 0 tutti i registri tranne x_1 di input.

Funzione stato prossimo

$$[\,]( ) : \text{W-COM} \times \text{W-STATI} \to \text{W-STATI}_\perp$$

È una funzione che, da un comando, preso in ingresso uno stato, resituisce lo stato prossimo: $[\mathbb{C}](\underline{x}) = \underline{x}'$. Questa funzione può essere definita induttivamente su tutte le strutture induttive del comando $\mathbb{C}$.

Nei casi base è semplice definire la funzione stato prossimo: semplicemente restituisce i 21 valori invariati tranne quello indicato nell'assegnamento, che sarà incrementato/decrementato o posto a 0.
In un passo composto conosciamo per induzione il comportamento della funzione stato prossimo: non ci resta che applicare iterativamente la funzione stato prossimo ad ogni comando, prendendo come stato iniziale lo stato prossimo del comando precedente.
Anche per il comando while conosciamo la funzione stato prossimo per ipotesi induttiva, e ci basta applicare la funzione stato prossimo il minor numero di volte necesario a mandare a 0 il registro di controllo. Lo stato prossimo di un comando while è quindi il risultato dopo queste applicazioni o indefinito se non si può mandare a 0 il registro di controllo.
Semantica di W-PROG: $\Psi_w(x) = Pro_0^{21}([w](w - in(x)))$.

Confronto tra sistemi di calcolo

Vogliamo ora confrontare i due sistemi RAM e WHILE per vedere se possiamo trarre conclusioni interessanti sulla calcolabilità in due sistemi diversi.
In generale poniamo di avere due sistemi di calcolo $\mathbb{C}_1$ e $\mathbb{C}_2$ e definiamo i rispettivi poteri computazionali:

$$F(\mathbb{C}_1) = \{f \in \mathbb{N}_\perp^{\mathbb{N}} : \exists P_1 \in \mathbb{C}_1\text{-PROG} | f = \Psi_{P_1}\} = \{\Psi_{P_1} : P_1 \in \mathbb{C}_1\text{-PROG}\}$$

$$F(\mathbb{C}_2) = \{f \in \mathbb{N}_\perp^{\mathbb{N}} : \exists P_2 \in \mathbb{C}_2\text{-PROG} | f = \varphi_{P_2}\} = \{\varphi_{P_2} : P_2 \in \mathbb{C}_2\text{-PROG}\}$$

Il fine sarebbe dimostrare $F(\mathbb{C}_1) \subseteq F(\mathbb{C}_2)$, ovvero

$$\forall f \in F(\mathbb{C}_1) \implies f \in F(\mathbb{C}_2)$$

$$\exists P_1 \in \mathbb{C}_1\text{-PROG} : f = \Psi_{P_1} \implies \exists P_2 \in \mathbb{C}_2\text{-PROG} : f = \varphi_{P_2}$$

In altre parole, per ogni programma del primo sistema ne esiste uno equivalente nel secondo (se esiste un programma che esprime una funzione f nel primo sistema, esiste un altro programma nel secondo sistema che esprime la stessa funzione f).
Tutto questo si può provare trovando un *compilatore* tra il primo e il secondo linguaggio.

Compilatore

In matematica la compilazione si chiama "traduzione".
Dati i sistemi $\mathbb{C}_1$ e $\mathbb{C}_2$, una traduzione tra i due è una funzione

$$T : \mathbb{C}_1\text{-PROG} \to \mathbb{C}_2\text{-PROG}$$

tale che:

1. sia programmabile effettivamente;
2. sia completa;
3. mantenga la semantica;

$\forall P \in \mathbb{C}_1\text{-PROG} : \varphi_{T(P)} = \Psi_P$
Teorema:
Se esiste $T : \mathbb{C}_1\text{-PROG} \to \mathbb{C}_2\text{-PROG}$ allora $F(\mathbb{C}_1) \subseteq F(\mathbb{C}_2)$
Dimostrazione:
$f \in F(\mathbb{C}_1) \implies \exists P \in \mathbb{C}_1\text{-PROG} : f = \Psi_P$. Applicando T ottengo $T(P) \in \mathbb{C}_2\text{-PROG}$ con $\varphi_{T(P)} = \Psi_P = f$.
Esiste quindi un programma in $\mathbb{C}_2\text{-PROG}$ per f per cui $f \in F(\mathbb{C}_2)$.

In un compilatore da WHILE a RAM possiamo tranquillamente spostare le 21 variabili nei primi 21 registri della macchina RAM senza problemi e usarli nello stesso modo.

Comandi base

Per l'azzeramento di una variabile:

$$\begin{aligned} Comp(x_k := 0) = \ & LP : \text{IF } R_k = 0 \text{ THEN GOTO EX} \\ & \quad\ R_k \leftarrow R_k \dot{-} 1 \\ & \quad\ \text{IF } R_{21} = 0 \text{ THEN GOTO LP} \\ & EX : R_k \leftarrow R_k \dot{-} 1 \end{aligned}$$

Nel caso di incremento/decremento, se $k = j$ abbiamo una istruzione apposita, quindi consideriamo solo il caso $k \neq j$:

$$\begin{aligned} Comp(x_k := x_j +/\dot{-} 1) = \; & LP : \text{IF } R_j = 0 \text{ THEN GOTO EX1} \\ & \quad R_j \leftarrow R_j \dot{-} 1 \\ & \quad R_{22} \leftarrow R_{22} + 1 \\ & \quad \text{IF } R_{21} = 0 \text{ THEN GOTO LP} \\ & EX1 : \text{IF } R_k = 0 \text{ THEN GOTO EX2} \\ & \quad R_k \leftarrow R_k \dot{-} 1 \\ & \quad \text{IF } R_{21} = 0 \text{ THEN GOTO EX1} \\ & EX2 : \text{IF } R_{22} = 0 \text{ THEN GOTO EX3} \\ & \quad R_{22} \leftarrow R_{22} \dot{-} 1 \\ & \quad R_j \leftarrow R_j + 1 \\ & \quad R_k \leftarrow R_k + 1 \\ & \quad \text{IF } R_{21} = 0 \text{ THEN GOTO EX2} \\ & EX3 : R_k \leftarrow R_k +/\dot{-} 1 \end{aligned}$$

Comando composto

Per ipotesi induttiva, conosco le compilazioni dei casi base che compongono un comando composto:

$$\begin{aligned} Comp(begin\ C_1; \ldots, C_m\ end) = \; & Comp(C_1); \\ & \vdots \\ & Comp(C_m); \end{aligned}$$

Comando while

Per ipotesi induttiva, assumo nota la compilazione di un comando composto

$$\begin{aligned} Comp(\text{WHILE } x_k := 0) = \; & LP : \text{IF } R_k = 0 \text{ THEN GOTO EX} \\ & \quad Comp(C) \\ & \quad \text{IF } R_{21} = 0 \text{ THEN GOTO LP} \\ & EX : R_k \leftarrow R_k \dot{-} 1 \end{aligned}$$

Conclusione

Questa traduzione è programmabile, completa e corretta: ciò significa che è effettivamente un compilatore tra i due linguaggi. Questo inoltre dimostra che la potenza computazionale di WHILE è inclusa in quella di RAM.

Interprete

Per dimostrare l'altro verso dell'inclusione, e quindi controllare se WHILE è incluso propriamente in RAM o se sono equipotenti, impiegheremo un *interprete*. Un interprete I differisce da un

compilatore in quanto è un programma scritto nel linguaggio di destinazione che permette di eseguire una istruzione alla volta del linguaggio di partenza.

Macro-while

Per comodità useremo una versione del WHILE con alcune macro per semplificare la scrittura del codice:

- $x_k := x_j + x_i$;
- $x_k :=< x_j, x_i >$;
- $x_k :=< a_1, \ldots, a_n >$;
- $x_k := Pro(x_j, x_i)$: mette in x_k il valore x_j-esimo della lista codificata in x_i;
- $x_k := incr(x_j, x_i)$: mette in x_k la codifica della lista in x_i col valore x_j-esimo incrementato di 1;
- $x_k := decr(x_j, x_i)$: mette in x_k la codifica della lista in x_i col valore x_j-esimo decrementato di 1;
- $x_k := sin(x_j)$;
- $x_k := des(x_j)$;
- if - then - else.

Corrispondenza memoria WHILE - RAM

Dato che WHILE ha solo 21 variabili mentre RAM ha infiniti registri, dobbiamo in qualche modo mappare i registri nelle 21 variabili.
Il primo particolare da notare è che RAM non userà mai veramente infiniti registri: dato che codificheremo programma e dato con Cantor, la macchina RAM non potrà aver usato un registro con indice maggiore del numero restituito dalla codifica di Cantor.

- $x_0 \leftarrow< R_0, \ldots, R_{n+2} >$: in x_0 codifichiamo con Cantor i registri della macchina RAM;
- $x_1 \leftarrow L$: in x_1 terremo il contatore;
- $x_2 \leftarrow x$: in x_2 il dato di input su cui lavora P;
- $x_3 \leftarrow cod(P)$: in x_3 metteremo la codifica di P;
- $x_4 \leftarrow Istr_L$: in x_4 l'istruzione corrente da eseguire.

Esecuzione

Per eseguire il programma RAM memorizzato in questi 5 registri semplicemente, dopo aver preparato le variabili, si imposta un ciclo while che dura finché il contatore è diverso da 0 o finchè non è maggiore della lunghezza di P; in questo ciclo si esegue il fetch dell'istruzione corrente (cioè si mette in x_4 la proiezione

dell'elemento contato da x_1 del listato del programma) e si confronta il suo modulo 3 per decidere quale istruzione eseguire; a questo punto si può banalmente eseguire una delle 3 istruzioni RAM tradotte in while.

