Informatica teorica

Appunti presi da Andrea Berlingieri durante le lezioni di Andrea Asperti nell'anno accademico 2018/2019

7 aprile 2019

Disclaimer

Questo documento e' stato creato sulla base dalle lezioni del professore Andrea Asperti di Informatica Teorica tenute nell'Anno Accademico 2018/2019 a Bologna. Il contenuto deriva in parte dalle lezioni frontali e in parte direttamente dal materiale di supporto del corso, che comprende dei lucidi ed una dispensa per la parte di Calcolabilita'. In ogni caso il contenuto e' stato soggetto di una interpretazione personale del sottoscritto per quanto riguarda la forma, l'ordine e alcuni dettagli degli argomenti presentati nel corso.

Di conseguenza, sebbene il professor Asperti sia a conoscenza dell'esistenza di questo documento e abbia dato il suo consenso alla pubblicazione, non si tratta di materiale ufficiale del corso. Puo' essere utilizzato come materiale di supporto allo studio della materia e spero che risulti utile a questo scopo. Cio' detto il documento non e' stato certificato da nessuno e non posso garantire che sia esente da refusi o errori logici, i quali possono essere stati frutto di distrazione personale durante le lezioni, di una interpretazione erronea dei contenuti trattati, o di altri fattori. In ogni caso mi assumo la responsabilita' di qualsiasi errore presente nel documento. Vi prego di non disturbare il professor Asperti al riguardo ma piuttosto di rivolgervi a me.

Ribadisco nuovamente che non c'e' alcuna garanzia di correttezza di questo documento, e che potete farne uso a vostro rischio e pericolo. Se un refuso nel documento vi portasse, ad esempio, a sbagliare una risposta all'esame sappiate che non c'e' nessuno a cui potreste fare ricorso per questo tipo di incidente. Vi assumete il rischio di sbagliare se vi basate interamente e ciecamente su questo documento.

Cio' detto ci tengo a dire che nel creare questo documento ci ho messo tutto il mio impegno per ottenere un risultato che fosse il piu' chiaro, corretto e completo possibile per i miei sforzi. Spero che questo si rifletta nel risultato finale e che lo apprezziate. Ogni correzione, suggerimento, o partecipazione costruttiva al lavoro e' bene accetta e sara' dovutamente riconosciuta quando andro' a stendere la lista delle persone che hanno contribuito a creare questo documento.

Spero infine, ancora una volta, che questo materiale vi aiuti nello studio della materia e che risulti il piu' utile possibile. Buono studio.

Indice

Parte I

Calcolabilità

Capitolo 1

Numerabilità e funzioni

Questi appunti riguardano le lezioni introduttive sulla numerabilità.

1.1 Finito vs. Infinito

Siamo interessati a questa problematica. Perché? Se avessimo solo da calcolare funzioni di input finiti con output finiti non avremmo troppe difficoltà. I nostri programmi diventano interessanti quando andiamo a lavorare su degli input infiniti. Ad esempio, potremmo avere una struttura dati dinamica come input. Ancora, un programma che lavora su una lista dovrebbe ragionevolmente funzionare per tutte le liste. I nostri algoritmi dovrebbero essere in grado di accettare una certa quantità di input infiniti diversi.

L'infinito è anche interessante a livello di complessità computazionale. Nella parte di complessità computazionale guarderemo al comportamento asintotico del programma al crescere della dimensione dell'input.

Un'altra problematica interessante è la seguente: a volte l'input stesso dei nostri programmi è infinito. Ad esempio, il nostro programma potrebbe essere un automa per un linguaggio libero da contesto. Il nostro input, un linguaggio, può essere un insieme infinito.

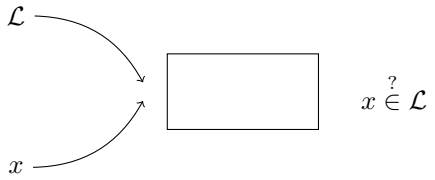


Figura 1.1: Schematizzazione di un automa per un linguaggio $\mathcal{L}$

Come facciamo a dare al programma un input infinito? L'unico modo è fornire al programma una descrizione finita dell'input. Creiamo quindi una

struttura generativa finita che, da un insieme finito di oggetti, ci permette di crearne dinamicamente di nuovi.

Questo esempio ci mostra come in questo ambito abbiamo un doppio livello di discussione:

- uno descrittivo: a questo livello parlo di quello che mi serve per descrivere ciò di cui voglio parlare (e.g. la grammatica per un linguaggio). Vi sono una serie di problematiche legate a questo livello: la descrizione che do è giusta? È unica? Ce n'è una canonica? Ad esempio, date due grammatiche posso decidere se queste definiscono lo stesso linguaggio?

- uno denotazionale: a questo livello parlo veramente del mio oggetto in questione, a prescindere da qualsiasi descrizione se ne possa dare (e.g. un linguaggio). Abbiamo le stesse problematiche già viste per il livello descrittivo

Il livello descrittivo è anche detto intensionale, quello denotazionale invece estensionale.

Anche per i programmi vale questa distinzione: la funzione calcolata da un programma fa parte del livello denotazionale. Una funzione è descritta mediante il suo grafo. Non avendo modi diretti, ovvero finiti, di descrivere la funzione usiamo un programma, che fa parte del livello descrittivo. Una problematica interessante è la seguente: esiste una maniera di descrivere una funzione che non mi dia anche un metodo effettivo per calcolarla (non un programma insomma)?

Siamo costretti ad avere questi due tipi di livelli? Sì se l'oggetto di cui vogliamo parlare è infinito. La comunicazione presuppone una descrizione finita dell'oggetto infinito.

Tutta la nostra intuizione è basata sul finito. Tuttavia molte cose che non valgono per il finito valgono per l'infinito. Ad esempio è possibile mettere in corrispondenza biunivoca un sottoinsieme stretto di un insieme infinito con l'insieme stesso, a differenza del caso finito. Questo è fuori dalla nostra intuizione immediata degli insiemi.

A volte vogliamo fare ricerche infinite, e.g. su strutture dati infinite. Ad esempio potrei voler verificare che una stringa x appartenga ad un linguaggio $\mathcal{L}$ generando tutte le stringhe da una grammatica per $\mathcal{L}$ e decidere "Sì" se x compare. Ho un modo per dire se in una ricerca troverò quello che cerco? In generale no. Questo problema è fondamentale perché è frequentissimo. Parleremo in questi casi qui di semi-decisione. Notiamo inoltre che sul complementare, ovvero $x \notin \mathcal{L}$, non possiamo dire niente.

Noi considereremo funzioni da $\mathbb{N}$ a $\mathbb{N}$ o, al massimo, da $\mathbb{N}^k$ a $\mathbb{N}$. Questo perché i numeri naturali sono la più semplice struttura infinita. Questo non ci pone limitazioni di alcun tipo. Se noi siamo interessati ad aspetti di calcolabilità tutto è alla fine riconducibile ad un numero binario. Questa traduzione da un oggetto qualsiasi ad un numero naturale prende il nome di Gödelizzazione. Ai tempi (~ 1930) fu un'idea rivoluzionaria, ma dal punto di vista di un informatico dovrebbe sembrare più naturale.

1.2 Numerabilità

Le seguenti sono note alle slide

Definizione 1.1. Un insieme A si dice numerabile se esiste una funzione suriettiva f dall'insieme dei numeri naturali $\mathbb{N}$ in A. f è detta funzione di enumerazione.

Un esempio di insieme numerabile è $\mathbb{N}$. Inoltre ogni sottoinsieme di un insieme numerabile è ancora numerabile. La funzione f ci dà anche un metodo di enumerazione degli elementi del codominio di f.

Lemma 1.1. *Sia A numerabile. Allora $\{*\} \oplus A$ è ancora numerabile.*

Dimostrazione. Sia $f : \mathbb{N} \to A$ la funzione di enumerazione di A. definiamo $g : \mathbb{N} \to \{*\} \oplus A$ nel modo seguente:

$$g(x) = \begin{cases} g(0) = * \\ g(x+1) = f(x) \end{cases}$$

Chiaramente g è suriettiva. □

Aggiungere un elemento non intacca la numerabilità di un insieme. Basta uno shift per avere una nuova enumerazione di A. La numerabilità è resistente all'operazione di estensione.

Corollario 1.1. *Sia A numerabile e D finito. Allora $D \oplus A$ è ancora numerabile.*

Lemma 1.2. *L'unione di due insiemi numerabili A e B è ancora numerabile.*

Dimostrazione. Siano f e g le due funzioni di enumerazione di A e B. $A \oplus B$ è allora enumerato dalla seguente funzione h:

$$h(x) = \begin{cases} h(2x) = f(x) \\ h(2x+1) = g(x) \end{cases}$$

□

Corollario 1.2. *Un unione finita di insiemi numerabili è numerabile.*

Corollario 1.3. *Se D è finito e A è numerabile allora $D \times A$ è numerabile.*

Dimostrazione. Per A, B numerabili abbiamo $A \oplus B$ numerabile. Prendiamo in considerazione l'unione disgiunta di A con se stesso k volte:

$$\overbrace{\underset{0}{A} \oplus \underset{1}{A} \oplus \cdots \oplus \underset{k}{A}}^{k \text{ volte}}$$

Questo insieme è chiaramente numerabile. Chiamiamolo A_k. Abbiamo che:

$$A_k = \bigcup_{i=0}^{k-1} \{< i, a > |\ a \in A\} = \{< i, a > |\ a \in A, i \in \{0, \ldots, k-1\}\} = A \times K$$

da cui l'asserto. □

Possiamo enumerare $\mathbb{N} \times \mathbb{N}$? Vediamo com'è fatto questo insieme. Avremo $(0,0), (0,1), (0,2), \ldots$, dopodiché, sulla prossima riga, avremo $(1,0), (1,1), (1,2), \ldots$, e così via all'infinito. Il modo più conveniente per visualizzarlo è pensare a dei punti nel piano.

Possiamo enumerarli? Sarebbe sbagliato numerare per righe o per colonne. Questo perché le righe e le colonne sono infinite, iterando su una riga non passerò mai alla prossima. Che tecnica usiamo? Dove tailing. Numeriamo per diagonali. Ce ne sono altre di numerazioni possibili. Va bene qualsiasi "gioco dell'oca" che passa per ogni coppia una e una sola volta.

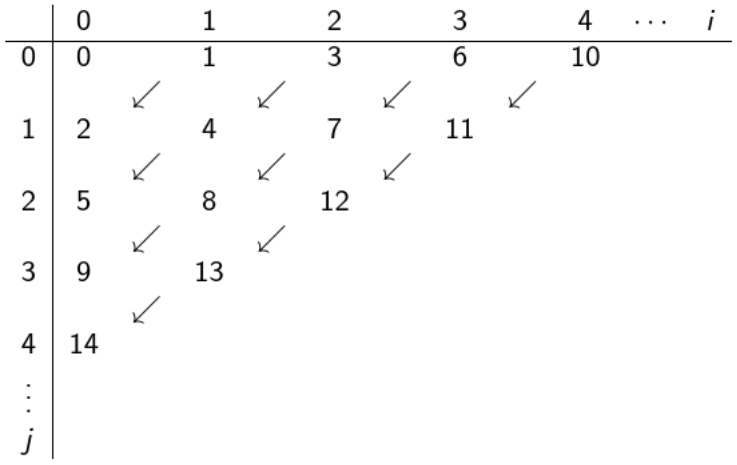


Figura 1.2: Dove tailing

Questa codifica si ottiene nel seguente modo:

$$< i, j >= j + \sum_{k=0}^{i+j} k = \frac{(i+j)^2 + i + 3j}{2}$$

ed è giustificata dal ragionamento seguente: la somma delle componenti i ed j per i punti su di una stessa diagonale è costante e pari al numero di diagonali già interamente percorse. Il numero di punti del piano già visitati in queste diagonali è

$$\sum_{k=0}^{i+j} k = \frac{(i+j)(i+j+1)}{2}$$

Per visitare l'elemento $< i, j >$ dovrò ancora percorrere j passi lungo l'ultima diagonale, da cui la formula precedente.

Importante: Il dove tailing non è la diagonalizzazione.

Come corollario abbiamo che il prodotto cartesiano di due insiemi numerabili è numerabile.

È facile dimostrare che posso avere delle biezioni tra $\mathbb{N}^k$ e $\mathbb{N}$, in modo da codificare delle tuple di numeri con un numero solo. In altre parole c'è una procedura algoritmica che mi permette di passare da una tupla al corrispondente numero n e da n alla corrispondente tupla. Di conseguenza $\mathbb{N}^k$, per k naturale, è ancora numerabile.

Lemma 1.3. *L'unione di un insieme numerabile di insiemi numerabili è ancora numerabile.*

Dimostrazione. Sia A un insieme numerabile e sia f la sua funzione di enumerazione. Sia $\{B_a \mid a \in A\}$ una collezione di insiemi numerabili, ciascuno enumerato da una funzione g_a. La funzione $h : \mathbb{N} \times \mathbb{N} \to \bigcup_{a \in A} B_a$ definita da

$$h(n, m) = g_{f(n)}(m)$$

è suriettiva. □

Lemma 1.4. *Se A è un insieme numerabile, anche $\bigcup_{i \in \mathbb{N}} A^i$ è numerabile.*

Consideriamo l'operatore di Kleene sull'alfabeto Σ: Σ^*. Anche questo insieme è numerabile, perché è l'unione numerabile di insiemi numerabili (ricordiamo che A^k è numerabile).

Un altro insieme numerabile interessante è l'insieme delle parti finite:

$$\mathscr{P}_{fin}(\mathbb{N}) = \{A \mid A \subseteq \mathbb{N}, A \text{ } \textit{è finito}\}$$

Abbiamo due piani di infinità nei programmi: l'infinità dell'input e l'infinità del tempo di calcolo, in quanto il mio programma può divergere su un dato input.

L'insieme di funzioni da A a B ha cardinalità B^A.

Possiamo caratterizzare l'insieme delle coppie su A, $A \times A = A^2$, come una funzione dall'insieme BOOL $= \{0, 1\}$ all'insieme A. $f(x)$ può essere così definita: data una coppia $< a_1, a_2 >$, $f(x)$: $x \mapsto$ if x then a_1 else a_2. Al posto dei booleani potremmo avere un qualsiasi insieme di cardinalità 2.

A^K è isomorfo all'insieme delle funzioni $f : K \to A$.

La funzione è utile sia come strumento di calcolo che come strumento di codifica dei dati.

Lo spazio delle funzioni da un insieme finito ad un insieme numerabile è ancora numerabile.

1.3 Diagonalizzazione

Consideriamo lo spazio delle funzioni da un insieme numerabile ad un altro insieme numerabile. Ancora più semplicemente, possiamo considerare le funzioni da $\mathbb{N}$ a 2 (o BOOL).

L'insieme delle funzioni da A a BOOL è isomorfo all'insieme delle parti di A.

Per denotare un sottoinsieme si può dare una funzione caratteristica, che restituisce 1 se un elemento fa parte del sottoinsieme e 0 altrimenti.

Teorema 1.1. *Per ogni insieme A non esiste una funzione suriettiva da A in $\mathscr{P}(A)$.*

Dimostrazione. Sia $f : A \to \mathscr{P}(A)$ una funzione suriettiva da A all'insieme delle parti di A. Data f posso costruire parametricamnente Δ_f così fatto:

$$\Delta_f = \{a \mid a \in A, a \notin f(a)\}$$

Essendo f suriettiva esiste a tale che $f(a) = \Delta_f$. Ci chiediamo ora, $a \in f(a)$?

$$a \in f(a) \iff a \in \Delta_f \iff a \in A \wedge a \notin f(a)$$

Ma questo è assurdo. □

La tecnica con cui si dimostra questo teorema è detta tecnica diagonale, o diagonalizzazione. Un'idea intuitiva che giustifica il nome è la seguente: consideriamo una tabella indicizzata per colonne dagli $a \in A$ e sulle righe dagli $f(a)$ per ogni $a \in A$. Nella cella (i, j) della tabella avrò 1 se $a_j \in f(a_i)$ e 0 altrimenti. Abbiamo per ogni riga la funzione caratteristica di $f(a)$, per ogni $a \in A$. Abbiamo ora che la funzione caratteristica di Δ_f è costruita andando sulla diagonale e prendendo l'elemento j in Δ se in corrispondenza sulla diagonale, ovvero alla riga i, ho 0, lasciandolo fuori altrimenti. Di conseguenza la funzione caratteristica che vado a costruire per definire Δ è diversa da tutte le funzioni caratteristiche sulle righe almeno per un elemento.

La diagonalizzazione è un metodo semplice per creare un nuovo elemento diverso da tutti quelli di una lista sotto certe ipotesi.

Dimostriamo che i numeri nell'intervallo $[0, 1[$ sono una quantità non numerabile. Come possiamo descrivere i numeri reali? Possiamo farlo, ad esempio, con la loro rappresentazione decimale: una infinita successione delle cifre del numero, per ogni numero.

Supponiamo di avere una enumerazione dei numeri reali $r_0, r_1, r_2, \ldots$. Disegnamo una tabella con le righe indicizzate dai numeri reali nell'ordine dato dall'enumerazione e con le colonne indicizzate dai numeri naturali. Per ogni numero della enumerazione possiamo scrivere, in corrispondenza della colonna j, la sua cifra j, in ordine da sinistra verso destra.

Consideriamo la diagonale della tabella. Consideriamo il mapping che per gli elementi dell'insieme $\{0, \ldots, 4\}$ restituisce 7 mentre per gli elementi dell'insieme $\{5, \ldots, 9\}$ restituisce 3. Se prendiamo la diagonale e alle sue cifre applichiamo il mapping creato otteniamo un numero che è diverso da tutti gli elementi dell'enumerazione almeno per la cifra sulla diagonale. Di conseguenza non fa parte dell'enumerazione. Ma questo è assurdo, dato che avevo supposto di poter enumerare tutti i numeri reali dell'intervallo $[0, 1[$.

Le funzioni da $\mathbb{N}$ a 2, a $\{0, \ldots, 9\}$, a $\mathbb{N}$ hanno la stessa cardinalità, quella del continuo, che non è numerabile.

Esistono quindi numeri reali per i quali non esiste una maniera per descriverli.

1.4 Definibilità vs. Calcolabilità

Le funzioni da $\mathbb{N}$ a $\mathbb{N}$ di cui possiamo parlare hanno cardinalità numerabile. Anche i programmi sono una quantità numerabile.

Non è però questo il problema che vogliamo affrontare. È evidente per ragioni di cardinalità che ci sono funzioni che non potremo calcolare. Il problema a cui siamo interessati è: tutte le funzioni che possiamo definire sono calcolabili o ci sono funzioni definibili ma non calcolabili?

Abbiamo inizialmente il problema di cosa significa definibile. La nozione di definibilità dipende dal contesto e dal potere espressivo del linguaggio che uso (e.g. linguaggio dell'aritmetica, linguaggio del secondo ordine, ecc.).

Possiamo formalizzare la nozione di definibilità? No. È possibile dimostrare che l'idea di definibilità non è definibile. Non è possibile dire quando una cosa è ben definita.

Supponiamo di avere un criterio che scansiona le stringhe e decide se una è una buona definizione o no. Possiamo quindi definire una sequenza di funzioni e quindi dare una numerazione delle funzioni definibili. Si può dimostrare che questa definizione di definibilità è incompleta, non cattura bene il nostro senso intuitivo di definibilità.

Intuitivamente, possiamo creare una tabella con le righe indicizzate dalle funzioni definite e per colonne dai numeri naturali. La casella (i, j) contiene il risultato della funzione f_i sul numero j. Supponiamo inoltre di avere funzioni binarie, ovvero che restituiscono 1 o 0. Questa semplificazione non fa perdere di generalità.

Con un procedimento diagonale possiamo definire una nuova funzione che non sta nell'enumerazione. Ad esempio, sia $f(n) = 1 - d_n(n)$. Se questa definizione intuitivamente valida non è presente nell'enumerazione allora il mio criterio è incompleto. Supponiamo che nella mia enumerazione la mia funzione f compaia in posizione m. Allora avremmo $d_m(m) = f(m) = 1 - d_m(m)$. Ma questo è assurdo.

Se fissiamo un linguaggio l'insieme delle funzioni definite su quel linguaggio è numerabile. Esiste quindi una funzione, per quel linguaggio, che mi dica se una definizione è valida. Questa funzione, tuttavia, non è calcolabile.

1.4.1 Paradosso di Russel

Questo è un vero paradosso, che mise in crisi la matematica agli inizi del XX secolo.

Possiamo pensare ad un insieme come ad una collezione di cose che rispetta certe proprietà. Data una proprietà $P(x)$ possiamo costruire un insieme che la rispetti:

$$\frac{P(t)}{t \in \{x \mid P(x)\}}$$

Vale anche l'implicazione inversa

Pensiamo alla proprietà $P(x) =$ "$x \notin x$". Possiamo costruire l'insieme $U = \{x \mid x \notin x\}$. Ci chiediamo ora: $U \in U$? Abbiamo che $U \in U \iff U \notin U$. Ma questo è paradossale.

Il problema è che all'inizio del secolo si era tentato di basare la teoria insiemistica sul principio di comprensione. Per quanto comodo e bello questo, nella forma che abbiamo visto, è causa di paradossi. Dobbiamo rinunciare all'idea che basti pensare ad una proprietà per poter creare un insieme che la rispetti.

1.4.2 Paradosso di Berry

C'è un principio importante nei numeri naturali che dice che dato un sottoinsieme finito dei numeri naturali esiste un elemento dei naturali che è il più piccolo numero naturale che non appartiene a questo sottoinsieme.

Supponiamo di definire i numeri con stringhe di lunghezza d. Abbiamo quindi un insieme di numeri che posso definire con al più d caratteri. Possiamo definire n_d come il più piccolo numero non definibile con meno di d caratteri, e sappiamo che esiste per il principio precedentemente riportato.

Prendiamo ad esempio $d = 100$. Abbiamo che n_{100} non è definibile con meno di 100 caratteri. Eppure è definito dalla stringa: "n_{100} non è definibile con meno di 100 caratteri", che ha meno di 100 caratteri. Questo è assurdo.

Ci sono paradossi e paradossi. Alcune sono cose che sembrano strane ma in realtà non lo sono così tanto. Altri sono proprio cattivi, mettono in questione aspetti fondamentali della nostra intuizione.

Dove sta il problema del paradosso di Berry? Nella stringa s, perché la nozione di definibilità non è definibile, ed s la usa. La definizione data non è una buona definizione.

Ci chiediamo: data una enumerazione θ delle sentenze dell'aritmetica è possibile generare una formula che mi dice il valore di verità di una sentenza dell'enumerazione?

Ad ogni sentenza dell'enumerazione associamo un numero, detto di Gödel. Questo perché non possiamo parlare delle sentenze in sè in aritmetica, ma possiamo parlare solo di numeri. Quindi ci serve un numero per parlare della sentenza.

La verità è intesa sul modello dei numeri naturali.

Vogliamo una formula *Vera* tale che

$$\mathcal{N} \models A \iff \mathcal{N} \models \mathit{Vera}(g(A))$$

dove $g(A)$ rappresenta il numero di Gödel di A.

Per i numeri naturali c'è un lemma detto lemma di diagonalizzazioe. Questo dice che dato un predicato è sempre possibile trovare una formula S tale che $\mathcal{N} \models S \iff \mathcal{N} \models P(g(S))$. È possibile, per una sentenza, dire "quel predicato P vale per me". La dimostrazione è interessante perché costruttiva.

Prendiamo come $P(x)$ la formula $\neg \mathit{Vero}(x) : \mathcal{N} \models P(x) \iff \mathcal{N} \models \neg \mathit{Vero}(x)$. Possiamo allora costruire S tale che $\mathcal{N} \models S \iff \mathcal{N} \models \neg Vero(S)$. Avremmo quindi $\mathcal{N} \models S \iff \mathcal{N} \models \neg S$. Ma questo è paradossale. Quindi non

è definibile, nel linguaggio dell'aritmetica, una formula che, data una sentenza dell'aritmetica, mi dice se questa è vera, in senso aritmetico. Questo risultato è soprendentemente forte e va sotto il nome di teorema di Tarski.

Un'idea simile viene usata nel teorema di Gödel non con la nozione di verità matematica ma con la nozione di dimostrabilità. Il risultato è che la formula che afferma la propria indimostrabilità non è dimostrabile. Da ciò il sistema logico è incompleto, ovvero c'è una formula indimostrabile.

Capitolo 2

Il formalismo primitivo ricorsivo

2.1 Ricorsione primitiva

Nei linguaggi di programmazione siamo abituati all'autoreferenzialità, o ricorsione. In generale bisogna fare attenzione.

Esistono forme esplicite ed implicite di autoreferenziazione. Per le seconde ho un oggetto che ha una sezione universale nel suo scope e tale che l'oggetto stesso ci rientra. È una sorta di ordine superiore.

Ad esempio:

> "Tutte le sentenze universali sono false"

Questa frase è implicitamente autoreferenziale. Nel suo raggio di applicabilità include se stessa.

Per le formule aperte non ha senso parlare di verità. Si può parlare solo di verità in caso di formule chiuse, ovvero sentenze. I teoremi sono tutti chiusi. La sentenza sopra è dimostrabilmente falsa.

Ci chiediamo se la definibilità di una funzione implichi la sua calcolabilità e anche se vale il viceversa. Sappiamo già che la nozione di definibilità è legata al linguaggio che utilizzo.

Possiamo dare una definizione precisa di calcolabilità? È una problematica delicata, legata al sistema di calcolo che utilizzo e alla definizione formale di algoritmo, che non è banale.

Ci chiediamo, vogliamo davvero avere un loop infinito? In certi casi sì, ad esempio per stampare una lista di numeri primi ed interromperla quando voglio. Ma non è comune, molto spesso ci basta una iterazione determinata.

Le funzioni dell'informatica sono di una categoria particolare: sono funzioni parziali. Le funzioni totali calcolabili sono un sottoinsieme delle funzioni parziali.

Ci poniamo il problema di giustificare l'inclusione, nei nostri linguaggi di programmazione, di costrutti che possono causare divergenza. Hanno un'utilità che giustifica il rischio di non terminazione dei programmi.

Noi siamo abituati a scrivere funzioni ricorsive. Ad esempio, $Fact(n+1) = (n+1) * Fact(n)$, assieme al caso base. Dal punto di vista matematico questo oggetto è strano. Non si può definire una cosa in funzione di se stessa. C'è modo di definire in una maniera più sensata a livello matematico una funzione ricorsiva, che sottintende una procedura? È una problematica interessante.

La ricorsione primitiva è un tipo di ricorsione simile a quella finora vista ma con dei limiti semantici. In particolare possiamo dare il vincolo: nel corpo della definizione di $f(n+1)$ ci si può richiamare ricorsivamente solo su n. È equivalente dal punto di vista dell'espressività un indebolimento di questo vincolo, in cui si limitano invece le chiamate ricorsive a oggetti più piccoli di $n+1$.

La ricorsione primitiva, per vincoli strutturali, garantisce la terminazione. Siamo interessati a questa classe di funzioni perché c'è un teorema che dice che le funzioni definibili in questo modo sono esattamente quelle definibili con un for.

Prendiamo come primitive alcune funzioni nel nostro linguaggio primitivo ricorsivo $\mathcal{L}$:

1. le funzioni costanti:

$$c_m^k(x_1, x_2, \ldots, x_k) = m \quad m \text{ in } \mathbb{N}$$

2. le proiezioni:

$$\pi_i^k(x_1, x_2, \ldots, x_k) = x_i \quad \text{per qualche } 1 \leq i \leq k$$

3. la funzione successore:

$$s(x) = x + 1$$

Ammettiamo inoltre due schemi composizionali per definire nuove funzioni a partire da quelle già esistenti:

1. Composizione: se $h : \mathbb{N}^n \to \mathbb{N}$ e $g_1, g_2, \ldots, g_n : \mathbb{N}^k \to \mathbb{N}$ appartengono a $\mathcal{L}$, anche la funzione $f : \mathbb{N}^k \to \mathbb{N}$ così definita:

$$f(\vec{x}) = h(g_1(\vec{x}), g_2(\vec{x}), \ldots, g_n(\vec{x}))$$

   appartiene a $\mathcal{L}$, con $\vec{x} = (x_1, x_2, \ldots, x_k)$.

2. Ricorsione primitiva: se $g : \mathbb{N}^k \to \mathbb{N}$,$h : \mathbb{N}^{k+2} \to \mathbb{N}$ appartengono a $\mathcal{L}$, allora che $f : \mathbb{N}^{k+1} \to \mathbb{N}$, così definita:

$$f(n, \vec{x}) = \begin{cases} f(0, \vec{x}) = g(\vec{x}) \\ f(y+1, \vec{x}) = h(y, f(y, \vec{x}), \vec{x}) \end{cases}$$

   appartiene a $\mathcal{L}$, con $\vec{x} = (x_1, x_2, \ldots, x_k)$.

Quando ho più strade di espansione nei sistemi di riscrittura il problema di capire se la strada che scelgo è influente è il problema della confluenza, legato alla terminazione dell'espansione.

Nell'esecuzione ci sono dei problemi che non abbiamo indicato esplicitamente, come la valutazione per valore e per nome, l'ordine di valutazione delle espressioni, i side effect, ecc.

L'idea delle funzioni ricorsive primitive è che abbiamo un argomento su cui facciamo la ricorsione ed una serie di parametri aggiuntivi.

Lavorare su una "sottostruttura" dell'input in ricorsione garantisce la terminazione della chiamata. La ricorsione primitiva è un sottocaso della ricorsione strutturale. Con la seconda ci si richiama solo su sottostrutture strette.

2.1.1 Esempi di funzioni primitive ricorsive

Vediamo ora alcuni esempi di definizioni di funzioni comuni nel formalismo primitivo ricorsivo.

$$add(x,y) = \begin{cases} add(0,y) = y \\ add(x+1,y) = succ(add(x,y)) \end{cases}$$

$$mult(x,y) = \begin{cases} mult(0,y) = 0 \\ mult(x+1,y) = add(mult(x,y),y) \end{cases}$$

Il meccanismo della definizione di funzioni per ricorsione primitiva è naturale. Molte funzioni sono strutturate in maniera ricorsiva.

$$fact(x) = \begin{cases} fact(0) = 1 \\ fact(x+1) = (x+1) * fact(x) \end{cases}$$

La seguente funzione mi restituisce 1 se l'input è 0 e 0 altrimenti.

$$test(x) = \begin{cases} test(0) = 1 \\ test(x+1) = 0 \end{cases}$$

Proviamo a definire la funzione differenza. Ricordiamo che abbiamo definito la calcolabilità sui numeri naturali, interi positivi. Di conseguenza non possiamo calcolare la differenza, ad esempio, tra 3 e 7. Noi vogliamo una funzione che calcoli $a - b$ se $a > b$. In tutti gli altri casi calcoliamo 0.

Una prima definizione naïve di $a - b$ è la seguente:

$$sub(a,b) = \begin{cases} sub(0,b) = 0 \\ sub(a+1,b) = succ(sub(a,b)) \end{cases}$$

Questa è sbagliata rispetto alla nostra specifica. Per $sub(1,1)$, ad esempio, restituisce 1. Una scelta migliore sarebbe andare in ricorsione su b.

Una delle scelte da fare è su cosa andare in ricorsione. Può essere uno dei parametri oppure un nuovo valore costruito a partire dai parametri.

Proviamo con la seguente definizione:

$$sub(a,b) = \begin{cases} sub(a,0) = a \\ sub(a,b+1) = pred(sub(a,b)) \end{cases}$$

dove *pred* restituisce 0 per 0 e $x-1$ in ogni altro caso.

Questa definizione è un pò borderline, perché andrebbe dimostrato che cambiando il parametro su cui vado in ricorsione ottengo un formalismo equivalente a quello introdotto. Tuttavia ciò è possibile perciò la definizione rientra nel formalismo primitivo ricorsivo.

Vogliamo ora una funzione che restituisca 1 se i parametri sono uguali, 0 altrimenti.

$$comp(n,m) = test(add(sub(n,m), sub(m,n)))$$

Benchè sembri limitante è veramente potente questo tipo di ricorsione.

Quando parliamo di predicati intendiamo funzioni che restituiscano un booleano.

Supponiamo di saper calcolare un certo predicato P. Possiamo calcolare anche la sua negazione.

Data la funzione caratteristica di P, c_P, possiamo calcolare la funzione caratteristica di $\overline{P}$: $c_{\overline{P}}(x) = 1 - c_P(x)$.

Date c_P e c_Q abbiamo che $c_{P \wedge Q}(x) = c_P(x) * c_Q(x)$.

È possibile definire $c_{P \vee Q}$ con le leggi di de Morgan: $A \vee B = \overline{\overline{A \vee B}} = \overline{\overline{A} \wedge \overline{B}}$. Un altro modo è usano la funzione normalizzazione, che normalizza i numeri a 0 e 1:

$$c_{P \vee Q}(x) = norm(c_P(x) + c_Q(x))$$

Dati dei predicati possiamo quindi calcolare i connettivi logici tra di loro nel nostro formalismo. Passiamo ora al discorso dei quantificatori.

Per calcolare il quantificatore esistenziale dovrei avere una procedura del genere:

```
x = 0
while ¬P(x):
    x = x + 1
```

Questo potrebbe ciclare all'inifinito se l'esiste non vale. Abbiamo un problema duale con il quantificatore universale.