Conseguenze

Avendo I_w, ossia un interprete WHILE per il linguaggio RAM, è possibile costruire un compilatore da RAM a WHILE semplicemente codificando il programma RAM ed usando le istruzioni eseguite dall'interprete. Questo porta al *teorema di Boehm-Jacopini* che afferma che per ogni programma in linguaggio con GOTO ne esiste uno equivalente in un linguaggio strutturato che si ottiene eliminando il GOTO.
Avere un interprete (e quindi un compilatore) significa quindi, come era nel caso precedente da while a RAM, che $F(\mathrm{RAM}) \subseteq F(\mathrm{WHILE})$. Le due inclusioni implicano che le due potenze computazionali sono coincidenti:

$$\mathbb{N}_{\perp}^{\mathbb{N}} \nsim \mathbb{N} \sim \mathrm{PROG} \sim F(\mathrm{RAM}) = F(\mathrm{WHILE})$$

Concludendo, i sistemi WHILE e RAM, pure profondamente diversi, calcolano le stesse cose, ed inoltre questo implica che esistono funzioni *non calcolabili*, dimostrato formalmente, da alcuno dei due sistemi.

Interprete universale RAM

Avevamo prima definito un compilatore che traduce ogni programma WHILE in un programma RAM; ora abbiamo un interprete WHILE che permette di eseguire in WHILE ogni programma RAM.
Cosa succede se traduciamo l'interprete WHILE col compilatore RAM in un programma RAM? Il risultato è un programma in linguaggio RAM che è in grado di eseguire ogni altro programma RAM.

$$U = Comp(I_w) \in \mathrm{PROG}$$
$$\varphi_U(< x, n >) = \Psi_{I_w}(< x, n >) = \phi_n(x)$$

U è detto interprete universale, e la sua semantica è la stessa dell'interprete while.

Funzioni calcolabili

Proveremo ora a fornire una definizione rigorosa e formale di "funzione calcolabile" in modo che astragga da ogni sistema di calcolo concreto.

Chiusura di insiemi rispetto ad operazioni

Dato un insieme U, si definisce *operazione* su U una qualunque funzione $op : U \times \cdots \times U \to U$, dove il numero di parametri di partenza è detto k-arietà dell'operazione.
Un insieme $A \in U$ è detto *chiuso* rispetto all'operazione $op : U^n \to U$ se e solo se

$$\forall a_1, \ldots, a_n \in A : op(a_1, \ldots, a_n) \in A$$

In generale, se Ω è l'insieme di operazioni su U, A è chiuso rispetto ad Ω se e solo se A è chiuso rispetto ad ogni op di Ω.

Chiusura di un insieme

Sia $A \subseteq U$ e $op : U^n \to U$. Qual è il più piccolo sottoinsieme di U che contenga A e sia chiuso rispetto ad op?
Questo insieme si ottiene calcolando la chiusura di A rispetto ad op, indicata con A^{op}, definita induttivamente come:

1. $\forall a \in A \implies a \in A^{op}$;
2. $\forall a_1, \ldots, a_n \in A^{op} \implies op(a_1, \ldots, a_n) \in A^{op}$;
3. nient'altro sta in A^{op}.

Operativamente, nella chiusura di A mettiamo tutti gli elementi di A, applichiamo op ad una k-tupla di A e, se non è già presente, aggiungiamo il risultato in A^{op}; ripetiamo fintantoché A^{op} cresce.

Definizione teorica di calcolabilità

Funzioni sicuramente calcolabili

Definiamo un insieme di 3 funzioni che siano calcolabili secondo ogni ragionevole idea di calcolabilità:

1. successore: $S(x) = x + 1$;
2. identicamente nulla: $0^n(x_1, \ldots, x_n) = 0, x_i \in \mathbb{N}$;
3. proiezione: $Pro_k^n(x_1, \ldots, x_n) = x_k, x_i \in \mathbb{N}$;

Operatore composizione di funzioni

sia $h : \mathbb{N}^n \to \mathbb{N}$ e $g_1, \ldots, g_n : \mathbb{N}^n \to \mathbb{N}$; denotiamo $\underline{x} \in \mathbb{N}^n$. Definiamo

$$Comp(h, g_1, \dots, g_n) : \mathbb{N}^n \to \mathbb{N}$$
$$Comp(h, g_1, \dots, g_n)(\underline{x}) = h(g_1(\underline{x}), \dots, g_n(\underline{x}))$$

Operatore ricorsione primitiva

Siano $g : \mathbb{N}^n \to \mathbb{N}$ e $h : \mathbb{N}^{n+2} \to \mathbb{N}$; $g(\underline{x})$ e $h(z, y, \underline{x})$ con $\underline{x} \in \mathbb{N}^n$.

$$RP(h, g) = f(\underline{x}, y) = \begin{cases} g(\underline{x}) & y = 0 \\ h(f(\underline{x}, y-1), y-1, \underline{x}) & y > 0 \end{cases}$$

Con questo riusciamo ad esprimere ad esempio somma, prodotto e predecessore:

$$somma(x, y) = \begin{cases} x = Pro_0^2(x, y) & y = 0 \\ successore(somma(x, y-1)) & y > 0 \end{cases}$$

$$prodotto(x, y) = \begin{cases} 0 & y = 0 \\ somma(x, prodotto(x, y-1)) & y > 0 \end{cases}$$

$$predecessore(x) = \begin{cases} 0 & x = 0 \\ x - 1 & x > 0 \end{cases}$$

$$x \dot{-} y = \begin{cases} x & y = 0 \\ predecessore(x) \dot{-} (y-1) & y > 0 \end{cases}$$

Funzioni ricorsive primitive

Questo è il nome della classe di funzioni ottenute partendo da $ELEM$ e chiudendolo rispetto all'operatore composizione e all'operatore ricorsivo primitivo.
Definiamo induttivamente le funzioni ricorsive primitive:

1. Le funzioni in $ELEM$ sono in $RICPRIM$;
2. $h, g_1, \dots, g_n \in RICPRIM \implies Comp(h, g_1, \dots, g_n) \in RICPRIM$;
3. $g, h \in RICPRIM \implies RP(h, g) \in RICPRIM$;
4. null'altro è in RICPRIM.

Per dimostrare che ogni funzione in RP è programmabile si procede sempre in maniera induttiva:

1. dimostro che $ELEM \subseteq F(\text{WHILE})$;
2. assumo che $h, g_1, \dots, g_n \in RICPRIM \subseteq f(\text{WHILE})$ e dimostro che $Comp(h, g_1, \dots, g_n) \in F(\text{WHILE})$;
3. assumo che $g, h \in RICPRIM \subseteq F(\text{WHILE})$ e dimostro che $RP(g, h) \in F(\text{WHILE})$.

Il primo punto è facile perché riesco facilmente a scrivere programmi per calcolare successivo, identità nulla e proiezione.
Per il secondo punto, assumendo (per ipotesi induttiva) di poter programmare le singole funzioni, mi basta mostrare un programma che

per ogni funzione g calcoli il suo risultato su x_1 e lo codifichi magari con Cantor in x_0 assieme al risultato della funzione precedente, e h finale semplicemente opera su questa codifica, quindi mostro un programma while la cui funzione espressa è la composizione delle funzioni: $\Psi_w(\underline{x}) = Comp(h, g_1, \dots, g_n)(\underline{x})$.
Per il terzo punto mostriamo che possiamo calcolare tutte le funzioni contenute in RP: semplicemente mostro un programma WHILE che richiama la routine $G(x)$ per cui vale $g = \varphi_G$, salva il risultato in un registro t, poi esegue in un ciclo di k iterazioni (fino a che vale $k \leq y$) la routine H per cui vale $h = \varphi_H$: $t := H(t, k-1, \underline{x})$.

Questo ci dimostra che $RP \subset F(\text{WHILE})$. Da notare che l'inclusione è propria in quanto le funzioni ricorsive primitive sono solo totali, mentre le macchine while possono andare in loop e quindi avere risultato indefinito.
Al massimo le funzioni ricorsive primitive sono coincidenti con le funzioni espresse dal linguaggio FOR, un linguaggio del tutto simile a WHILE se non per i cicli in cui si esegue l'iterazione solo un massimo numero di volte indicato da una variabile di controllo.
Oltre ai loop infiniti però la macchina WHILE è in grado di esprimere funzioni che crescono troppo velocemente per la macchine FOR (e quindi per le ricorsive primitive), come ad esempio la seguente funzione di Ackerman:

$$\mathbb{A}(m,n) = \begin{cases} n+1 & m = 0 \\ \mathbb{A}(m-1, 1) & m > 0, n = 0 \\ \mathbb{A}(m-1, \mathbb{A}(m, n-1)) & m, n > 0 \end{cases}$$

Operatore di minimalizzazione di funzioni

Siccome le funzioni ricorsive primitive ancora non catturano la completezza delle funzioni calcolabili da WHILE (e quindi da RAM), le ampliamo ancora chiudendole rispetto ad un nuovo operatore che prende una funzione e restituisce la funzione minimalizzata.
Sia $f : \mathbb{N}^{n+1} \to \mathbb{N}$, $f(\underline{x}, y)$ con $\underline{x} \in \mathbb{N}^n$ e $y \in \mathbb{N}$. Definiamo

$$MIN(f(\underline{x}, y)) = g(x) = \begin{cases} y & f(\underline{x}, y) = 0 \land \forall t < y : f(\underline{x}, t) \neq 0 \\ \perp & \text{altrimenti} \end{cases}$$

$$= \mu y (f(\underline{x}, y) = 0)$$

Funzioni ricorsive parziali

La classe $\mathbb{P}$ delle *funzioni ricorsive parziali* amplia RICPRIM chiudendo ELEM rispetto a Comp, RP, e infine MIN:

$$\text{ELEM}^{\text{Comp, RP, MIN}} = \mathbb{P} = \{\text{funzioni ricorsive parziali}\}$$

Sicuramente $\mathbb{P}$, che grazie a MIN contiene anche funzioni parziali, amplia RICPRIM, ma rispetto a $F(\text{WHILE})$?

$\mathbb{P} \subseteq F(\text{WHILE})$

Dimostrazione: definiamo induttivamente $\text{ELEM}^{\text{Comp, RP, MIN}} = \mathbb{P}$.