È evidente che c'è un problema ma non è detto che questo non sia insormontabile.

Quello che sappiamo senza dubbio calcolare è la quantificazione limitata, o bound. L'idea è che ho un upper bound finito alla mia quantificazione. Calcoleremmo quindi $g(n) = \exists x \leq n, P(x)$.

Ovviamente possiamo calcolare la quantificazione limitata con la ricorsione primitiva. Nel caso del quantificatore universale:

$$f(0) = P(0)$$
$$f(n+1) = P(n+1) * f(n)$$

Tanti problemi aperti dell'aritmetica sono legati alla quantificazione: sapere se esiste un numero che ripetti tale proprietà o se una tale proprietà vale per tutti i numeri.

È anche una maniera per approcciarsi ai problemi. Domandarsi se c'è un bound permette di rendere l'algoritmo più efficiente e certamente terminante.

2.2 Test di primalità

Vediamo ora un altro predicato interessante, il test di primalità.

Perché escludiamo 1 dai numeri primi? Perché uno dei risultati più importanti dell'aritmetica è la fattorizzazione unica, che vale per tutti i numeri escludendo 1. Per mantenere quel teorema 1 viene escluso.

Definiamo la proprietà $P(x)$:

$$x > 1 \wedge \forall z(z|x \implies (z = 1 \vee z = x))$$

$z|x$ è notazione per z divide x (più precisamente, z è un fattore di x).

$$z|x := \exists a, z * a = x := \exists a \leq x, z * a = x$$

Possiamo porre un bound anche al test di primalità: possiamo fermarci ad x. Siamo quindi in grado di definire (e calcolare) il test di primalità.

2.3 Minimizzazione

Vediamo ora un'altra operazione che opera su domini limitati che useremo molto spesso: la minimiazzazione.

$$\mu x, P(x)$$

Possiamo vederla come uno snippet del genere:

```
x = 0
while ¬P(x):
    x = x + 1
return x
```

Fissiamo un ordinamento e cerchiamo il primo x che soddisfa $P(x)$. Quello che ci interessa alla fine del ciclo è il valore di x. Questo è, sostanzialmente, cosa l'operazione di minimizzazione fa.

While è un costrutto imperativo. Noi preferiamo, per la nostra teoria della calcolabilità, un costrutto funzionale, da cui l'introduzione del μ. Il risultato di questo operatore è x.

Al solito col while abbiamo il problema che il while potrebbe non fermarsi mai. Quello che possiamo certamente sperare di scrivere, e quindi calcolare, nel nostro formalismo è una forma limitata di μ. Cerchiamo il più piccolo x minore di un certo y su cui vale un predicato $P(x, \vec{z})$.

$$f(y, \vec{z}) = \mu x \leq y, P(x, \vec{z})$$

I parametri di $\vec{z}$ rappresentano parametri ulteriori che possono essere utili.

Cosa vogliamo restituire se non troviamo x nell'intervallo fissato? Dobbiamo restituire un valore e ne possiamo restituire uno qualunque. La cosa più naturale è restituire $y + 1$, se y è il mio upper bound.

Vogliamo trovare un modo per scrivere questa operazione, non necessariamente per calcolarla efficientemente.

Definiamo prima il predicato R:

$$R(y, \vec{z}) = \forall x \leq y, \neg P(x, \vec{z})$$

Questo mi dice "fino a y non ho trovato il minimo", con y compreso.

R, come funzione, è una costante di valore 1 fino all'y_0 minimo per cui vale $P(y_0, \vec{z})$. Da lì in poi il suo valore diventa costantemente 0.

Possiamo ora scrivere μx:

$$\mu x \leq y, P(x, \vec{z}) = \sum_{w \leq y} \neg R(w, \vec{z})$$

Possiamo definirlo in un altro modo. Prendiamo in considerazione il predicato M:

$$M(x, \vec{z}) = P(x, \vec{z}) \wedge \forall y < x, \neg P(y, \vec{z})$$

Questo predicato mi dice "x è il più piccolo valore per cui vale P". Come faccio però a trovare x? Si può moltiplicare x per $M(x, \vec{z})$. Dato che dobbiamo testare tutti gli $x \leq y$, abbiamo che μ può essere espresso come:

$$\mu x \leq y, P(x, \vec{z}) = \sum_{x \leq y} x \cdot M(x, \vec{z})$$

2.4 Generazione numeri primi

Come possiamo trovare il più piccolo numero primo successivo ad un numero i? Con la seguente funzione, ad esempio:

$$\Pi(i) = \begin{cases} \Pi(0) = 2 \\ \Pi(x+1) = \mu p.prime(p) \wedge p > \Pi(x) \end{cases}$$

Cosa manca? Bisogna dare un bound alla minimizzazione. C'è un teorema che dice che è sempre possibile trovare un numero primo tra n e $2n$. Il bound che possiamo dare è $2\Pi(x)$.

L'importante è dare un bound. C'è un altro bound, molto più grande, che andrebbe bene comunque: $\Pi(x)!$.

Come si dimostra l'infinità dei numeri primi? Supponiamo che siano finiti e siano $p_1, p_2, \ldots, p_n$. Prendiamo il numero $p_1 \cdot p_2 \cdots p_n + 1$. Questo numero non è divisibile per nessun p_i della mia enumerazione. Quindi o è un numero primo oppure i suoi fattori non fanno parte di quella lista. In ogni caso ho un assurdo.

È una dimostrazione bella. La tecnica è analoga alla diagonalizzazione: costruisco un nuovo elemento da quelli di una lista che prova il mio assurdo.

2.5 Ricorsione multipla

Consideriamo una possibile codifica del piano e consideriamo la coppia $< n, m >$. Non siamo troppo interessati alla codifica della coppia in sè, ma alle funzioni che mi restituiscono le componenti della coppia.

Abbiamo che, in generale, le componenti sono $\leq$ della (codifica della) coppia: $n \leq < n, m >$ e $m \leq < n, m >$.

Supponiamo di voler calcolare $\pi_1(x)$, ovvero la prima proiezione della coppia x. Partiamo dalla seguente definizione:

$$\pi_1(x) = \mu n, \exists m, < n, m >= x$$

Manca un bound. Quale prendiamo? x:

$$\pi_1(x) = \mu n \leq x, \exists m \leq x, < n, m >= x$$

Talvolta la ricorsione può essere necessaria su più di un valore. Nel formalismo che abbiamo visto finora non abbiamo questa possibilità.

Consideriamo la sequenza di Fibonacci:

$$\begin{aligned} \mathit{fib}(0) &= 1 \\ \mathit{fib}(1) &= 1 \\ \mathit{fib}(x+2) &= \mathit{fib}(x+1) + \mathit{fib}(x) \end{aligned}$$

La funzione di Fibonacci è intrinsicamente esponenziale, ma questo è il peggior metodo per calcolarlo. Siamo esponenziali nel numero di chiamate oltre ad esserlo nell'input. Inoltre così com'e scritta ora non rispetta i vincoli del formalismo primitivo ricorsivo.

Qual è l'idea? Bisogna portarsi dietro un accumulatore.

Vogliamo definire la seguente funzione:

$$\mathit{fibo'}(x) =< \mathit{fib}(x), \mathit{fib}(x+1) >$$

Possiamo definirla nel formalismo primitivo ricosivo nel seguente modo:

$$\begin{aligned} \mathit{fibo'}(0) &=< 1, 1 > \\ \mathit{fibo'}(x+1) &=< \mathit{fib}(x+1), \mathit{fib}(x+2) > \\ &=< \pi_2(\mathit{fibo'}(x)), \pi_1(\mathit{fibo'}(x)) + \pi_2(\mathit{fibo'}(x)) > \end{aligned}$$

L'unica chiamata ricorsiva che faccio è *fibo'*(x).

Questa funzione corrisponde all'incirca al seguente snippet:

```
acc_0 = 1
acc_1 = 1
for i in range(x+1):
    tmp = acc_0
    acc_0 = acc_1
    acc_1 = tmp + acc_1
return acc_1
```

Questa tecnica è generale è prende il nome di ricorsione di coda.

Tutte le funzioni ricorsive primitive possono essere espresse mediante un for. È possibile dimostrare anche il viceversa. Il formalismo primitivo ricorsivo è equivalente, a potere espressivo, ai programmi scrivibili con il for, ovvero senza while e senza ricorsione generale.

2.6 Funzione di Ackermann

Vediamo ora una funzione che non è possibile scrivere nel formalismo primitivo ricorsivo. Va immaginata come una famiglia di funzioni dove il primo parametro mi istanza la particolare funzione. Abbiamo ack_0, ack_1, etc.

$$\begin{aligned} ack(0,0,y) &= y \\ ack(0,x+1,y) &= ack(0,x,y)+1 \\ ack(1,0,y) &= 0 \\ ack(z+2,0,y) &= 1 \\ ack(z+1,x+1,y) &= ack(z, ack(z+1,x,y), y) \end{aligned}$$

I casi sono mutalmente disgiunti. Data una tripla qualsiasi di valori solo una riga si applica.

La funzione termina? Sì, ma non è banale. L'argomento più importante della funzione è il primo. Se il primo decresce bene. Altrimenti vado a vedere gli altri. Questo mi dà un ordinamento delle mie chiamate ricorsive che mi da un'idea del fatto che la funzione è terminante.

Cosa calcola? ack_0 è abbastanza banale: è la somma. ack_1 invece calcola il prodotto tra x e y. ack_2 calcola l'elevamento a potenza (y^x).

ack_i itera ack_{i-1}. Il prodotto itera la somma. L'elevamento a potenza itera la moltiplicazione la tetrazione itera l'elevamento a potenza.

La funzione, benchè terminante, ha una complessità spaventosa.

Perché non posso scriverla nel formalismo primitivo ricorsivo? Perché questa funzione cresce troppo velocemente. Cresce più velocemente di qualsiasi funzione esprimibile col formalismo primitivo ricorsivo. La funzione di Ackermann mi dà un bound computazionale alle funzioni esprimibili con il formalismo primitivo ricorsivo. Di conseguenza non può essere esprimibile nello stesso formalismo.

Se riesco ad esprimere un bound computazionale alla complessità di un programma so che non è possibile calcolare quel bound nello stesso formalismo in cui ho espresso il mio programma.

Tutte le singole istanze di Ackermann sono scrivibili nel formalismo primitivo. Ma non posso scrivere un programma che le esprima tutte. Il formalismo non è abbastanza parametrico.

È sufficiente aggiungere l'ordine superiore al formalismo primitivo per poter esprimere la funzione di Ackermann.

La funzione di Ackermann è una funzione chiaramente calcolabile, essendo esprimibile in un qualche linguaggio di programmazione, ma non è esprimibile nel formalismo primitivo ricorsivo.

Si può dimostrare che c'è un ordinamento ben fondato e quando facciamo le chiamate ricorsive nella funzione di Ackermann le variabili decrescono secondo questo ordine.

Un ordinamento si dice ben fondato se non esistono catene discendenti infinite.

Dire che ho un ordinamento ben fondato non è la stessa cosa di dire che di elementi minori di un dato elemento m ce ne sono una quantità finita.

Vediamo un esempio: l'ordinamento lessicografico. Considerando $\mathbb{N} \times \mathbb{N}$. Definiamo l'ordinamento $<_p$ nel seguente modo: $< n_1, m_1 ><_p< n_2, m_2 > \iff n_1 < n_2 \vee (n_1 = n_2 \wedge m_1 < m_2)$. Questo ordinamento è ben fondato. Vale che $\forall m, < 2, m ><_p< 3, 7 >$.

Non è possibile costruire catene discendenti infinite. Proviamo a costruirne una: $< 3, 7 >>_p< 3, 6 > ... < 3, 0 >>_p< 2, 10^4 >$

Arrivati qui posso ripetere il giochetto di prima finché non arrivo a 0, dopodiché dovrò decrementare la prima componente. Di queste sequenze ne posso fare di lunghezza arbitraria ma sempre finita. Questo ragionamento ci dà un'idea del perché la funzione di Ackermann termina (il principio alla base della dimostrazione è lo stesso).

Capitolo 3

Altri formalisimi e Macchine di Turing

3.1 Formalismi totali e problema dell'interprete

Ci sono varie estensioni possibili al formalismo primitivo ricorsivo. Un esempio è il sistema T di Gödel, che aggiunge l'ordine superiore. Questo ci permette di scrivere la funzione di Ackermann. Un'altra possibilità è quella di rilassare la ricorsione primitiva permettendo una ricorsione che permetta di andare in ricorsione su ordinamenti ben fondati. Questa condizione è verificabile in termini sintattici, entro certi limiti.

Esteso il mio linguaggio ottengo la possibilità di scrivere la funzione di Ackermann e le mie funzioni sono totali. Sono complete? O c'è qualche funzione totale che posso pensare ma non scrivere? C'è una dimostrazione che dice che sì, tutti questi formalismi saranno incompleti. Un formalismo che permette di scrivere solo programmi terminanti sarà sempre incompleto.

La dimostrazione è abbastanza semplice. L'idea è che un programma che non riesco a scrivere (tra i tanti) è l'interprete per il linguaggio stesso.

Cosa intendiamo per interprete? Qui parliamo sempre di funzioni da $\mathbb{N}$ a $\mathbb{N}$. Dovremmo quindi definire l'interprete in questi termini.

L'input dell'interprete non è un programma in senso astratto. Prende in input una stringa che esprime il programma. Questa stringa può essere letta come un numero. Dire che una funzione è calcolabile è equivalente a definire l'interprete per tale funzione, se vogliamo dare una definizione operativa.

Sia data una enumerazione effettiva (e.g. lessicografica) P_n dei programmi del linguaggio $\mathcal{L}$, e sia φ_n la funzione calcolata dal programma P_n.

P_n è un livello intensionale, una descrizione di una funzione. φ_n è la funzione calcolata dal programma n-esimo.

Un interprete per $\mathcal{L}$ è una funzione che preso in input l'indice n di un programma ed un input m simula il comportamento di P_n su tale input, cioè una

funzione I tale che

$$I(n, m) = \varphi_n(m)$$

Se la numerazione dei programmi è effettiva, I è intuitivamente calcolabile.

È fondamentale la numerazione: dato n devo poter sapere qual'è la funzione n-esima da interpretare.

Ci chiediamo se l'interprete per $\mathcal{L}$ può essere scritto in $\mathcal{L}$, cioè se esiste u tale che $I = \varphi_u$.

La risposta è no. Supponiamo infatti che esista u tale che $I(n, m) = \varphi_u(n, m) = \varphi_n(m)$. Consideriamo la funzione

$$f(x) = \varphi_u(x, x) + 1 = \varphi_x(x) + 1$$

Se il linguaggio $\mathcal{L}$ è chiuso per composizione $f \in \mathcal{L}$, e dunque deve esistere un programma i tale che $\varphi_i = f$.

Ma allora

$$\varphi_i(i) = f(i) = \varphi_i(i) + 1$$

che è assurdo

L'argomento è sempre diagonale. Mi muovo sulla digaonale mentre sui lati ho due infinità (numero dei programmi e lunghezza dei programmi ad esempio).

L'interprete è un esempio di funzione intuitivamente calcolabile ma non esprimibile in un formalismo totale. Tipicamente è sempre possibile, dato un linguaggio che esprime funzioni totali, trovare un linguaggio più espressivo.

Teorema 3.1. *Nessun formalismo totale è in grado di esprimere il proprio interprete.*

La completezza algoritmica è un concetto obsoleto. Gli algoritmi a cui siamo spesso interessati sono quelli polinomiali in complessità. A questo punto, perché non restringersi a linguaggi di programmazione che permettono di scrivere solo programmi con questa complessità? Si può fare, ce ne sono tanti di linguaggi del genere, basta imporre i giusti vincoli sul linguaggio.

Da un punto di vista operativo si perde in praticità. Ma questo è anche legato al capire la complessità computazionale dell'algoritmo. Se però si capisce bene perché un dato algoritmo ha una certa complessità allora possiamo strutturare il programma che lo realizza in modo che rispetti i vincoli del linguaggio. Il nostro approccio è al contrario: scriviamo i programmi e poi li analizziamo. Questo approccio non ci dà un granchè di informazioni. Non abbiamo alcun metodo generalizzato che ci dia informazioni o garanzie su programmi. Sarebbe più bello avere queste informazioni a priori.

Nei linguaggi di programmazione comuni (C, Python, ecc.) posso scrivere l'interprete per il linguaggio stesso. Perché? Perché non ho garanzie di terminazione ($\varphi_i(i)$ divergerebbe).

Abbiamo una gerarchia nota dei linguaggi totali. Un'interessante caratterizzazione del sistema T è che le funzioni calcolabili in questo sistema sono esattamente quelle calcolabili nell'aritmetica di Peano al primo ordine.

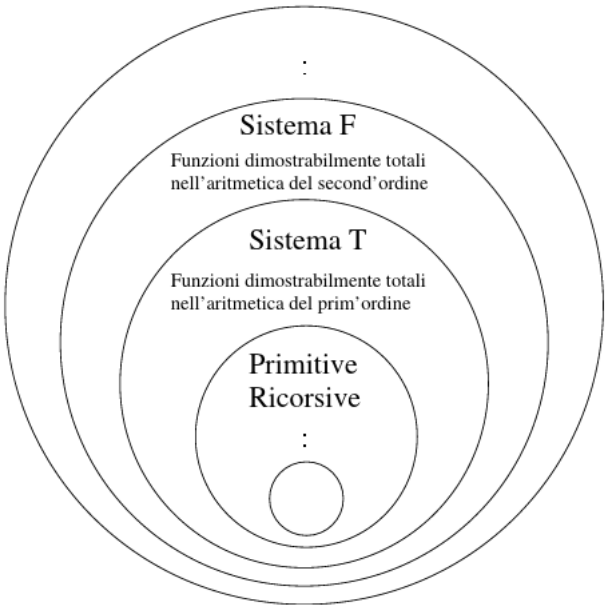


Figura 3.1: Gerarchia dei formalismi totali

3.2 Tesi di Church

Che succede se rinunciamo a questa idea della totalità? Sappiamo che non daremo luogo a paradossi. Tuttavia rimane il problema: siamo nella situazione dei formalismi totali, e cioè esiste una gerarchia di formalismi più potenti, o colpiamo un upper bound tale che in un formalismo del genere posso calcolare tutte le funzioni che posso calcolare in qualsiasi altro formalismo? Quest'ultima situazione sembrerebbe miracolosa dati i risultati visti finora.

La cosa che era ragionevole aspettarsi era la prima. Quello che sembra essere, ma che non è dimostrabile, è che la nozione di calcolabilità è indipendente dal formalismo che uso. Se ho un formalismo abbastanza espressivo posso esprimere qualsiasi funzione intuitivamente calcolabile. Questo è il succo della tesi di Church.

Quello che possiamo fare è confrontare formalismi a livello di potere espressivo. Più formalismi si sono considerati più si è avvalorata l'idea che la classe di funzioni che possiamo calcolare sia la stessa, ed in particolare quella delle funzioni calcolabili da una macchina di Turing.

La tesi di Church può essere espressa nella maniera seguente:

Teorema 3.2. *Le funzioni calcolabili sono esattamente quelle intuitivamente calcolabili medi- ante una procedura effettiva di calcolo.*

Ci sono delle funzioni intuitivamente definibili ma non calcolabili? Non lo sappiamo. Rimane un problema aperto.

3.3 Alcuni formalismi Turing completi

Dimostrare la Turing-completezza dei linguaggi moderni è complesso. Molti formalismi sono stati studiati e sono stati trovati tutti equivalenti a potere espressivo.

Un sistema interessante è la logica combinatoria. Ho due costanti, che per ragione storiche si chiamano S e K. I programmi sono scritti come grosse combinazioni di S e K. Ad esempio:

$$(K((SK)K))$$

Abbiamo due regole di risctittura:

- $((KM)N) \to M$
- $(((SP)Q)R) \to (PR)(QR)$

È dimostrabile che questo sistema è Turing completo. Ovviamente c'è il problema di codificare i dati di input e output. Questi sono quelli che sono chiamati combinatori.

Questi formalismi sono semplici. Si può dimostrare quindi l'equivalenza tra due formalismi come meta-teorema in maniera agevole.

Il λ-calcolo è una di quelle cose dell'informatica che è così e non potrebbe essere altrimenti. Si giustifica intrinsicamente. È la cosa più naturale, in un certo senso. La logica combinatoria, ad esempio, non è così invece.

L'idea del λ-calcolo è che voglio un linguaggio per descrivere funzioni. Ho bisogno di:

- variabili
- un meccanismo per definire funzioni. Introduciamo quindi, dato un termine M,$\lambda x.M$. Questa è l'operazione di introduzione (o costruzione) della funzione. Manca l'operazione di eliminazione (o distruzione)
- introduciamo quindi l'applicazione: (MN)

Posso partire da x e, perché no, applicare x a se stesso. Dopodiché introduco l'astrazione. Ottengo $\lambda x.(xx)$.

Qual è la regola di calcolo? Questa nasce dall'interazione tra costruttore e distruttore.

Cosa mi aspetto da $(\lambda x.M)N$? Mi aspetto M con tutte le occorrenze (libere) di x sostituite da N:

$$(\lambda x.M)N \to M[N/x]$$

L'operatore visto prima è noto comunemente come $\delta = \lambda x.(xx)$. Definiamo I come $I = \lambda x.x$. È un formalismo di alto livello. Non ci sono tipi, posso applicare espressioni ad altre espressioni senza limiti.

Consideriamo la seguente sequenza di calcolo dell'espressione (δI):

$$(\delta I) \to (II) \to I$$

I è un termine in forma normale: non c'è più alcuna riduzione possibile per questo termine.

Cosa succede con $(\delta\delta)$? Si riscrive se stesso all'infinito.

Un altro formalismo è quello delle funzioni μ-ricorsive.Cosa sono le funzioni μ-ricorsive? Si prende il formalismo primitivo ricorsivo e si aggiunge la minizzazione unbound. È Turing completo.

Quando definiamo un formalismo abbiamo due modi. Un modo è quello descrittivo, ovvero definisco un linguaggio ed eventualmente delle regole. È una descrizione ad alto livello. L'altro modo, un pò più simile alle macchine di Turing o alle macchine a registri, è di dare un'architettura di basso livello e scrivere i propri programmi utilizzando questa architettura.

3.4 Macchine di Turing

Finché lavoro con linguaggi ad alto livello è difficile convicersi che abbiano lo stesso potere espressivo delle macchine di Turing. Inoltre rimane il dubbio: siamo sicuri di non aver tralasciato un costrutto che permette di fare un balzo nel potere espressivo?

La macchina di Turing che consideriamo ha un nastro solo. È un nastro di memoria infinito. Esiste? No, è un'astrazione matematica. E' diviso in celle di dimensione fissata. Ogni cella può contenere un carattere di un alfabeto dato, compreso un carattere b (bianco) di inizializzaizone.

Abbiamo una testina di lettura mobile e un automa a stati finiti per la definizione del programma.

Un programma è composto da una lista infinita di operazioni che associano ad una coppia $<$ carattere, stato interno $>$ una tripla <nuovo carattere, nuovo stato, mossa>, dove mossa è *dx* o *sx*.

Quello che possiamo fare è:

- leggere e scrivere il carattere individuato dalla testina
- spostare la testina di una posizione verso destra o verso sinistra
- leggere e scrivere il carattere individuato dalla testina

Una Macchina di Turing (one-tape, deterministica) è una tupla $< Q, \Gamma, b, \Sigma, \delta, q_0, F >$ dove:

- Q è un insieme finito di stati
- Γ è l'alfabeto finito del nastro
- b è il carattere bianco
- $\Sigma \subseteq \Gamma \setminus \{b\}$ è l'insieme dei caratteri di input/output

- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali (o di accettazione)
- $\delta : Q \setminus F \times \Gamma \to Q \times \Gamma \times \{L, R\}$ è la funzione di transizione.

La funzione di transizione ha un grafo finito. Ogni elemento del grafo è una quintupla (q, a, q', a', M) tale che $(q, a) = (q', a', M)$. L'insieme finito delle quintuple può essere visto come l'insieme delle istruzioni della macchina (programma), e la determina univocamente.

La computazione deve essere deterministica. Dato un input alla funzione di transizione ci può essere un solo output.

Dobbiamo rispondere ad alcune considerazioni. Ad esempio, dove si trova la testina rispetto all'input? Innanzitutto supponiamo che il nastro sia inizializzato con la stringa di input (un carattere per ogni cella). Supponiamo poi che la testina parta dall'inizio dell'input. Tutte le altre celle del nastro sono inizializzate col carattere speciale b.

Se abbiamo un nastro solo su quello scriviamo l'input e alla fine su quello troviamo l'output. Dobbiamo decidere come interpretarlo; ci sono vari modi standard. Per noi nel momento in cui la macchina si arresta l'output è la più lunga stringa di caratteri in Γ (in particolare, senza b) alla destra della testina

Vediamo un esempio di macchina di Turing

Supponiamo di essere nello stato q_0 e di essere in posizione @. Consideriamo il seguente programma:

$$
\begin{aligned}
< q_0, 0 > &\to < q_1, 0, R > \\
< q_0, 1 > &\to < q_2, 1, R > \\
< q_1, 0 > &\to < q_1, 0, R > \\
< q_1, 1 > &\to < q_1, 1, R > \\
< q_1, \# > &\to < q_3, 0, R > \\
< q_3, 1 > &\to < q_2, 0, R > \\
< q_3, 0 > &\to < q_4, \#, L > \\
< q_3, 1 > &\to < q_4, \#, L > \\
< q_3, b > &\to < q_4, \#, L > \\
< q_2, 0 > &\to < q_2, 0, R > \\
< q_2, 1 > &\to < q_2, 1, R > \\
< q_2, \# > &\to < q_3, 1, R >
\end{aligned}
$$

Possiamo associare q_1 allo stato "ho letto uno zero".

Questo programma copia il primo carattere in input nella posizione di # e poi sposta # a destra.

Abbiamo $Q = \{q_0, \ldots, q_4\}$ e $F = \{q_4\}$

Ad ogni coppia stato simbolo viene associata una tripla nuovo stato, nuovo simbolo e mossa. Ci sono una infinità di varianti (tutte equivalenti dal punto di vista del potere formale).

La mossa da fare sarà unica perché la macchina è deterministica.

Una configurazione istantanea è una descrizione istantanea della configurazione della macchina. Non corrisponde allo stato interno della macchina, ma quest'ultimo ne fa parte. La si può pensare come ciò che devo ricordare per riprendere una computazione interrotta più tardi. Si parla di configurazione solo in relazione ad un dato nastro di input.

Ho bisogno di salvare tre informazioni, avendo un nastro solo:

- lo stato interno
- il nastro
- la posizione della testina sul nastro

L'ultimo passaggio è delicato: avrei bisogno di un'origine per definire la posizione. Non è però chiaro definire dove sta questa origine. Avendo nastri infiniti non ho un'idea ben definita di origine. Potrei fissarne una ma poi dovrei separare il nastro in due parti. Con un seminastro sarebbe facile. La cosa più semplice è definire come origine il dove si trova la testina. A questo punto mi interessa memorizzare solo il seminastro a destra della testina e quello a sinistra.

La mia configurazione sarà quindi:

$$\alpha, q, \beta$$

α è il seminastro sinistro, β quello destro. Possiamo ora descrivere la comuputazione come una sequenza di configurazioni.

Descriviamo la computazione di prima:

$$\begin{aligned}
&\emptyset, q_0, 0110\# \\
&0, q_1, 110\# \\
&01, q_1, 10\# \\
&011, q_1, 0\# \\
&0110, q_1, \# \\
&01100, q_3, \emptyset \\
&0110, q_4, 0\#
\end{aligned}$$

Per comodità è sempre utile far vedere il primo carattere a destra e a sinistra della testina. Se un seminastro è blank posso immaginare di avere b come primo carattere:

$$\alpha, q, \beta \equiv \alpha_1 a, q, b\beta_1$$

Supponiamo che questa sia la configurazione

$$\alpha a, q, b\beta$$

Supponiamo che questa sia l'istruzione

$$< q, b > \rightarrow < q', b', R >$$

Allora

$$< \alpha a, q, b\beta > \vdash < \alpha a b', q', \beta >$$

Analogamente, se $< q, b > \rightarrow < q', b', L >$ allora

$$< \alpha a, q, b\beta > \vdash < \alpha, q', a b' \beta >$$

Questa è la semantica della macchina di Turing.

Il processore è a tutti gli effetti una macchina a stati finiti

Perché è importante l'idea della macchina di Turing? Perché la macchina di Turing racchiude il concetto di operatore di calcolo più naturale che possiamo immaginare.

Quello che l'agente esecutore ha a sua disposizione è una memoria illimitata. L'agente è un qualcosa di finitistico, della memoria ha una visione finita. Quello che vede è la cella, di dimensione finita ma senza alcuna assunzione sulla dimensione della cella. Non ha una visione sinottica dell'intero nastro, ha una visione limitata dalla sua natura. L'agente può modificare una porzione finita del nastro. Per semplicità diciamo che può modificare solo la cella. Può spostarsi e modificare il suo stato interno. Di quanto può spostarsi? Di una porzione finita. Può ripetere queste azioni. Più di questo non può fare.

Perché è così potente questa nozione? Perché per andare oltre a questa nozione di calcolabilità dovrei visionare e realizzare un agente di calcolo con capacità maggiori di quello descritto.

3.5 Macchine universali

Un'altra macchina interessante è la macchina universale. Questa è una macchina capace di eseguire una qualsiasi macchina di Turing.

Perché ho bisogno di un nasto per memorizzare gli stati? Perché questi devono essere codificati, e non so a priori quanto grande sarà la mia codifica.

```
-------------------------------------------------------
    ||q_{0}|0|q_{1}|0|R||q_{1}|0|q_{1}|0|R||
-------------------------------------------------------
-------------------------------------------------------
    ||q_{0}|0||
-------------------------------------------------------
-------------------------------------------------------
    |0|1|1|0|#|
-------------------------------------------------------
```

Ho tre nastri: il nastro degli stati, il nastro che simula la macchina di Turing ed il nastro dell'input. Ognuno ha la sua testina.

Perché è interessante questa macchina? Perché questa è, in sostanza, la macchina di Von Neumann, eccetto per alcune differenze non significative. Ad esempio VN ha accesso random invece che un nastro sequenziale.

Con le macchine di Turing ogni automa definiva un'architettura: servirebbero tante macchine quante funzioni esprimibili. La macchina universale invece può emulare le macchine di Turing con un'unica architettura. Ci sono differenze con la macchina di Turing ma sono dettagli, la struttura è simile e le capacità della macchina universale non sono maggiori: il potere espressivo è lo stesso. Prendiamo come input una macchina e l'input della funzione e la macchina universale fa da interprete.

Noi passeremo ad una nozione ancora più astratta. Questo perché vogliamo una teoria della calcolabilità che sia machine-independent. Non vogliamo essere costretti a ridurci sempre ad un modello computazionale particolare.

L'idea è che dobbiamo pensare di avere una enumerazione dei programmi. φ_i è la funzione calcolata dall'i-esimo programma. Noi diremo la funzione i-esima.

Vogliamo assicurarci che la numerazione dei programmi sia effettiva. Ad esempio, dato 100 voglio sapere qual è il programma 100. Vogliamo quindi una funzione universale μ tale che:

$$\mu(i, x) = \varphi_i(x)$$

Questa è la macchina universale o interprete. Possiamo vedere i come la descrizione del programma.

Possiamo riformulare la tesi di Church in questo contesto come:

"f è intuitivamente calcolabile $\implies \exists i.\varphi_i = f$"

Capitolo 4

Problemi indecidibili

Ci chiediamo ora se ci sono dei problemi che non sono decidibili, ovvero non calcolabili in un formalismo Turing completo. Introduciamo l'argomento con uno dei problemi più noti in letteratura, l'*Halting problem*.

Prima però chiariamo meglio i concetti di totalità, calcolabilità e la relazione tra i due, oltre a introdurre i concetti di divergenza e convergenza.

4.1 Note introduttive su programmi e funzioni

Una qualsiasi funzione φ_i della nostra enumerazione delle funzioni calcolabili è una funzione parziale calcolabile: su alcuni input può non essere definita.

La calcolabilità non coincide con la totalità: esistono funzioni parziali calcolabili e funzioni totali non calcolabili.

Le funzioni possono essere non definite in un punto. Se questo è il caso per una funzione calcolabile i ed un punto x, avremo allora la divergenza del *programma* i: $\varphi_i(x) \uparrow \iff \varphi_i$ non è definita su input x. Divergenza e convergenza sono più proprietà del programma che della funzione che calcola. Nonostante qui usiamo il simbolo φ sia per i programmi che per le funzioni è bene ricordare che può avere un doppio ruolo e una semantica diversa associata ad esso. La relazione tra i due è: la funzione i nella mia enumerazione di funzioni calcolabili è quella calcolata dal programma i della mia enumerazione dei programmi.

La divergenza non è una proprietà generale delle funzioni. Non ha senso dire $\varphi_i \uparrow$ senza specificare dove diverge; questo perché la convergenza e la divergenza sono proprietà puntuali: valgono per un dato input. Al massimo $\varphi_i(x)$ per un qualche x può divergere.

4.2 Il problema della fermata

Il problema che ci interessa è il cosiddetto "problema della terminazione". Le funzioni che stiamo calcolando sono funzioni parziali: i programmi corrispondenti possono potenzialmente divergere. Noi vorremmo calcolare la seguente

funzione:

$$Term(i,x) = \begin{cases} 1 & \text{se } \varphi_i(x) \downarrow \\ 0 & \text{se } \varphi_i(x) \uparrow \end{cases}$$

Questà è la specifica della mia funzione. È una funzione totale. Ci chiediamo a questo punto, è anche calcolabile? La risposta, come vedremo, sarà negativa.

Per dimostrarlo supponiamo che *Term* sia calcolabile e prendiamo in considerazione ora la funzione intermedia g:

$$g(x) = \begin{cases} 1 & \text{se } Term(x,x) = 0 \\ \uparrow & \text{se } Term(x,x) = 1 \end{cases}$$

Se *Term* fosse calcolabile allora anche g sarebbe calcolabile. Se g è calcolabile deve esistere un k tale che $\varphi_k = g$. Ci può essere più di un programma che calcola g, ma a me ne basta uno.

Ci chiediamo ora, legittimamente, qual è il comportamento di $\varphi_k(k)$? Converge?

Abbiamo che $\varphi_k(k) = g(k)$. Quindi $\varphi_k(k) \uparrow \iff Term(k,k) = 0 \iff g(k) = 1 \iff g(k) \downarrow \iff \varphi_k(k) \downarrow$. Questo è contraddittorio. Verrebbe da concludere che $\varphi_k(k)$ converge. È tuttavia vero che $\varphi_k(k) \downarrow \iff Term(k,k) = 1 \iff g(k) \uparrow \iff \varphi_k(k) \uparrow$. Ma anche questo è contradditorio. L'ipotesi da cui siamo partiti è che *Term* fosse calcolabile. Concludiamo quindi che *Term* non è calcolabile.

È il primo caso di una funzione che possiamo pensare ma che non riusciamo a calcolare, almeno con questa formulazione qui.

La dimostrazione è un semplice ragionamento diagonale. Esistono quindi funzioni ben definite ma non calcolabili: non esiste un algoritmo che mi calcoli questo problema. Nella sua forma generale il problema delle terminazione non è algoritmico.

Cosa vuol dire nella sua forma generale? Significa che valga per tutti gli i e per tutti gli x. Per alcuni programmi e alcuni input è possibile dimostrare che il programma termina. Ci sono dei casi particolari che sono gestibili, ma non esiste un unico algoritmo che in modo uniforme su tutti gli i e tutti gli x sappia decidere se il programma i termini su input x.

Quali erano le nostre ipotesi? La calcolabilità dell'interprete e qualche piccola proprietà di chiusura sul mio formalismo. Non molto.

Questa funzione non è esprimibile in un formalismo Turing completo. Non è tuttavia assurda l'idea che esista un agente di calcolo con più capacità della macchina di Turing e che sia in grado di calcolare *Term*. È difficile da immaginare. Nella calcolabilità relativa si parte immaginando un oracolo che sia capace di calcolare *Term* e ci si chiede da lì cosa si possa calcolare (e cosa no).

4.3 Proprietà s-m-n

L'ipotesi della calcolabilità dell'interprete è un'ipotesi importante. Supponiamo, nella nostra teoria della calcolabilità, che esista un modo per calcolarla. Questa

proprietà è detta proprietà UTM, Universal Turing Machine: $\exists u, \varphi_u(i,x) = \varphi_i(x)$. È dimostrabile in tutti i formalismi Turing completi.

La proprietà che andiamo ora a considerare è la cosiddetta proprietà s-m-n. Supponiamo di avere una funzione calcolabile $g(x,y)$. Cosa posso dire delle sue istanze? Supponiamo di fissare x, ad esempio a 0. Ottengo $g(0,y)$, che è una funzione unaria che dipende solo da y. Possiamo indicare $g(0,y)$ come $f_0(y)$. In generale posso fare questo per $f_k(y) = g(k,y)$.

Tutte queste funzioni sono calcolabili e quindi ognuna di esse farà parte della mia enumerazione delle funzioni calcolabili. Esisterà quindi, per ognuna, un programma che la calcola. Esiste quindi $\varphi_{h(k)}(y) = f_k(y) = g(k,y)$ che mi calcola $g(k,y)$ per ogni y. h è una funzione che mi restituisce l'indice dell'istanza di g che mi interessa. L'esistenza di h è praticamente ovvia: essendo tutte le istanze di g relative a k calcolabili avranno un indice nella mia enumerazione delle funzioni calcolabili e questo indice dipende da k. La domanda ora è: h è calcolabile? La risposta è sì.

Noi supporremo la proprietà s-m-n, ma questa è facilmente dimostrabile in tanti formalismi.

Teorema 4.1. ***Proprietà s-m-n.*** *$\forall g$ calcolabile $\exists h$ totale e calcolabile tale che*

$$\forall x, y. \varphi_{h(x)}(y) = g(x,y)$$

Quello che stiamo facendo è una curryficazione. C'è un importante isomorfismo a livello di funzioni: lo spazio delle funzioni $(A \times B) \to C$ è isomorfo allo spazio delle funzioni $A \to (B \to C)$. L'operazione alla base della dimostrazione di questa affermazione e che mi permette di passare da una funzione del primo spazio alla sua corrispondente nel secondo si chiama curryficazione. L'idea è: data $g(x,y)$ fissiamo x. In questo modo ottengo $g(x,y)$, con x fissato, ovvero una funzione unaria in y da B a C.

Si può dimostrare anche con un argomento sulla cardinalità. Sappiamo che la cardinalità del primo spazio è $|C|^{|A\times B|} = |C|^{|A|\cdot|B|} = (|C|^{|B|})^{|A|}$, che è la cardinalita di $A \to B \to C$. Questo mi garantisce l'esistenza dell'isomorfismo. La dimostrazione precedente è un pò più strutturale (e costruttiva se così si vuol dire).

h fondamentalmente mi dà l'indice della funzione curryficata.

In un certo senso s-m-n mi dice che il mio formalismo è chiuso rispetto alle curryficazioni/λ-astrazioni. UTM mi dice che il mio formalismo è chiuso rispetto alle λ-applcazioni.

Possiamo generalizzare s-m-n a vettori x e y di parametri:

Teorema 4.2. *$\forall g \forall m \forall n \exists s$ totale calcolabile tale che*

$$\varphi_{s(\vec{x}_m)}(\vec{y}_n) = g(\vec{x}_m, \vec{y}_n)$$

s-m-n serve per calcolare un numero come indice di un programma. Se non bisogna calcolare un indice di un programma in funzione di qualcos'altro non mi serve s-m-n. Noi vedremo molti casi in cui avremo proprio bisogno di quello.

Vediamo un esempio di applicazione di s-m-n nella dimostrazione della non calcolabilità di alcune funzioni.

La funzione che ci interessa indagare adesso è *Tot* che determina se un certo programma è totale o meno. È anche questo un problema di decisione. La specifica della mia funzione è:

$$Tot(i) = \begin{cases} 1 & \text{se } \forall x \varphi_i(x) \downarrow \\ 0 & \text{altrimenti} \end{cases}$$

Abbiamo bisogno però prima di un risultato intermedio. Un caso particolare caso del problema della terminazione è il problema della terminazione diagonale:

$$Term(i) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \\ 0 & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Se la terminazione diagonale non è calcolabile la terminazione generabile non è calolabile. Non è vera l'implicazione inversa. Ha senso quindi chiedersi se la terminazione diagonale è calcolabile. La risposta è no, e la dimostrazione è analoga a quella della terminazione generale.

Dimostrare che una funzione non sia calcolabile non è banale. Devo dimostrare che non esiste un algoritmo tale che la calcoli. È molto più difficile rispetto a dimostrare la calcolabilità di una funzione.

Abbiamo due strade. Assumiamo la calcolabilità di *Tot* e troviamo un assurdo. Oppure usiamo un procedimento di riduzione: se è calcolabile *Tot* deve essere calcolabile $Term_i$. Da lì poi dimostriamo la non calcolabilità di $Term_i$.

Prendiamo $g(i, x) = \varphi_i(i)$. Abbiamo due casi: o converge o diverge. Nel primo caso se fisso i la funzione curryficata che ottengo è totale. Nell'altro caso no.

Per s-m-n abbiamo h totale e calcolabile tale che

$$\varphi_{h(i)}(x) = g(i, x) = \varphi_i(i)$$

Ora mi chiedo: quanto vale $Tot(h(i))$? Abbiamo che

$$Tot(h(i)) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \\ 0 & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Componendo *Tot* con h risolverei il problema della terminazione diagonale. Ma questo è assurdo. Da cui *Tot* non è calcolabile.

In questa dimostrazione parto da funzioni calcolabili che vado a comporre con la mia funzione di partenza (*Tot*) e ottengo una funzione che mi potrebbe calcolare qualcosa che so non essere calcolabile. Da ciò concludo che la mia funzione di partenza non è calcolabile. È una dimostrazione diversa da quella del problema della terminazione.

È interessante vedere questa dimostrazione anche "al contrario", in versione bottom up. Come faccio a dimostrare la non calcolabilità di *Tot*? Assumo che

sia calcolabile e la uso per risolvere un problema indecidibile. In questo caso vogliamo ridurci alla terminazione diagonale. Come facciamo? Possiamo farlo cercando una funzione h tale che, per ogni i, $\varphi_{h(i)}$ è totale sse $\varphi_i(i)\downarrow$. Esiste questa funzione h? Sì, e possiamo dimostrarlo in due passaggi.

Per prima cosa definiamo una funzione calcolabile binaria $g(i,x)$ che, in base alla terminazione di o meno di $\varphi_i(i)$, ha un comportamento diverso che rispetti le condizioni che abbiamo posto su $\varphi_{h(i)}$. Ad esempio

$$g(i,x) = \begin{cases} 1 & \text{se } \varphi_i(i)\downarrow \\ \uparrow & \text{se } \varphi_i(i)\uparrow \end{cases}$$

Questa funzione è calcolabile? Sì, è facilmente scrivibile in un linguaggio di programmazione. Ma a questo punto ho trovato la h che cercavo, grazie alla proprietà s-m-n: $\exists h$ tot. e calc.$\varphi_{h(i)}(x) = g(i,x)$. A questo punto, se mi chiedo se la funzione $\varphi_{h(i)}$ è totale, utilizzando la funzione *Tot*, mi sono ridotto al problema della terminazione diagonale.

Consideriamo un caso particolare e poi proviamo a generalizzare. Stiamo analizzando programmi. Mi chiedo se un mio programma calcola la funzione identità. Si potrebbe generalizzare al capire se il mio programma ha un certo comportamento rispetto ad una funzione di riferimento.

Vogliamo quindi

$$ID(i) = \begin{cases} 1 & \text{se } \forall x, \varphi_i(x) = x \\ 0 & \text{altrimenti} \end{cases}$$

Possiamo dimostrare che questa funzione non è calcolabile, con la stessa tecnica di prima. Ci riduciamo al problema della terminazione.

Costruiamo g tale che

$$g(i,x) = \begin{cases} x & \text{se } \varphi_i(i)\downarrow \\ \uparrow & \text{se } \varphi_i(i)\uparrow \end{cases}$$

Per s-m-n esiste h tale che $\varphi_{h(i)}(x) = g(i,x)$. Mi chiedo ora, $h(i)$ è la funzione identita?

$$ID(h(i)) = \begin{cases} 1 & \text{se } \varphi_i(i)\downarrow \\ 0 & \text{se } \varphi_i(i)\uparrow \end{cases}$$

Bisogna partire da una funzione calcolabile, altrimenti non si può applicare s-m-n. È parte fondamentale della dimostrazione mostrare che g è calcolabile.

4.4 Il predicato T di Kleene

Ci chiediamo ora se esista un predicato calcolabile $T(i,x,t)$ che mi dice se il programma i termina la computazione su input x entro il tempo t.

C'è il problema di come definiamo il tempo; tuttavia l'idea è che tutti i programmi di cui parliamo sono di tipo discreto, hanno un passo discreto di calcolo.

È calcolabile? Intuitivamente sì. Immaginiamo di avere un interprete. Interpretiamo il programma i, seguendolo passo per passo per input x. Se dopo t passi la computazione è terminata allora ho una risposta positiva; in caso contrario no.

Esistono tante varianti del predicato T di Kleene. Questa è una forma "ternaria". Possiamo pensare ad una versione "quaternaria" $T^4(i, x, y, t)$ che mi dice se il mio programma termina su output y in t o meno passi su input x.

La forma originale di Kleene era un predicato ternario $T(i, x, tr)$, dove tr è una traccia computazionale. Si può vedere come una sequenza di configurazioni istantanee. Non deve essere necessariamente completa, deve essere corretta. È ancora chiaramente decidibile se la traccia seguita sia quella passata in input. Questa forma in un certo senso comprende le prime due. L'idea è che la lunghezza della traccia è il tempo passato dall'inizio della computazione.

Si può addirittura dimostrare che T è primitivo ricorsivo in generale. Inoltre ha una complessità computazionale relativamente bassa. La complessità è lineare in t e nelle altre componenti.

C'è un corollario del predicato T di Kleene. Se supponiamo questo predicato come primitivo possiamo scrivere, sulla base di questo, la macchina universale, la cui esistenza smette di essere primitiva nella nostra teoria della calcolabilità.

Infatti possiamo definire u nel seguente modo:

$$u(i, x) = fst(\mu < y, t >, T(i, x, y, t))$$

Questo corrisponde a scrivere $fst(\mu w, T(i, x, fst(w), snd(w)))$. Il fst più esterno serve perché a me non interessa t, interessa y.

Questo è un risultato importante e si chiama forma normale di Kleene. C'è un corollario importante. Si potrebbe limitare sintatticamente un programma ad un while con all'interno un for e questo non diminuirebbe il potere espressivo del formalismo, poichè l'interprete è scrivibile in questi termini.

Un altro problema non calcolabile è il problema del raggiungimento di codice. Ovvero, dato un programma e un'istruzione il problema di decidere se il programma raggiungerà mai quell'istruzione non è calcolabile. Esistono delle tecniche ma queste non sono generali. Sono tipicamente usate dai compilatori per ottimizzare il codice oggetto ed eliminare parti di codice inutile.

Capitolo 5

Insiemi ricorsivi e ricorsivamente enumerabili

Supponiamo di voler trasmettere un insieme. Se questo è finito non c'è nessun problema, basta mandare i suoi elementi. Se ho un insieme infinito posso trasmettere il programma che calcola la funzione caratteristica del mio insieme. Non posso immaginare di poter trasmettere tutti gli insiemi in questo modo, altrimenti potrei risolvere il problema della terminazione diagonale. Quelli per cui è possibile sono detti ricorsivi.

Definizione 5.1. Un insieme A è ricorsivo sse c_A è calcolabile.

Un altro modo per trasmettere il mio insieme in maniera effettiva è dare un metodo di calcolo, detto funzione enumerativa. Supponiamo di avere un insieme $a_0, a_1, a_2, \ldots$. Diamo una funzione f tale che $f(0) = a_0$, $f(1) = a_1$, ecc. Abbiamo che $A = cod(f)$.

Quando posso dare l'insieme in questa maniera ed f è calcolabile si dice che l'insieme è ricorsivamente enumerabile. Ricorsivamente è terminologia vecchia, degli anni trenta del XX secolo. In queste due accezioni ricorsivo va visto come sinonimo di calcolabile.

Definizione 5.2. A è ricorsivamente enumerabile sse esiste una funzione di enumerazione f per A calcolabile: $A = cod(f)$, per f calcolabile.

5.1 Relazione tra insiemi ricorsivi e r.e.

Esistono quindi sostanzialmente due modi per descrivere in maniera effettiva un insieme A. Vogliamo capire che relazione c'è tra queste due nozioni: qual è più potente? Come si rapportano?

Per descrivere una classe è spesso utile capire rispetto a quali operazioni la classe è chiusa. In particolare, se A è ricorsivo cosa possiamo dire del complementare di A, o dell'unione/intersezione/differenza di A con un altro insieme ricorsivo?

Ricordiamo che $A \subseteq \mathbb{N}$. Abbiamo che:

- $\emptyset$ è ricorsivo: la funzione costante **0** è calcolabile;
- $\mathbb{N}$ è ricorsivo: la funzione costante **1** è calcolabile;
- ogni insieme finito è ricorsivo: D è finito $\implies$ D è ricorsivo:

$$c_D(x) = \begin{cases} 1 & \text{se } x = d_1 \vee x = d_2 \vee \ldots \vee x = d_n \\ 0 & \text{altrimenti} \end{cases}$$

  Sarebbe esprimibile anche con una serie di *if*. Qualunque sia D ho un modo per scrivere la mia funzione caratteristica.

Con lo stesso ragionamento posso dimostrare che molte altre funzioni sono calcolabili. Ad esempio una funzione con *Range* finito è sicuramente calcolabile; è scrivibile con un *case* ad esempio. Tutte le funzioni scrivibili con un *case* sono sicuramente calcolabili. Queste funzioni hanno sempre un numero finito di elementi su cui hanno un comportamento interessante e nei restanti hanno lo stesso comportamento.

L'insieme dei numeri pari è ricorsivo. Tutti gli insiemi primitivi ricorsivi, ovvero con funzione caratteristica primitiva ricorsiva, sono sicuramente ricorsivi. Infatti le funzioni primitive ricorsive sono un caso particolare di algoritmo.

Esistono insiemi non ricorsivi. Ad esempio $K = \{i \mid \varphi_i(i) \downarrow\}$ non è ricorsivo.

Supponiamo che A,B siano ricorsivi. Abbiamo che:

- $\overline{A}$ è ricorsivo. Questo perche $c_{\overline{A}}(x) = 1 - c_A(x)$
- $A \cup B$ è ricorsivo. Questo perche $c_{A \cup B}(x) = \max(c_A(x), c_B(x))$
- $A \cap B$ è ricorsivo. Questo perche $c_{A \cap B}(x) = \min(c_A(x), c_B(x))$

Gli insiemi ricorsivi formano un'algebra di Boole, essendo chiusi per queste tre operazioni. Sono una sottostruttura interessante degli insiemi.

Data una funzione caratteristica è facile costruire una funzione di enumerazione. In altri termini, se un insieme è ricorsivo è anche ricorsivamente enumerabile. C'è un piccolo inghippo però a cui bisogna fare attenzione.

La funzione di enumerazione potrebbe essere costruita nel seguente modo ad esempio:

```
def e_A(i):
    c = 0
    j = 0
    while c ≤ i:
        if c_A(j):
            c++
        j++
    return j − 1
```

In versione funzionale potremmo scrivere scrivere e_A come $e_A(i+1) = \mu j, c_A(j) = 1 \wedge j > e_A(i)$, con $e_A(0) = \mu j, c_A(j) = 1$. Il piccolo dettaglio a cui fare attenzione è che la funzione di enumerazione la vorremmo totale e calcolabile. Tuttavia questo non è un vincolo così restringente.

Diamo la seguente definizione: A è r.e. se esiste f totale calcolabile tale che $A = cod(f)$ oppure A è vuoto.

Il problema della nostra funzione di enumerazione è che può divergere con insiemi finiti. Se, ad esempio, ho un A con 7 elementi e cerco l'ottavo con la mia funzione avrò che divergerà.

Il teorema rimane, se A è ricorsivo è ricorsivamente enumerabile. Bisogna giusto ricordarsi che per gli insiemi finiti va fatto un caso speciale che gestisca input maggiori della cardinalità di A.

Più precisamente, se A è finito, con $A = a_0, \ldots, a_n$, allora definiamo e_A nel seguente modo:

$$e_A(x) = \begin{cases} x = 0 \Rightarrow a_0 \\ x = 1 \Rightarrow a_1 \\ \vdots \\ x = n \Rightarrow a_n \\ default \Rightarrow a_n \end{cases}$$

Abbiamo quindi che Ricorsivo $\implies$ R.E. Vale il viceversa? La risposta è no, ma non è del tutto ovvia.

Sia A un insieme r.e. Potremmo pensare di calcolare la funzione caratteristica di A con la sua funzione di enumerazione, nel seguente modo:

```
def c_A(x):
    i = 0
    while e_A(i) ≠ x:
        i++
    return 1
```

Questa c_A però non calcola la funzione caratteristica di A, bensì la funzione semicaratteristica di A,che indicheremo con s_A.

$$s_A(x) = \begin{cases} 1 & \text{se } x \in A \\ \uparrow & \text{altrimenti} \end{cases}$$

Questa funzione è calcolabile e parziale. Per ora abbiamo dimostrato che la funzione semicaratteristica di A è calcolabile, non abbiamo ancora dimostrato che r.e. $\not\implies$ ricorsivo. Dobbiamo mostrare un esempio di insieme r.e. che non sia ricorsivo. K è non ricorsivo. K è r.e.?

Proviamo a scrivere una funzione di enumerazione per K:

```
def e_K(< i, t >):
    if T^3(i, i, t) = 1:
        return i
```

```
else:
    return k_0
```

Nell'else è importante che restituisca un programma che sta in K. Ce ne sono tanti e di semplici; ad esempio le funzioni costanti. Diamo indice k_0 ad un tale programma.

È una buona funzione di numerazione? È sicuramente calcolabile. È suriettiva? È facile vedere che numero solo cose che stanno in K: k_0 sta in K e se restituisco i allora i stava in K. Se i sta in K vuol dire che esiste un t tale che $\varphi_i \downarrow$ in t passi. Di conseguenza vengono enumerate tutte le funzioni convergenti su i di indice i.

Il dovetailing è al variare dell'input. Mi muovo sugli input e sul tempo di computazione.

Si potrebbe fare anche nel seguente modo:

$$e_K(c) = fst(\mu < i, t >, < i, t > \geq c \wedge T(i, i, t))$$

Il risultato finale è che K è ricorsivamente enumerabile.

C'è un'altra relazione notevole tra insiemi r.e. e insiemi ricorsivi.

Teorema 5.1. *Sia A un insieme tale che sia A che $\overline{A}$ sono r.e. Allora A è ricorsivo.*

Dimostrazione. Supponiamo di aver e_A ed $e_{\overline{A}}$. Informalmente, possiamo far partire la ricerca per entrambe le funzioni. Prima o poi una terminerà, e in base a quale termina ho la mia risposta. La computazione parallela è assolutamente algoritmica e non c'è problema al riguardo.

Più formalmente, sia h così definita:

$$h(x) = \begin{cases} e_A(x) & \text{se } x = 2n \text{ per qualche } n \\ e_{\overline{A}}(x) & \text{se } x = 2n+1 \text{ per qualche } n \end{cases}$$

Possiamo scrivere $c_A(n) = pari(\mu i, h(i) = n)$ □

Come corollario di questo teorema abbiamo che esiste un insieme nè ricorsivo nè r.e. Quale? $\overline{K}$. Se infatti $\overline{K}$ fosse r.e. allora K sarebbe ricorsivo.

Abbiamo una gerarchia degli insiemi fatta in questo modo:

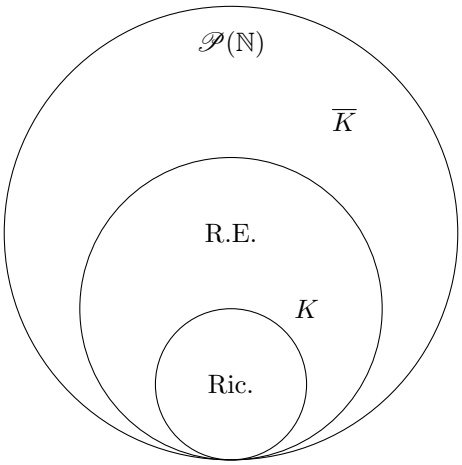


Figura 5.1: Gerarchia degli insiemi r.e. e ricorsivi

5.2 Enumerazioni iniettive e monotone

Ci concentriamo ora sulle proprietà dell'enumerazione. Ci chiediamo in particolare cosa possiamo dire riguardo alla sua monotonia e alla sua iniettività.

Supponiamo di poter enumerare A con una funzione effettiva. Ci chiediamo se possiamo trasformare f in modo da non avere ripetizioni. Se pensiamo all'enumerazione come ad uno stream infinito di numeri in input ci chiediamo se possiamo creare un filtro che elimini i duplicati dallo stream.

Questo si può fare in maniera algoritmica tenendo una lista dei numeri ricevuti dallo stream e, ogni volta che ho un nuovo numero, controllo se è gia nella lista. In base al risultato decido se aggiungerlo o meno alla lista. La mia lista è ora il mio nuovo stream senza ripetizioni. Abbiamo quindi che non è restittivo richiedere che la mia enumrazione sia iniettiva.

Diamo quindi la seguente definizione:

Definizione 5.3. A è R.E. sse $\exists f$ iniettiva tot. calc. tale che $A = cod(f)$ oppure A è finito.

A livello formale data l'enumerazione f possiamo costruire una nuova enumerazione g per lo stesso insieme che sia iniettiva nel seguente modo:

$$g(x) = \begin{cases} f(0) & \text{se } x = 0 \\ f(\mu i, \forall y \leq x-1, f(i) \neq g(y)) & \text{altrimenti} \end{cases}$$

Questo ha senso per insiemi infiniti.

L'iniettività è quindi una proprietà accettabile per la mia enumerazione. Posso trasformarla anche in una funzione monotona crescente? La risposta sarà no, non posso enumerare gli elementi di A in maniera crescente se A è r.e.

Il problema è fondamentalmente che non posso assicurarmi che un dato elemento sia il più piccolo nello stream. La ricerca divergerebbe. Ogni volta che cerchiamo un minimo c'è un problema del genere.

L'idea è che se potessimo ordinare gli elementi di A allora A sarebbe ricorsivo. Dato che non tutti gli insiemi r.e. sono ricorsivi non si può fare in generale.

Sia f un'enumerazione strettamente crescente. Essendo discreta cresce strettamente più dell'identità. Se voglio sapere se x fa parte della mia numerazione posso controllare gli elementi fino ad x, oltre sicuramente non lo troverò.

Più formalmente, la ricerca di x può essere fatta col seguente algoritmo:

i_x = $\mu i, f(i) \geq x$
return $(f(i_x) == x)$

In questo modo sono sicuro, se f è strettamente crescente, che la mia ricerca termina. Per capire se x stava nel mio stream controllo che $f(i_x)$ sia uguale ad x.

Questo algoritmo funziona se la mia funzione è non decrescente? Se la mia funzione diventa indefinitamente costante da un certo punto in poi potrei andare avanti all'infinito. La mia minimizzazione non ha più garanzia di terminare.

Questo problema sorge quando $cod(f)$ è finito. Quando $cod(f)$ è infinito prima o poi la funzione assumerà un nuovo valore. Tuttavia se $cod(f)$ è finito allora l'insieme enumerato da f è ricorsivo.

Ogni insieme ricorsivo infinito è enumerabile mediante una funzione di enumerazione crescente. Ogni insieme ricorsivo non vuoto è enumerabile mediante una funzione di enumerazione non decrescente.

Se si è in grado di ordinare si è anche in grado di decidere. Gli approcci generativi sono generalmente semidecidibili e permettono di generare elementi in maniera disordinata.

Riprendiamo ora $K = \{i \mid \varphi_i(i) \downarrow\}$. Come decidiamo se un certo i sta in K? Possiamo far partire il mio interprete e aspettare. Se termina allora i appartiene a K. Se non appartiene a K non lo saprò mai, perché dovrei aspettare che il programma termini. Di K non possiamo quindi calcolare la funzione caratteristica, bensì la funzione semicaratteristica:

$$s_K(i) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

In generale la funzione semicaratteristica di A, con A r.e., è fatta nel seguente modo:

$$s_A(x) = \begin{cases} \downarrow & \text{se } x \in A \\ \uparrow & \text{se } x \notin A \end{cases}$$

In alcuni casi posso semidecidere la appartenenza di un elemento ad un insieme, in altri casi potrei non esserne in grado.

5.3 Insiemi r.e. e semidecisione

Ci chiediamo ora che legame esista tra la capacità di enumerare e la capacità di semidecidere. Ce lo chiediamo in generale, ricordando che per K l'abbiamo già visto.

Supponiamo che A sia vuoto oppure $A = \mathit{cod}(f)$, con f tot. calcolabile. Come faccio a costruire l'algoritmo di semidecisione per A? Semplicemente comincio a cercare x nell'enumerazione.

$$s_A(x) = (\mu i, (f(i) = x))$$

Come facciamo per il viceversa? Abbiamo la procedura di semidecisione per A e vogliamo costruire la funzione di enumerazione per A. f è delicata perché potrebbe divergere. Dobbiamo essere più astuti. L'idea è muoversi mediante dove tailing sull'input e sul tempo di computazione.

$$f(< x, t >) = \begin{cases} x & \text{se } s_A(x) \text{ termina nel tempo } t \\ a_0 & \text{altrimenti} \end{cases}$$

f è totale e calcolabile. Abbiamo quindi un'enumerazione per A. Non è iniettiva, ma questo non importa. a_0 è un elemento qualsiasi di A. Possiamo quindi vedere la semidecisione come enumerazione di un insieme r.e. e viceversa.

Non siamo sicuri di poter trovare a_0. Questo pezzo della dimostrazione è non costruttivo. Dal punto di vista classico però o l'insieme è vuoto, e quindi r.e., oppure non lo è e deve avere almeno un elemento a_0. Deve quindi esistere la funzione calcolabile f, anche se non è detto che possiamo trovarla.

Vediamo ora un'applicazione. Quali sono le proprietà di chiusura per gli insiemi r.e.? Sono chiusi per complementazione? No, altrimenti gli insiemi r.e. sarebbero tutti ricorsivi, e sappiamo che questo non è vero. Ad esempio K è r.e. e $\overline{K}$ non lo è.

Cosa possiamo dire di $A \cup B$ e $A \cap B$? Per $A \cup B$ è abbastanza ovvio. Dati f_A e f_B possiamo costruire $f_{A \cup B}$ in modo che $f_{A \cup B}(2x) = f_A(x)$ e $f_{A \cup B}(2x+1) = f_B(x)$. Per $A \cap B$ è semplice costruire il semidecisore di $A \cap B$ dati quelli per A e per B. Basta concatenare i due: $s_{A \cap B}(x) = s_A(x); s_B(x)$.

Piccola nota sulla notazione. Noi consideriamo funzioni parziali da $\mathbb{N}$ a $\mathbb{N}$: $f : \mathbb{N} \multimap \mathbb{N}$. Indichiamo, in maniera impropria, $\mathit{cod}(f) = \{y \mid \exists x, y = f(x)\}$ e $\mathit{dom}(f) = \{x \mid f(x) \downarrow\}$. Il nostro codominio in realtà è il *Range* della funzione ed il nostro dominio è il dominio di congergenza (o co-*Range*). In termini del grafo abbiamo che $\mathit{dom}(f) = \{x \mid \exists y, < x, y > \in \mathit{grafo}(f)\}$.

5.4 Caratterizzazioni alternative di un insieme r.e.

Vediamo ora alcune caratterizzazioni, tutte equivalenti, degli insiemi r.e. che sono più adatte in certi contesti.

Se si può caratterizzare un insieme r.e. A come il dominio di convergenza di una funzione s abbiamo che s rappresenta la funzione di semidecisione di A.

Abbiamo visto che A è r.e. sse $A = \emptyset \vee A = cod(f)$, f tot. calc.

Abbiamo però che A può essere visto come $dom(g)$, con g parz. calc. È una definizione equivalente a quella appena citata.

Si può infine definire A anche come $A = cod(h)$, con h parz. calc.

Solitamente quando costruiamo una funzione di enumerazione la vogliamo totale, è meno problematica.

Le tre definizioni sono equivalenti appena viste sono equivalenti.

Teorema 5.2. *Le tre seguenti affermazioni sono equivalenti e definiscono tutte un insieme r.e. A:*

1. $A = \emptyset \vee A = cod(f)$, *con* f *totale calcolabile*
2. $A = dom(g)$, *con* g *parziale calcolabile*
3. $A = cod(h)$, *con* h *parziale calcolabile*

Dimostrazione. Lo dimostriamo facendo vedere che (1) $\implies$ (2), (2) $\implies$ (3) e (3) $\implies$ (1).

- (1) $\implies$ (2):

  Se $A = \emptyset$, allora abbiamo che $A = dom(f_\emptyset)$, se indichiamo con $f_\emptyset$ la funzione ovunque divergente.

  Se invece $A \neq \emptyset$ sia f tale che $A = cod(f)$. Definiamo g come

  $$g(x) = \mu y, f(y) = x$$

  Si ha che $A = dom(g)$.

- (2) $\implies$ (3):

  Sia $A = dom(g)$, con g parziale calcolabile. Sia h la funzione calcolata dal seguente programma:

  ```
  def h(x):
      g(x)
      return x
  ```

  Se $g(x)$ termina restituisco x. Si ha che $A = cod(h)$.

- (3) $\implies$ (1):

  Abbiamo due casi: A è vuoto oppure no.

  Se $A = \emptyset$ l'asserto è già dimostrato.

Se $A \neq \emptyset$ allora $\exists a \in A$. A questo punto definisco la mia f come:

$$f(< x,t >) = \begin{cases} h(x) & \text{se } h(x) \text{ termina entro il tempo } t \\ a & \text{altrimenti} \end{cases}$$

f è totale, perché do sempre in output un valore. È calcolabile, sotto l'ipotesi che h sia calcolabile. Infine $cod(f) = A$ perché f restituisce solo elementi in A e me li restituisce tutti, dato che $A = cod(h)$.