1. Le funzioni ELEM sono in $\mathbb{P}$;
2. $h, g_1, \dots, g_n \in \mathbb{P} \implies COmp(h, g_1, \dots, g_n) \in \mathbb{P}$;
3. $h, g \in \mathbb{P} \implies RP(h, g) \in \mathbb{P}$;
4. $f \in \mathbb{P} \implies MIN(f) \in \mathbb{P}$;
5. null'altro è in $\mathbb{P}$.

Ora come nella dimostrazione precedente dimostriamo che ogni punto fa parte di $F(\text{WHILE})$.
Per il primo punto si parla di ELEM quindi già le operazioni elementari sono parte di $F(\text{WHILE})$.
Per il secondo punto abbiamo che le singole funzioni sono per ipotesi WHILE-programmabili, quindi la loro composizione è anch'essa WHILE-programmabile.
Il tezo punto si basa ancora sulla dimostrazione precedente quindi è dimostrato facilmente.
Manca quindi la dimostrazione del quarto punto, che è l'aggiunta all'insieme delle funzioni ricorsive primitive. Anche questo è facilmente dimostrabile: per ipotesi la funzione $f(\underline{x}, y) \in \mathbb{P}$ e quindi esiste anche un programma WHILE F che calcola quella funzione. Per calcolare MIN ci basta provare tutti gli y partendo dal più piccolo e controllare che il programma F restituisca 0. Se troviamo 0 restituiamo y altrimenti continueremo all'infinito (quindi $\perp$), esattamente come la semantica di MIN.
La classe delle funzioni ricorsive parziali è WHILE-programmabile.

$F(\text{WHILE}) \subseteq \mathbb{P}$

Per mostrare se l'inclusione è propria o impropria, proviamo a mostrare anche l'altro verso dell'inclusione.
Formalmente, la semantica di un programma WHILE è definita come $\Psi_w(x) = Pro_0^{21}([w](w - in(x)))$ dove $[\,](\,)$ è la funzione stato prossimo e $w - in()$ è l'inizializzazione.
Dato che la proiezione è già inclusa nelle ricorsive parziali e le ricorsive parziali sono chiuse alla composizione, ci basta mostrare che anche l'argomento sia incluso.
Tecnicamente, le ricorsive parziali hanno come codominio i naturali mentre la semantica della funzione stato prossimo ha come codominio un vettore di 21 componenti: passeremo quindi dai vettori ai

naturali utilizzando la codifica fornita dalla coppia di Cantor, ovvero mostreremo che $f_c(x) = y \in \mathbb{P}$ con $x = [\underline{x}]$ e $y = [\underline{y}]$ dove $[\,]$ è la codifica di Cantor di vettori di dimensione fissa.
Procediamo ancora passo per passo. L'azzeramento di un registro è esprimibile con funzioni in $\mathbb{P}$:

$$f_{x_k:=0}(x) = [Pro(0,x), \ldots, 0_k, \ldots, Pro(20,x)]$$

e allo stesso modo l'incremento, decremento:

$$f_{x_k:=x_j+/\dot{-}1}(x) = [Pro(0,x), \ldots, Pro(j,x) + /\dot{-}1, \ldots, Pro(20,x)]$$

Induttivamente siamo quindi in grado di esprimere comandi WHILE con funzioni ricorsive parziali, e possiamo facilmente esprimere anche comandi composti, in quanto composizione di comandi base, e l'insieme delle ricorsive parziali è chiuso rispetto alla composizione.
Per esprimere un ciclo WHILE, per induzione suppongo di saper scrivere comandi base e composti con funzioni ricorsive parziali, e semplicemente eseguo la funzione contenuta nel ciclo $e(x)$ volte: $f_c^{e(x)}(x)$ con $e(x) = \mu y(Pro(x, f_c^y(x)) = 0) = \mu y(Pro(x, T(x,y)) = 0)$.
Da notare che per fare questo mi serve comporre f_c un numero $e(x)$ non costante di volte, che è un'operazione che non so eseguire ad ora: mi serve definire $T(x,y) = f_c^y(x)$ e fare in modo che sia ricorsiva parziale.
Questa funzione T è definita così:

$$T(x,y) = \begin{cases} x & y = 0 \\ f_c(T(x, y-1)) & y > 0 \end{cases}$$

Questa funzione T è sostanzialmente una minimalizzazione di una funzione ricorsiva parziale, quindi è anch'essa ricorsiva parziale $f_{c'}(x) = f_c^{e(x)}(x) = T(x, e(x)) \in \mathbb{P}$.
In questo modo abbiamo mostrato che tutti i comandi WHILE sono esprimibili tramite funzioni ricorsive parziali, dimostrando quindi il secondo verso dell'inclusione: $F(\text{WHILE}) \subseteq FRP$

Conclusioni e Tesi di Church-Turing

Abbiamo definito matematicamente una classe di funzioni che riteniamo calcolabili. La *Tesi di Church-Turing* asserisce che la classe delle funzioni intuitivamente calcolabili coincida proprio con questa classe di funzioni ricorsive parziali: essa è solo una tesi in quanto "intuitivamente calcolabile" o "modello di calcolo ragionevole" non sono caratterizzazioni formali e completi degli strumenti.

Assiomatizzazione di sistemi di calcolo

Vogliamo ora cercare di definire caratteristiche generali che deve avere un sistema di calcolo per essere "buono", su cui poi potremo dimostrare proprietà basandoci solo su tali assiomi e applicandole ad ogni sistema che li rispetti.
Indichiamo con $\{\varphi_i\}_{i\in\mathbb{N}}$ i sistemi di programmazione.

1. $\{\varphi_i\} = \mathbb{P}$: vogliamo che il sistema di programmazione sia coerente con la Tesi di Church-Turing, ovvero che possa calcolare le funzioni ricorsive parziali;
2. interprete universale: la presenza di questo programma, in grado di simulare ogni altro programma del sistema di programmazione, permette una *algebra dei programmi*;
3. validità del teorema S_1^1, ovvero deve essere possibile costruire automaticamente programmi più specifici a partire da programmi generali.

Teorema S_1^1

Il programma S_1^1 implementa la funzione

$$S_1^1(n,y) = \bar{n} : \varphi_{\bar{n}}(x) = \varphi_n(x,y)$$

Il programma che implementi questo, in linguaggio RAM, semplicemente per ogni input da fissare esegue n incrementi iniziali, dove n è il valore da fissare per l'input, per poi metterlo in coppia di Cantor col resto dell'input in R_1, e chiamare poi semplicemente il normale programma P.
Questa funzione è quindi sia programmabile che totale.
Ne consegue il teorema S_1^1: dato $\{\varphi_i\}$

$$\exists S_1^1 \in \mathbb{T} : \forall n,x,y \in \mathbb{N},\ \varphi_n(< x,y >) = \varphi_{S_1^1(n,y)}(x)$$

.

Sistemi di programmazione accettabili - SPA

Sistemi che soddisfino queste tre caratteristiche sono detti *sistemi di programmazione accettabili*:

1. $\{\varphi_i\} = \mathbb{P}$: aderenza alla Tesi di Church-Turing;
2. $\exists u \in \mathbb{N} : \varphi_u(< x,n >) = \varphi_n(x)$: esistenza dell'interprete universale;
3. $\exists S_n^m \in \mathbb{T} : \forall n \in \mathbb{N}, \underline{x} \in \mathbb{N}^m, \underline{y} \in \mathbb{N}^n$

Mostrare un risultato usando solo questi assiomi significa renderlo valido per tutti gli SPA.

Compilatori tra SPA

Dati due SPA, esiste sempre un compilatore tra essi.
La sua esistenza è facilmente dimostrata: ogni funzione in un sistema di calcolo può essere simulata dall'interprete universale, e dato che questo interprete è una funzione ricorsiva parziale, anche il secondo sistema deve avere una funzione che esprima la stessa semantica; a questo punto basta, grazie al teorema S_1^1, specificare il programma per ottenere la funzione voluta nel secondo sistema di programmazione. In simboli, cerchiamo $t \in \mathbb{T}$, quindi totale e corretta:

$$\varphi_i(x) \overset{(2)}{=} \varphi_u(< x, i >) \overset{(1)}{=} \Psi_e(< x, i >) \overset{(3)}{=} \Psi_{S_1^1(e,i)}(x)$$

Quindi la funzione cercata è proprio $t(i) = S_1^1(e, i)$ e possiamo dire che appartiene alle funzioni totali in quanto $S_1^1 \in \mathbb{T}$ ed è corretta in quanto $\varphi_i = \Psi_{t(i)}$.

Inoltre, in conseguenza di ciò, possiamo affermare che esiste sempre un compilatore tra due linguaggi scritto in un terzo linguaggio, dato che sappiamo che esiste un compilatore tra i primi due che è una funzione ricorsiva parziale, che è esprimibile da qualche programma del terzo linguaggio.

Teorema di ricorsione

Dato $\{\varphi_i\}$ SPA, $\forall t : \mathbb{N} \to \mathbb{N}, t \in \mathbb{T}, \exists n \in \mathbb{N} : \varphi_n = \varphi_{t(n)}$.
In parole povere, considerando t come una funzione che manipola funzioni, esisterà sempre un programma, all'indice n, la cui semantica non verrà modificata dall'applicazione di t.

Dimostrazione

Consideriamo una funzione h: possiamo, applicando il teorema S_1^1, specializzarla con la funzione $s_1^1(h, h)$, e a questa specializzazine applicare una trasformazione t.
A questo punto, essendo t una composizione di funzioni totali, sappiamo che esiste una funzione che calcola t, ovvero una funzione implementata dal programma m la cui semantica è come segue: $\varphi_m(h) = \varphi_{t(s_1^1(h,h))}$.
Diciamo ora che n è il programma che implementa questa

specializzazione: $n = s_1^1(m, m)$, e costruiamo la seguente catena di uguaglianze:

$$\varphi_n = \varphi_{s_1^1(m,m)} = \varphi_m(m) = \varphi_{t(s_1^1(m,m))} = \varphi_{t(n)}$$

Questa catena ci porta chiaramente a concludere che $n = t(n)$, ovvero, in accordo con la tesi del teorema, che $\varphi_n = \varphi_{t(n)}$.