□

La definizione (3) ci permette di non doverci soffermare sul caso particolare dell'insieme vuoto, che non ha implicazioni tecniche importanti.

5.4.1 Teorema della proiezione

Un'ulteriore caratterizzazione degli insiemi r.e. è la seguente:

Teorema 5.3. *A è r.e. sse $\exists B$ ricorsivo tale che $A = \{x \mid \exists y, < x,y > \in B\}$.*

Dimostrazione. Vediamo i due versi del $\iff$ separatamente:

- $\implies$ Sia c_B la funzione caratteristica di B. Allora $A = dom(f)$, con

$$f(m) = \mu n, c_B(< n,m >) = 1$$

- $\impliedby$ Sia $A = dom(\varphi_i)$. Quindi $m \in A$ sse $\varphi_i(m) \downarrow$, ovvero se $\exists t, T^3(i,m,t) = 1$, dove T è il predicato ternario di Kleene. Basta quindi porre:

$$B = \{< m,t > \mid \exists t, T^3(i,m,t)\}$$

□

Possiamo vedere B come una codifica delle coppie oppure come un insieme di coppie, non è importante.

A è la proiezione esistenziale di B

Il teorema è importante. Se ho un insieme ricorsivo e ne faccio la proiezione esistenziale ottengo un insieme r.e. $x \in A$ sse $\exists y, (x,y) \in B$. Posso verificare se (x,y) sta in B perché B è ricorsivo. Posso quindi semidecidere l'appartenenza di x ad A con un algoritmo del genere ad esempio: $\mu y, (x,y) \in B$.

Se io faccio una ricerca in un dominio ricorsivo non è detto che questa termini. Se potessi dare dei bound alla mia ricerca o avessi altre ipotesi potrei trasformare il mio algoritmo di semidecisione in un algoritmo di decisione, ma in generale non è questo il caso. Avremo in generale una funzione di semidecisione.

L'altra conseguenza importante è che qualunque problema semidecidibile può essere visto come una ricerca in uno spazio decidibile.

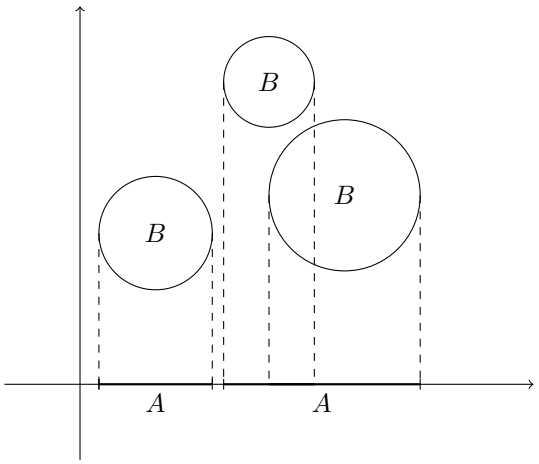


Figura 5.2: Proiezione esitenziale di B

È utile vedere y come certificato di x, c_x. Il fatto che x appartenga ad A è cerficato da c_x. Se $x \in A$ sappiamo che questo ceritificato esiste ma non sappiamo quale sia. La sua ricerca non è però un problema decidibile.

Prendiamo per esempio la logica del prim'ordine. Se ho una formula in generale non è decidibile se questa sia dimostrabile. Tuttavia abbiamo un algoritmo che genera tutte le formule dimostrabili nella logica del prim'ordine. Dimostrare F con questo algoritmo significa andare a cercare, nell'insieme delle formule generate, un certificato per F.

Un altro esempio è: dato un programma datemi un certificato per la sua appartenenza a K. Questo certificato potrebbe essere il tempo t in cui $\varphi_x(x)$ termina. La dimostrazione che abbiamo visto procede proprio in questo modo.

Il certificato non è un concetto ben definito, sta a noi decidere cosa sia un certificato. Un certificato per un teorema potrebbe benissimo essere sia una prova che semplicemente la dimensione della prova.

Ritorneremo su questo concetto quando parleremo di $\mathbb{P}$ e $\mathbb{NP}$. Vedremo che $\mathbb{NP}$ è l'insieme dei linguaggi che sono proiezione esistenziale di un qualche linguaggio in $\mathbb{P}$.

La caratterizzazione $A = dom(\varphi_i)$ è così importante che abbiamo della notazione specifica per indicarlo: W_i. Possiamo con questa notazione definire K come $K = \{i \mid i \in W_i\}$. L'importanza di questa caratterizzazione è data dal fatto che essa induce una enumerazione di tutti gli insieri r.e., basata sui domini di convergenza delle funzioni della mia enumerazione delle funzioni calcolabili.

5.5 Ulteriori proprietà di chiusura degli insiemi r.e.

Teorema 5.4. *Siano $A, B \subseteq \mathbb{N}$ e $f : \mathbb{N} \to \mathbb{N}$. Abbiamo che:*

1. *se A è ricorsivo e f è totale calcolabile, allora $f^{-1}(A)$ è ricorsivo*
2. *se A è r.e. e f è calcolabile, allora $f^{-1}(A)$ è r.e.*
3. *se A è r.e. e f è calcolabile, allora $f(A)$ è r.e.*

Dimostrazione. Vediamo i tre punti uno per uno:

1. $c_{f^{-1}(A)}(x) = c_A(f(x))$
2. Sia $A = dom(g)$, con g calcolabile. Consideriamo la seguente funzione:
$$h(x) = g(f(x))$$
Questa funzione termina su x sse $x \in dom(f) \wedge f(x) \in dom(g)$. Ma $dom(g) = A$, quindi h termina su tutti quegli x tali che $f(x) \in A$. Ma quindi $f^{-1}(A) = dom(h)$.
3. Sia $A = cod(g)$. Consideriamo la seguente funzione:
$$h(x) = f(g(x))$$
Abbiamo che $cod(h) = f(cod(g))$. Ma $cod(g) = A$, quindi $f(A) = cod(h)$.

□

Vediamo ora che le famiglie r.e. di insieme r.e. sono chiuse per unione ma non per intersezione. Per farlo ci servirà l'enumerazione degli insiemi r.e. indotta dai domini delle funzioni calcolabili.

Lemma 5.1. *Valgono le seguenti proprietà per gli insiemi r.e.:*

1. $\forall x, \bigcup_{i \in W_x} W_i$ *è r.e.;*
2. $\exists x, \bigcap_{i \in W_x} W_i$ *non è r.e.;*

Dimostrazione. Vediamo i due punti uno per uno:

1. possiamo scrivere il semidecisore di $\bigcup_{i \in W_x} W_i$ come segue, mediante dovetailing:
$$s_{\bigcup_{i \in W_x} W_i}(x) = \mu < i, t >, T(i, x, t)$$
2. Sia g la funzione così definita:
$$g(i, x) = \begin{cases} \uparrow & \text{se } T(x, x, i) \\ 0 & \text{altrimenti} \end{cases}$$
Per s-m-n esiste h totale calcolabile tale che:
$$W_{h(i)} = \{x \mid \neg T(x, x, i)\}$$

dove T è il predicato ternario di Kleene. Si dimostra facilmente che:

$$\bigcap_{i \in \mathbb{N}} W_{h(i)} = \overline{K}$$

Poichè $\bigcap_{i \in \mathbb{N}} W_{h(i)} = \bigcap_{i \in cod(h)} W_i$ l'asserto risulta verificato.

□

Ulteriore nota, nella seconda dimostrazione abbiamo che gli insiemi $W_{h(i)}$ sono ricorsivi e h può essere scelta monotona crescente, quindi una intersezione ricorsiva di insiemi ricorsivi non è in generale r.e.

Capitolo 6

I teoremi di Rice e di Rice-Shapiro

Andiamo ora a vedere un teorema importante: il teorema di Rice.

Pensiamo alla proprietà $P(i) =$ "$\forall n, \varphi_i(n) \geq k$". Possiamo già intuire dalla quantificazione universale che la mia proprietà non è decidibile. Inoltre rimane il problema della divergenza. Un'altra proprietà interessante è analoga ma ha il quantificatore esistenziale al posto di quello universale. Questa è semidecidibile sicuramente, muovendosi con dove-tailing su input e tempo. È decidibile? Intuitivamente no, dovrei fare una ricerca su uno spazio infinito. Se sono fortunato in questi casi ho una proprietà semidecidibile.

Quello che andiamo a dimostrare adesso è che riguardo alle proprietà dei programmi non possiamo decidere niente. Non esiste alcuna proprietà decidibile, ad esclusione delle proprietà di falsità/verità costanti. Questo è dimostrabile in generale.

Questo qua è il succo del teorema di Rice.

6.1 Proprietà estensionali

Come caratterizziamo il comportamento delle funzioni calcolate dai programmi? Nel seguente modo: data una proprietà P ci chiediamo se $\varphi_i = \varphi_j$ implicihi $P(i) = P(j)$. Se questo è il caso allora dico che P è estensionale rispetto a φ. È importante il legame con l'enumerazione, che dato un numero mi restituisce la funzione calcolata dal programma con indice quel numero.

Possiamo caratterizzare un predicato con l'insieme degli indici delle funzioni per cui il predicato è vero. Parliamo in generale di insiemi estensionali. A è estensionale se c_A è tale che $\varphi_i = \varphi_j \implies c_A(i) = c_A(j)$. Gli insiemi estensionali sono chiusi rispetto a complementazione, unione e intersezione; formano quindi un'algebra di Boole.

φ è una funzione dai naturali alle funzioni parziali calcolabili: $\varphi : \mathbb{N} \to \mathcal{PC}$. Prendiamo $B \subseteq \mathcal{PC}$. A è estensionale se $A = \varphi^{-1}(B)$. Queste due caratterizzazioni sono equivalenti. La dimostrazione è lasciata al lettore.

6.2 Il teorema di Rice

Teorema 6.1. *Un insieme A estensionale è ricorsivo sse $A = \emptyset$ o $A = \mathbb{N}$ (o, equivalentemente, A è banale).*

Banale in matematica significa solitamente degenere.

Dimostrazione. L'implicazione inversa è banale. Dimostriamo solo quella diretta. Sia A estensionale. Supponiamo che A sia ricorsivo. Vogliamo dimostrare che A è vuoto oppure $A = \mathbb{N}$. Procediamo per assurdo. Supponiamo che $A \neq \emptyset$ e $A \neq \mathbb{N}$. Esistono allora $a_0 \in A$ e $a_1 \notin A$. Sia m un indice per la funzione che diverge sempre. Abbiamo due casi: o $m \in A$ o $m \in \overline{A}$. I due casi sono assolutamente simmetrici. Supponiamo, senza perdita di generalità, che $m \in \overline{A}$. A seconda che $\varphi_i(i)$ converga o diverga voglio costruire un programma che ha lo stesso comportamento di m in un caso e lo stesso comportamento di a_0 nell'altro. Vogliamo g tale che:

$$g(i,x) = \begin{cases} \varphi_{a_0}(x) & \text{se } \varphi_i(i) \downarrow \\ \varphi_m(x) & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Per s-m-n abbiamo $\varphi_{s(i)}(x) = g(i,x)$. Come calcoliamo $g(i,x)$? Col seguente algoritmo: $g(i,x) = \varphi_i(i); \varphi_{a_0}(x)$. Di conseguenza g è calcolabile.

Ci chiediamo ora: $s(i) \in A$? Dipende se $\varphi_i(i)$ converge. $s(i) \in A \iff \varphi_i(i) \downarrow$. Ma quindi anche K sarebbe ricorsivo. Ma questo è assurdo. □

6.3 Teorema di Rice-Shapiro

6.3.1 Proprietà compatte e monotone

Cerchiamo ora di generalizzare il teorema di Rice. Cosa possiamo semidecidere del comportamento dei programmi, tenuto vero quanto espresso dal teorema di Rice?

Prendiamo la proprietà essere totali: $P(i) =$ "φ_i è totale". Intuitivamente questa proprietà non è semidecidibile, perché dovrei esplorare tutti gli input prima di rispondere.

Proviamo con il complementare. Con un pò di abuso di notazione scriviamo $\overline{P}(i) =$ "φ_i non è totale" = "$\exists n, \varphi_i(n) \uparrow$". La divergenza non è testabile: si tratta sempre di una ricerca in uno spazio infinito, dove l'infinità è data dal tempo. Per questo motivo neanche questa proprietà è semidecibile, almeno a livello intuitivo.

Cosa posso sicuramente semidecidere? La convergenza puntuale ad esempio. L'algoritmo è semplice: lancio il programma i sul mio input x e aspetto. Se converge mi restituirà qualcosa, altrimenti divergerà.

Intuitivamente, le proprietà semidecidibili sono i test che riguardano la convergenza su un numero finito di input e i risultati relativi. Questo assomiglia un pò alla procedura di testing: osserviamo il comportamento del programma su un numero finito di input e verifichiamo se passa il mio test. È un testing semidecidibile. La cosa importante è che sia finito, non è neanche richiesto che sia determinato.

Ad esempio, prendiamo $P(i) =$ "$\exists n, \varphi_i(n) \downarrow$". Questo test è chiaramente semidecibie, mediante dove-tailing su n e tempo.

Vogliamo ora formalizzare l'idea di cosa è semidecidibile e cosa no. Cerchiamo alcune proprietà interessanti che sono soddisfatte da questi test.

Ci serve un ordinamento tra funzioni. Cerchiamo una nozione di ordinamento basata sul grafo delle funzioni. Diciamo che $\varphi_i \subseteq \varphi_j$ se $\mathit{grafo}(\varphi_i) \subseteq \mathit{grafo}(\varphi_j)$. In simboli questo corrisponde a $\forall n \forall m, \varphi_i(n) = m \implies \varphi_j(n) = m$. Quand'è che j può avere un comportamento diverso da i? Quando i diverge. In questo senso j è una "estensione" di i.

$\varphi_i \subseteq \varphi_i$, in base alla nostra definizione. φ_j è, in un certo senso, più informativa di φ_i, dato che dove φ_i è definita il suo valore è identico a quello di φ_j, e dove φ_i non è definita φ_j può darmi una nuova informazione.

Qualunque funzione è un'estensione della funzione che diverge sempre. Sia $P(i)$ una proprietà "testabile". Supponiamo che $P(i)$ e supponiamo che $\varphi_i \subseteq \varphi_j$. Cosa possiamo dire su φ_j rispetto a P? Per fissare le idee, prendiamo $P(i) =$ "φ_i converge su tutti i numeri minori di 100". Supponiamo esista un a per cui vale P. Se ho che b estende a, ovvero $\varphi_a \subseteq \varphi_b$, allora necessariamente $P(b)$: qualunque estensione di un programma che converge su input minori di 100 continuerà a farlo.

Un predicato che rispetta la proprietà appena discussa è detto monotono. La monotonia è una proprietà del mio predicato P se per ogni estensione del mio programma a per cui vale P continua a valere la proprietà P.

Vediamo ora un'altra caratteristica che può avere un predicato estensionale. Supponiamo di avere il nostro test e che il test sia vero per un certo programma a. Allora, se esiste un b tale φ_b è una restrizione finita di φ_a e $P(b)$, diciamo che P è compatto. Equivalentemente possiamo esprimere la compattezza come $\exists b$ tale che $\varphi_b \subseteq \varphi_a$, φ_b è finito (come grafo) e $P(b)$. Questa proprietà dei predicati estensionali è detta compattezza.

Abbiamo dei massimi nell'ordinamento basato su estensione? Sì, tutte le funzioni totali, dette in questo contesto massimali. Non sono però in relazione tra loro, sono distinte.

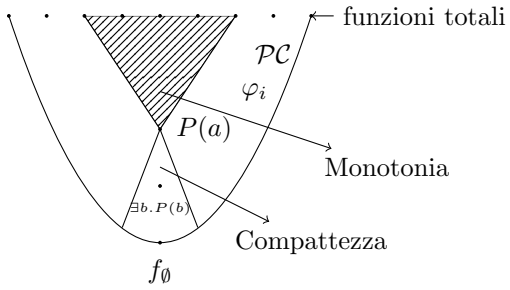


Figura 6.1: Una proprietà P è monotona se tutte vale per tutte le estensioni di una funzione a per cui vale. È compatta se vale per una qualche restrizione finita b di a

Vediamo ora alcuni esempi di proprietà e chiediamoci quali sono compatte e quali monotone (e quali entrambe).

- $P(i) =$ "φ_i è totale". È monotona? Sì. È compatta? No.
- $P(i) =$ "φ_i non è totale". È monotona? No. È compatta? Sì.
- $P(i) =$ "$4 \in cod(\varphi_i)$". È monotona? Sì. È compatta? Sì.
- $P(i) =$ "$cod(\varphi_i) = 4$". È monotona? No. È compatta? Sì.

Dimostreremo che monotonia e compattezza sono condizioni necessarie ma non sufficienti per la semidecidibilità.

Prendiamo $P(i) =$ "$dom(\varphi_i)$ finito". È monotona? No. È compatta? Sì. Per il complementare? È monotono? Sì. È compatto? No.

Una proprietà come $P(i) =$ "φ_i è primitiva ricorsiva" non è neppure semidecidibile, oltre a non essere decidibile per il teorema di Rice. Il motivo è che $P(i)$ non è compatta: il formalismo primitivo ricorsivo mi permette di definire solo funzioni totali.

6.3.2 Il teorema di Rice-Shapiro

I nomi di questo teorema variano nella letteratura. Per noi è il teorema di Rice-Shapiro.

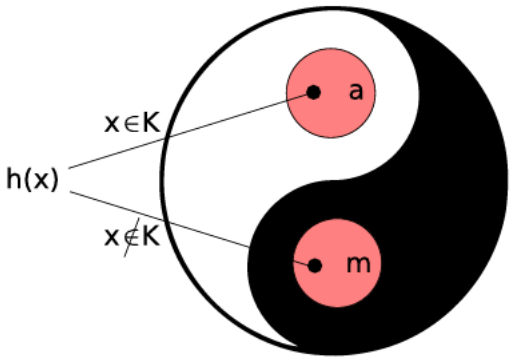


Figura 6.2: Visualizzazione della dimostrazione del teorema di Rice

Il simbolo dello yin-yang mi deve sottolineare l'impossibilità di separare le due aree, perché le due cose che sto tentando di separare sono processi in movimento che continuamente si trasformano nell'una e nell'altra. Non posso operare una distinzione tra le due.

I due cerchi rappresentano degli intorni che non cambiano, che appartengono all'altra area. Da un punto di vista topologico sono due aperti. Due aperti non possono dividere uno spazio, perché il complementare di un aperto è un chiuso.

Possiamo immaginare un cerchio come un intorno di tutti i programmi che hanno lo stesso comportamento di un certo programma. Se cerchiamo di dividere l'insieme di tutti i programmi in due spazi non possiamo aspettarci di poterlo fare in modo algoritmico. Questo è il succo del teorema di Rice. La proprietà di estensionalità, che è una proprietà di chiusura, non ci permette di fare questa divisione.

Nella mia dimostrazione cerco un s che mi permette di trovare una funzione che si comporta come a o m a seconda del comportamento di $\varphi_i(i)$. Questo lo posso fare, è più debole di chiedere che $s(i)$ mi restituisca a o m a seconda del comportamento di $\varphi_x(x)$ e sfrutta l'estensionalità. La prima cosa posso calcolarla, la seconda no.

Teorema 6.2. ***Teorema di Rice-Shapiro.*** *Sia A un insieme estensionale. Se A è r.e. allora:*

- *A è monotono*
- *A è compatto*

Dimostrazione. La monotonia mi dice che $\forall i, j$ se $i \in A$ e $\varphi_i \subseteq \varphi_j$ allora $j \in A$. Supponiamo che A sia estensionale e r.e. Supponiamo che A non sia monotono. Allora devono esistere i, j tali che $i \in A$, $\varphi_i \subseteq \varphi_j$ e $j \notin A$. Proveremo a costruire un programma che si comporta come i in un caso e come j nell'altro caso. Costruiamo h totale e calcolabile che vogliamo si comporti come i se $\varphi_x(x) \uparrow$, come j se $\varphi_x(x) \downarrow$. Supponiamo di averla già: vedremo poi se possiamo

calcolarla. Avremmo che $\varphi_x(x) \in \overline{K} \iff h(x) \in A$. L'implicazione inversa di questo $\iff$ può essere dimostrata dimostrando $\neg B \implies \neg A$. Ma questo vuol dire che se A fosse semidecidibile avrei un modo per calcolare, in modo semidecidibile, l'appartenenza a $\overline{K}$.

Tornando alla definizione del programma che calcola h, con s-m-n passiamo a $\varphi_{h(x)} = g(x,y) = \varphi_i(y)||(\varphi_x(x)\varphi_j(y))$. Questo programma ha lo stesso comportamento di j anche se $\varphi_x(x)$ converge e termina prima $\varphi_i(y)$ perché j è un'estensione di i.

Questa dimostrazione non è costruttiva. È costruttiva invece la dimostrazione per contraddizione: A non monotono $\implies$ A non r.e.

Questa dimostrazione implica, come corollario, il teorema di Rice.

Passiamo alla compattezza. Questa mi dice che $\forall i, i \in A \implies \exists j, \varphi_j \subseteq \varphi_i, \varphi_j$ è finita, $j \in A$.

Andiamo nuovamente per assurdo. Andiamo a negare la compattezza di A: $\exists i, i \in A \forall j, \varphi_j \subseteq \varphi_i, \varphi_j$ è finita $\implies j \notin A$.

Vogliamo costruire una funzione parametrica in x tale che si comporti come i se $\varphi_x(x) \uparrow$ e come una qualsiasi restrizione finita di i quando $\varphi_x(x) \downarrow$. La conclusione della dimostrazione è identica a quella della monotonia, quindi concentriamoci su h.

Utilizzando s-m-n abbiamo $\varphi_{h(x)} = g(x,y)$, con

$$g(x,y) = \begin{cases} \varphi_i(y) & \text{se } T^3(x,x,y) = \text{False} \\ \uparrow & \text{se } T^3(x,x,y) = \text{True} \end{cases}$$

Analizziamo il comportamento di questo programma. Ipotizziamo che $\varphi_x(x)$ sia divergente. Siamo nel primo caso, quindi il comportamento del mio programma è identico a quello di φ_i. Se $\varphi_x(x)$ converge invece ad un certo punto T mi restituirà True e da lì in poi la risposta rimarrà quella. A quel punto la mia funzione diverge. Questa corrisponde ad una restrizione finita di i: non so quanto lunga, so che esiste. Questa funzione è calcolabile, quindi h è calcolabile, da cui l'asserto. □

Capitolo 7

Teorema del punto fisso

7.1 Il teorema del punto fisso di Kleene

Teorema 7.1. *Per ogni funzione totale calcolabile f esiste m tale che*

$$\varphi_{f(m)} = \varphi_m$$

Dimostrazione. Sia g così definita:

$$g(x, y) = \varphi_{f(\varphi_x(x))}(y)$$

Per s-m-n esiste h tale che $\varphi_{h(x)}(y) = g(x, y)$, per x, y arbitari. Sia p un indice per h e poniamo $m = \varphi_p(p) = h(p)$. m è sicuramente definito, essendo h totale. Abbiamo ora che, per ogni y:

$$\varphi_m(y) = \varphi_{\varphi_p(p)}(y) = \varphi_{h(p)}(y) = g(p, y) = \varphi_{f(\varphi_p(p))}(y) = \varphi_{f(m)}(y)$$

□

Se prendiamo per f la funzione successore ho per forza che nel mio ordinamento avrò due programmi adiacenti che si comportano nella stessa maniera. In questo modo non è possibile creare un ordinamento tale che questo non si verifichi. La ragione fondamentale è che non ho controllo sul comportamento dei programmi quando vado ed enumerarli.

Punti fissi e ricorsione sono fondamentalmente la stessa cosa vista da due punti di vista diversi.

La ricorsione è legata alla possibilità che nel corpo del mio programma sia visibile la funzione che sto definendo.

Supponiamo di voler costruire un programma che stampi se stesso, ovvero che stampi il suo indice. Per semplicità lo vogliamo costante, ovvero tale che per ogni input stampi il suo indice: $\forall x, \varphi_p(x) = p$.

Prendiamo $g(i, x)$ così definita:

$$g(i, x) = i$$

Per s-m-n otteniamo una classe di programmi costanti $\varphi_{h(i)}(x)$. Questi però stampano sempre i, anche se $h(i)$ è diverso da i, quindi non vanno bene. Sfruttando il teorema del punto fisso abbiamo però che esiste $\varphi_p(x) = \varphi_{h(p)}(x) = p$. Di conseguenza è sufficiente prendere il punto fisso di h, p.

Se io voglio realizzare un comportamento ricorsivo posso farlo con il teorema del punto fisso di Kleene. Supponiamo di voler una funzione calcolabile φ_p tale che $\varphi_p(x) = f(\varphi_p(x))$, per qualche f. Si può? Sì. Abbiamo che $g(i, x) = f(\varphi_i(x))$. Ma ora per s-m-n abbiamo che questo corrisponde a $\varphi_{h(i)}(x)$. Ma ora abbiamo che $\varphi_p(x) = \varphi_{h(p)}(x) = f(\varphi_p(x))$, se prendiamo come p il punto fisso di h.

In generale per punto fisso di una funzione intendiamo un input x tale che $f(x) = x$. Il nostro senso è un pò particolare perché è un punto fisso estensionale.

Sfruttando questo risultato vogliamo costruire un programma che dato i mi restituisca qualcosa diverso dalla funzione φ_i per almeno un input, ovvero vogliamo i tale che:

$$\exists x, g(i, x) \neq \varphi_i(x).$$

Proviamo con $g(i, x) = \varphi_i(x)+1$. Per s-m-n esiste h tale che $\varphi_{h(i)}(x)$ identico a g. Funziona? No, se i è un indice per la funzione sempre divergente. Si può fare di meglio? No, per il teorema del punto fisso. Per qualunque trasformazione di programmi ho un punto fisso tale che $\varphi_{h(m)} = \varphi_m$. Non ho speranza di fare una modifica effettiva ed uniforme tale che il mio programma sia diverso da quello di partenza, e questo vale per ogni programma.

Vediamo una nuova dimostrazione del teorema di Rice basata sul teorema del punto fisso di Kleene. È concettualmente diversa da quella vista in precedenza.

Sia A un insieme estensionale. A è ricorsivo sse è banale. Supponiamo per assurdo che A sia ricorsivo ma non banale, ovvero esista $i \in A$ e $j \in \overline{A}$. Posso definire h tale che:

$$h(x) = \begin{cases} j & \text{se } x \in A \\ i & \text{se } x \notin A \end{cases}$$

Questa funzione sarebbe calcolabile per la ricorsività di A. Per il teorema del punto fisso, essendo h totale e calcolabile, deve esistere m tale che $\varphi_m = \varphi_{h(m)}$. Ci chiediamo ora, dove sta m? Se $m \in A$ allora stanno in A anche tutti i programmi che si comportano come m, tra cui anche $h(m)$. Ma se $m \in A$ allora $h(m) = j$, quindi $h(m) \notin A$. Se invece $m \notin A$ allora $h(m) \notin A$. Ma per definizione di h si ha che $h(m) = i \in A$. Ecco la mia contraddizione.

Tutte le volte che cerchiamo programmi che dipendono, intensionalmente o meno, dall'indice del programma che sto cercando devo fare riferimento al teorema del punto fisso.

7.2 Il secondo teorema del punto fisso

Pensiamo ora ad una funzione binaria $f(x, y)$. Possiamo generalizzare il teorema del punto fisso:

Teorema 7.2. *$\forall f^2$ totale calcolabile $\exists s$ totale calcolabile tale che $\forall y, \varphi_{f(s(y),y)} = \varphi_{s(y)}$*

Dimostrazione. Definiamo la funzione g come

$$g(x, y, z) = \varphi_{f(\varphi_x(x),y)}(z)$$

Per s-m-n esiste h totale calcolabile tale che:

$$\varphi_{h(x,y)}(z) = g(x, y, z)$$

Definiamo ora $g'(y, x) = h(x, y)$. Per s-m-n esiste r totale calcolabile tale che

$$\varphi_{r(y)}(x) = g'(y, x) = h(x, y)$$

Definiamo ora la funzione $s(y) = \varphi_{r(y)}(r(y))$. Abbiamo che, per ogni z:

$$\varphi_{s(y)}(z) = \varphi_{\varphi_{r(y)}(r(y))}(z) = \varphi_{h(r(y),y)} = \varphi_{f(\varphi_{r(y)}(r(y)),y)}(z) = \varphi_{f(s(y),y)}(z)$$

□

Capitolo 8

Riducibilità

8.1 *Many-to-one* reducibility

Abbiamo in precedenza visto dei procedimenti di "riduzione" da un insieme A a K per dimostare, ad esempio, che se A fosse ricorsivo anche K lo sarebbe. Andiamo ora a definire formalmente la nostra nozione di riducibilità. Ne esistono tante versioni diverse di questa nozione, noi ne vediamo una concettualmente semplice che va sotto il nome di m-riducibilità. La m sta per *many-to-one*; esiste ance la riduzione *one-to-one*.

Definizione 8.1. Siano $A, B \subseteq \mathbb{N}$. Un insieme A è m-riducibile a B ($A \leq_m B$) sse $\exists f$ totale calcolabile tale che $x \in A \iff f(x) \in B$.

Si parla di riducibilità perché possiamo ridurre la appartenenza ad A all'appartenenza a B. Per ogni x mi basta calcolare $f(x)$ e vedere se sta in B per sapere automaticamente se x appartiene ad A. Nella *one-to-one* reducibility si richiede anche che f sia iniettiva. È una nozione che troveremo molto simile nella parte di complessità, avremo solo degli ulteriori limiti sulla complessità di f.

Per dimostrare che $A \leq_m B$ devo fare due cose:

- per prima cosa devo definire $f : \mathbb{N} \to \mathbb{N}$ totale calcolabile;
- dopodiché va dimostrato che è una buona funzione di riduzione, ovvero va dimostrato il $\iff$ della definizione.

È un esercizio creativo, cambia per diversi A a B. La m-riducibilità è un ordine parziale. Infatti $A \leq_m A$, con la funzione identità, e se $A \leq_m B$ e $B \leq_m C$, allora, componendo le due funzioni di riduzione, otteniamo una funzione di riduzione da A a C:$A \leq_m C$.

Notazione: $A \leq_m^f B$ se A è riducibile a B attraverso la funzione f.

Se $A \leq_m B$ e $B \leq_m A$ allora scriveremo $A =_m B$. Intuitivamente sono equivalenti dal punto di vista della riducibilità.

La nozione di riducibilità ci dà un'idea di quanto complicato è un problema. Se $A \leq_m B$ allora B è tanto difficile quanto lo è A. Quanto fine è la mia misurazione dipende dalla potenza della mia nozione di riducibilità. Se la mia nozione di riducibilità non ha troppe pretese rischio di passare da un problema ricorsivo ad uno ricorsivamente enumerabile, e di avere molte riduzioni possibili non molto significative.

Tutti i problemi ricorsivi sono mutuamente riducibili ad esempio. Infatti supponiamo $x \in A \iff f(x) \in B$. A sto punto se B è ricorsivo ovviamente posso decidere l'appartenenza ad A, e quindi A è ricorsivo. Se invece B è r.e. allora, per lo stesso motivo, anche A risulta essere r.e.

L'altra implicazione non è vera. È possibile passare da un problema ricorsivo ad uno ricorsivamente enumerabile. Il minore o uguale ha il significato intuitivo di "A non è più difficile di B".

Supponiamo di avere due insiemi ricorsivi non banali. Mostriamo che sono sempre mutuamente riducibili l'uno all'altro. Dimostriamo solo $A \leq_m B$; il verso opposto usa le stesse ipotesi. Esistono allora $b_1 \in B$ e $b_2 \in \overline{B}$. È banale costruire ora $f(x)$ tale che:

$$f(x) = \begin{cases} b_1 & \text{se } x \in A \\ b_2 & \text{se } x \notin A \end{cases}$$

È una funzione totale calcolabile? Sì. Vale il sse della definizione di riducibilità? Anche, e si vede quasi subito. È importante l'ipotesi di non banalità di A e B. Non abbiamo usato l'ipotesi di ricorsività di B. Se B è un qualsiasi insieme non banale posso ridurre A a B. Senza altre ipotesi potrei non essere in grado di tornare indietro.

Prendiamo un insieme non ricorsivo come K e supponiamo che K sia riducibile ad A. Cosa abbiamo concluso? Che A non è ricorsivo. Analogamente se $\overline{K} \leq_m A$ allora A non è r.e.

Abbiamo infine che $A \leq_m B \iff \overline{A} \leq_m \overline{B}$.

8.2 Completezza

Passiamo ora alla nozione di completezza. Di solito è relativa ad una classe di problemi. Qua siamo interessati alla classe dei problemi semidecidibili.

Definizione 8.2. Diciamo che un certo problema B è completo (per i problemi r.e.) se

- B è r.e.
- $\forall A$, se $A \in \mathcal{RE}$, allora $A \leq_m B$

Potrei definire la stessa nozione per altre classi di problemi.

Il primo punto che ci interessa è, esistono dei problemi completi? La risposta è sì. Prendiamo ad esempio $K_1 = \{< x, y > |\ \varphi_x(y) \downarrow\}$. È banalmente semidecibile. È completo? Supponiamo che A sia r.e. Esiste quindi il semidecisore di A,

s_A. Supponiamo che sia il programma di indice a nella mia enumerazione dei programmi: $\varphi_a(x)$. Quindi $x \in A$ sse $\varphi_a(x) \downarrow$ sse $< a, x > \in K_1$. La funzione f che mi interessa è la funzione che mappa x nella coppia $< a, x >$: $x \mapsto < a, x >$.

Ne esistono altri? Si, ad esempio K è completo. Si può dimostrare direttamente. Noi però dimostreremo che $K \equiv_m K_1$. È facile vedere che $K \leq_m K_1$. La mia f è quella tale che $x \mapsto < x, x >$. Abbiamo che $x \in K \iff \varphi_x(x) \downarrow \iff < x, x > \in K_1$.

La parte $K_1 \leq_m K$ è un pò più tricky. Vogliamo catturare un comportamento puntuale sulla diagonale. Questo è difficile; possiamo piuttosto catturare un comportamento uniforme che valga ovunque, e che quindi in particolare valga anche per la diagonale.

Sia f tale che

$$f(x, y, z) = \begin{cases} 1 & \text{se } \varphi_x(y) \downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Utilizzo ora s-m-n sulla coppia $< x, y >$: esiste $\varphi_{s(x,y)}(z)$ con lo stesso comportamento. La mia funzione s è quella che cerco (questo è il mio claim).

Supponiamo che $(x, y) \in K_1$. Allora $\forall z \varphi_{s(x,y)}(z) \downarrow$. In particolare converge anche sulla diagonale. In particolare $\varphi_{s(x,y)}(s(x, y)) \downarrow$, e qindi $s(x, y) \in K$. Inoltre $(x, y) \notin K_1 \implies \forall z \varphi_{s(x,y)}(z) \uparrow \implies \varphi_{s(x,y)}(s(x, y)) \uparrow \implies s(x, y) \notin K$.

Come conseguenza abbiamo che tutti gli insiemi completi sono mutuamente riducibili: A e B completi $\implies A \equiv_m B$.

8.3 Insiemi produttivi e creativi

Cerchiamo ora di dare altre caratterizzazioni degli insiemi completi. Vediamo se hanno delle proprietà interessanti. Una proprietà curiosa è la creatività.

Definizione 8.3. Sia $A \subseteq \mathbb{N}$. A è produttivo se esiste f totale calcolabile, detta funzione di produzione, tale che comunque prendo un sottoinsieme $W_i \subseteq A$, si ha che $f(i) \in A \setminus W_i$.

Ricordiamo che W_i è la numerazione dei sottoinsiemi r.e. basata sul dominio di convergenza della funzione i della mia numerazione delle funzioni calcolabili.

Cos'è che produce la funzione di produzione? Uno potrebbe essere interessato a dare una approssimazione r.e. di questi insiemi. Cosa intendiamo? Dare un sottoinsieme di A ricorsivamente enumerabile. f mi produce una dimostrazione, per ogni approssimazione, dell'incompletezza della mia approssimazione in funzione dell'algoritmo con cui enumeriamo i programmi. f è unica, definita a prori dall'insieme A. Questo concetto è stato suggerito dai problemi di incompletezza logica.

C'è una caratterizzazione degli insiemi completi attraverso la produttività.

Possiamo pensare ad A come all'insieme delle formule vere dell'aritmetica. Quando costruiamo un sistema formale per ragionare sull'aritmetica, ovvero un sistema di assiomi fondamentalmente e le regole logiche per passare alle conseguenze logiche degli assiomi, avremo, per il teorema di incompletezza,

che ci saranno sempre formule vere che non sono però dimostrabili nel sistema formale.

È una tecnica simile alla diagonalizzazione.

Un insieme produttivo non è, per sua natura, r.e.; altrimenti potrei approssimarlo con l'insieme stesso.

Definizione 8.4. Un insieme $A \subseteq \mathbb{N}$ è creativo se A è r.e. e $\overline{A}$ è produttivo.

Non tutti gli insiemi produttivi hanno un complementare r.e.

K è un esempio di insieme creativo. Per dimostrarlo ci manca solo da dimostrare che $\overline{K}$ è produttivo. Questo è semplice però perché la funzione di produzione per $\overline{K}$ è la funzione identità.

Sia $W_i \subseteq \overline{K}$. Dobbiamo dimostrare che $i \in \overline{K}$ e $i \notin W_i$.

Per la prima appartenenza andiamo per assurdo. Supponiamo che $i \in K$. Allora $i \in W_i$. Ma essendo W_i sottinsieme di $\overline{K}$ si ha $i \in \overline{K}$, il che è assurdo. Quindi $i \in \overline{K}$.

Per la seconda appartenenza andiamo nuovamente per assurdo. $i \in W_i \implies i \in K \implies \bot \implies i \notin W_i$.

8.4 Relazione tra creatività e completezza

Nota: La seguente è l'ultima dimostrazione richiesta per l'orale.

Da qui in poi verrà usato il simbolo $\leq$ sottintendendo $\leq_m$. Vediamo un risultato importante per gli insiemi produttivi

Teorema 8.1. *Un insieme $A \subseteq \mathbb{N}$ è produttivo sse $\overline{K} \leq A$.*

Questo implica come corollario che A è creativo sse A è r.e. e $K \leq A$. Di conseguenza insiemi creativi ed insiemi completi coincidono.

Dimostrazione. Dimostriamo il primo teorema. Abbiamo i due versi del $\iff$ da dimostrare. Il più semplice dei due è quello inverso: $\overline{K} \leq A \implies A$ è produttivo.

Sia quindi $\overline{K} \leq A$. Sia f la funzione di riducibilità di $\overline{K}$ in A. Sappiamo anche che $\overline{K}$ è produttivo, con funzione di produzione uguale ad *Id*. L'idea è di ritornare a $\overline{K}$ da A attraverso f. Dopodiché, utilizzando la funzione di produzione di $\overline{K}$, troviamo un nuovo elemento che, attraverso alla funzione f, sta in A e non in W_i.

Ricordiamo che la controimmagine di una insieme A attraverso una funzione f è definita come l'insieme $f^{-1}(A) = \{x \mid f(x) \in A\}$).

Consideriamo $f^{-1}(W_i)$, la controimmagine di W_i via f. Abbiamo che sicuramente $f^{-1}(W_i) \subseteq \overline{K}$ e che è r.e. L'indice di $f^{-1}(W_i)$ può essere calcolato in maniera effettiva attraverso una funzione totale calcolabile h. Abbiamo quindi che corrisponde a $W_{h(i)}$. Pensiamo alla funzione $\varphi_i(f(x)) = \varphi_{h(i)}(x)$ per s-m-n. Dove sta $h(i)$? Sta in $\overline{K}$ ma non in $f^{-1}(W_i)$, essendo $\overline{K}$ produttivo con funzione di produzione uguale a *Id*. Passando quindi, attraverso f, al corrispondente in A (ovvero $f(h(i))$) abbiamo che sta in A ma non in W_i. Quindi A è produttivo.

La dimostrazione poteva essere fatta con un altro insieme produttivo. In particolare, se B è produttivo e $B \leq A$ allora A è produttivo.

Passiamo ora al verso opposto: A produttivo $\implies \overline{K} \leq A$.

Sia A produttivo e sia f la funzione di produzione per A. Vado alla ricerca di $W_{s(i)}$ tale che:

$$W_{s(i)} = \begin{cases} \{f(s(i))\} & \text{se } i \in K \\ \emptyset & \text{se } i \in \overline{K} \end{cases}$$

Non sappiamo, per ora, se esiste s; supponiamo però di poterla definire. Per dimostrare la sua esistenza ricorreremo al teorema del punto fisso.

Vogliamo dimostrare che $i \in \overline{K}$ sse $f(s(i)) \in A$. Supponiamo che $i \in \overline{K}$. Allora $W_{s(i)} = \emptyset \subseteq A$. Posso applicare la produttività e quindi concludere che $f(s(i)) \in A$.

Supponiamo che $i \in K$. Allora $W_{s(i)} = \{f(s(i))\}$. Supponiamo per assurdo che $f(s(i)) \in A$. Allora $W_{s(i)} \subseteq A$. Sfruttando la produttività avremmo che $f(s(i)) \in A$ e che $f(s(i)) \notin W_{s(i)}$. Ma questo è assurdo. Perciò $f(s(i)) \notin A$.

La dimostrazione è incompleta, manca la dimostrazione dell'esistenza di s. Ci serve il secondo teorema di ricorsione. Partiamo da $g(i, z, x)$ tale che:

$$g(i, z, x) = \begin{cases} 1 & \text{se } \varphi(i) \downarrow \wedge\, x = f(z) \\ \uparrow & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Per s-m-n esiste $h(i, z)$ tale che $\varphi_{h(i,z)}(x)$ ha lo stesso comportamento. Esiste ora s tale che $\forall i, \varphi_{s(i)} = \varphi_{h(i,s(i))}$. Abbiamo quindi che

$$\varphi_{s(i)}(x) = \varphi_{h(i,s(i))}(x) = g(i, s(i), x) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \wedge\, x = f(s(i)) \\ \uparrow & \text{se } \varphi_i(i) \uparrow \end{cases}$$

□

Dimostrazione. Dimostriamo il corollario. Sia A creativo. Allora A è r.e. e $\overline{A}$ è produttivo. Allora $\overline{K} \leq \overline{A}$. Ma questo implica che $K \leq A$.

Sia A r.e. e completo. Allora $K \leq A$. Allora $\overline{K} \leq \overline{A}$, quindi $\overline{A}$ è produttivo, di conseguenza A è creativo. □

8.5 Insiemi r.e. non completi

Supponiamo di avere un insieme A r.e. Vogliamo dimostrare che A non è ricorsivo. Posso condurre questa dimostazione sempre dimostrando che $K \leq A$? Limitatamente agli insiemi creativi/completi allora sì. Il punto è, esiste un insieme r.e. non creativo? Questo va sotto il nome di problema di Post. È rimasto aperto per un certo numero di anni e veniva considerato un problema difficile. È un analogo del problema $\mathbb{NP}$ vs $\mathbb{P}$. Nella teoria della complessità però si sà molto di meno. È più facile per un insieme r.e. essere completo che il contrario.

Per rispondere a questa domanda abbiamo bisogno di due risultati intermedi.

Teorema 8.2. *Sia A un insieme r.e. infinito. Allora esiste B ricorsivo infinito sottoinsieme di A.*

Dimostrazione. Una caratterizzazione degli insiemi ricorsivi è che sono enumerabili in maniera crescente. Se prendo l'enumerazione f di A e la "taglio" in pezzetti crescenti e li uso per definire la mia enumerazione di B ho che questa è crescente, e quindi B è ricorsivo infinito.

Più precisamente, la mia enumeraziona g di B sarà fatta nel modo seguente, se f è la mia enumerazione di A: $g(0) = f(0)$ e $g(x+1) = f(\mu y.f(y) \geq g(x))$. □

Teorema 8.3. *Sia A un insieme ricorsivo infinito. Allora $\exists B$ r.e. non ricorsivo tale che $B \subseteq A$.*

Dimostrazione. Consideriamo una funzione di enumerazione f per A iniettiva. Consideriamo l'insieme B così definito:

$$B = \{f(i) \mid i \in K\}$$

Abbiamo che B è sottoinsieme di A e che B è r.e. non ricorsivo. Infatti se B fosse decidibile lo sarebbe anche K. In particolare, abbiamo che $i \in K \implies f(i) \in B$. Viceversa essendo f iniettiva non è possibile che un elemento $i \notin K$ venga mappato da f in un elemento $f(i) \in B$. Di conseguenza $i \notin K \implies f(i) \notin B$. □

Se vediamo l'estrazione di sottoinsiemi come un'operazione di focalizzazione possiamo focalizzarci su insiemi di complessità desiderata

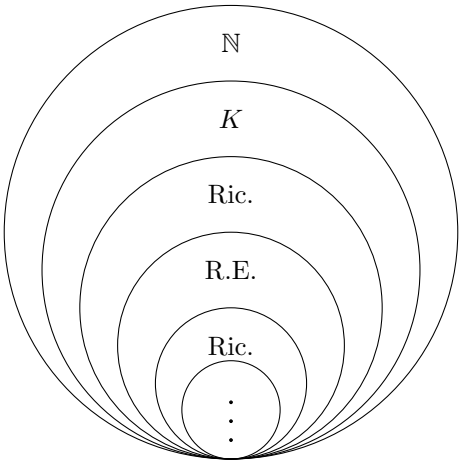


Figura 8.1: Focalizzazione su insiemi

Una proprietà interessante degli insiemi produttivi è che ogni insieme produttivo contiene un sottoinsieme r.e. infinito.

Teorema 8.4. *Sia $A \subseteq \mathbb{N}$ produttivo. Allora $\exists B$ r.e. infinito tale che $B \subseteq A$.*

È una proprietà cruciale che ci serve perché costruiremo un insieme infinito che non contiene alcun sottoinsieme r.e. In pratica abbiamo un insieme tale che qualsiasi suo sosttoinsieme infinito è talmente caotico da non essere nemmeno r.e.

Dimostrazione. Per la dimostrazione useremo la produttività. Proviamo ad approssimare A. Inizieremo con l'insieme vuoto. Per la produttività esiste $a \in A$. Ora nella mia approssimazione includo pure a. Per produttività esiste $a' \in A$ fuori dalla mia approssimazione, e continuo così all'infinito. Questo procedimento è costruttivo e mi crea la mia approssimazione r.e. di A.

Questo procedimento ha un analogo con il tentativo di approssimare l'insieme delle conseguenze di un insieme di assiomi. Per l'incompletezza esiste una formula che non è dimostrabile dal mio sistema formale. A questo punto potrei aggiungere la validità di questa formula al mio sistema formale e crearne uno nuovo. Ma a questo punto ho un nuovo sistema formale, e il ragionamento precedente si applicherebbe identicamente. Quindi non potrò mai avere un insieme che contenga tutte le conseguenze dei miei assiomi.

Vogliamo dimostrare che $W_{h(i,a)} = W_i \cup a$. Più precisamente vogliamo dimostrare che esiste un modo effettivo per calcolare un insieme r.e. ottenibile mediante estensione di W_i con a. $W_{h(i,a)}$ è il dominio di una certa funzione $\varphi_{h(i,a)}$. Vogliamo quindi che $\varphi_{h(i,a)}(x) = g(i,a,x) = \varphi_i(x)|(x = a)$. In pratica voglio che converga se $x = a$ oppure se x fa parte del dominio di φ_i. g è effettivamente calcolabile e ho quindi un metodo effettivo per calcolare $W_{h(i,a)}$ ogni volta.

Possiamo ora costruire la mia sequenza di approssimazione. $W_m = \emptyset$. Sia s la funzione di produzione per A. Ho che $s(m) \in A \setminus W_m$. Passo quindi a $W_{h(m,s(m))=m_1} = \{s(m)\}$. La mia seconda approssimazione sarà $W_{h(m_1,s(m_1))} = \{s(m), s(m_1)\}$.

Se definiamo $next(x) = h(x, s(x))$ possiamo definire $g(x) = next^x(m_0)$ e abbiamo che B è uguale al codominio di g. □

8.5.1 Complessità di Kolmogorov

Se noi vogliamo trasmettere un'informazione, diciamo il numero n, abbiamo due modi per farlo: uno è trasmettere direttamente n, ad esempio con la sequenza di bit che rappresnta n, che ha un costo che dipende da n; un'altra possibilità è trasmettere un modo per costruire n. Ad esempio un algoritmo.

In particolare, immaginiamo un programma i tale che $\varphi_i(0) = n$. Posso trasmettere direttamente i. Qual è più conveniente? Intuitivamente quello più piccolo. Confronto i con n. Se i è più piccolo trasmetto i, altrimenti trasmetto n.

Definizione 8.5. La complessità di Kolmogorov di n, $K(n)$, è il minimo i tale che $\varphi_i(0) = n$ $(= \min\{i \mid \varphi_i(0) = n\})$.

È una astrazione di una problematica reale legata alla comprimibilità delle informazioni.

Il mio confronto è tra n e $K(n)$.

8.5.2 Numeri random

Definizione 8.6. $n \in \mathbb{N}$ è un numero random se $n \leq K(n)$.

Dobbiamo pensare a n come alla sua espansione decimale: una stringa di bit.

Perché usiamo la nozione di *Random* per descrivere questa caratteristica? Perché è una idea del concetto. Se ci fosse una grande regolarità nella stringa per n è chiaro che posso creare un programma piccolino che lo generi. Si parla di dati lawful o lawless. I dati che seguono una regola non saranno random. Se sono random sono talmente disordinati che non riesco a dare nessun metodo generativo più breve dei dati stessi. È un approccio formale alla nozione di random legata alla comprimibilità del dato.

Come l'abbiamo vista la nostra nozione di "casualità" dipende dalla nostra enumerazione dei programmi. Per numeri grandi però non dovrebbe cambiare niente se cambia l'enumerazione.

Un'altra questione è che la nostra nozione di random va bene per sequenze finite. Noi in generale vorremmo idealmente una nozione che valga per sequenze infinite. In questo contesto però ci è sufficiente.

La randomicità di un numero è decidibile? È semidecidibile? Potrei pensare di fare una ricerca limitata ad n per un programma che genera n. Il problema è che le mie funzioni sono parziali calcolabili, potrebbero divergere. Quindi sembrerebbe che la risposta sia no.

Cosa possiamo però semidecidere? Se un numero non è random. Se $\mathcal{R}$ è l'insieme dei numeri random allora abbiamo che $\overline{\mathcal{R}}$ è semidecidibile. La nostra congettura è che $\mathcal{R}$ non sia nemmemno semidecidibile.

Abbiamo un altro modo per dimostrare quanto detto su $\overline{\mathcal{R}}$, oltre al lanciare in parallelo tutti i φ_i e aspettare che uno di essi termini. Lanciamo $\varphi_i(0)$ e vediamo se è uguale a n. Se $\varphi_i(0) = n$ e $i < n$ allora restituisco n, altrimenti divergo. Questa funzione $g(< i, n >)$ è una funzione di numerazione parziale che dà fuori numeri che non sono random. Prima o poi tutti i numeri non random verranno enumerati.

Il nostro claim è che l'insieme dei numeri random non è produttivo.

Vogliamo dimostrare che $\mathcal{R}$ non è produttivo. Lo dimostriamo facendo vedere che $\mathcal{R}$ non contiene nessun sottoinsieme r.e. infinito.

Noi dimostriamo ciò dimostrando che in $\mathcal{R}$ non ci sia nessun sottoinsieme ricorsivo infinito. Grazie ai risultati dimostrati in precedenza è equivalente.

Supponiamo, per assurdo, che A sia ricorsivo e che $A \subseteq \mathcal{R}$. Definiamo $g(i, x) = \mu n, n \in A$ e $n > i$. L'idea è che A è ricorsivo, e quindi possiamo decidere $n \in A$, e possiamo andare alla ricerca del più piccolo n maggiore di i. Questa funzione è calcolabile e per s-m-n abbiamo che possiamo calcolarla con $\varphi_{h(i)}(x)$.

Esitste p tale che $\varphi_p(x) = \varphi_{h(p)}(x) = \mu n, n \in A$ e $n > p$, per il teorema del punto fisso. Ora mi chiedo $\varphi_p(0)$ è random? Dovrebbe essere random, dato

che l'output sta in $A \subseteq \mathcal{R}$. Ma al tempo stesso $\varphi_p(0)$ è generato da p e, per costruzione, $p < \varphi_p(0)$. E quindi sarebbe non random. Assurdo.

Questa parte è legata al paradosso di Berry.

Un insieme che non contiene insiemi r.e. infiniti è detto immune e il suo complementare è detto semplice.

Abbiamo quindi che l'insieme dei numeri random non è produttivo. Di conseguenza l'insieme $\overline{\mathcal{R}}$ è r.e. non ricorsivo e non creativo, ovvero non completo.

Capitolo 9

La gerarchia aritmetica

9.1 Aritmetica e incompletezza

Quando vogliamo parlare di qualcosa abbiamo bisogno di un linguaggio. Partiamo da un insieme di segni.

Il linguaggio dell'aritmetica è il linguaggio del primo ordine basato sulla seguente segnatura:

$$0, S, +, \cdot, =$$

Indichiamo con $\bar{n}$ il termine che rappresenta il numero n. Occhio a non fare confusione tra termine che rappresenta un numero e numero stesso. Ad esempio 3 (il numero) vs $S(S(S(0)))$ (il termine per il numero 3).

Definizione 9.1. $A \subseteq \mathbb{N}^k$ si dice aritmetico se esiste una formula aritmetica $\psi(x_1, \ldots, x_k)$ tale che

$$(n_1, \ldots, n_k) \in A \iff \mathcal{N} \models \psi[\bar{n_1}/x_1, \ldots, \bar{n_k}/x_k]$$

ovvero $\psi(x_1, \ldots, x_k)$ è vera sui numeri naturali. Diremo in questo caso che ψ è una descrizione aritmetica di A.

Ad esempio, l'insieme dei numeri primi è aritmetico, in quanto può essere descritto dalla seguente formula aritmetica:

$$\psi(n) = n \geq 2 \wedge \forall y, 1 < y \wedge y < n \implies \forall z, y \cdot z \neq n$$

Non tutti gli insiemi sono aritmetici. I possibili sottoinsiemi di $\mathbb{N}$ sono più che numerabili, non posso aspettarmi di descrivere tutti questi nell'aritmetica.

Gli insiemi aritmetici sono chiusi rispetto alle operazioni di unione, intersezione e complementazione.

Definizione 9.2. Una funzione f si dice aritmetica se il suo grafo è un insieme aritmetico.

Le seguenti funzioni sono aritmetiche:

- la costante 0, descritta dalla formula $\psi(y) := y = 0$;
- la funzione unaria successore, descritta dalla formula $\psi(x, y) := y = S(x)$;
- la funzione binaria somma, descritta dalla formula $\psi(x_1, x_2, y) := y = x_1 + x_2$;
- la funzione binaria prodotto, descritta dalla formula $\psi(x_1, x_2, y) := y = x_1 \cdot x_2$;
- la proiezione k-aria π_i^k, descritta dalla formula $\psi(x_1, x_2, \ldots, x_k, y) := y = x_k$;

È ragionevole chiedersi se anche le funzioni aritmetiche sono chiuse rispetto ai metodi tipici per comporre le funzioni: definire nuove funzioni a partire da quelle esistenti.

Il primo modo che ci può venire in mente è la composizione di funzioni.

Lemma 9.1. *La composizione di funzioni aritmetiche è aritmetica.*

Dimostrazione. Sia $f(x) = g(h(x))$ e siano $\psi_g(x, y)$ e $\psi_h(x, y)$ le descrizioni aritmetiche di g e h.

Definiamo

$$\psi_f(x, z) = \exists y, \psi_h(x, y) \wedge \psi_g(y, z)$$

Si può dimostrare che ψ_f è una descrizione aritmetica di f. □

Altre operazioni di definizione di nuove funzioni tipiche sono la ricorsione primitiva e la minimizzazione.

Se abbiamo la minimizzazione (che corrisponde circa ad un while) possiamo avere subito la ricorsione primitiva (che corrisponde ad un for).

Lemma 9.2. *Una funzione definita per minimizzazione di una funzione aritmetica è ancora aritmetica.*

Dimostrazione. Sia

$$f(x) = \mu y, (g(x, y) = 0)$$

e supponiamo che ψ_g sia una descrizione aritmetica di g. f è descritta dalla seguente formula:

$$\psi_f(x, y) = \psi_g(x, y, 0) \wedge \forall z, z < y \implies \exists m, (\psi_g(x, z, m) \wedge m \neq 0)$$

□

Possiamo prendere la descrizione delle funzioni come la specifica di un programma. Ci poniamo quindi il problema di quali funzioni sono specificabili.

Teorema 9.1. *Tutte le funzioni calcolabili sono aritmetiche.*

Questo è conseguenza del fatto che ogni funzione calcolabile può essere espressa in un formalismo che contiene somma, prodotto, costanti, proiezioni ed è chiuso per composizione e minimizzazione.

Mi serve anche l'uguaglianza nel mio formalismo.

Teorema 9.2. *Ogni insieme ricorsivamente enumerabile è aritmetico.*

Dimostrazione. Se A è r.e. esiste f parziale calcolabile tale che $A = dom(f)$. Sia $\psi_f(x, y)$ una descrizione aritmetica di f. Allora:

$$n \in A \iff \mathcal{N} \models \exists y, \psi(\bar{n}, y)$$

e dunque $\psi_A(x) = \exists y, \psi(x, y)$ è una descrizione aritmetica di A. □

Teorema 9.3. *L'insieme delle formule aritmetiche vere non è ricorsivamente enumerabile.*

Dimostrazione. Sia $\{\psi_n\}_{n\in\mathbb{N}}$ una enumerazione effettiva delle formule aritmetiche in una variabile. Consideriamo l'insieme A così definito

$$n \in A \iff \mathcal{N} \models \neg\psi_n(\bar{n})$$

Se la verità aritmetica fosse semidecidibile allora A sarebbe r.e. Dunque A dovrebbe essere aritmetico e dovrebbe esistere una formula ψ_a per cui

$$n \in A \iff \mathcal{N} \models \neg\psi_a(\bar{n})$$

Ma per $n = a$ otteniamo una contraddizione. □

Teorema 9.4. *Ogni sistema formale aritmetico, se consistente, è incompleto (i.e. esistono formule aritmetiche valide ma non dimostrabili).*

Un sistema formale è definito da un insieme ricorsivo di assiomi e un insieme di regole di inferenza che permettono di dedurre nuovi teoremi a partire dagli assiomi in un numero finito di applicazioni.

Pertanto le formule aritmetiche dimostrabili costituiscono un insieme ricorsivamente enumerabile.

Poichè le formule vere non sono r.e. esistono necessariamente delle formule vere ma non dimostrabili.

È difficile descrivere in maniera categorica una teoria. Esistono poche teorie per cui non sia possibile cambiare il modello e ribaltarla completamente.

9.2 La gerarchia aritmetica

L'uguaglianza tra numeri naturali è decidibile. Questo non è banale, ad esempio per i numeri reali non è nemmeno semidecidibile l'uguaglianza; per essi è semidecidibile la disuguaglianza.

Data una formula aritmetica possiamo spostare tutti quantificatori all'inizio della formula. La mia formula acquista quindi la forma $Q_1x_1Q_2x_2Q_3x_3 \dots Q_nx_nP$,

con P senza quantificatori e con P decidibile. Se abbiamo dei quantificatori uniformi che si susseguono possiamo collassarli. Ad esempio $\forall x_1 \forall x_2 \ldots$ diventa $\forall < x_1, x_2 > \ldots$. Nella formula si parlerà poi di proiezioni della codifica al posto delle singole istanze delle variabili collassate. Ad esempio $fst(x)$ invece che x_1, $\pi_n(x)$ invece che x_n.

Nel collasso c'è un cambio di notazione. Il $\forall$ diventa Π_n, con n uguale al numero di inversioni $\forall$ $\exists$. Lo stesso discorso vale per $\exists$ e Σ_n. Ad esempio, $\forall\exists\forall$ è di classe Π_3, e $\exists\forall$ è di classe Σ_2. Esiste poi $\Delta_n = \Pi_n \cap \Sigma_n$.

Quali sono le formule Σ_1? Quelle semidecidibili. Perché? Perché sono proiezione esistenziale di precidati decidibili.

Se $A \in \Sigma_n$ allora $\overline{A} \in \Pi_n$. Π_1 corrisponde alla classe degli insiemi co-r.e. Come conseguenza ho che Δ_1 è la classe degli insiemi ricorsivi.

La gerarchia degli insiemi è interessante perché dà una classificazione degli insiemi per complessità computazionale.

Il ragionamento vale per tutti i tipi di proprietà, non solo quelle estensionali. In generale si vuole dare una descrizione aritmetica di un programma. Da questa poi sarà possibile fare dei ragionamenti per capire "quanto sia difficile". Ad esempio, sia $P(i) =$ "W_i è finito". La sua descrizione aritmetica sarebbe $\exists x \forall y \forall t \forall m, y \geq x \implies \neg T(i, y, m, t)$. Ho che $P(i) \in \Sigma_2$. Mi aspetto che non sia ne r.e. ne co-r.e., e in effetti non lo è. Va comunque ricordato che l'appartenenza ad una classe non è mutalmente esclusiva. Potrei avere proprietà Σ_2 che appartengono a Σ_1. Questo di solito accade quando i quantificatori sono bound, che è la stessa situazione che avremmo se non esistessero.

In Δ_n stanno insiemi descrivibili sia con una formula in Π_n che con una formula in Σ_n.

Capitolo 10

Esercizi svolti in classe

10.1 Compito Parziale Gennaio 2018

10.1.1 Esercizio 1

Data una funzione $f : \mathbb{N} \to \mathbb{N}$ e un insieme $A \subseteq N$, la controimmagine di A via f è l'insieme

$$f^{-1}(A) = \{x \mid f(x) \in A\}$$

(a) dare un esempio di una funzione parziale calcolabile f e di un insieme ricorsivo A tale che $f^{-1}(A)$ è r.e. ma non ricorsivo

(b) dimostrare che per ogni funzione parziale calcolabile f, se A è r.e. allora anche $f^{-1}(A)$ è r.e

Soluzione

(a) Basta prendere un semidecisore per un insieme r.e. non ricorsivo e $\mathbb{N}$.

(b) Possiamo definire $s_{f^{-1}(A)}$ a partire da s_A:

$$s_{f^{-1}(A)}(x) = s_A(f(x))$$

Approfondimento

Cosa posso dire della controimmagine di un insieme A attraverso una funzione f?

f \ A	Ricorsivo	R.E.
totale	$f^{-1}(A)$ è Ricorsivo	$f^{-1}(A)$ è r.e.
parziale	$f^{-1}(A)$ è r.e.	$f^{-1}(A)$ è r.e.

Questo vale in generale. Per alcuni casi particolari potrei avere situazioni più particolari. Ad esempio funzione parziale e A ricorsivo e controimmagine ricorsiva.

E dell'immagine cosa possiamo dire? L'immagine di un insieme r.e. attraverso f rimane r.e. (un'ovvietà). E se f è totale e A è ricorsivo? Non posso dire con certezza che l'immagine sarà ricorsiva. Se ad esempio prendo la funzione di enumerazione di un insieme r.e. ottengo un'immagine r.e. a partire da un insieme ricorsivo ($\mathbb{N}$).

10.1.2 Esercizio 2

È possibile enumerare ogni insieme r.e. infinito mediante una funzione di enumerazione crescente? Motivare adeguatamente la risposta.

Soluzione

La funzione di enumerazione di un insieme r.e. può sempre essere iniettiva (*i.e.*posso enumerare senza ripetizioni), ma non può essere mai crescente. Questo perchè se posso enumerare in maniera crescente un insieme A ho che A è ricorsivo. Dato che esistono insiemi r.e. non ricorsivi ho che non posso enumerare tutti gli insiemi r.e. mediante una funzione crescente.

10.1.3 Esercizio 3

Data una funzione di enumerazione f totale e calcolabile, la funzione $count_f(n)$ conta quante volte n compare nella enumerazione, fino ad un massimo di 100 (se n compare più di 100 volte, l'output è 100):

$$count_f(n) = \min(100, |\{x \mid f(x) = n\}|)$$

È possibile calcolare $count_f$?

Soluzione

La specifica di $count_f$ è totale, vediamo se esiste un programma che la rispetti. Intuitivamente sembrerebbe di no.

Posso scegliere la mia f come voglio per mostrare un controesempio nella mia dimostrazione che $count_f$ non è calcolabile.

Scegliamo per f la funzione di enumerazione di K. Abbiamo che

$$count_{e_K}(n) = 0 \iff n \notin K.$$

Quindi se $count_f$ potrei risolvere la terminazione diagonale. Ma questo è assurdo, da cui la non calcolabilità di $count_f$.

10.1.4 Esercizio 4

È possibile calcolare il più piccolo input x su cui un programma da un output maggiore o uguale di x?

Soluzione

Se φ_i fosse totale non avrei problemi, visiterei progressivamente i miei input finché non trovo il più piccolo per cui vale quella proprietà.

La congettura è che g non sia calcolabile. Costruiamo una funzione $\varphi_{h(i)}$ tale che g non sia calcolabile per $\varphi_{h(i)}$.

Sia $f(i,x)$ tale che:

$$f(i,x) = \begin{cases} 0 & \text{se } x = 0 \text{ e } \varphi_i(i) \downarrow \\ 1 & \text{se } x \geq 1 \end{cases}$$

Per s-m-n ho h che mi curryfica la mia funzione. Ora, cosa posso dire per $g(h(i))$?

$$g(h(i)) = \begin{cases} 0 & \text{se } \varphi_i(i) \downarrow \\ 1 & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Da cui la non calcolabilità di g.

10.1.5 Esercizio 5

Classificare il seguente insieme:

$$A = \{i \mid \varphi_i \text{ è una funzione (parziale) periodica}\}$$

Soluzione

In questo tipo di esercizi posso usare i meta-teoremi di Rice e Rice-Shapiro.

Abbiamo che A è compatto ma non monotono. Infatti se una funzione phi_i ha un periodo T per un certo input k su cui diverge possiamo restingere la funzione alla funzione sempre divergente che ha un periodo T su qualsiasi input. Per quanto riguarda la monotonia abbiamo che la funzione sempre divegente, che è periodica, può essere estesa ad una funzione non periodica.