Programmi autoreplicanti - quine

$$\exists j \in \mathbb{N} : \Phi_j(x) = j$$

Consideriamo come esempio un programma magari RAM che porta in output un numero j. Consideriamo anche una codifica $Z(j) \in \mathbb{T}$ di questo programma, magari con Cantor. Abbiamo quindi ora che la semantica del programma codificato secondo Z è $\Phi_{Z(j)} = j$.
Per il teorema di Ricorsione sappiamo che deve esistere $i \in \mathbb{N}$ tale che $\Phi_i(x) = \Phi_{Z(i)}(x) = j$.

Più semplicemente, il teorema di ricorsione mostra che esiste una funzione h che mantiene la sua semantica attraverso una trasformazione $t(h)$: poniamo che questa trasformazione sia una funzione che produce il listato del suo argomento, $h = t(h)$ significa che esiste un programma il cui output è il listato di se stesso.

Compilatore errato

Dati due SPA $\{\Phi_i\}, \{\Psi_i\}, \nexists t \in \mathbb{T} : \forall i \in \mathbb{N}\ \Phi_i \neq \Psi_{t(i)}$.
Per convincercene, prendiamo $t : \mathbb{N} \to \mathbb{N} \in \mathbb{T}$ e scriviamo la seguente catena:

$$\Psi_{t(i)}(x) \overset{2}{=} \Psi_U(< x, t(i) >) \overset{1}{=} \Phi_j(< x, t(i) >) \overset{3}{=} \Phi_{S_1^1(j,t(i))}(x) = \Phi_{g(i)}(x)$$

Sostanzialmente partendo da una funzione in uno SPA, passiamo al suo interprete universale (seconda proprietà degli SPA), e poi passiamo all'interprete universale di un secondo SPA (prima proprietà degli SPA), poi specializziamo la funzione (terza proprietà degli SPA). A questo punto, grazie al teorema della Ricorsione, sappiamo che deve esistere un programma la cui semantica non è cambiata da questa catena di uguaglianze: ciò significa che abbiamo costruito un "compilatore" con queste uguaglianze, e che questo esiste almeno un programma che, passando attraverso questo compilatore, esprimerà ancora la sua semantica originale.

A mio parere si può anche vedere come una conseguenza banale del teorema di ricorsione: dato che un compilatore è sostanzialmente una traformazione esprimibile da una funzione, per il teorema di

ricorsione appunto deve esistere un programma la cui semantica non viene modificata da tale trasformazione.

Problemi di decisione

Un problema è decidibile se ho un algoritmo che mi dà sempre la risposta corretta per ogni input oppure se la sua funzione decisione è totale.

$$\Phi_\pi : D \to \{0,1\} : \Phi_\pi(x) = \begin{cases} 1 & p(x) \\ 0 & \neg p(x) \end{cases}$$

Un problema π è decidibile se e solo se $\Phi_\pi \in \mathbb{T}$.

Esempi

- Definire se n è pari: $\Phi_{PARI}(n) = 1 \dot{-} (n \; mod \; 2)$
- Equazione diofantea: $\Phi_{ED}(a,b,c) = 1 \dot{-} (c \; mod \; mcd(a,b))$

Problemi sui grafi:

- l'esistenza di un cammino tra due nodi, problema di raggiungibilità, viene risolto con algoritmi di visita oppure sapendo che, data la matrice di adiacenza M_G del grafo G, $(M_G^k)_{ij} = 1$ se e solo esiste un cammino di lunghezza k tra il nodo i e il nodo j;
- l'esistenza di un circuito hamiltoniano è un problema NP-completo con complessità fattoriale nel numero dei nodi, in quanto consiste nel provare tutte le permutazioni dei nodi e controllare se è un circuito valido (rimane comunque decidibile);
- l'esistenza invece di un circuito euleriano è provata in modo molto più semplice: grazie al teorema di Eulero sappiamo infatti che un grafo contiene un circuito euleriano se e solo ogni nodo ha grado pari.

Problema dell'arresto

AR_p indica il problema dell'arresto per il programma p, ovvero chiede se il programma termina per qualsiasi input: è equivalente a chiedersi se una funzione è totale, ovvero se $\forall x \in \mathbb{N} \; \varphi_p(x) \downarrow$.
Per alcuni programmi, per i quali riesco a trovare la funzione calcolata, il problema è decibile.
Invece ad esempio, facendo eseguire all'interprete universale un programma su se stesso abbiamo un programma indecibile, in quanto non solo il dato ma anche il programma non è fissato.

Lo dimostriamo per assurdo: assumiamo $AR_{\hat{p}}$ decidibile. Dunque la funzione

$$\Phi_{AR_{\hat{p}}}(x) = \begin{cases} 1 & \varphi_{\hat{p}}(x) = \varphi_x(x) \downarrow \\ 0 & \varphi_{\hat{p}}(x) = \varphi_x(x) \uparrow \end{cases}$$

è ricorsiva totale e quindi esiste un programma $\hat{P}$ che la calcola e termina sempre.
Possiamo quindi implementare la seguente funzione:

$$f(x) = \begin{cases} 0 & \Phi_{AR_{\hat{p}}}(x) = 0 \\ \varphi_{\hat{p}}(x) + 1 & \Phi_{AR_{\hat{p}}}(x) = 1 \end{cases}$$

La semantica di questa funzione, che immaginiamo sia codificata dal numero a, è

$$\varphi_a(a) = \begin{cases} 0 & \varphi_a(a) \uparrow \\ \varphi_a(a) + 1 & \varphi_a(a) \downarrow \end{cases}$$

Ma entrambi i risultati portano a degli assurdi: è 0 se è indefinita, ma non può terminare se è indefinita; inoltre se è definita è uguale a se stessa + 1, il che è assurdo.

Insiemi ricorsivi

$A \subseteq \mathbb{N}$ è un *insieme ricorsivo* se e solo se esiste un programma P_a che si arresta su ogni input classificando correttamente gli elementi in base alla loro appartenza o meno ad A.
Equivalentemente definiamo una funzione caratteristica $\chi_a : \mathbb{N} \to \{0, 1\}$ tale che $\chi_a(x) = 1$ se $x \in A$, 0 altrimenti. A è ricorsivo se $\chi_a \in \mathbb{T}$.
Le due definizioni sono equivalenti: se P_a implementa χ_a significa che χ_a è totale, e se χ_a è totale significa che esiste un programma P_a che la implementa.
Con abuso di linguaggio possiamo dire che un insieme ricorsivo è decidibile perché ad ogni insieme possiamo associare un corrispondente problema di riconoscimento: infatti la funzione riconoscimento che restituisce 1 se un elemento appartiene all'insieme e 0 in caso contrario coincide con χ_a ed è quindi totale.
Analogamente se un problema π è decidibile allora $\Phi_\pi \in \mathbb{T}$ e quindi un programma che calcola Φ_π riconosce automaticamente l'insieme $A_\pi \implies A_\pi$ è ricorsivo.

Insiemi non ricorsivi

Dato che ogni problema decidibile corrisponde ad un insieme ricorsivo, problemi indecidibili corrispondono ad insiemi non

ricorsivi. Prendiamo ad esempio il problema dell'arresto: definiamo $A = \{x \in \mathbb{N} : \varphi_x(x) \downarrow\}$ ovvero l'insieme dei programmi che terminano su sé stessi.
Se fosse ricorsivo, significherebbe avere un programma P_a che restituisce 1 se un programma termina e 0 se un programma non termina, ma un programma del genere risolverebbe il problema dell'arresto che abbiamo dimostrato essere indecidibile: A quindi non è ricorsivo.

Relazione ricorsiva

$R \subseteq \mathbb{N} \times \mathbb{N}$ è una relazione ricorsiva se e solo se l'insieme R è ricorsivo, ovvero $\chi_R \in \mathbb{T}$, ovvero esiste un programma P_R che risponde 1 se xRy, 0 altrimenti.

Un'importante relazione ricorsiva

Prendendo un programma P fissato, definiamo la seguente relazione:

$$R_P = \{(x, y) \in \mathbb{N}^2 : P(x) \text{ termina in } y \text{ passi}\}$$

Questa relazione è ricorsiva.
Si implementa facilmente utilizzando l'interprete universale e un meccanismo per contare il numero di passi.
Si può usare questa relazione per esprimere in modo diverso l'insieme corrispondente al problema dell'arresto:

$$A = \{x \in \mathbb{N} : \varphi_x(x) \downarrow\} \to B = \{x \in \mathbb{N} : \exists y \in \mathbb{N}(xR_{\hat{p}}y)\}$$

Insiemi ricorsivamente enumerabili

Sono insiemi che non ammettono algoritmi di classificazione, il meglio a disposizione è l'elenco degli elementi di appartenenza: un programma che scrive in sequenza ogni elemento appartenente all'insieme.
L'insieme $A \subseteq \mathbb{N}$ si dice *ricorsivamente enumerabile* se e solo se:

- $A = \emptyset$;
- $A = Im_f$ con $f : \mathbb{N} \to \mathbb{N} \in \mathbb{T}$;

Le seguenti condizioni sono equivalenti:

1. A è ricorsivamente enumerabile, ovvero $A = Im_f \in \mathbb{T}$;
2. $A = Dom_f$ con $f \in \mathbb{P}$;
3. $\exists R \subseteq \mathbb{N}$ ricorsiva tale che $A = \{x \in \mathbb{N} : \exists y \in \mathbb{N}((x, y) \in R)\}$;

Per dimostrare il secondo punto prendiamo per vero il primo. Dato che è quindi per ipotesi ricorsivamente enumerabile, significa che esiste un programma associato la cui semantica è 1 se l'elemento appartiene all'insieme, 0 altrimenti. Questa semantica è quindi una funzione il cui dominio è proprio l'insieme, e possiamo anche dire che è ricorsiva parziale perché è calcolabile da un programma.