Approfondimento

Il seguente insieme è compatto, monotono non r.e.:

$$A = \{i \mid W_i \cap \overline{K} \neq \emptyset\} = \{i \mid dom(\varphi_i) \cap \overline{K} \neq \emptyset\} = \{i \mid \exists x, \varphi_i(x) \downarrow \wedge \varphi_x(x) \uparrow\}$$

Se estendo una funzione φ_i, con $i \in A$, avrò sempre intersezione non vuota, quindi vale la monotonia. Se l'intersezione è diversa dal vuoto ho che, per ogni funzione φ_i con $i \in A$, esiste un x che appartiene all'intersezione. Posso restringermi a quell'x e ottengo una funzione il cui indice sta ancora in A, e quindi vale la compattezza.

A non è r.e. Dato un numero x esiste sempre una funzione totale calcolabile tale che $W_{h(x)} = \{x\}$ (si costruisce con s-m-n). Ora, $h(x) \in A \iff W_{h(x)} = \{x\} \cap \overline{K} \neq \emptyset \iff x \in \overline{K}$.

In generale per dimostrae che un insieme A non è r.e. posso provare a dimostrare che $\overline{K} \leq A$. Per mostrare che A non è ricorsivo cerco in genere una riduzione da K ad A ($K \leq A$). Va tuttavia ricordato che non tutti gli insieri r.e. sono completi, quindi questa seconda procedura non vale per tutti gli insiemi r.e. non ricorsivi.

10.2 Altri esercizi

10.2.1 Esecuzione parallela

Vediamo una definizione del parallelo, ovvero a dare una definizione più precisa di cosa significa lanciare in parallelo due programmi.

Definiamo:

$$(f||g)(x) = \begin{cases} \uparrow & \text{se } f(x) \uparrow \text{ e } g(x) \uparrow \\ f(x) & \text{se } f(x) \downarrow \text{ e } g(x) \uparrow \\ g(x) & \text{se } f(x) \uparrow \text{ e } g(x) \downarrow \\ \max\{f(x), g(x)\} & \text{se } f(x) \downarrow \text{ e } g(x) \downarrow \end{cases}$$

Questa funzione è calcolabile? Intuitivamente no: supponiamo che lanciando il nostro programma f termini. Non possiamo determinare se g divergerà, quindi non saremo mai certi di cosa restituire.

Più precisamente, scegliamo per f il semidecisore di K e per g la funzione costante $\mathbf{0}$. Abbiamo che

$$(f||g)(x) = \begin{cases} 1 & \text{se } \varphi_x(x) \downarrow \\ 0 & \text{se } \varphi_x(x) \uparrow \end{cases}$$

Quindi la funzione non è calcolabile.

10.2.2 Estensione totale di funzioni parziali

Sarebbe bello se, dato un programma qualsiasi, ne esistesse una estensione totale. In questo modo potrei lavorare sempre con l'estensione e avrei una funzione totale.

La domanda è, può una data funzione φ_i essere estesa a una funzione totale calcolabile? La risposta è in generale no. Esistono funzioni parziali per cui non può esistere una estensione totale.

La funzione che consideriamo, tra le tante, è $f(x) = \varphi_x(x) + 1$. Sia $\hat{f}$ una estensione totale di f con indice m (φ_m). Abbiamo che $\varphi_m(m) \downarrow$, essendo φ_m totale, ed $f(m)$ deve di conseguenza convergere, per sua definizione. Ora, $\hat{f}(m)$ dovrebbe evere lo stesso valore di $f(m)$, essendo una sua estensione. Ma allora avremmo $\varphi_m(m) = f(m) = \varphi_m(m) + 1$, il che è assurdo.

Consideriamo l'insieme $A = \{i \mid \varphi_i \text{ è estendibile}\}$. Come lo classifichiamo? Essendo una proprietà estensionale possiamo applicare i nostri meta-teoremi. Abbiamo inoltre, grazie al risultato precedente, che esistono funzioni non estendibili (il che non è banale). Ora immediatamente per Rice so che A non è ricorsivo. È sicuramente compatto, dato che $f_\emptyset$ è estendibile banalmente. Non è invece monotono, dato che se estendo la funzione $f_\emptyset$ con la funzione $f(x) = \varphi_x(x) + 1$ ho che f non è più estendibile per il risultato precedente.

10.2.3 Classificazione di un insieme non estensionale

Supponiamo di voler classificare l'insieme $A = \{i \mid \varphi_i(i) = i\}$. Non essendo la proprietà caratterizzante estensionale non posso applicare Rice o Rice-Shapiro. Ho che la proprietà è sicuremante semidecidibile, dato che il semidecisore è il programma che, lanciando φ_i su input i, mi dica 1 se ho output i, 0 altrimenti e che diverge se $\varphi_i(i)$ diverge.

È ricorsivo? Se abbiamo il sospetto che non lo sia possiamo provare a fare una riduzione da K ad A.

Consideriamo la funzione binaria $g(i, x)$:

$$g(i,x) = \begin{cases} x & \text{se } \varphi_i(i) \downarrow \\ \uparrow & \text{altrimenti} \end{cases}$$

Per s-m-n esiste h totale calcolabile tale che $\varphi_{h(i)}(x) = g(i, x)$. Ora, $h(i)$ è una buona funzione di riduzione?

Abbiamo che:

- $i \in K \implies \varphi_{h(i)}(h(i)) = h(i) \implies h(i) \in A$
- $i \notin K \implies \varphi_{h(i)}$ diverge sempre, in particolare $\varphi_{h(i)}(h(i)) \neq h(i)$, essendo h totale $\implies h(i) \notin K$

Abbiamo quindi che A non è ricorsivo.

10.2.4 Differenza tra un insieme r.e. ed un insieme finito

Sia A r.e. non ricorsivo. Sia B finito. Consideriamo $A \cup B$. Vogliamo dimostrare che è r.e. ma non ricorsivo.

L'ipotesi di finitezza di B è fondamentale. Se avessi avuto solo B ricorsivo avrei potuto prendere $\mathbb{N}$ e ottenere un insieme ricorsivo.

Supponiamo di avere il decisore per $A \cup B$, $c_{A \cup B}(n)$. Quanto è diversa questa funzione dal semidecisore di A? È diversa per un sottoinsieme di B, ovvero un numero finito di punti.

Consideriamo i punti in $B \setminus A = \{b_1, \ldots, b_k\}$. Posso ora costruire un decisore per A nel seguente modo:

```
def c_A(x):
    if x = b_1:
```

```
    return 0
else if x = b_2:
    return 0
⋮
else:
    return c_{A∪B}(x)
```

Ma quindi se $A \cup B$ fosse decidibile anche A lo sarebbe. Ma questo è assurdo.

Non è un problema la non calcolabilità di $B \setminus A$, l'importante è l'esistenza di questi punti.

10.3 Remark finale sulla calcolabilità di certe funzioni

Un conto è quando non esiste un programma, un conto è quando un programma esiste ma non ho modo di calcolarlo a partire da certe informazioni. Nel primo caso sto trattando un problema non calcolabile, come la terminazione generale, nel secondo caso il problema è calcolabile ma non è calcolabile l'indice del programma che mi calcola il mio problema sulla base di certe informazioni.

Ad esempio, prendiamo la seguente famiglia di funzioni:

$$g(x) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \\ 0 & \text{se } \varphi_i(i) \uparrow \end{cases}$$

Qui i è una variabile libera. Fissato un i la funzione g è calcolabile. Se ho che $\varphi_i(i) \downarrow$ allora g corrisponde alla costante $\mathbf{1}$. Altrimenti corrisponde alla costante $\mathbf{0}$. Queste sono due funzioni calcolabili. Quello che non posso calcolare sono gli indici i per cui g corrisponde a una a all'altra funzione.

In particolare non posso calcolare:

$$g(i, x) = \begin{cases} 1 & \text{se } \varphi_i(i) \downarrow \\ 0 & \text{se } \varphi_i(i) \uparrow \end{cases}$$

analogo della funzione precedente ma in cui i è legata.

Parte II

Complessità

Capitolo 11

Introduzione

11.1 Complessità, costi, analisi

In questa parte di corso siamo interessati a definire le classi di complessità. Ci sono alcune differenze rispetto alla complessità intesa in senso algoritmo.

Siamo interessati a cosa siamo in grado di calcolare in tempi ragionevoli. Il fatto che un problema sia calcolabile non implica necessariamente che si tratti di un problema risolvibile in modo pratico.

Come si valuta la complessità computazionale, ovvero il costo di far funzionare dei programmi? Tipicamente si misura il costo in funzione del consumo di una determinata risorsa. Abbiamo fondamentalmente due risorse principali di riferimento nella computazione: tempo e spazio.

Perché non usiamo altre risorse, come ad esempio il consumo di energia elettrica o la temperatura della CPU? Potremmo. Ciò pone una questione interessante: che relazione c'è tra queste risorse? Ad esempio, sapendo quanto tempo richiede un programma per essere eseguito sappiamo qualcosa sull'occupazione di memoria?

Come misuriamo il costo computazionale? Tipicamente si è interessati al comportamento asintotico di un programma, al crescere della dimensione dell'input. Si protrebbe anche fare un'analisi puntuale. Di solito questa è il punto di partenza dell'analisi del costo. Si può però fare una considerazione diversa e chiedersi quanto scala il programma al crescere della dimensione dei dati di input.

Si possono ovviamente avere delle fluttuazioni nei dati e quindi nel comportamento del programma. Noi consideriamo di solito il caso pessimo. Questo è utile perché mi dà un limite al peggio che può succedere. Potrebbe a volte essere interessante fare un'analisi del caso medio, che però è in genere molto più complessa della corrispettiva per il caso pessimo. Ad esempio è necessario conoscere la distribuzione dei dati di input, il che non è sempre semplice.

Può succedere che un programma abbia un comportamento "pesante" all'inizio e leggero al crescere dell'input. La nostra analisi ignora il primo aspetto.

11.2 Modelli di calcolo

Un'altra parte importante riguarda il meccanismo di calcolo.

Ci chiediamo anche quanto la misura del tempo e dello spazio dipenda dal particolare modello computazionale che usiamo.

Noi faremo analisi e prescidere dalle costanti. $1000 \cdot n^2$ e $10 \cdot n^2$ sono equivalenti per noi. È un tentativo di dissociarsi dal particolare modello di calcolo usato. Ma funziona? O la nostra misura di complessità rimane vincolata ad un particolare modello di calcolo?

Ad esempio, abbiamo programmi con complessità diverse a seconda dell'architettura (e.g. n^2 vs n^6)? Non è nemmeno scontato che il processo di compilazione mantenga la complessità del programma scritto in codice sorgente, ad esempio per considerazioni legate all'implementazione del linguaggio.

Noi faremo riferimento a macchine teoriche (Macchine di Turing). Ci chiederemo: data una macchina con n nastri riusciamo ad eseguire uno stesso algoritmo con la stessa complessità che avremmo con una macchina con 1 nastro? La risposta sarà no. Questa è una forte differenza rispetto alla teoria della complessità, dove il numero di nastri non ha alcuna influenza sulla calcolabilità di una funzione.

Consideremo quanto sia significativo dire che un certo problema ha una data complessità.

Tra i modelli che considereremo ci sarà quello non deterministico, che è un pò strano. Dovremo capire perché siamo interessati a questa nozione. Il motivo principale è che ci interessa la classe $\mathbb{NP}$, che contiene tanti problemi con una complessità computazionale elevata con gli algoritmi odierni e con delle buone euristiche. Non si è ancora dimostrato che non esista un modo di risolvere questi problemi in tempo polinomiale con una macchina deterministica. Cercheremo di capire perché è così complicato da dimostrare.

Purtroppo in complessità siamo dipendenti dal modello di calcolo (in particolare nella definizione del costo). È importante capire quanto è forte questa dipendenza.

Siamo interessati inoltre a trasformazioni che ci portino da un modello di calcolo ad un altro. Siamo interessati anche a capire quanto ci costa simulare, in modo deterministico, modelli di calcolo non deterministici.

11.3 La classe $\mathbb{NP}$

Perché $\mathbb{NP}$ è così interessante? Perché è la classe di problemi per i quali non sappiamo se abbiamo degli algoritmi efficienti per risolverli ma per i quali abbiamo metodi di verifica di una soluzione efficienti. Non è scontato che esista un modo di verificare una soluzione in maniera efficiente per un dato problema.

Prendiamo ad esempio il problema della soddisfacibilità proposizionale. Come tanti dei problemi che vedremo è un problema decisionale (risposta sì/no). Il nostro programma deve dirmi sì se una data formula è soddisfacibile e no altrimenti. È possibile avere un certificato che "giustifichi" la risposta. Per SAT

questo certificato può essere l'assegnazione di valori alle variabili che soddisfa la proposizione. In tempo lineare posso fare la sostituzione e verificare se la formula ottenuta è valida.

Il problema della tautologicità non è di questa categoria. Si può dare un certificato compatto per questo problema? Si direbbe di no, ma non si è ancora dimostrato il contrario.

Il certificato per essere compatto deve avere dimensione polinomiale.

Che un problema sia in $\mathbb{NP}$ è utile da sapere per decidere come verificare in maniera efficiente che una soluzione sia valida.

I problemi in $\mathbb{NP}$ sono spesso detti intrattabili. Questa è forse un'esagerazione. Si tratta di problemi comuni, per i i quali si usano tante euristiche e tecniche per ottenere delle soluzioni parziali che vanno già abbastanza bene. C'è molto di peggio. Perfino in $\mathbb{P}$ ci sono problemi "intrattabili" (ad esempio con complessità superiore al n^3) molto peggiori dei più famosi problemi in $\mathbb{NP}$.

Ci sono inclusioni che sono congetturate che tuttavia non sono facili da dimostrare. La più famosa è la relazione tra $\mathbb{P}$ e $\mathbb{NP}$. Pare che manchi ancora la tecnica matematica corretta per affrontare queste problematiche; la scienza è ancora incompleta.

Molti problemi con algoritmi di complessità esponenziale nel caso pessimo fanno anche parte di $\mathbb{NP}$. Quantomeno ci è agevole verificare che una soluzione sia valida.

11.4 Dimensione dei dati di input

Noi misuriamo la complessità computazionale in funzione della dimensione dei dati di input. Questo perché di solito accettiamo stringhe di bit che rappresentano l'input in modo compatto. Con un alfabeto almeno binario abbiamo almeno una rappresentazione logaritmica dei dati in input. Ovvero, la rappresentazione di un numero n ha lunghezza che è di ordine $O(\log(n))$.

Tutte le volte che abbiamo algoritmi che lavorano con numeri la dimensione dell'input è logaritmica. Se l'input è n e la complessità è lineare in n la complessità dell'algoritmo non è lineare, bensì esponenziale nella dimensione dell'input.

Qui c'è una leggera differenza tra la teoria della complessità e la teoria algoritmica della complessità. In algoritmica si considera cosatante il costo delle operazioni artimetiche. Ciò non è necessariamente sbagliato, dato che si considerano interi con una dimensione massima (limitata dalla grandezza della parola di memoria). Fintanto che esiste un bound alla dimensione dei dati tutte le operazioni hanno costo costante. Se facessimo operazioni su interi di grandezza arbitraria queste assunzioni non varrebbero più, e il costo dipenderebbe dall'implementazione usata, e sicuramente non sarebbe più costante.

Noi siamo interessati a complessità asintotiche, non possiamo assumere che i nostri interi abbiano dimensione fissata. Per noi possono avere dimensione arbitraria.

11.5 Notazioni d'ordine

Rispetto alle funzioni $f : \mathbb{N} \to \mathbb{N}$, che utilizziamo per misurare il costo di un'operazione, possiamo fare una suddivisione in classi, dette **notazioni d'ordine**.

Le notazioni d'ordine sono insiemi di funzioni. La classe più importante, a cui faremo riferimento, è la $O(f)$.

Definizione 11.1. Sia $f : \mathbb{N} \to \mathbb{N}$. Definiamo le seguenti notazioni d'ordine:

1. $O(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \exists c \forall n, g(n) \leq cf(n) + c\}$; è la classe delle funzioni che crescono al più come f;
2. $o(f) = \{g : \mathbb{N} \to \mathbb{N} \mid \forall c \exists n_0 \forall n \geq n_0, cg(n) + c \leq f(n)\}$; è la classe delle funzioni che crescono meno rapidamente di f;
3. $\Omega(f) = \{g : \mathbb{N} \to \mathbb{N} \mid f \in O(g)\}$; è la classe delle funzioni che crescono almeno quanto f;
4. $\Theta(f) = O(f) \cap \Omega(f)$; è la classe delle funzioni che crescono come f;

Abbiamo che valgono le seguenti proposizioni:

- $\forall c > 0, O(cf) = O(f)$
- se $f_1 \in O(g_1)$ e $f_2 \in O(g_2)$ allora $f_1 + f_2 = O(g_1 + g_2)$
- se $f_1 \in O(g_1)$ e $f_2 \in O(g_2)$ allora $f_1 \cdot f_2 = O(g_1 \cdot g_2)$

La definizione d'ordine è indipendente dalle costanti. Questo è importante perché vogliamo renderci indipendenti dalle unità di misura. Se cambia l'unità di misura non vogliamo che cambi anche l'ordine di complessità. È analogo al rendersi indipendenti dalle prestazioni del processore rispetto alla complessità.

Sia $g(n) > 0$ per ogni n. Valgono le seguenti proposizioni:

- se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \neq 0$, allora $f \in O(g)$ e $g \in O(f)$
- se $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ o $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$, allora $f \in O(g)$ e $g \notin O(f)$
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ se e solo se $f \in o(g)$

$f \in o(g)$ implica $f \in O(g)$, ma non vale il viceversa.

Scrivere $g \in O(f)$ è equivalente a scrivfere $O(g) \subseteq O(f)$.

Se il limite all'infinito del rapporto tra f e g è uguale ad una costante diversa da 0 abbiamo che f e g hanno lo stesso comportamento asintotico.

Quando cerchiamo l'ordine di grandezza cerchiamo la funzione più semplice che mi indichi il comportamento asintotico della mia funzione. Un polinomio di quarto grado è sicuramente $O(n^5)$, ma l'ordine più semplice che useremmo è $O(n^4)$.

Abbiamo che:

- per ogni costante c, $n^c \in O(c^n)$, ma $c^n \notin O(n^c)$: l'esponenziale cresce più velocemente di qualsiasi polinomio;
- per ogni a, b, $\log_a(n) \in O(\log_b(n))$;
- per ogni a, b, se $a < b$, $b^n \notin O(a^n)$;
- $O(n \log(n)) = O(\log(n!))$ e $O(n^n) = O(n!)$

Per le complessità logaritmiche la base è ininfluente. Per le complessità esponenziali invece la base è influente.

Le ultime uguaglianze sono vere in luce delle disuguaglianze di Stirling:

$$e\left(\frac{n}{e}\right)^n \leq n! \leq en\left(\frac{n}{e}\right)^n$$

Nella tabella 11.1 è possibile osservare le complessità di alcuni algoritmi noti:

Ordine	**Nome**	**Esempio**
$O(1)$	costante	operazioni su strutture dati finite
$O(\log n)$	logaritmico	ricerca di un elemento in un array ordinato o in un albero bilanciato
$O(n)$	lineare	ricerca di un elemento in array disordinati/-liste; somma di due interi con la tecnica del riporto
$O(n \log n)$	quasi-lineare	Fast Fourier transform; merge-sort, quicksort (caso medio)
$O(n^2)$	quadratico	prodotto di due interi; bubble sort e insertion sort
$O(n^c)$, per $c > 1$	polinomiale	parsing di grammatiche contestuali; simplesso (caso medio)
$O(c^n)$, per $c > 1$	esponenziale	soluzione del problema del commesso viaggiatore con tecniche di programmazione dinamica; costruire la tabella di verità di una proposizione
$O(n!)$	fattoriale	soluzione del problema del commesso viaggiatore mediante ricerca esaustiva

Tabella 11.1: Complessità di alcuni algoritmi noti

La tabella 11.1 considera complessità in tempo.

11.5.1 Complessità sublineari

Ha senso di parlare di complessità in tempo sublineari? Si potrebbe rispondere istintivamente di sì, portando due esempi famosi (binary search e alberi bilanciati). Questo però ha senso se supponiamo che la struttura dati che stiamo usando

è già data e non dobbiamo rileggerla ogni volta. Se dovessimo leggere ogni volta una struttura dati di grandezza n allora la complessità sarà quantomeno lineare nella dimensione dell'input, poichè l'input va letto.

Si suppone spesso che le complessità in tempo sublineari non abbiano molto senso. Lo acquistano solo se facciamo significative assunzioni sul problema in questione.

11.6 Grafi

Il grafo è una struttura dati molto ricca in informatica. Ha inoltre una definizione matematica molto precisa.

Definizione 11.2. Un grafo finito è una coppia (V, E):

1. V è un insieme finito di vertici;
2. $E \subseteq V \times V$ è una relazione che definisce l'insieme degli archi.

Un grafo $G = (V, E)$ è **non orientato** quando la relazione E è simmetrica e non riflessiva.

Noi considereremo sempre grafi finiti (V e E finiti).

Definizione 11.3. Sia $G = (V, E)$ un grafo.

- due vertici $u, v \in V$ sono adiacenti se esiste un arco $(u, v) \in E$;
- un cammino è una sequenza di coppie di vertici dove per ciascuna coppia consecutiva i due vertici sono adiacenti;
- un cammino è semplice se tutti i vertici sono distinti;
- un ciclo è un cammino dove il primo e l'ultimo vertice coincidono e dove non ci sono ulteriori ripetizioni di vertici

11.6.1 Problemi tipici sui grafi

Definizione 11.4. Sia $G = (V, E)$ un grafo.

- Un cammino (ciclo) Hamiltoniano in G è un cammingo (ciclo) che comprende ciascun vertice del grafo una sola volta;
- Un ricoprimento di vertici per G è un sottoinsieme $V_0 \subseteq V$ tale che ogni arco $e \in E$ ha almeno un'estremità in V;
- G è n-colorabile se esiste una funzione di colorazione $col : V \to c_1, \ldots, c_n$ tale che vertici adiacenti hanno colori diversi, ovvero

$$(u, v) \in E \implies col(u) \neq col(v)$$

- G è completo se ogni coppia di nodi distinti è connessa da un arco

- Una cricca di G è un suo sottografo completo $G' = (V', E')$, ovvero tale che $V' \subseteq V$ e $E' = V' \times V' \subseteq E$
- Un insieme indipendente in G è un sottoinsieme di vertici $V' \subseteq V$ tale che per ogni coppia di vertici $u, v \in V' \implies (u, v) \notin E$.

Un grafo $G = (V, E)$ ammette sempre dei ricoprimenti (V stesso ad esempio). Inoltre se R è un ricoprimento di G ogni suo sovrainsieme lo è. Quello a cui siamo interessati in genere è un ricoprimento minimo. Non è detto che un ricoprimento di una certa dimensione sia unico.

Un grafo ammette sempre anche degli insiemi indipendenti (ad esempio l'insieme vuoto o il singoletto $\{v\}$, se $v \in V$). Inoltre se I è un insieme indipendente di G ogni suo sottoinsieme lo è. Siamo interessati a insiemi indipendenti significativi, ovvero massimi.

Il complementare di un ricoprimento è sempre un insieme indipendente, e viceversa. In particolare il complementare di un ricoprimento minimo è un insieme indipendente massimo. Cercare uno è equivalente a cercare l'altro.

Ogni grafo ammette delle cricche (ad esempio la cricca di dimensione 1 o 2). Inoltre se C è una cricca di G allora ogni suo sottoinsieme lo è. Siamo ancora una volta interessati a cricche di dimensione massima.

Questi sono tutti problemi interessati e tipicamente $\mathbb{NP}$ completi.

Il problema della colorabilità cambia complessità in base al valore di n. Ad esempio, la 2-colorabilità e un problema in $\mathbb{P}$, la 3-colorabilità e un problema $\mathbb{NP}$ completo. È una tecnica comune quella di restringere un problema per diminuire la complessità. A volte si riesce a ridursi ad un problema che ha algoritmi polinomiali (per casi particolari).

11.6.2 Rappresentazione di un grafo

È importante fare delle considerazioni sulla rappresentazione che facciamo dei grafi. Questa infatti influenza l'analisi della complessità che andremo a fare.

Si rappresenta solitamente un grafo mediante matrice di adiacenza.

Definizione 11.5. La matrice di adiacenza M_G di un grafo $G = (V, E)$ è la matrice definita nel modo seguente:

$$M_G(u, v) = 1 \iff (u, v) \in E$$

Se non abbiamo ipotesi sul numero di archi in un grafo possiamo supporre che il grafo abbia $O(n^2)$ archi. In questo caso la rappresentazione con matrice è conveniente.

Se il grafo è sparso sono più convenienti altre rappresentazioni.

La complessità di solito è data in funzione di V e di E.

11.6.3 Raggiungibilità

Un algoritmo molto importante sui grafi è quello della raggiungibilità, ovvero determinare se esiste un cammino tra due nodi del grafo. Si può fare mediante una visita.

Una visita in generale si svolge nel seguente modo:

1. si partiziona il grafo in tre parti:
   (a) nodi già processati (nodi Done)
   (b) frontiera corrente (nodi Current)
   (c) nodi ancora da visitare (nodi Todo)
2. si prende un nodo da cardine e lo si processa (nel caso della raggiungibilità vediamo se siamo arrivati in fondo).
3. per ogni nodo adiacente al nodo cardine, in base al suo tipo, si svolge una delle seguenti azioni:
   (a) se il nodo è Done lo si ignora
   (b) se il nodo è Todo si estende Current con questo
4. si estende Done con il nodo cardine
5. si riparte da 2 fino a che l'insieme Current non risulti vuoto

Una rappresentazione grafica del processo è data dalla figura 11.1.

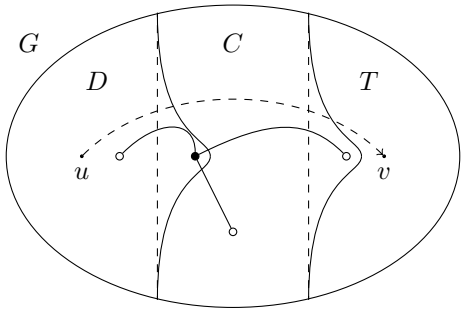


Figura 11.1: Rappresentazione grafica dello schema di algoritmo di visita

La complessità è lineare nel numero degli archi. Tutti gli archi devono essere visitati almeno una volta, se vogliamo fare una visita completa.

Tanti algoritmi di visita possono essere visti come modifiche di questo algoritmo qui andando a lavorare su come viene gestita l'aggiunta di nuovi nodi a Current. Se si aggiunge in cima si ha una politica in profondità (LIFO). Nel caso duale si ha una visita in larghezza. Queste sono fondamentalmente le uniche due visite che hanno senso.

Visita in profondità e larghezza hanno caratteristiche diverse. La visita in profondità non garantisce di trovare il cammino di lunghezza minima, quella in larghezza invece permette di trovare sempre il cammino di lunghezza minima che collega il mio nodo di partenza ad un altro nodo del grafo. La ricerca di un cammino minimo tra due nodi è un'operazione semplice.

È semplice calcolare il cammino più lungo in un grafo? Se lo fosse potremmo controllare la sua lunghezza. Se questa fosse uguale a n allora avremmo anche un cammino hamiltoniano. La ricerca di un cammino lungo è quindi equivalente alla ricerca di un cammino hamiltoniano. E quindi è anche un problema in $\mathbb{NP}$ (lo vedremo più avanti).

In un certo senso il duale di un problema semplice è un problema difficile (cammino minimo vs. cammino massimo). Non basta una semplice scansione dei cammini per trovare il più lungo, ma bisogna considerarli tutti. Questo rende più complessa la ricerca.

Cercare cammini brevi è un'operazione facile, cercare cammini lunghi è difficile.

11.7 Analisi di problemi

Quando abbiamo un problema dobbiamo innanzitutto trovare un algoritmo stupido per dare un primo upper bound alla complessità del problema. Gli algoritmi stupidi in genere fanno ricerche esaustive. L'algoritmo stupido aiuta a cominciare a comprendere il problema.

Ad esempio, per trovare il cammino più lungo possiamo scansionare tutti i cammini semplici e dire qual è il più lungo. Che upper bound abbiamo? Supponiamo di avere un grafo completamente connesso. Possiamo calcolare il numero di cammini con tutte le permutazioni possibili tra u e v. Abbiamo quindi un numero fattoriale di cammini rispetto al numero dei nodi, e di conseguenza rispetto alla dimensione del grafo. Questa complessità è dell'ordine di n^n: $n! \sim n^n$.

Un grafo completamente connesso rappresenta un caso particolarmente sfavorevole. Tuttavia anche per i grafi sparsi abbiamo un numero notevole di cammini. Tipicamente il numero è esponenziale o più che esponenziale. Ad esempio in un grafo come quello della figura 11.2 abbiamo un numero di archi che è lineare nell'ordine dei nodi ma abbiamo comunque un numero esponenziale di cammini tra due nodi.

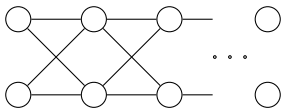


Figura 11.2: Esempio di grafo "semplice" con un grande numero di cammini possibili

La ricerca esaustiva è costosa. In linea di massima potremmo dare lo stesso upper bound per la ricerca di cammini minimi. In realtà per fortuna con la ricerca breadth-first possiamo visitare i nodi in "ordine di distanza". In questo modo possiamo individuare in maniera semplice i cammini minimi. È quasi un

miracolo in considerazione dello spazio di ricerca delle soluzioni di dimensione esponenziale che abbiamo.

Accade spesso che algoritmi polinomiali derivino dal fatto che invece di una ricerca esaustiva possiamo farne una più mirata, grazie a particolari condizioni e risultati.

L'altro test semplice che si deve sempre fare è: "data una soluzione esiste un modo semplice per verificare che una soluzione sia corretta in modo efficace"? È possibile "certificare" in maniera semplice una soluzione?

Il certificato per un cammino massimo non è banale: come facciamo a verificare che sia davvero il massimo in maniera efficiente? Viceversa è invece banale il certificato per un cammino di lunghezza k. Questo ammmette una verifica semplice, e questo colloca il problema di cercare un cammino di lunghezza k tra due nodi in un grafo in $\mathbb{NP}$, che è una classe piccola di problemi esponenziali.

Una volta che abbiamo capito che sta in $\mathbb{NP}$ possiamo passare a chiederci se sta anche in $\mathbb{P}$ e cominciare la ricerca di un algoritmo efficiente.

11.7.1 Colorabilità

Un altro problema interessante sui grafi è la colorabilità: capire se un dato grafo è colorabile con meno di n colori. La "difficoltà" del problema problema dipende dal valore di n. La bicolorabilità di un grafo è un problema semplice. Non tutti i grafi sono bicolorabili (vedi Figura 11.3). Un grafo può essere bicolorabile in più di un modo, a seconda di quale colore scegliamo per iniziare la colorazione. Tuttavia questo non è un problema, dato che possiamo sempre passare da una coloraziona all'altra semplicemente complementando i colori.

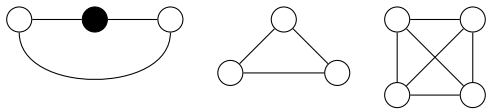


Figura 11.3: Esempi di grafi bicolorabili e di grafi non bicolorabili

Se abbiamo una cricca di dimensione n abbiamo bisogno di almeno n colori per colorarla, se questa è in effetti colorabile. La colorabilità di un grafo è legata alla densità degli archi del grafo.

La bicolorabilità è facilmente risolvibile mediante una visita. Il procedimento è a grandi linee questo:

- Ci poniamo la seguente invariante: i nodi già visitati e i nodi nella frontiera sono già stati colorati. Coloriamo quando aggiungiamo alla frontiera;
- Nella frontiera corrente selezioniamo un nodo;
- Andiamo a guardare i nodi connessi al corrente e facciamo la verifica di coerenza:

- Per ogni arco che ci riporta in Done verichiamo che i colori siano diversi, altrimenti ci blocchiamo. Questo perché il colore è forzato, se troviamo un'incoerenza non è possibile procedere altrimenti;
- Facciamo lo stesso controllo in Current.

- Per i nodi in Todo li spostiamo in Current, assegnandoli un colore. Quale? Il colore complementare del current.

Se l'algoritmo arriva a termine senza bloccarsi il grafo è bicolorabile.

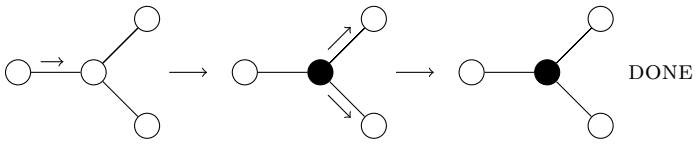


Figura 11.4: Esempio di bicolorazione di un grafo

Possiamo svolgere questo processo in una sola passata. Come mai? Perché l'assegnazione di colore è univoca. Questo non è il caso con più di due colori: con tre colori potremmo scegliere tra due alternative, e dovremmo fare backtracking. Il backtracking può portare (e in genere porta) ad un'esplosione esponenziale.

Già la 3-colorabilità è un problema $\mathbb{NP}$-completo. Perché sta in $\mathbb{NP}$? Perché è facile verficare la correttezza della colorazione, con una semplice visita arbitraria. $\mathbb{NP}$ è una classe ragionevolmente vicina a $\mathbb{P}$.