Proviamo il terzo punto basandoci sul secondo. Dato che l'insieme A è il dominio di una funzione ricorsiva parziale, questa funzione deve essere calcolabile da un programma P, sul quale possiamo indurre la relazione che pone in relazione x con y se P termina su x in y passi. Definiamo $B = \{x \in \mathbb{N} : \exists y \in \mathbb{N}(xRy)\}$, ovvero l'insieme degli x per i quali esiste un y col quale sono in relazione. Possiamo quindi affermare sia che A è incluso in B sia che B è incluso in A, e questo significa che sono coincidenti.

Ora implichiamo il primo punto partendo dal terzo. Essendo R ricorsiva parziale esiste un programma che la calcola. Impostiamo una funzione che, preso un numero, lo interpreta come coppia di Cantor e se la parte sinistra è in relazione secondo R con la parte destra, allora resituisce la parte sinistra, altrimenti restituisce un elemento a che ipotizziamo essere in A. Questa funzione è ricorsiva totale dato che si appoggia ad R che è ricorsiva. Inoltre, l'immagine di questa funzione è proprio A: la funzione o restituisce la parte sinistra (che è in relazione con la parte destra, quindi proprio la definizione di A) o direttamente $a \in A$.

Insiemi ricorsivi $\subset$ ricorsivamente enumerabili

Dimostriamo che gli insiemi ricorsivi sono meno espressivi degli insieme ricorsivamente enumerabili.
Mostrare che un insieme ricorsivo è anche ricorsivamente enumerabile è facile, basta modificare la funzione che classifica in $\{0,1\}$ tutti gli elementi in base all'appartenenza all'insieme e farle ritornare invece $\perp$ quando ritornerebbe 0.
Ovviamente l'inclusione è propria in quanto esistono problemi, ad esempio il problema dell'arresto, che non sono ricorsivi ma ricorsivamente enumerabili.

Chiusura degli insiemi ricorsivi

Se pensiamo a due insiemi A e B ricorsivi, abbiamo le rispettive funzioni caratteristiche e i rispettivi programmi di decisione:

1. Sono in grado di dimostrare che $A \cup B$ è ricorsivo?
2. Sono in grado di dimostrare che $A \cap B$ è ricorsivo?
3. Sono in grado di dimostrare che $A^{\complement}$ è ricorsivo?

Per tutti e tre i quesiti posso facilmente trovare un programma che li decide, semplicemente impiegando il minimo, il massimo e l'opposto sul programma normale, o equivalentemente

1. $\chi_{A \cup B}(x) = \chi_A \cdot \chi_B$;
2. $\chi_{A \cap B}(x) = 1 \dot{-} (1 \dot{-} \chi_A(x))(1 \dot{-} \chi_B(x))$;
3. $\chi_{A^{\complement}}(x) = 1 \dot{-} \chi_A(x)$.

Gli insiemi ricorsivi sono perciò un'algebra di Boole.

Insiemi non ricorsivamente enumerabili

Consideriamo il complemento del problema dell'arresto. Teorema: $A^{\complement} = \{x \in \mathbb{N} : \varphi_x(x) \uparrow\}$ non è ricorsivo.
Se lo fosse, grazie alle proprietà precedenti, avremmo $A^{\complement} \text{ ricorsivo} \implies (A^{\complement})^{\complement} = A \text{ ricorsivo}$, ma è assurdo perché sappiamo che il problema dell'arresto non è ricorsivo.

Teorema: A ricorsivamente enumerabile e $A^{\complement}$ ricorsivamente enumerabile $\implies A$ ricorsivo.
Questo ci obbliga a dire che se $A^{\complement}$ fosse ricorsivamente enumerabile, A dovrebbe essere ricorsivo; noi però sappiamo che A nel caso del problema dell'arresto non è ricorsivo, e questo obbligatoriamente porta a concludere che $A^{\complement}$ non è nemmeno ricorsivamente enumerabile.
Dimostrazione: consideriamo un insieme ed il suo complemento, significa che esistono due funzioni f e g implementate dai programmi F e G che elencano uno alla volta gli elementi dei rispettivi insiemi. Un programma che riconosce il primo insieme esegue un passo di F e uno di G e controlla se l'elemento considerato è uguale all'output di uno dei due programmi, ed emette l'appartenenza di conseguenza.
Questo programma riconosce un insieme e termina sempre, in quanto un elemento è sicuramente o in un insieme o nel suo complemento.
L'insieme riconosciuto è pertanto ricorsivo.

Chiusura degli insiemi ricorsivamente enumerabili

Chiaramente gli insiemi ricorsivamente enumerabili non sono un'algebra di Boole in quanto non sono chiusi al complemento. Sono però chiusi ad intersezione ed unione: è possibile mostrare due

programmi che, utilizzando le funzioni di decisione dei due insiemi, risolvono i problemi dell'intersezione e dell'unione, mostrando quindi che sono operazioni incluse nell'insieme delle funzioni ricorsive parziali e quindi proprie di insiemi risorsivamente enumerabili.

Teorema di Rice

Insiemi che rispettano le funzioni

$I \in \mathbb{N}$ rispetta le funzioni se e solo se $a \in I \land \varphi_a = \varphi_b \implies b \in I$.
Sostanzialente sono insiemi che se contengono un programma che implementa una funzione, allora contengono tutti i programmi che implementano tale funzione.

Il teorema di Rice dice che un insieme che rispetta le funzioni non è mai ricorsivo: lo è solo nei casi banali, ovvero quando tale insieme è vuoto o corrispondente a $\mathbb{N}$.

Dimostriamo per assurdo: assumiamo che I rispetti le funzioni e non sia né $\emptyset$ né $\mathbb{N}$, perciò esiste sia $a \in I$ sia $\bar{a} \notin I$. Definiamo ora $t : \mathbb{N} \to \mathbb{N} \in \mathbb{T}$:

$$t(n) = \begin{cases} \bar{a} & n \in I \\ a & n \notin I \end{cases}$$

La funzione t è totale in quanto posso fornire un programma P che usa P_I, ovvero il programma di decisione di I: questo P_I è totale, ovvero si arresta sempre su ogni input, perché per assurdo abbiamo supposto I ricorsivo.
A questo punto, il teorema di ricorsione assicura l'esistenza di un altro programma la cui semantica non viene modificata se viene dato come input a t: $\varphi_d = \varphi_{t(d)}$.
Posso ora individuare due casistiche:

- $d \in I$: poiché I rispetta le funzioni, dato che d sta in I e dato che le loro semantiche sono uguali ($\varphi_d = \varphi_{t(d)}$), allora anche $t(d)$ deve stare in I, ma per definizione in questo caso t restituisce $\bar{a}$, che non sta in I, contraddizione;
- $d \notin I$: in questo caso, la semantica di t è tale per cui $t(d)$ restituisce $a \in I$ per elementi che non appartengono ad I, ma dato che ancora $\varphi_d = \varphi_{t(d)}$ dovremmo avere che $d \in I$, contraddizione.

Si vede quindi che un insieme può essere ricorsivo solo nei casi banali.

Il teorema di Rice fornisce una metodologia per mostrare che un insieme non è ricorsivo: innanzitutto mostro che l'insieme in questione rispetta le funzioni, in secondo luogo mostro che non è né vuoto né l'insieme dei naturali, ed ho come conseguenza che non è ricorsivo.

Verifica automatica del software

Il teorema di Rice ha importanti conseguenze anche per la verifica del software: la verifica consiste nel ricevere specifiche e controllare che la semantica espressa dal programma le rispetti. Controllare automaticamente questa proprietà significa chiedersi se l'insieme PC dei programmi corretti sia ricorsivo.
Tale insieme rispetta le funzioni, perché un programma corretto fa parte di PC e un altro programma che calcola la stessa funzione del primo ne fa anch'esso parte.
Questo però implica automaticamente che, a meno che le specifiche non siano banali, PC non è ricorsivo e non può quindi essere riconosciuto da una funzione totale.
Ovviamente, mentre la correttezza semantica non può essere controlata automaticamente, la correttezza sintattica lo è.

Teoria della complessità

Dato un problema π la teoria della calcolabilità studia se sia possibile risolvere π in modo automatico; la teoria della complessità si chiede invece come funzionino i programmi per risolvere π e quante risorse computazioniali vengano impiegate. Queste risorse sono il *tempo* e lo *spazio*, considerando sistemi di calcolo monoprocessore. Studiare la complessità aiuta anche a definire un modello "quantitativo" della Tesi di Church-Turing.

Macchina di Turing

Usiamo come modello di calcolo formale la Macchina di Turing, che permette di definire formalmente

- il passo di computazione;
- la computazione;
- tempo e spazio (le risorse) consumati dai programmi.

Essa ci permette di applicare metodi matematici per la misurazione del consumo delle risorse e dell'efficienza.

Teoria dei Linguaggi Formali - TLF

Le macchine di Turing sono viste come dispositivi per riconoscere linguaggi.

- Alfabeto: insieme finito di simboli, $\Sigma = \{\sigma_1, \dots, \sigma_n\}$
- Stringhe su Σ: sequenze di simboli sull'alfabeto, $x = x_1 \cdots x_n$ con $x_i \in \Sigma$
- Lunghezza di una stringa: numero di simboli che la compongono, $|x| = n$
- Stringa nulla: stringa composta da 0 simboli, $|\varepsilon| = 0$
- Σ^* indica tutte le stringhe su Σ, Σ^+ indica tutte le stringhe tranne la stringa nulla
- Linguaggio su Σ: $L \subseteq \Sigma^*$. Può essere sia finito che infinito

Macchina di Turing Deterministica - DTM

È composta da un nastro di celle infinito a destra, in ognuna delle quali una testina che si muove di un solo passo a destra o a sinistra ad ogni mossa può sia leggere che scrivere. Questa testina è collegata ad un controllo a stati finiti.
La stringa da analizzare viene posta carattere per carattere sul nastro, e le celle non utilizzate contengono un carattere BLANK che non appartiene all'alfabeto.
La mossa è un passo della computazione della macchina, determinata dallo stato del controllore e dal simbolo letto.
La funzione di transizione calcola proprio sulla base di questi due input (stato e carattere di input) il prossimo stato del controllore, il simbolo da sostituire a quello letto, e il movimento della testina.
La macchina si arresta se si trova in uno stato per cui la funzione di transizione non è definita: *accetta* una stringa se si arresta in uno stato finale o accettante, la *rifiuta* se si arresta in un altro tipo di stato; la macchina può anche andare in loop. Il linguaggio della macchina è composto da tutte le stringhe per cui la macchina stessa termina in uno stato finale, e si dice che una stringa non è accettata anche se va in loop.