11.7.2 Problemi di flusso

Una classe di problemi tipici sui grafi è quella dei problemi di flusso.

Definizione 11.6. Una rete N è un grafo orientato con una sorgente s, un pozzo t e una capacità $c(u, v)$ associata ad ogni arco.

Definizione 11.7. Un flusso in N è una funzione $f(u, v)$ che ad ogni arco (u, v) associa un intero positivo tale che

- $f(u, v) \leq c(u, v)$;
- la somma dei flussi entranti in ogni nodo (esclusi s e t) è uguale alla somma dei flussi uscenti.

Il problema del flusso massimo è capire qual è il flusso massimo che la rete supporta, ovvero il flusso tale che la somma dei flussi uscenti da s (o equivalentemente di quelli entranti in t) sia massima. Ci chiediamo quanto "fluido" riusciamo a far passare dalla sorgente al target. Un upper bound è dato dal minimo tra la somma delle capacità degli archi uscenti dalla sorgente e la somma di quelli entranti nel target. È un'importante ipotesi che le capacità siano intere.

Vediamo un algoritmo semplice per risolvere il problema.

Per risolvere il problema inizialmente cerchiamo dei flussi banali, ovvero dei flussi lineari: la quantità di fluido si incanala lungo un unico cammino. Quello che cerchiamo quindi è un cammino qualunque tra sorgente e target. Lungo quel cammino cerchiamo di capire quanto fluido riusciamo a far passare. Questo è limitato dalla capacità dell'arco di capacità minima nel cammino. Di conseguenza definiamo un flusso che faccia uscire questa quantità di fluido dalla sorgente e la faccia entrare nel target lungo il cammino.

Si procede in questo modo finché non ci sono più cammini tra s e t, con l'accorgimento di fare degli "aggiustamenti" alla rete legati al fatto che su alcuni archi stiamo già facendo passare del flusso. Un'operazione è quella di diminuire la capacità di un arco lungo cui stiamo facendo già passare del fluido, un'altra è quella di diminuire il fluido lungo un arco per farlo passare lungo un altro.

L'algoritmo è dimostrabilmente corretto; andiamo quindi a discutere la complessità computazionale dell'algoritmo.

La ricerca di flussi e la somma di flussi sono operazioni lineari nella dimensione del grafo.

Quante iterazioni faremo al massimo? Con ogni operazione diminuiamo almeno di uno il flusso massimo che passa attraverso la rete. Usando gli upper bound sul flusso massimo dati prima abbiamo un upper bound alle iterazioni.

Più formalmente procediamo per maggiorazioni. Indichiamo la capacità massima degli archi nel grafo con C. Se supponiamo che tutti gli archi abbiano quella capacità abbiamo un upper bound nC al flusso massimo. La complessità delle operazioni di ricerca del flusso e applicazione del flusso è n^2, quindi abbiamo una complessità dell'ordine $O(n^3C)$. È una complessità polinomiale? No, C è un numero. In complessità bisogna stare attenti, specie quando abbiamo queste strutture etichettate. La complessità dovrebbe essere un logaritmo di C se volessimo un algoritmo polinomiale. In effetti esistono casi patologici che fanno esplodere la complessità per via di C.

Data questa considerazione il nostro algoritmo non è polinomiale. Da ricordare è che se la dimensione dei numeri è logaritmica, la complessità deve essere lineare nella dimensione dei dati.

Possiamo migliorare l'algoritmo? Sì, facendo una ricerca di cammino minimo quando andiamo a cercare dei flussi. In questo caso la complessità diventa polinomiale. Possiamo risolvere il problema del flusso massimo in tempo polinomiale.

11.8 Problemi decisionali

Un altro concetto importante è la differenza tra problemi di ottimizzazione e problemi di decisione. Nel caso del flusso massimo abbiamo un problema di ottimizzazione. Abbiamo dei vincoli e cerchiamo la soluzione minima (o massima) che soddisfa tali vincoli. In teoria della complessità siamo interessati a problemi decisionali, ovvero con soluzione booleana Sì/No.

In un certo senso possiamo vedere i problemi di ottimizzazione come dei particolari problemi di decisione, specie per i problemi di ottimizzazione discreti. Il trucco è, invece di chiederci se esiste un flusso massimo ci chiediamo "esiste un flusso maggiore di n?". È un problema meno informativo, ma se so risolvere il problema decisionale posso pensare ad un algoritmo efficiente per risolvere il problema di ottimizzazione. Ad esempio potremmo creare un loop che esegue il nostro algoritmo per il problema decisionale per valori diversi dell'input finchè non troviamo il massimo o minimo. Questo tuttavia non è molto astuto come metodo, ne esistono di migliori.

Di solito risolvendo il problema di ottimizzazione non abbiamo la soluzione in sè, ma un valore numerico tramite cui possiamo risalire alla soluzione in modo agevole. In altri termini otteniamo la x invece che la $f(x)$. Ma la f è facilmente calcolabile, di conseguenza questo non è un problema. Nel caso del problema del flusso massimo di solito la soluzione che otteniamo è il flusso, non il suo valore. Tuttavia questo è facilmente calcolabile con la somma dei valori del flusso sugli archi uscenti dalla sorgente, ad esempio.

Restringersi a problemi decisionali non è una decisione problematica, ed è interessante a livello matematico. Perciò in complessità di solito si fa questa assunzione. Per noi il problema della cricca sarà "questo grafo ha una cricca di dimensione k?".

11.9 Problemi e linguaggi

Nel problema decisionale abbiamo delle stringhe che descrivono l'input, in un qualche alfabeto. Per alcune stringhe la risposta al problema sarà Sì, per altre No. Il problema mi dà quindi un linguaggio di stringhe che rappresentano input per cui la soluzione è Sì. La teoria della complessità si può ridurre al problema di riconoscere stringhe di un linguaggio. L'algoritmo di decisione del problema può essere visto come un algoritmo di riconoscimento del linguaggio delle stringhe corrette. Le classi di complessità saranno quindi insiemi di linguaggi riconoscibili da una macchina di Turing deterministica o meno con una data complessità. Gli insiemi di linguaggi sono pensati senza un'architettura specifica in mente.

11.10 Riduzioni

Un'altra problematica importante in Complessità e il problema della riduzione. Lo vediamo guardando il problema del matching bipartito.

11.10.1 Matching bipartito

Definizione 11.8. Un grafo bipartito è una tripla $B = (U, V, E)$ dove U e V sono insiemi di uguale cardinalità e $E \subseteq U \times V$ è un insieme di archi.

Definizione 11.9. Un matching per un grafo bipartito $B = (U, V, E)$ è un insieme $M \subseteq E$ che associa ad ogni elemento in U uno e un solo elemento in V.

Un grafo può essere suddiviso, a livello dei nodi, in due insiemi, tipicamente dalla stessa cardinalità, i cui elementi non sono collegati. Otteniamo così un grafo bipartito.

Il problema del matching bipartito consiste nello stabilire se per un grafo bipartito esista o meno un matching.

Non per tutti i grafi bipartiti esiste un accoppiamento, e non è del tutto ovvio come van create le coppie.

Come lo risolviamo? Un approccio possibile è quello di sfruttare un algoritmo già noto e corretto per un altro problema simile. Nel nostro caso possiamo sfruttare l'algoritmo per il problema del flusso massimo. L'idea è che possiamo ridurre il problema del matching bipartito ad un problema di flusso massimo.

Per fare ciò facciamo la seguente trasformazione al grafo bipartito $B = (U, V, E)$, ottenendo quindi un grafo B':

- aggiungiamo un nodo s ed un nodo t;
- aggiungiamo un arco da s ad ogni nodo in U;
- aggiungiamo un arco a t da ogni nodo in V;
- assegnamo ad ogni arco una capacità unitaria.

A questo punto per B esiste un matching bipartito se e solo se B' ammette un flusso massimo di entità $n = |U| = |V|$, ovvero tale che il flusso totale uscente da s sia uguale a n.

Questo è un tipico esempio di riduzione. Abbiamo un problema, il matching bipartito, e lo riduciamo ad un altro problema, che sappiamo già risolvere. Utilizziamo il problema del flusso massimo per risolvere il nuovo problema. Che operazione facciamo per fare ciò? L'unica cosa che facciamo è trasformare i dati di input in una rete di flusso. Dopodiché li diamo in pasto al mio algoritmo per il flusso massimo. In base alla risposta del primo abbiamo automaticamente la risposta del secondo.

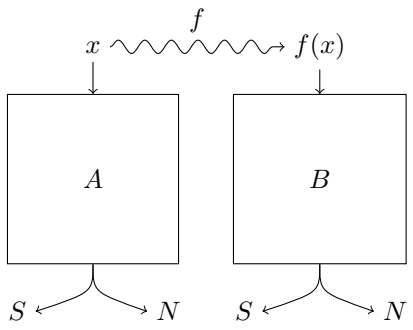


Figura 11.5: Schema di riduzione di un problema A in un problema B

La tentazione di solito è quella di cambiare un algoritmo per adattarlo ad un nuovo problema. In questo caso non facciamo quello; noi prendiamo i dati di input e li trasformiamo in modo che l'algoritmo iniziale lo possa accettare. L'algoritmo rimane lo stesso, quello che viene adattato è l'input. Questo significa fare una riduzione.

Avremo però qui dei vincoli in più rispetto alla riduzione della teoria della calcolabilità. Ad esempio avremo vincoli sul costo computazionale di f. Qui è in un certo senso più intuitivo rispetto alla teoria della calcolabilità, dato che facciamo trasformazioni da strutture dati a strutture dati (che per noi sono sempre stringhe), mentre in teoria della calcolabilità si trasformano numeri in numeri.

Esiste un'altra nozione di riducibilità, la Turing riducibilità. Immaginiamo un oracolo che risolve B e che possiamo interrogare quante volte vogliamo. Possiamo ora risolvere A? In questo caso avremmo che A è Turing-riducibile in B.

Per quanto riguarda il concetto di riduzione possiamo pensare ad un problema A e ad un problema B come linguaggi. Con una riduzione da A a B abbiamo che $x \in A \iff f(x) \in B$. Questa è una nozione di riducibilità, ce ne sono altre più potenti.

11.10.2 Riducibilità e classi di complessità

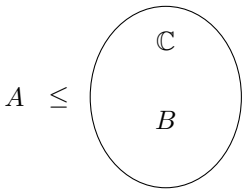
La riducibilità è interessante ai fini della complessità. Supponiamo di avere una classe di complessità $\mathbb{C}$ e supponiamo che $B \in \mathbb{C}$. Se riusciamo a ridurre A a B vorremmo poter concludere che anche $A \in \mathbb{C}$.

Per fare ciò è importante prendere una nozione di riducibilità abbastanza fine per le classi a cui siamo interessanti. Una riduzione troppo forte, come la Turing-riducibilità, potrebbe dare problemi.

Ad esempio pensiamo di poter complementare la risposta di un oracolo. In questo caso potremmo ridurre ogni linguaggio al suo complementare, che non è sempre quello che vogliamo. Potremmo non essere in grado di osservare differenze tra classi con una nozione così forte, mentre con una più debole (o fine) possiamo osservare certe differenze. Con riduzioni forti le classi in un certo senso collassano. Questo è interessante se vogliamo studiare classi di complessità (o linguaggi) che non sono chiuse per complementazione.

Tutte le classi non deterministiche non sono chiuse per complementazione (come non lo erano insiemi r.e.). Ad esempio i complementari di linguaggi in $\mathbb{NP}$ stanno in **CO-**$\mathbb{NP}$. In questo caso non ha senso usare una nozione troppo forte di riducibilità.

Concludere l'appartenenza di A in una classe $\mathbb{C}$ in base alla sua riducibilità ad un problema B in $\mathbb{C}$ non è immediato e dipende dalla nozione di riducibilità. In particolare andrà dimostrato.



$$B \in \mathbb{C} \wedge A \leq B \implies A \in \mathbb{C}$$

Figura 11.6: Chiusura di classi di complessità in base alla nozione di riducibilità

11.10.3 Complessità di f

La nozione di base di riducibilità l'abbiamo data, ma non basta. Si pongono dei limiti su f: si richiede che f abbia complessità polinomiale. Questo perché vogliamo la proprietà di chiusura della figura 11.6. Infatti se risolviamo B in tempo polinomiale e f opera una trasformazione anch'essa polinomiale componendo i due algoritmi avremmo un nuovo algoritmo polinomiale per risolvere A.

Richiedere che f sia addirittura lineare sarebbe più fine di quanto ci interessa: sarebbero troppi pochi i problemi riducibili ad un dato problema. Un vincolo polinomiale sembra essere abbastanza ragionevole. Ammettere funzioni di riduzione esponenziali permetterebbere di ridurre fondamentalmente qualsiasi problema a qualsiasi altro.

Problemi che appartengono apparentemente a domini completamente diversi possono essere agevolmente ridotti l'uno all'altro. Ad esempio SAT è riducibilie alla copertura di vertici di un grafo. La soddisfacibilità è stato il primo problema ad essere stato dimostrato essere $\mathbb{NP}$-completo. Se abbiamo un problema A che è $\mathbb{NP}$-completo e abbiamo B tale che $A \leq B$ diremo che B è $\mathbb{NP}$-hard.

Le riduzioni aiutano a vedere i problemi in un modo diverso dal solito.

Vediamo alcuni esempi di riduzioni polinomiali tra problemi.

11.10.4 Colorabilità

Vediamo ora che la n-colorabilità e riducibile alla $n+1$-colorabilità: n-COLOR $\leq$ $n+1$-COLOR.

Cosa prende in input n-COLOR? Un grafo e dice se è n colorabile. Fissiamo n a 3 e dimostriamo che 3-COLOR $\leq$ 4-COLOR.

Possiamo fare la riduzione con la nozione di riducibilità che abbiamo visto prima?

Bisogna fare attenzione a "non dare per ovvia la soluzione". Ad esempio è sbagliato pensare sempre che la funzione di riduzione sia l'identità, dato che l'identità solitamente permette solo di ridurre un linguaggio a se stesso. In questo caso avremmo un verso ma non avremmo l'altro: un grafo 3-colorabile

è sicuramente 4-colorabile ma non vale il viceversa, il che è importante per la riduzione.

Cosa dobbiamo fare? Dobbiamo prendere il grafo G in input per 3-COLOR e trasformarlo in un grafo G' che soddisfi la condizione di riduzione. Prendiamo G e aggiungiamo un nodo che colleghiamo a tutti i nodi di G. Per poter colorare questo nodo abbiamo bisogno di un colore nuovo non in G. Se G è 3-colorabile allora G' è 4-colorabile. Viceversa se G' è 4-colorabile il colore usato dal nodo aggiunto deve essere diverso da quello dei nodi del sottografo che corrisponde a G, che quindi risulta 3-colorabile.

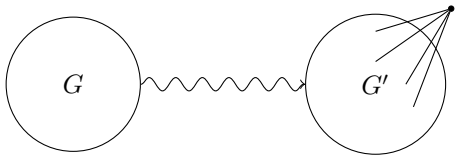


Figura 11.7: Riduzione del problema della n-colorabilità alla $n+1$-colorabilità

11.10.5 Cricca e insieme indipendente

Vediamo un altro esempio. Proviamo a ridurre il problema dell'insieme indipendente al problema della cricca: IS $\leq$ CRICCA.

L'input di IS è una coppia (G, k) e ci chiediamo se esista un insieme indipendente di dimensione k. L'input di CRICCA è una coppia (G, k) e chi chiediamo se esista una cricca di dimensione k.

La trasformazione è semplice: complementiamo il grafo. Dove c'era un insieme indipendente di nodi avremo tutti i collegamenti possibili tra di essi, e quindi una cricca.

11.11 Ricerca vs. Verifica

Un conto è cercare una soluzione, un conto è verificare la correttezza di una soluzione.

Di nuovo, quando incontriamo un problema bisogna pensare innanzitutto all'algoritmo stupido di ricerca esaustiva.

Non sempre il certificato è una soluzione al problema. È un'informazione in più che ci viene data e che ci permette di verificare in modo agevole la correttezza della mia soluzione. Non per tutti i problemi è possibile effettuare la verifica in tempo polinomiale (e.g. torri di Hanoi).

$\mathbb{NP}$ è una classe interessante perché contiene problemi facilmente certificabili.

11.12 Relazioni tra alcune classi di complessità

Cosa significa $A \in \mathbb{NP}$? Significa che $\exists B \exists p, B \in \mathbb{P} \wedge p$ *polinomio* tale che

$$x \in A \iff \exists c_x, |c_x| \leq p(x) \wedge \langle x, c_x \rangle \in B$$

La seconda parte dello statement ci dice che possiamo verificare che c_x è un buon certificato per x in tempo polinomiale. La prima parte dice che il certificato deve avere dimensione polinomiale.

Il linguaggio delle coppie $\langle x, c_x \rangle$ deve far parte di B sse $x \in A$.

C'è una differenza tra parlare di linguaggi e di macchine. Un linguaggio appartiene ad una certa classe se esiste una MdT che lo riconosce secondo le condizioni della classe, ma non è la macchina a far parte della classe. Le classi comprendono linguaggi, non macchine.

Per tanti problemi possiamo certificare la correttezza con un certificato, per altri no. Ad esempio per la soddisfacibilità si può fare, per la tautologicità no. Il problema è la dimensione del certificato.

Per tutti i problemi abbiamo un algoritmo generate and test, il problema sta nella ricerca che è costosa.

Ci sono alcuni problemi in $\mathbb{NP}$ di cui non si è dimostrata la completezza ma di cui se si potesse dimostrare l'incompletezza avremmo $\mathbb{P} \neq \mathbb{NP}$. Finora non si è dimostrata l'incompletezza di nessun problema in $\mathbb{NP}$.

Si congettura la gerarchia di classi di complessità mostrata in figura 11.8.

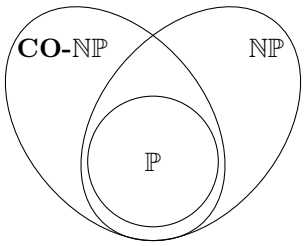


Figura 11.8: Gerarchia congetturata di alcune classi di complessità

Si congettura che $\mathbb{NP}$ sia diverso da $\mathbb{P}$, e che l'intersezione tra **CO-**$\mathbb{NP}$ e $\mathbb{NP}$ sia diversa da $\mathbb{P}$. Abbiamo che **CO-**$\mathbb{P}$ è uguale a $\mathbb{P}$. Se si dimostra, come si congettura, che **CO-**$\mathbb{P}$ sia diverso da **CO-**$\mathbb{NP}$ allora avremmo anche che $\mathbb{NP}$ sarebbe diverso da $\mathbb{P}$. Se si dimostra $\mathbb{P} = \mathbb{NP}$ allora collasserebbero tutte su $\mathbb{P}$. Abbiamo inoltre che ogni problema non banale in $\mathbb{P}$ è $\mathbb{P}$-completo. Se si dimostrasse che esiste un problema in $\mathbb{NP}$ incompleto avremmo $\mathbb{P} \neq \mathbb{NP}$.

Un problema molto studiato è quello dell'isomorfismo tra grafi. Il problema è capire se due grafi G, G' sono isomorfi, ovvero hanno la stessa struttura topologica. È un problema che ha un sacco di applicazioni concrete.

Possiamo certificare una soluzione per questo problema? Sì, con il mapping usato. Il numero di mapping è esponenziale, perciò una ricerca esaustiva non

è un granché. Di solito si cerca di pilotare la ricerca in modo da renderla più veloce, ad esempio sapendo che certi nodi non possono essere mappati in certi altri in base a certe condizioni.

Si congettura che questo problema, in questa forma, non sia completo. Non se ne è ancora dimostrata la completezza. C'è un problema simile, che è quello dell'isomorfismo di sottografo. Quello è un problema completo. Questo perché generalizza il problema della cricca.

Vediamo un problema che sta in **CO-**$\mathbb{NP} \cap \mathbb{NP}$ che si congettura non essere completo (altrimenti collasserebbe tutto).

Questo problema è la fattorizzazione di un numero intero. La versione decisionale prende (n, k) e ci chiediamo se n è fattorizzabile con meno di k fattori. Sta in $\mathbb{NP}$? Sì. Non è banale perché fino a poco tempo fa non si sapeva se il test di primalità fosse polinomiale. Ma sappiamo ora che lo è, quindi la verifica della fattorizzazione è semplice.

Abbiamo un certificato per dire che un numero non è fattorizzabile in meno di k fattori? Sì, sempre la fattorizzazione. Questo perché la fattorizzazione è unica.

Il certificato va bene per entrambi i casi, basta cambiare il metodo di verifica.

Capitolo 12

Classi deterministiche di complessità

12.1 Macchine di Turing

La macchina di Turing è un celebre modello di calcolo introdotto in 3.4. Lo rivediamo qui per comodità in una versione leggermente diversa (multi-tape).

Definizione 12.1. Una Macchina di Turing (multi-tape, deterministica) è una tupla $\langle Q, \Gamma, b, \Sigma, k, \delta, q_0, F \rangle$ dove

- Q è un insieme finito di stati
- Γ è l'alfabeto finito del nastro
- b è il carattere bianco
- $\Sigma \subseteq \Gamma \setminus \{b\}$ è l'insieme dei caratteri di input/output
- $k \geq 1$ è il numero di nastri
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali (o di accettazione)
- $\delta : Q \setminus F \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$ è la funzione di transizione.

Intuitivamente abbiamo una serie di nastri con memoria illimitata, divisi in celle di dimensione fissata. Ogni cella può contenere un carattere di un alfabeto dato, compreso un carattere b (bianco) di inizializzazione. Abbiamo una testina mobile di controllo per ogni nastro ed un automa di controllo a stati finiti.

Le operazioni fondamentali sono la lettura o scrittura di caratteri individuati dalla testina, lo spostamento della testina verso destra o verso sinistra, e la modifica dello stato interno dell'automa.

Per le misure di complessità in genere si tende sempre a prendere come riferimento la macchina di Turing. Questo perché si tratta di un modello di calcolo semplice in cui sono ben chiari i concetti di unità di tempo e di unità di spazio, che corrispondono rispettivamente ad una transizione e ad una cella di memoria. Ci chiederemo quanti nastri ci permettiamo di usare. Vedremo che questo farà la differenza.

I nastri sono infiniti, ma in ogni momento si suppone che solo una parte finita del nastro sia stata scritta. I nastri partono blank, su di essi viene scritto l'input e su di essi si svolge la computazione.

È la funzione di transizione che governa il comportamento della macchina.

12.1.1 Convensioni di input output

Si suppone in genere di avere un nastro di input sul quale possono essere effettuate solo letture e su cui ci si può muovere in una sola direzione. Non è limitante dato che possiamo sempre copiare il nastro di input in un nastro di lavoro e fare quello che vogliamo sul nastro copia.

Abbiamo anche un nastro di output, di sola scrittura, la cui testina avanza in una sola direzione.

Si suppone in genere di avere già l'input su nastro prima di cominciare la computazione. Questa ipotesi non cambia le considerazioni già fatte sulle complessità sublineari, in genere poco interessanti. Per convenzione supponiamo che la computazione cominci con la testina sul primo carattere dell'input, con le altre celle che hanno il carattere blank.

Per convenzione supponiamo che, alla fine della computazione, l'output sia la più lunga stringa di caratteri dell'alfabeto non blank alla sinistra della testina del nastro di output.

È importante non confondere configurazione iniziale e stato iniziale.

12.1.2 Configurazioni

Una configurazione è una descrizione dello stato della computazione ad un dato istante della computazione.

Rappresentiamo le configurazioni mediante tuple così fatte:

$$q, (\sigma_1, \tau_1), \ldots, (\sigma_k, \tau_k)$$

dove q è lo stato dell'automa e σ_i e τ_i sono due stringhe di caratteri che descrivono la porzione non (definitivamente) bianca del nastro i alla sinistra e alla destra della relativa testina. Il carattere in lettura è il primo carattere di τ_i.

La computazione avviene per passi discreti: una transizione tra due configurazioni è una relazione $\vdash$ governata dalla funzione di transizione:

$$(q, (\sigma_1 b_1, a_1\tau_1), \ldots, (\sigma_k b_k, a_k\tau_k)) \vdash (q', (\sigma_1\beta_1, \alpha_1\tau_1), \ldots, (\sigma_k\beta_k, \alpha_k\tau_k))$$

se

- $\delta(q, a_1, \ldots, a_k) = (q', a'_1, \ldots, a'_k, D_1, \ldots, D_k)$

- se $D_i = R$, allora $\beta_i = b_i a'_i$ e $\alpha_i = \varepsilon$
- se $D_i = L$, allora $\beta_i = \varepsilon$ e $\alpha_i = b_i a'_i$

La relazione $\vdash^*$ denota la chiusura transitiva e riflessiva della relazione $\vdash$.

Definizione 12.2. Una funzione $f : \Sigma^* \to \Sigma^*$ è calcolata da una macchina di Turing M se per ogni α esiste $q_f \in F$ tale che

$$q_0, (\varepsilon, \alpha), \ldots, (\varepsilon, \varepsilon) \vdash^* q_f, (\gamma_1, \tau_1), \ldots, (\gamma_k, \tau_k)$$

e $f(\alpha)$ è il più lungo prefisso di γ_k appartenente a Σ^*

Detto in altri termini una computazione parte dallo stato iniziale, con i nastri vuoti, con l'input sul nastro di input e procede per configurazioni succesive, fino ad arrivare ad una configurazione finale (se la funzione calcolata è definita per il dato input).

$$\begin{array}{c} \varepsilon, q_0, x \\ \| \\ c_0 \end{array} \to c_1 \to c_2 \to \cdots \to c_F$$

Dato che in complessità noi consideriamo problemi decisionali di solito l'output non è interessante di per sè. Potremmo avere due stati interessanti, uno di accettazione e uno di rifiuto.

Nel modello deterministico l'evoluzione della macchina è completamente determinata e univoca per la macchina considerata dato lo stato iniziale. Lo schema è lineare. La macchina non deterministica invece non ha una sola mossa disponibile, ne ha un set. Ci possono essere più scelte riguardo alla mossa successiva.

Lo schema corrispondente smette di essere una sequenza e diventa un grafo, come si puo notare in figura 12.1.

La computazione della macchina deterministica è un grafo con milioni di nodi. Inoltre questi nodi sono "grossi", un sacco di informazione è codificata in ogni nodo. Questo è un esempio vero di grafo "grosso" (Internet a confronto è un grafo piccolo).

Questo è fondamentale per quanto riguarda la simulazione deterministica della macchina non deterministica. In sostanza questa si riduce alla visita di tutti i cammini del grafo della computazione della macchina non deterministica.

Il problema è che dobbiamo visitare tutti i cammini del grafo. Solitamente non basta una visita semplice. Abbiamo il problema di come svolgere questo processo. Lo rivedremo più avanti.

12.2 Classi di complessità

Prima di vedere le classi di complessità definiamo un pò di operazioni: $time_M$, $space_M$, t_M, s_M.

Definizione 12.3. Sia M una Macchina di Turing.

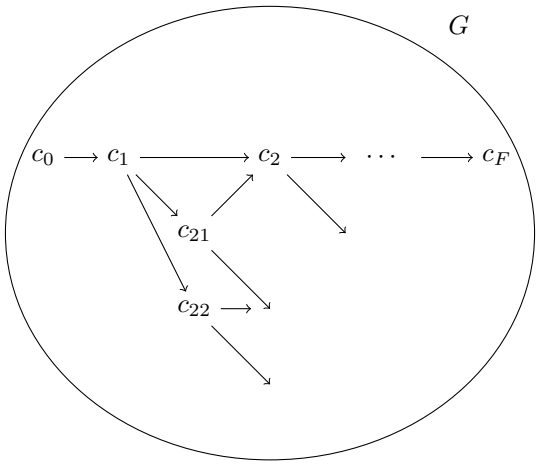


Figura 12.1: Schema di una computazione non deterministica

- $time_M(x)$ è il tempo di esecuzione di M su input x: il numero di passi richiesti per la computazione
- $space_M(x)$ è il numero massimo di celle visitate da una qualche testina durante la computazione (per macchine a più nastri si considerano solo i nastri di lavoro)
- $t_M(n) := \max\{time_M(x) \mid |x| = n\}$
- $s_M(n) := \max\{space_M(x) \mid |x| = n\}$

Se in una computazione abbiamo una configurazione $\langle \beta, q, \alpha \rangle$ su qualche nastro dove la lunghezza di β e α sono massime, lo spazio consumato è uguale alla somma tra la lunghezza di β e α.

Le prime due funzioni calcolano la complessità in funzione dell'input, ma noi vogliamo calcolarla in funzione della dimensione dell'input. Definiamo quindi le altre due operazioni. t_M rappresenta il caso pessimo di complessità in tempo per stringhe di lunghezza n. s_M è un analogo per lo spazio.

Definizione 12.4. Sia $f : \mathbb{N} \to \mathbb{N}$. Definiamo le seguenti classi di complessità:

- $\text{DTIME}(f) := \{\mathcal{L} \subseteq \Sigma^* \mid \exists M, \mathcal{L} = \mathcal{L}_M \wedge t_M \in O(f)\}$
- $\text{DSPACE}(f) := \{\mathcal{L} \subseteq \Sigma^* \mid \exists M, \mathcal{L} = \mathcal{L}_M \wedge s_M \in O(f)\}$

Le classi di complessità sono classi definite in termini di funzioni. La funzione di input mi dà un ordine di grandezza, non una misura precisa. In altre parole siamo interessati ad $O(f)$.

Nota bene: le classi **non rappresentano tempi o spazi**. Rappresentano classi di problemi, o, più formalmente, insiemi di linguaggi.

DTIME(f) rappresenta l'insieme di linguaggi che sono accettabili da una macchina di Turing in un tempo $O(f)$.

f è una funzione della dimensione dell'input. DTIME(n^2) è l'insieme dei linguaggi che sono riconosciuti da una MdT in tempo quadratico.

12.3 Relazione tra spazio e tempo

Il teorema che vediamo ora è molto importante. È analogo, a livello di importanza, al problema della terminazione della teoria della calcolabilità. Ci dice qualcosa sulla relazione tra complessità in tempo e complessità in spazio.

Sapendo che un certo linguaggio ha una certa complessità in tempo possiamo dire qualcosa sulla sua complessità in spazio, e viceversa?

Partiamo dalla parte più facile. Supponiamo di avere la complessità in tempo di un algoritmo.

Che motivazioni abbiamo per voler sapere la complessità in tempo? Una è sicuramente che il tempo è una risora critica. Inoltre è una misura più fine di quella di spazio, ci dà un upper bound a quest'ultimo. Questo perché se vogliamo aumentare l'occupazione di spazio, con la MdT, dobbiamo scrivere nuove celle. Sporcare una nuova cella richiede almeno un'unità di tempo. La complessità in tempo dà un upper bound stretto alla complessità in spazio.

Questo è importante, ed è anche una delle ragioni per cui il tempo è più interessante dello spazio.

Questo discorso vale a meno di costanti, dato che definiamo noi l'unità di tempo e spazio. Ma una volta definite queste il ragionamento resta immutato.

Abbiamo quindi che DTIME(f) $\subseteq$ DSPACE(f). A prima vista sembrerebbe un'"inversione". In realtà se ci pensa se abbiamo un upper bound alla complessità in tempo richiesta per risolvere un problema abbiamo che lo stesso upper bound si applica per la complessità in spazio, e quindi i linguaggi di DTIME(f) fanno sicuramente parte di DSPACE(f).

Teorema 12.1. *Sia* $f : \mathbb{N} \to \mathbb{N}$. *Allora* DTIME($f$) $\subseteq$ DSPACE(f).

Dimostrazione. Supponiamo che $\mathcal{L} \in$ DTIME(f). Abbiamo quindi che $\exists M, \mathcal{L} = \mathcal{L}_M \wedge t_M \in O(f)$. Ma se $t_M \in O(f)$ allora anche $s_M \in O(f)$. Questo perché il tempo è un upper bound allo spazio. Ma quindi $\mathcal{L} \in$ DSPACE(f). □

La D in queste due classi sta per determinstic.

Per dimostrare il teorema abbiamo svolto la nostra analisi sulla stessa macchina. Questo non è richiesto per la dimostrazione, dato che potremmo immaginare di svolgere la dimostrazione con un'altra macchina per la parte della complessità in spazio. In questo caso non è stato necessario, il teorema è puntuale sulla macchina. Se una certa macchina ha una certa complessità in tempo ha la stessa complessità, al più, in spazio.

Si può occupare anche meno spazio, il tempo fa da upper bound.

Questa era la parte facile, ragioniamo ora sul viceversa. Data la complessità in spazio posso dire qualcosa sulla complessità in tempo?

Supponiamo ad esempio che una certa macchina di Turing lavori in spazio costante. Quando misuriamo la complessità in spazio in genere abbiamo macchine multinastro. Di solito non consideriamo lo spazio del nastro di input. Noi consideriamo solo lo spazio aggiuntivo di cui la macchina ha bisogno per svolgere la computazione.

In questo ha caso ha senso parlare di complessità sublineari, dato che magari non abbiamo bisogno di uno spazio aggiuntivo della stessa dimensione dell'input. In particolare possiamo avere spazio costante, indipendente dall'input.

Un punto importante è che la complessità la misuriamo sempre al termine della computazione. Se la computazione non termina la complessità è indefinita, quindi non è un caso interessante quello di loop infinito. Se la complessità è definita non possiamo divergere.