Definizione formale

Si può formalizzare una DTM con la seguente sestupla:

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

dove:

- Q è l'insieme degli stati assumibili dal controllo a stati finiti;

- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ sono gli stati finali;
- Σ è l'alfabeto di input per il quale vale $L_M \subseteq \Sigma^*$;
- Γ è l'alfabeto di lavoro per cui vale $\Sigma \subset \Gamma$ dato che deve contenere sicuramente tutti i i simboli dell'alfabeto di input più almeno BLANK;
- $\delta : Q \times \Gamma \to Q \times (\Gamma \setminus \{BLANK\}) \times \{-1, 0, +1\}$ è la funzione di transizione che definisce le mosse: è una funzione parziale perché non è definita per ogni possibile combinazione di stati e input, nei quali casi la macchina si arresta. Inoltre BLANK può solo essere letto, non scritto.

Configurazione

Una fotografia dello stato della macchina in un momento è una tripla che registra lo stato del nastro, la posizione della testina e lo stato del controllore:

$$c = (q, k, w)$$

- $q \in Q$ è lo stato corrente;
- $k \in \mathbb{N}^+$ è la posizione della testina;
- $w \in \hat{\Gamma}$ è il contenuto del nastro tranne i BLANK (non utili).

Configurazione iniziale: $c_0 = (q_0, 1, x)$ con x stringa in input.
Configurazione accettante: $c = (q, k, w)$ con $q \in F$.
Configurazione d'arresto: $c = (q, k, w)$ con $\delta(q, \gamma) = \perp$.
Computazione di M su $x \in \Sigma^*$: è una sequenza di configurazioni indotte dalla funzione di transizione: $c_0 \to c_1 \to \cdots \to c_i \to c_{i+1} \to \dots$ dove $\forall i$ vale che ci si possa spostare da c_i a c_{i+1} grazie a δ.
M accetta x se e solo se esiste una sequenza di configurazioni in cui l'ultima configurazione termina in uno stato finale: $c_0 \overset{*}{\to} c_f$ con $c_f = (q, k, w) | q \in F$.

Riconoscimento di insiemi

Una DTM riconosce linguaggi, ma codificando l'insieme dei naturali come se fosse un alfabeto, posso vedere un insieme $A \subseteq \mathbb{N}$ come un linguaggio su tale alfabeto e quindi $L_A = \{cod(a) : a \in A\}$, e a questo punto posso dire che A è riconoscibile su DTM se esiste una DTM M tale che $L_M = L_A$.

Teorema: la classe degli insiemi riconosciuti da DTM (riconosciuti da procedure) coincide con la classe degli insiemi ricorsivamente numerabili.

Definizione: un *algoritmo deterministico* per il riconoscimento di un insieme $A \subseteq \mathbb{N}$ è una DTM M tale che $L_A = L_M$ e tale che M si arresta su ogni input.

Teorema: la classe degli insiemi riconoscibili da algoritmi deterministici coincide con la classe degli insiemi ricorsivi.

Problemi di decisione

Una DTM M risolve un problema π se e solo se M è un algoritmo deterministico per L_π, dove L_π contiene la codifica in un alfabeto riconosciuto da M di tutti gli elementi $x \in Dom_\pi$ per cui $p(x) = 1$.

Specificare una DTM

Le specifiche di una DTM sono pesanti e di difficile comprensione: bisogna specificare ogni stato e ogni transizione in forma tabellare. Per questo si usa lo pseudocodice per esplicitare le operazioni eseguite dalla sequenza di transizione.

Calcolo di funzioni

Data una funzione $f : \Sigma^* \to \Gamma^*$, una DTM M calcola f se e solo se

- se $f(x)\downarrow$ allora la computazione di M su x termina con $f(x)$ sul nastro;
- se $f(x)\uparrow$ allora la computazione di M su x va in loop.

Si dimostra che le DTM calcolano tutte e sole le funzioni ricorsive parziali, in accordo con la Tesi di Church-Turing.
Le DTM sono SPA:

- calcolano tutte e sole le funzioni ricorsive parziali;
- esiste una DTM universale;
- vale il teorema S_n^m.

Tempo

Si consideri la DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. Definiamo il *tempo di calcolo* $T(x)$ della macchina M su input x come il numero di mosse della computazione, eventualmente infinito.
La complessità in tempo (worst-case) è una funzione $t : \mathbb{N} \to \mathbb{N}$ tale che $t(n) = max\{T(x) : x \in \Sigma^* \wedge |x| = n\}$, ovvero il tempo massimo impiegato da una computazione su tutti i problemi con input di lunghezza n.

Complessità temporale di linguaggi, insiemi, decisioni

Un linguaggio $L \subseteq \Sigma^*$ è riconosciuto in tempo deterministico $f(n)$ se e solo se esiste una DTM M tale che

1. $L = L_M$;
2. $t(n) \leq f(n)$.

Allo stesso modo possiamo dire che

- l'insieme $A \subseteq \mathbb{N}$ è riconosciuto in tempo $f(n)$ se e solo se lo è il linguaggio $L_A = \{cod(a) : a \in A\}$;
- il problema di decisione π è risolvibile in tempo $f(n)$ se e solo se il linguaggio $L_\pi = \{cod(x) : p(x)\}$ è riconosciuto in tempo $f(n)$.

Assumendo di poter riconoscere il lunguaggio L_{PARI} dei numeri pari in tempo $t(n) = n + 1$, le seguenti affermazioni sono equivalenti:

- il linguaggio L_{PARI} è riconoscibile in tempo $n + 1$;
- l'insieme dei numeri PARI è riconoscibile in tempo $n + 1$;
- il problema di decisione PARI è risolubile in tempo $n + 1$.

Classi di complessità

Una classe di complessità è un insieme di problemi che sono risolti entro gli stessi limiti di una o più risorse computazionali:

$$\text{DTIME}(f(n)) = \{\text{linguaggi che vengono accettati da DTM in tempo deterministico } O(f(n))\}$$

Possiamo allo stesso modo definire la classe di funzioni calcolate in tempo $f(n)$ come le funzioni risolte da una macchina di Turing deterministica in $O(f(n))$:

$$\text{FTIME}(f(n)) = \{\text{funzioni (problemi) calcolate (risolti) da DTM in tempo deterministico } O(f(n))\}$$

P ed FP

$$P = \bigcup_{k \geq 0} \text{DTIME}(n^k) = \{\text{linguaggi accettati da DTM in tempo polinomiale}\}$$

$$FP = \bigcup_{k \geq 0} \text{FTIME}(n^k) = \{\text{funzioni calcolate da DTM in tempo polinomiale}\}$$

Raccolgono quei problemi che vengono considerati risolti in tempo efficiente.

Proprietà composizionale

Poniamo che una procedura P sia composta da una prima procedura $p(x)$ il cui output y è riproposto in input alla procedura $q(y)$; supponiamo anche che $p, q = O(n^k)$.
Allora possiamo dire che $t_P(n) = t_p(n) + t_q(|y|)$, ossia la complessità di q dipende dalla lunqhezza di y. Ma y è generato da p in un numero $p(x)$ di passi, quindi deve essere $|y| \leq p(n) = O(n^k)$. Possiamo concludere che $t_P(n) = p(n) + t_q(|y|) \leq p(n) + q(p(n))$ che sono entrambe quantità polinomiali.
Programmi efficienti in tempo che richiamano routine efficienti in tempo rimangono efficienti in tempo.

"Robustezza" dell'efficienza

Problemi risolti in modo efficiente sono risolti efficientemente anche su altre piattaforme di calcolo: se una DTM risolve un problema in tempo polinomiale, allora anche un sistema RAM o WHILE richiederà le stesse risorse.
Come ulteriore prova, se un problema è risolto da una DTM multinastro con k nastri (k costante) in tempo $t(n)$, una DTM mononastro può risolvere lo stesso problema in tempo $O(t^2(n))$.

Tesi di Church-Turing estesa

La classe dei problemi efficientemente risolubili in tempo coincide con la classe dei problemi risolti in tempo polinomiale su DTM (algoritmi con tempo di sviluppo polinomiale).
Problemi in P:

- test di primalità ($t(n) = O(n^6)$);
- raggiungibilità su grafi;
- vari problemi di parsing;
- ...

Problemi in FP:

- problemi di ordinamento;
- alberi minimi;
- cammini minimi;
- vari problemi di ottimizzazione (lineare);
- ...

Spazio

Si consideri la DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. Definiamo lo *spazio di calcolo* $S(x)$ della macchina M su input x come il numero di celle diverse del nastro occupate/visitate durante la computazione.

La complessità in spazio (worst-case) è una funzione $s : \mathbb{N} \to \mathbb{N}$ tale che $s(n) = max\{S(x) : x \in \Sigma^* \wedge |x| = n\}$, ovvero lo spazio massimo occupato da una computazione su tutti i problemi con input di lunghezza n.

DTM a due nastri

Usando questo modello però avremo sempre, con input x, almeno $|x|$ celle occupate del nastro, di conseguenza la complessità spaziale non potrà mai essere sub-lineare.
Definiamo quindi un modello di Macchina di Turing a due nastri: nel primo, di sola lettura, viene depositato l'input, circondato da due caratteri che indicano l'inizio e la fine della stringa da leggere; nel secondo, il nastro di lavoro, una seconda testina potrà sia leggere che scrivere, permettendoci di calcolare lo spazio utilizzato senza contare la lettura dell'input.
Nella definizione formale di DTM non cambia altro che la definizione dell'alfabeto di lavoro, che include ora i due caratteri di terminazione, e la funzione di transizione, che a differenza di prima deve dare l'indicazione per il movimento di due testine, non più una sola.
Possiamo ulteriormente affinare la definizione di spazio utilizzato aggiungendo anche un nastro di output: questo nastro, di sola scrittura, manterrà il risultato ad esempio del calcolo di una funzione, senza occupare spazio nel nastro di lavoro.

Complessità spaziale di linguaggi, insiemi, decisioni

Un linguaggio $L \subseteq \Sigma^*$ è riconosciuto in spazio deterministico $f(n)$ se e solo se esiste una DTM M tale che

1. $L = L_M$;
2. $s(n) \leq f(n)$.

Allo stesso modo possiamo dire che

- l'insieme $A \subseteq \mathbb{N}$ è riconosciuto in spazio $f(n)$ se e solo se lo è il linguaggio $L_A = \{cod(a) : a \in A\}$;
- il problema di decisione π è risolvibile in spazio $f(n)$ se e solo se il linguaggio $L_\pi = \{cod(x) : p(x)\}$ è riconosciuto in spazio $f(n)$.

Classi di complessità

$$\mathrm{DSPACE}(f(n)) = \{\text{linguaggi accettati da DTM in spazio deterministico } O(f(n))\}$$

$$\mathrm{FSPACE}(f(n)) = \{\text{funzioni (problemi) calcolate (risolti) da DTM in spazio deterministico } O(f(n))\}$$

L ed FL

$$L = \text{DSPACE}(log\ n) = \{\text{linguaggi accettati da DTM in spazio logaritmico}\}$$

$$FL = \text{FSPACE}(log\ n) = \{\text{funzioni calcolate da DTM in spazio logaritmico}\}$$

Raccolgono quei problemi che vengono considerati risolti in spazio efficiente.

Proprietà composizionale

Programmi efficienti in spazio che richiamano routine efficienti in spazio rimangono efficienti in spazio.

"Robustezza" dell'efficienza

Problemi risolti in modo efficiente sono risolti efficientemente anche su altre piattaforme di calcolo: se una DTM risolve un problema in spazio logaritmico, allora anche un sistema RAM o WHILE richiederà le stesse risorse.

Tesi di Church-Turing estesa

La classe dei problemi efficientemente risolubili in spazio coincide con la classe dei problemi risolti in spazio logaritmico su DTM (con nastro di input).
Problemi in L:

- raggiungibilità di due nodi in un grafo non diretto;
- parsing di linguaggi regolari;
- parsing di linguaggi context-free (al momento $s(n) = O(log^2 n)$);
- ...

Problemi in FL:

- operazioni aritmetiche;
- permanenti di matrici booleane;
- aritmetica modulare;
- ...

Un problema in L o in FL è anche efficientemente parallelizzabile; al momento non esistono compilatori perfettamente parallelizzabili.

Tempo *vs* spazio

Spesso tempo e spazio sono richieste contrastanti e si lavora molto sul trade-off tra le due: esiste una qualche implicazione tra limiti

temporali e spaziali e viceversa?
Teorema:

$$\mathrm{DTIME}(f(n)) \subseteq \mathrm{DSPACE}(f(n))$$

$$\mathrm{FTIME}(f(n)) \subseteq \mathrm{FSPACE}(f(n))$$

Dimostrazione: $L \in \mathrm{DTIME}(f(n)) \implies \exists$ DTM M che riconosce L in tempo $t(n) = O(f(n)) \implies$ su un input x di lunghezza $n = |x|$, M compie $O(f(n))$ passi. In tale computazione non può occupare più di di $O(f(n))$ celle del nastro di lavoro, potendone occupare al più una per ogni passo $\implies$ M ha complessità spaziale $s(n) = O(f(n)) \implies L \in \mathrm{DSPACE}(f(n))$.

Questo implica che efficienza temporale non porta necessariamente ad efficienza spaziale, perché le risorse rimangono polinomiali (non logaritmiche).

Si può in qualche modo mostrare che anche $\mathrm{DSPACE}(f(n)) \subseteq \mathrm{DTIME}(f(n))$?
Consideriamo un linguaggio $L \in \mathrm{DSPACE}(f(n))$: questo significa che esiste una DTM M che riconosce L in spazio $s(n) = O(f(n))$.
La computazione di M è quindi una sequenza di configurazioni, ma quanto lunga? Può essere al massimo lunga tanto quante le configurazioni possibili di M, se fosse più lunga (per il principio della piccionaia) vorrebbe dire che siamo rientrati in una configurazione già visitata, quindi abbiamo trovato un loop: tanto vale rifiutare.
Una macchina che risolve un problema in spazio $O(f(n))$ quindi si arresta dopo un numero di passi pari al numero di configurazioni che M può assumere su input di lunghezza n:

- numero di stati: $|Q|$;
- posizioni sul nastro di input: $n + 2$;
- posizioni sul nastro di lavoro: $s(n) + 1 = \alpha f(n) + 1$;
- contenuto sul nastro di lavoro: $|\Gamma|^{s(n)}$.

In input possiamo trovarci su uno dei caratteri di input o sui due terminatori; similmente, sul nastro di lavoro possiamo trovarci in una delle $s(n)$ celle sporcate o nel BLANK; il contenuto del nastro di lavoro è una stringa appartenente a Γ e ce ne sarebbero infinite, ma possiamo limitarci a quelle di lunghezza $s(n)$.
Eseguendo vari conti:

$$\begin{gathered} t(n) \leq |Q| \cdot (n+2) \cdot (\alpha f(n) + 1) \cdot |\Gamma|^{\alpha f(n)} \leq \\ \leq |Q| \cdot (n+2) \cdot |\Gamma|^{\alpha f(n)+1} \cdot |\Gamma|^{\alpha f(n)} \leq \\ \leq |Q| \cdot |\Gamma| \cdot (n+2) \cdot |\Gamma|^{2\alpha f(n)} = \\ = |Q| \cdot |\Gamma| \cdot (n+2) \cdot 2^{2\alpha f(n) \cdot log(|\Gamma|)} \end{gathered}$$

$$\implies O(n \cdot 2^{O(f(n))})$$

Questo significa che M deve accettare o rifiutare entro questo numero di passi.

Questo prova che

$$\mathrm{DSPACE}(f(n)) \subseteq \mathrm{DTIME}(n \cdot 2^{O(f(n))})$$

e quindi, in generale, un limite sullo spazio porta ad un limite esponenziale nel tempo.
Teorema:

1. $\mathrm{DSPACE}(f(n)) \subseteq \mathrm{DTIME}(n \cdot 2^{O(f(n))})$
2. $\mathrm{FSPACE}(f(n)) \subseteq \mathrm{FTIME}(n \cdot 2^{O(f(n))})$

Teorema: $L \subseteq P$ e $FL \subseteq FP$
Dimostrazione:

$$L = \mathrm{DSPACE}(log\, n) \overset{1}{\subseteq} \mathrm{DTIME}(n \cdot 2^{O(log\, n)}) = \mathrm{DTIME}(n^2) \subseteq \bigcup_{k \geq 0} \mathrm{DTIME}(n^2) = P$$

Allo stesso modo usando il teorema 2 ottengo $FL \subseteq FP$:

- non si sa se le inclusioni siano strette;
- in teoria quindi algoritmi efficienti in spazio portano ad algoritmi efficienti in tempo (non vale il contrario);
- in pratica il grado del polinomio ottenuto da un algoritmo efficiente in spazio è troppo alto, perciò gli algoritmi efficienti in tempo vanno progettati a sé.

Zona grigia e nondeterminismo

Esiste tutta una classe di problemi, anche di importanza pratica, per i quali ancora non si è trovata una soluzione che sia in tempo meno che esponenziale, e si congettura che non esista. Tali problemi:

- pare abbiano soluzioni inefficienti;
- sono efficientemente verificabili, ovvero data un'istanza e un'informazione generata "magicamente" posso efficientemente decidere l'istanza.

Il paradigma nondeterministico cattura sostanzialmente questa zona grigia.

Algoritmi nondeterministici

Per problemi di decisione di facile verifica, si ipotizza una struttura non-deterministica per cui, presa un'istanza del problema, in una prima *fase congetturale* viene generato "magicamente" un albero di computazioni ed in seguito, nella *fase di verifica*, ogni computazione viene eseguita in modo deterministico.
Un algoritmo non deterministico, per essere corretto, deve dare risposta corretta su tutte le istanze sia con risposta negativa sia con risposta positiva.

Formalizzazione di un modello di calcolo nondeterministico

Definiamo una macchina di Turing nondeterministica per questi algoritmi.
La struttura di una NDTM è la stessa di una DTM, se non per l'aggiunta di un "modulo congetturale" il cui compito è scrivere sulla parte sinistra del nastro (che ora è biinfinito) tutte le strutture magicamente generate che rappresentano i vari universi in cui verranno eseguite le computazioni deterministiche.
La fase congetturale rimane "magica" e quindi senza costo, mentre la fase di verifica lavora su (γ, x), con $\gamma \in \Gamma^*$ struttura per il nondeterminismo e x input del problema; la NDTM accetta $x \in \Sigma^*$ se e solo se $\exists \gamma \in \Gamma^*$ tale che (γ, x) viene deterministicamente accettata.

Riconoscimento di linguaggio e soluzione di problemi di decisione

Analogamente al modello deterministico, possiamo semplicemente dire che:

- un linguaggio $L \subseteq \Sigma^*$ è accettato da un algoritmo nondeterministico se e solo se esiste una NDTM M tale che $L = L_M$;
- un algoritmo nondeterministico per la soluzione del problema π è una NDTM M tale che $L_\pi = L_M$.

Complessità in tempo

Le singole computazioni sono normali computazioni deterministiche, quindi possiamo basarci su queste e dire che una NDTM M ha complessità in tempo $t : \mathbb{N} \to \mathbb{N}$ se e solo se per ogni input $x \in L_M$ con $|x| = n$ esiste una computazione accettata di M che impiega $t(n)$ passi.
Analogamente un linguaggio è accettato con complessità in tempo

nondeterministica $t(n)$ se e solo se esite una NDTM con complessità $t(n)$ che lo accetta.

Classi di complessità nondeterministiche

$\text{NTIME}(f(n)) = \{\text{linguaggi (problemi di decisione) accettati (risolti) con complessità nondeterminis}$

Definiamo inoltre la classe dei problemi efficientemente verificabili nondeterministicamente:

$NP = \bigcup_{k \geq 0} \text{NTIME}(n^k) = \{\text{problemi di decisione che ammettono algoritmi nondeterministici polinom}$

Relazioni tra classi temporali deterministiche e nondeterministiche

Quello che possiamo sicuramente notare ad ora è che $P \subseteq NP$.
Pensando ad esempio ad un linguaggio $L \in \text{DTIME}(f(n))$ accettato dalla DTM M, possiamo vedere questa macchina M come una NDTM N che ignora il modulo congetturale ed esegue sempre la computazione di M su ogni struttura generata: è palese che N accetta quindi L in tempo $t(n) = O(f(n)) \implies L \in \text{NTIME}(f(n))$.
Quindi

$$P = \bigcup_{n \geq 0} \text{DTIME}(n^k) \subseteq \bigcup_{k \geq 0} \text{NTIME}(f(n)) = NP$$

Per quanto riguarda l'inclusione inversa, dobbiamo provare a chiederci quanto costa togliere la "magia" nondeterministica, in quanto le macchine nondeterministiche sono un costrutto teorico non realizzabile concretamente.
Una DTM M che implementi deterministicamente una NDTM N potrebbe generare tutte le strutture $\gamma \in \Gamma^*$ e su ognuna di esse eseguire M: possiamo limitare i passi eseguiti da ogni computazione ricordando che $\text{DSPACE}(f(n)) \subseteq \text{DTIME}(n \cdot 2^{O(f(n))})$.
Resta il problema che le strutture sono infinte: anche queste possono essere limitate, dato che dovendo M compiere al più $O(f(n))$ passi, non ha senso generare strutture più lunghe di $O(f(n))$.
La complessità però esplode dalla polinomiale nondeterministica a una esponenziale deterministica: $t(n) = |\Gamma|^{O(f(n))} \cdot O(f(n)) = O(f(n) \cdot 2^{O(f(n))})$.
Questo porta a dire che

$$\text{NTIME}(f(n)) \subseteq \text{DTIME}(f(n) \cdot 2^{O(f(n))})$$

Con questo riusciamo a dire che i problemi in NP hanno sicuramente soluzione deterministica in tempo esponenziale, ma non siamo riusciti a confermare né escludere che $P = NP$.

È da notare che NP non significa Non-Polinomiale, ma *Nondeterministico Polinomiale*, ovvero racchiude gli algoritmi risolti efficientemente col paradigma nondeterministico.

Gerarchia in NP

Per dimostrare che $NP \subseteq P$ (e quindi di conseguenza $P = NP$) potrei prendere ogni problema in NP e trovarne una soluzione in P efficiente. Ciò è impossibile in quanto i problemi in NP sono infiniti: per questo si procede coi seguenti passi.

1. Stabilisco una *relazione di difficoltà* tra problemi in NP: dati $\pi_1, \pi_2 \in NP$, $\pi_1 \leq \pi_2$ significa che trovare una soluzione efficiente per π_2 porta automaticamente ad una soluzione efficiente per π_1, ovvero π_1 non è più difficile di π_2.
2. Trovo i problemi più difficili in NP: π è più difficile in NP se $\pi \in NP \land \forall \tilde{\pi} \in NP : \tilde{\pi} \leq \pi$.
3. Restringo la ricerca di algoritmi efficienti ai soli problemi difficili di NP.

Riduzione polinomiale

Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$ diciamo che L_1 si riduce polinomialmente ad L_2, e scriviamo $L_1 \leq_P L_2$, se e solo se esiste $f : \Sigma^* \to \Sigma^*$ tale che:

1. $f \in FP$, ovvero è calcolabile in tempo polinomiale da un DTM;
2. $\forall x \in \Sigma^* : x \in L_1 \iff f(x) \in L_2$.

Teorema: siano due linguaggi $A, B \subseteq \Sigma^*$ tali che $A \leq_P B$. Allora $B \in P \implies A \in P$.
Dimostrazione.
$A \leq_P B \implies \exists f$ che effettui la riduzione polinomiale.
Questa funzione f può semplicemente tradurre $a \in A$ e restituire $b \in B$, applicare un algoritmo per B e rispondere per a.
Se l'algoritmo per riconoscere B è efficiente, allora la procedura appena illustrata è composta da due procedure efficienti, che come sappiamo è ancora efficiente.
Questo comporta che riconoscere A è un problema risolto efficientemente, non più difficile che riconoscere B.

Problemi NP-completi

Un problema di decisione π è NP-completo se e solo se:

1. $\pi \in NP$;

2. $\forall \tilde{\pi} \in NP : \tilde{\pi} \leq_P \pi$.

Teorema: sia $\pi \in NP\text{-C}$ e $\pi \in P$. Allora $NP \subseteq P$ ($P = NP$).
Dimostrazione.
Se $\pi \in NP\text{-C}$ ogni altro problema in NP si riduce polinomialmente a π. Ma assumendo che $\pi \in P$ e ricordando che $A \leq_P B \wedge B \in P \implies A \in P$ abbiamo che ogni problema in NP appartiene a P, e possiamo quindi concludere che P ed NP coincidono.

Dimostrare che un problema è in $NP\text{-C}$

1. Dimostrare che $\pi \in NP$;
2. scegliere un problema x notoriamente $NP\text{-completo}$;
3. dimostrare $x \leq_P \pi$;
4. per la transitività della riduzione polinomiale si ottiene che $\pi \in NP\text{-C}$.

Ancora su relazioni tra complessità spaziali e temporali

Ricordiamo che $L = \text{DSPACE}(log\, n)$ e $P = \bigcup_{k \geq 0} \text{DTIME}(n^k)$ sono le classi dei problemi risolti efficientemente rispettivamente in spazio e in tempo. Abbiamo potuto dimostrare che $L \subseteq P$, ovvero che una soluzione temporalmente efficiente porta ad una soluzione spazialmente efficiente.
Per provare il verso opposto dell'inclusione ci muoviamo in maniera analoga a quanto appena fatto: selezioniamo dei problemi $\pi \in L$ che siano difficili, ovvero che risolti questi tutti gli altri troveranno automaticamente una soluzione efficiente.

Gerarchia in P

Per dimostrare che $P \subseteq L$ (e quindi di conseguenza $P = L$) potrei prendere ogni problema in P e trovarne una soluzione in P efficiente in spazio. Ciò è impossibile in quanto i problemi in P sono infiniti: per questo si procede coi seguenti passi.

1. Stabilisco una *relazione di difficoltà* tra problemi in P: dati $\pi_1, \pi_2 \in P$, $\pi_1 \leq \pi_2$ significa che trovare una soluzione efficiente per π_2 porta automaticamente ad una soluzione efficiente per π_1, ovvero π_1 non è più difficile di π_2.
2. Trovo i problemi più difficili in P: π è più difficile in P se $\pi \in P \wedge \forall \tilde{\pi} \in P : \tilde{\pi} \leq \pi$.
3. Restringo la ricerca di algoritmi efficienti ai soli problemi difficili di P.

Riduzione logaritmica

Dati due linguaggi $L_1, L_2 \subseteq \Sigma^*$ diciamo che L_1 si log-space riduce ad L_2 , e scriviamo $L_1 \leq_l L_2$, se e solo se esiste $f : \Sigma^* \to \Sigma^*$ tale che:

1. $f \in FL$, ovvero è calcolabile in spazio logaritmico da un DTM;
2. $\forall x \in \Sigma^* : x \in L_1 \iff f(x) \in L_2$.

Teorema: siano due linguaggi $A, B \subseteq \Sigma^*$ tali che $A \leq_l B$. Allora $B \in L \implies A \in L$.
Dimostrazione.
$A \leq_P B \implies \exists f$ che effettui la log-riduzione.
Questa funzione f può semplicemente tradurre $a \in A$ e restituire $b \in B$, applicare un algoritmo per B e rispondere per a.
Se l'algoritmo per riconoscere B è efficiente, allora la procedura appena illustrata è composta da due procedure efficienti, che come sappiamo è ancora efficiente.
Questo comporta che riconoscere A è un problema risolto efficientemente, non più difficile che riconoscere B.

Problemi P-completi

Un problema di decisione π è P-completo se e solo se:

1. $\pi \in P$;
2. $\forall \tilde{\pi} \in P : \tilde{\pi} \leq_l \pi$.

Teorema: sia $\pi \in P\text{-C}$ e $\pi \in L$. Allora $P \subseteq L$ $(L = P)$.
Dimostrazione.
Se $\pi \in P\text{-C}$ ogni altro problema in P si log-riduce a π. Ma assumendo che $\pi \in L$ e ricordando che $A \leq_l B \wedge B \in L \implies A \in L$ abbiamo che ogni problema in P appartiene a L, e possiamo quindi concludere che L e P coincidono.

Dimostrare che un problema è in $P\text{-C}$

1. Dimostrare che $\pi \in P$;
2. scegliere un problema x notoriamente $P\text{-completo}$;
3. dimostrare $x \leq_l \pi$;
4. per la transitività della riduzione polinomiale si ottiene che $\pi \in P\text{-C}$.

Grazie!

Carlo Meneghetti