Siamo in loop se passiamo per due volte sulla stessa configurazione. Ma noi questo non lo permettiamo. Perciò abbiamo un insieme finito di possibili configurazioni che verranno assunte in una computazione che giunge a termine.

Proviamo a calcolare il numero possibile di configurazioni immaginando di lavorare con uno spazio costante. Consideriamo i possibili nastri diversi di lunghezza L. Questi sono $|\Sigma|^L$. Moltiplichiamo per $|Q|$, il numero di stati. Infine moltiplichiamo per L, dato che la testina potrebbe essere su qualsiasi cella del nasto. Otteniamo la seguente quantità:

$$L \times |Q| \times |\Sigma|^L$$

Questo rappresenta un upper bound alla complessità in tempo del problema.

Il numero degli stati è fissato in base alla macchina. L'ordine è dato da L. La crescita è esponenziale. Non è così grave, c'è molto di peggio.

Questa analisi vale in caso di complessità costante. Cambia qualcosa se la complessità in spazio ha un ordine $O(s(n))$? No, basta che la mia L diventi una $L(s(n))$. Il discorso si applica invariato. Sappiamo che o la macchina termina entro questo upper bound oppure è in loop.

Anche qui l'analisi è puntuale. Data una macchina e una complessità in spazio possiamo dare una complessità in tempo sulla stessa macchina, non ne dobbiamo produrre un'altra.

In un certo senso per alcuni programmi non possiamo decidere la terminazione perché non possiamo neanche decidere la quantità di spazio che verrà richiesto. Si può dimostrare che se è decidibile il problema di sapere qual è l'occupazione di una certa risora sarebbe decidibile il problema di sapere qualcosa per un altra risorsa. Questo perché tutte le misure di complessità sono correlate. Questa correlazione può essere pessima a piacere, nel nostro caso era esponenziale.

In questa analisi stiamo dimenticando qualcosa. Sembrerebbe che se abbiamo una complessità in spazio costante avremmo una complessità in tempo costante, e quindi sublineare. Tuttavia in realtà noi dobbiamo sapere anche fino a che punto siamo arrivati nell'input. Questo va ad influire sul numero delle combinazioni. In particolare va aggiunta la lunghezza dell'input alla misura data prima.

$$n \times L \times |Q| \times |\Sigma|^L$$

Quindi abbiamo che, se $M \in \text{DSPACE}(f(n))$ allora $\exists c, M \in \text{DTIME}(c^{f(n)+\log(n)})$. Ricordiamo che $n = c^{\log_c(n)}$.

Teorema 12.2. *Sia* $f : \mathbb{N} \to \mathbb{N}$. *Allora* $\text{DSPACE}(f) \subseteq \bigcup_{c \in \mathbb{N}} \text{DTIME}(c^{log+f})$.

Ancora una volta l'esponenziale non è così male. Tutta la teoria della complessità si gioca tra esponenziale e polinomiale, è un vincolo abbastanza vicino.

12.4 Dipendenza dal modello di calcolo

La macchina di Turing è molto flessibile dato che ci sono tante variabili che possono influire. Ci chiediamo quanto questo ci influenzi nella nostra analisi.

Se rimpiccioliamo l'alfabeto abbiamo bisogno di stati nuovi per ricordare che ruolo ha un simbolo del nuovo alfabeto rispetto a quello che aveva nel vecchio.

Lo slowdown è lineare. Se passiamo, ad esempio, da ASCII a binario dovremo fare 8 passi per uno equivalente. Era di moda tempo fa la ricerca della macchina di Turing universale che minimizzasse la somma tra grandezza dell'alfabeto e numero di stati.

12.4.1 Riduzione dei nastri

Vediamo qualcosa di più importante, ovvero la riduzione dei nastri.

Immaginiamo di passare da due nastri ad un solo nastro. Perché è problematico lavorare con un nastro? Immaginiamo di avere una stringa l sul nastro e di volerla copiare più avanti. Con un nastro solo questa operazione è molto complessa. Dobbiamo leggere i caratteri $x_0, \ldots, x_n$ uno alla volta, memorizzando il carattere con uno stato interno. Non possiamo memorizzare l'intera stringa usando uno stato interno perché non abbiamo un upper bound alla lunghezza della stringa.

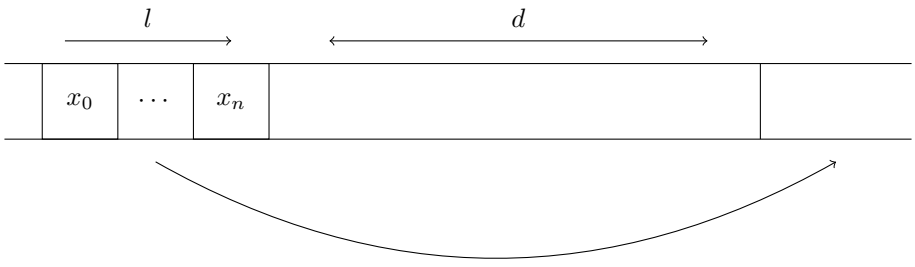


Figura 12.2: Copia di una stringa con due nastri

Se la distanza è d sono necessari $d \cdot l$ giri avanti e indietro. Essendo d almeno lineare in l (non consideriamo condizioni di overlap) abbiamo complessità almeno quadratica.

Con due nastri è semplice. Basta copiare la stringa di input su un altro nastro, posizionare la testina del nastro di input dove vogliamo copiare la stringa e posizionare la testina del secondo nastro all'inizio della stringa. A questo punto è sufficiente scrivere nel primo nastro il carattere letto dalla testina sul secondo nastro fino a raggiungere la fine della stringa.

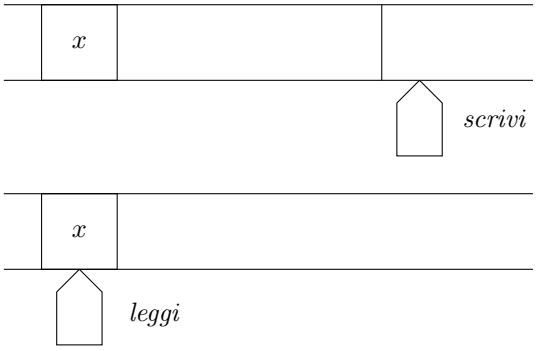


Figura 12.3: Copia di una stringa con due nastri

Passando da due nastri ad uno abbiamo uno slowdown che può essere quadratico. Questo è pesante. La complessità computazionale dipende dal modello di calcolo. Nell'ambito delle macchine di Turing questi slowdown restano però polinomiali.

C'è una sottigliezza da considerare riguardo la macchina a due nastri. Le due testine possono essere lontane a piacere, e il tempo di trasmissione tra le due può quindi aumentare arbitrariamente. Non è sempre corretto considerare il costo della tramsissione pari ad 1.

Generalizzando, proviamo a pensare ad una macchina a k nastri e a come simularla con un nastro solo. Se avessimo un modo efficiente per fare cio' avremmo un modo generale per eseguire un qualsiasi algoritmo eseguito sulla macchina a k nastri su un solo nastro. La complessita' dell'emulatore ci da' un upper bound alla complessita' legato dall'overhead richiesto per la simulazione.

La prima ipotesi che facciamo e' la seguente: non poniamo restrizioni sull'alfabeto dei nastri. Possiamo pensare di avere un alfabeto molto ricco per la macchina ad un nastro. In particolare possiamo immaginare di avere a disposizione un alfabeto i cui simboli rappresentano l'unione di vari pezzi dell'alfabeto originale. E' in un certo senso analogo all'utilizzo di una parola di memoria a 64 bit per memorizzare due parole di memoria a 32 bit. Come abbiamo visto cambiare l'alfabeto comporta un overhead lineare.

Facciamo poi un'altra ipotesi: da qualche parte dobbiamo memorizzare la posizione della testina.

Noi immaginiamo di simulare k nastri su un nastro solo utilizzando un alfabeto per il nostro nastro in cui ogni simbolo rappresenta k nastri e k "tracce".

I simboli sui nastri simulati sono simboli dell'alfabeto originale, e sulle "tracce" memorizziamo la posizione della testina di ogni nastro rispetto al nastro simulato. In figura 12.4 e' possibile vedere una rappresentazione grafica di questo processo.

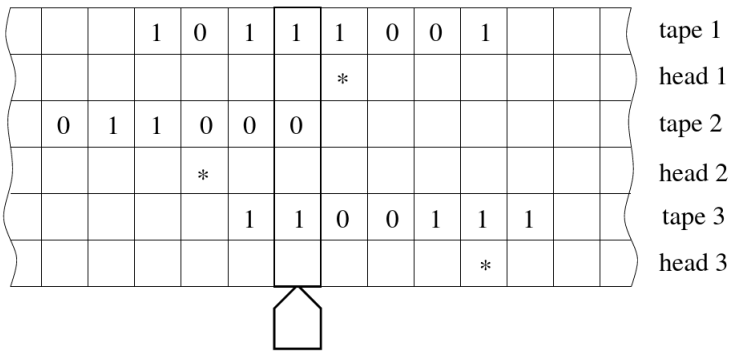


Figura 12.4: Simulazione su una MdT con un nastro di una MdT con piu' nastri

Se Σ e' l'alfabeto originale, il nuovo alfabeto e' $\Sigma' = (\Sigma \times \{*, B\})^k$.

Proviamo ora a capire quanto ci costa simulare una operazione della macchina a k nastri su questa macchina.

Ci sono dei piccoli accorgimenti che si possono fare per avere dei miglioramenti dal punto di vista della complessita' in tempo, a dispendio della memoria. Ad esempio potremmo immaginare di memorizzare dell'informazione ulteriore che ci dia una direzione nella quale dovremmo cercare la testina, in modo da non dover scansionare l'intero nastro. Tuttavia queste ottimizzazioni non sono significative a livello di complessita'.

Supponiamo di avere due nastri per semplicita'. La nostra macchina sarebbe descritta da n-uple cosi' fatte:

$$((a_1, a_2), q, (a_1', a_2'), q', (M_1, M_2))$$

dove a_1, a_2 sono i caratteri in lettura, M_1 e M_2 sono le due mosse sulle testine, a_1' e a_2' sono i due nuovi caratteri da scrivere e q e q' rappresentano, rispettivamente, lo stato attuale e il nuovo stato. Questa e' una tipica istruzione della macchina a due nastri.

Simulando questa macchina con un nastro solo abbiamo quindi bisogno di una prima passata del nastro per capire quali sono i caratteri in lettura e salvarli in uno stato interno. A questo punto riportiamo la testina all'"inizio" del nastro e facciamo un'altra passata per scrivere i nuovi caratteri. Infine facciamo una terza passata per simulare la mossa sui nastri simulati.

Sono quindi necessarie piu' passate per simulare un'operazione della macchina a piu' nastri. Un'operazione che aveva costo 1 ora ha un costo che dipende dall'occupazione di memoria. Possiamo dare un upper bound alla dimensione

del nastro al tempo t? Si', abbiamo un limite lineare in t. Siamo passati da costo 1 a costo $O(t)$. Inoltre il nostro upper bound non e' costante, peggiora col tempo.

Qual e' il costo totale della simulazione? Abbiamo una somma di operazioni di costo lineare. Di conseguenza avremo un costo totale quadratico.

Abbiamo quindi visto che con una simulazione fatta in questo modo abbiamo un overhead quadratico. Non abbiamo pero' dimostrato che passando da n nastri ad un nastro abbiamo necessariamente un overhead quadratico. Ci potrebbe essere un modo migliore per fare simulazioni che non implica questo aggravio nella complessita'.

Un altro modo in cui potremmo fare la nostra simulazione e' quello di utilizzare un alfabeto $\Sigma' = \Sigma^k$, se Σ era l'alfabeto di partenza, e immaginare di tenere le testine tutte allineate e che siano i nastri a muoversi sotto le testine.

Con una simulazione fatta in questo modo non dobbiamo piu' cercare la testina, i caratteri in lettura li troviamo subito, le scritture le facciamo immediatamente. Quello che diventa complicato e' fare la mossa. Dobbiamo fare una riscrittura dell'intero nastro per fare questo shift dei nastri simulati. Questo richiede quantomeno una scansione della memoria occupata.

L'overhead di simulare una macchina a k nastri con una macchina ad un nastro solo sembra quindi essere intrinsicamente quadratico.

A riprova di questo fatto esistono linguaggi che sono riconoscibili in tempo lineare con una macchina a due nastri, in tempo quadratico con una macchina ad un nastro, ma che non sono riconoscibili in un tempo che sia un $o(n^2)$ con una macchina ad un nastro. Un linguaggio con queste caratteristiche e' il seguente:

$$\mathcal{L} = \{w\#^{|w|}w \mid w \in \{0,1\}^*\}$$

La dimostrazione di questo fatto e' interessante perche' e' in generale difficile dimostrare che un certo problema non si possa risolvere con un algoritmo di complessita' migliore di una nota. La dimostrazione fa inoltre uso della complessita' di Kolmogorov.

Nota sugli slowdown

Nel calcolare gli slowdown prendiamo un'operazione di costo costante e vediamo quanto costa eseguirla in un altro modo. In questo modo la complessita' dell'altro modo corrisponde allo slowdown che abbiamo.

Se non abbiamo costo costante dobbiamo fare un'analisi piu' sofisticata. O prendiamo il caso pessimo, e cosi' diamo un upper bound. Questo pero' potrebbe essere troppo esagerato, e sarebbe il caso di dare un upper bound definito meglio.

12.4.2 Random Access Machine

Quanto dipende la complessita' dalla modello che utlizziamo? Per rispondere a questa domanda vediamo un modello di calcolo molto piu' "liberale" della MdT: la Random Access Machine (RAM).

In una RAM abbiamo delle celle di memoria tutte indirizzabili, come se fossero dei registri. Di queste ne abbiamo una quantita' numerabile. Possiamo accedere a queste mediante "indirizzi", che possiamo immaginare per semplicita' esssere dei numeri interi. L'ipotesi forte che facciamo e' che ogni registro possa avere dimensione illimitata: in ogni registro possiamo scrivere numeri arbitrariamente grandi.

Abbiamo, tra i vari registri, il program counter, che e' un registro particolare che mantiene la prossima istruzione da eseguire.

Non e' un modello realistico di complessita', proprio per via di questa ipotesi non realistica.

Usiamo le seguenti notazioni:

- r_i rappresenta il contenuto del registro i-esimo. Utilizziamo il registro r_0 come accumulatore;
- i_j rappresenta il j-esimo input;
- k rappresenta il program counter.

Il comportamento della macchina e' descritto da programmi composti da una sequenza di istruzioni di uno dei seguenti tipi:

- READ, che ha due varianti:
  - READ j, con la seguente semantica: $r_0 \leftarrow i_j$
  - READ $\uparrow j$, con la seguente semantica: $r_0 \leftarrow i_{r_j}$
- STORE, che ha due varianti:
  - STORE j, con la seguente semantica: $r_j \leftarrow r_0$
  - STORE $\uparrow j$, con la seguente semantica: $r_{r_j} \leftarrow r_0$
- LOAD x, con semantica $r_0 \leftarrow x$
- ADD x, con semantica $r_0 \leftarrow r_0 + x$
- SUB x, con semantica $r_0 \leftarrow r_0 - x$
- HALF, con semantica $r_0 \leftarrow r_0/2$
- JUMP j, con semantica $k \leftarrow j$
- JPOS j, con semantica **if** $r_0 > 0$ **then** $k \leftarrow j$
- JZERO j, con semantica **if** $r_0 = 0$ **then** $k \leftarrow j$
- HALT, con semantica $k \leftarrow 0$

E' molto simile ad un linguaggio Assembler di un tipico calcolatore. Questo e' un modello piu' vicino alla macchina di Von Neumann rispetto alla MdT. Consideriamo di costo 1 tutte le operazioni della RAM, indipendentemente dal fatto che lavorino su interi di grandezza arbitraria. Inoltre se immaginiamo di avere un problema per cui la dimensione dei nostri registri e' sufficiente il modello risulta abbastanza fedele a quello di Von Neumann, e quindi alla realta'.

Questo e' un modello abbastanza flessibile e generale, e' difficile immaginare di meglio. Questo giustifica anche l'irrealta' di questo modello.

Quanto ci costa simulare questa macchina con una MdT?

Abbiamo delle ipotesi. All'inizio della computazione e' caricato in un particolare registro con l'input, il PC e' settato all'inizio del nostro programma e tutte le altre celle sono memorizzate a vuoto, non contengono informazione.

Il problema principale e' come gestiamo la memoria di questa macchina nella MdT. Immaginiamo di usare un nastro della MdT per la simulazione della memoria. Dobbiamo simulare, registro per registro, il contenuto dei registri. All'inizio abbiamo solo il registro di input con informazione interessante.

Non sappiamo in che modo saranno scritti i registri della macchina. Ad esempio non possiamo supporre che avremo accessi sequenziali (prima il primo registro, poi il secondo, ecc.). Un modo semplice per simulare la memoria e' descrivere ogni registro con due campi: il nome (il numero) e il valore. Ogni registro corrisponde ad una coppia nome-valore.

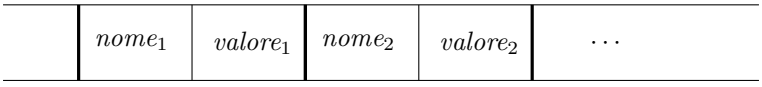


Figura 12.5: Simulazione dei regisri di una RAM in una MdT

In generale non possiamo dare un upper bound ne' al numero dei registri ne' alla dimensione di questi. Potremmo avere un numero arbitrario di registri occupati con valori arbitrariamente grandi.

Sul nastro della memoria memorizziamo le coppie in maniera sequenziale.

Immaginiamo di voler simulare un'operazione, ad esempio la somma. Solitamente si immagina il registro r_0 come l'accumulatore utilizzato per le operazioni aritmetiche.

Immaginiamo di sommare, ad esempio, al registro r_0 il valore del registro r_{25}. Dovremo fare una ricerca in memoria del registro r_{25}, leggere il valore e sommarlo al contenuto del registro r_0. Non abbiamo problemi di dimensione dell'accumulatore se ad esso dedichiamo un intero nastro.

Non abbiamo limiti sul numero dei nastri, purche' sia finito. Non possiamo immaginare un nastro per registro.

Immaginiamo di voler simulare una memorizzazione, magari del risultato della somma. Dobbiamo quindi scrivere un nuovo valore al posto del valore del registro r_0. Dovremmo quindi fare delle operazioni ulteroiri per simulare la memorizzazione, ad esempio uno shift delle coppie nome-valore verso destra per fare spazio.

Quello che facciamo e' una cosa brutale: cancelliamo il registro (nome-valore) e lo riscriviamo in fondo. Ha lo stesso costo dello shift, ovvero quello di una scansione della memoria.

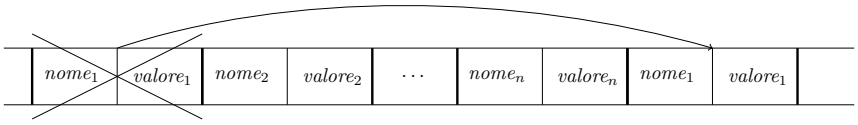


Figura 12.6: Simulazione di una memorizzazione in una MdT

Con questo approccio abbiamo problemi di frammentazione e un'occupazione di memoria maggiore rispetto a quella che avremmo con lo shift, ma in termini di complessita' questo non fa alcuna differenza.

Cosa possiamo dire sulla dimensione della memoria al tempo t della macchina simulata? Ci chiediamo quanti registri possiamo avere e quanto questi possono essere grandi.

Il numero di registri referenziati e' limitato da $O(t)$. Quant'e' l'occupazione massima di questi registri? Sempre $O(t)$. Perche'? Abbiamo una sola operazione che aumenta i valori dei registri, e quindi l'occupazione di memoria: la somma. Abbiamo che la somma di due numeri di dimensione n puo' avere al piu' dimensione $n + 1$. Di conseguenza, al tempo t, abbiamo una occupazione al piu' quadratica della memoria della macchina simulata.

Simulare una operazione della RAM ci costa quindi $O(t^2)$. Il costo complessivo della somma sara' la somma dei t^2 per t che va da 1 a t_{fin}. Abbiamo quindi un costo totale $O(t^3)$.

Simulare una RAM con una MdT puo' portare ad uno slowdown cubico. Potrebbe andare meglio, ma nel caso pessimo abbiamo questa complessita'. Questa inoltre potrebbe peggiorare se facessimo la simulazione con un solo nastro.

Possiamo vedere questo risultato in maniera positiva o negativa. Da un certo punto di vista questo slowdown e' gravoso. Dall'altro questo slowdown e' comunque polinomiale. In ogni caso un algoritmo che su RAM aveva complessita' polinomiale se simulato con una MdT continuera' ad avere una complessita' polinomiale.

Questo e' vero per un modello di calcolo molto liberale. Di conseguenza la complessita' computazionale di un algoritmo dipende dal modello di calcolo, ma se un modello e' "ragionevole" la complessita' dell'algoritmo rimarra' polinomiale anche su questo. La classe polinomiale dei problemi e' quindi una classe relativamente stabile rispetto al modello di calcolo con cui la studiamo.

Questo e' vero entro i limiti di ragionevolezza. Cosa intendiamo? Supponiamo di avere come primitiva anche l'operazione di moltiplicazione. Quello che succede e' che nella simulazione l'occupazione di memoria esplode: diventa esponenziale. Questo perche' con la moltiplicazione l'occupazione di memoria puo' crescere di n con un'operazione, dato che il prodotto di numeri di dimensione n ha puo' avere dimensione $2n$. Avremmo un overhead esponenziale.

E' quindi importante che il modello che utilizziamo abbia un minimo di ragionevolezza. Non ha senso contare 1 il costo dell'operazione di moltiplicazione. In un certo senso e' gia' strano che l'addizione abbia costo 1, ma nella RAM lo ammettiamo, e questo si rivela non causare problemi.

Abbiamo quindi due conclusioni importanti:

- la complessita' computazionale dipende dal modello di calcolo;
- se usiamo modelli di calcolo ragionevoli la complessita' computazionale su di essi non differisce piu' che polinomialmente.

C'e' chi parla di generalizzazione della tesi di Church, dove alla calcolabilita' corrisponde la complessita' polinomiale e ai modelli Turing completi corrispondono modelli di calcolo ragionevoli. Tuttavia questo dipende troppo dalle assunzioni che facciamo sul modello di calcolo, e quindi non ha la generalita' e lo spessore della tesi di Church.

Capitolo 13

Gerarchie in spazio e tempo

13.1 I teoremi della gerarchia

I teoremi che seguono sono tra i piu' importanti della teoria della calcolabilita'. Riflettono l'idea intuitiva che disponendo di una quantita' maggiore di risorse e' effettivamente possibile affrontare problemi piu' complessi.

Consideriamo $\text{DTIME}(n^2)$, ovvero la classe dei linguaggi riconoscibili in tempo quadratico e $\text{DTIME}(n^3)$. Qual'e' la relazione di inclusione tra queste due classi? Abbiamo sicuramente che $\text{DTIME}(n^2) \subseteq \text{DTIME}(n^3)$.

In generale abbiamo che, se f, f' sono funzioni da $\mathbb{N}$ a $\mathbb{N}$:

$$O(f) \subseteq O(f') \implies \text{DTIME}(f) \subseteq \text{DTIME}(f')$$

Vale lo stesso per lo spazio. Ricordiamo che $O(f) \subseteq O(f') \iff f \in O(f')$.

Ci chiediamo ora, questa inclusione e' stretta? E sotto che ipotesi possiamo affermare cio'? Esistono, ad esempio, dei linguaggi riconoscibili in tempo $O(n^3)$ ma non in tempo $O(n^2)$? E analogamente per lo spazio?

La risposta e', in un certo senso, positiva, sia per tempo che spazio. Vale piu' per lo spazio che per il tempo, poiche' lo spazio e' piu' "grezzo" del tempo come misura. La separazione in classi risulta piu' difficile con una misura fine come quella del tempo.

13.1.1 Il teorema della gerarchia in spazio

Vediamo innanzitutto come formalizzare il teorema ponendo la nostra attenzione sullo spazio. Stiamo facendo il confronto tra $\text{DSPACE}(n^2)$ e $\text{DSPACE}(n^3)$. Quello che vogliamo concludere e' che dato che $n^3 \notin o(n^2)$, allora $\text{DSPACE}(n^3) \not\subseteq \text{DSPACE}(n^2)$. Da cui si ottiene, data l'inclusione precedente, $\text{DSPACE}(n^2) \subset \text{DSPACE}(n^3)$.

Teorema 13.1. *Siano $f, f' : \mathbb{N} \to \mathbb{N}$. Si ha che:*

$$f \in O(f') \implies \text{DSPACE}(f) \subset \text{DSPACE}(f')$$

Per dimostrare il teorema, dando per ovvia l'inclusione non stretta, ci e' sufficiente dimostrare che:

$$f \notin O(f') \implies \text{DSPACE}(f) \not\subseteq \text{DSPACE}(f')$$

Il teorema in questa forma e' dimostrabile. Esiste tuttavia una forma piu' debole ma piu' intuitiva:

$$f' \in o(f) \implies \exists \mathcal{L}, \mathcal{L} \in \text{DSPACE}(f) \wedge \mathcal{L} \notin \text{DSPACE}(f')$$

Abbiamo rafforzato la premessa dell'implicazione e quindi indebolito il teorema. Ricordiamo che $f' \in o(f) \implies f \notin O(f')$.

E' uno dei pochi teoremi della teoria della complessita' che ci permettono di separare nettamente classi di complessita'. Questo e' in contrasto con le tante inclusioni congetturate che non riusciamo a dimostrare, la piu' famosa delle quali e' la questione $\mathbb{P}$ vs $\mathbb{NP}$.

Dimostrazione. Questo teorema e' sicuramente intuitivo nell'enunciato ma non e' ovvio nella dimostrazione.

Per dimostrare il teorema dobbiamo, sotto l'ipotesi $f' \in o(f)$, definire il nostro linguaggio $\mathcal{L}$ e poi dimostrare le proprieta' attese per $\mathcal{L}$.

La tecnica tipica che si usa quando si vuole creare un elemento che non faccia parte di una certa classe e' la diagonalizzazione. In questo caso definiamo il nostro $\mathcal{L}$ per diagonalizzazione in modo che non sia riconoscibile in spazio $O(f')$.

Vediamo prima una versione sbagliata del linguaggio, per poi passare a quella corretta. Questo passaggio ci portera' a capire una cosa importante della complessita'.

Usiamo la notazione $\lceil M \rceil$ per denotare il codice della MdT M.

Definiamo $\mathcal{L}$ come:

$$\mathcal{L} = \{\lceil M \rceil \mid \lceil M \rceil \notin \mathcal{L}_M \wedge s_M(|\lceil M \rceil|) \leq f(|\lceil M \rceil|)\}$$

La seconda parte della congiunzione ci garantisce che la complessita' in spazio delle macchine il cui codice fa parte di $\mathcal{L}$ su stringhe di lunghezza $|\lceil M \rceil|$ sia $O(f)$.

Procediamo per contraddizione. Supponiamo che $\mathcal{L} \in \text{DSPACE}(f')$. Ovvero, $\exists M_0$ tale che $\mathcal{L}_{M_0} = \mathcal{L}$ e inoltre M_0 lavora in spazio $O(f')$.

Ci chiediamo ora, $\lceil M_0 \rceil \in \mathcal{L}$? Questo e' vero sse $\lceil M_0 \rceil \in \mathcal{L}_{M_0}$. Dalla definizione di $\mathcal{L}$ abbiamo $\lceil M_0 \rceil \notin \mathcal{L}_{M_0} \wedge s_{M_0}(|\lceil M_0 \rceil|) \leq f(|\lceil M_0 \rceil|)$.

Abbiamo quindi:

$$\lceil M_0 \rceil \in \mathcal{L} \iff \lceil M_0 \rceil \notin \mathcal{L}_{M_0} \wedge s_{M_0}(|\lceil M_0 \rceil|) \leq f(|\lceil M_0 \rceil|)$$

Abbiamo che la prima parte del $\iff$ e' incompatibile con la prima parte della congiunzione. Per avere una contraddizione dobbiamo fare in modo che la seconda parte della congiunzione risulti vera.

Ci bastano le ipotesi che M_0 lavora in spazio $O(f')$ e che $f' \in o(f)$ per affermare che, su input $\lceil M_0 \rceil$, avremo che M_0 lavorera' in spazio al piu' limitato da $f(|\lceil M_0 \rceil|)$? La risposta e' no, poiche' quando parliamo di complessita'

computazionale parliamo di complessita' asintotica, mentre qui ci stiamo chiedendo quale sia il comportamento della funzione in un punto particolare. $\lceil M_0 \rceil$ potrebbe essere un punto sfortunato, come mostrato in figura 13.1.

La complessita' asintotica non ci dice niente su cosa succede su un determinato input.

Come procediamo? In questo caso il problema non e' particolarmente complicato. Siccome sappiamo che la complessita' asintotica della macchina M_0 sara' piu' bassa di f dobbiamo "spostare" l'input un po' piu' in la', per essere certi del comportamento di M_0 in quel punto. Per fare cio' andiamo a rivedere la definizione del linguaggio $\mathcal{L}$.

$$\mathcal{L} = \{\langle \lceil M \rceil, x \rangle \mid \langle \lceil M \rceil, x \rangle \notin \mathcal{L}_M \wedge s_M(|\langle \lceil M \rceil, x \rangle|) \leq f(|\langle \lceil M \rceil, x \rangle|)\}$$

La x e' una stringa aggiuntiva che non fa altro che accrescere la dimensione dell'input.

Come prima, supponiamo che $\mathcal{L} \in \text{DSPACE}(f')$, con macchina M_0 che riconosce $\mathcal{L}$ e lavora in tempo $O(f')$. Sappiamo che $f' \in o(f)$, ovvero esiste un punto m tale che, per valori in input maggiori di m, abbiamo che f' sta definitivamente al di sotto di f. Scegliamo ora un x_0 la cui rappresentazione in memoria ha una dimensione maggiore di m. Avremo sicuramente che $f'(|x_0|) < f(|x_0|)$, e questo varra' maggiormente per stringhe con una dimensione maggiore di quella di x_0.

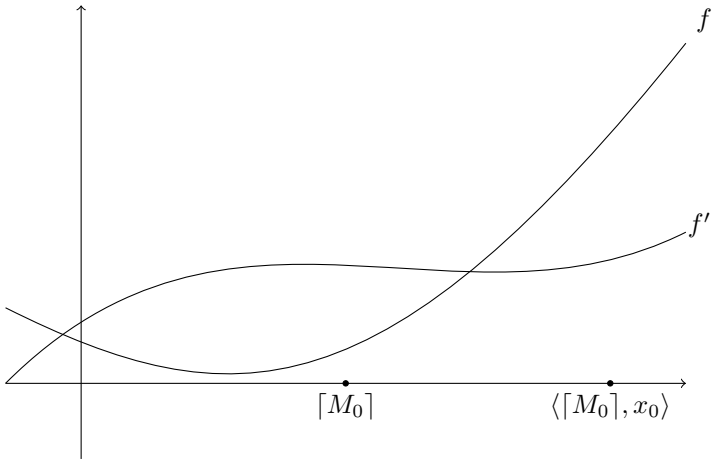


Figura 13.1: L'input $\lceil M_0 \rceil$ potrebbe essere in una posizione sfortunata. In generale non possiamo fare assunzioni su dove si trovi. La soluzione e' usare un x che sposti piu' in la' l'input, dove abbiamo la certezza che f sia al di sopra di f'

Quello che vogliamo e' quindi spostare l'input in un punto tale che da li' in poi f sara' definitivamente al di sopra di f'. Di fatto noi siamo interessati alla complessita' della macchina M_0, ma sapendo che questa lavora in spazio

$O(f')$ avremo che dal punto che raggiungiamo con il "padding" dato da x_0 in poi questa lavorera' in uno spazio che sara' definitivamente minore di f a meno di una costante.

Ci chiediamo ora se $\langle \lceil M_0 \rceil, x_0 \rangle \in \mathcal{L}$. Questo succede sse $\langle \lceil M_0 \rceil, x_0 \rangle \in \mathcal{L}_{M_0}$. Ma questo succede sse, per definizione di $\mathcal{L}$, $\langle \lceil M_0 \rceil, x_0 \rangle \notin \mathcal{L}_{M_0} \wedge s_{M_0}(|\langle \lceil M_0 \rceil, x_0 \rangle|) \leq f(|\langle \lceil M_0 \rceil, x_0 \rangle|)$. La seconda parte della congiunzione e' vera per come abbiamo scelto x_0, e a questo punto abbiamo una contraddizione.

Dobbiamo ora dimostrare che questo linguaggio e' riconoscibile in spazio $O(f)$. Quando vogliamo dimostrare che un certo linguaggio fa parte di una certa classe di complessita' purtroppo l'unico metodo che abbiamo e' quello di andare a definire un programma che riconosce il linguaggio con la complessita' della classe. Dobbiamo quindi progettare un algoritmo.

Come procediamo? Prendiamo una stringa $\langle \lceil M \rceil, x \rangle$. Dovremmo quindi verificare, inizialmente, che la prima parte della stringa rappresenti una MdT. Questo e' abbastanza semplice, e si puo' fare con dei semplici controlli sintattici. Dopodiche' andiamo a fare l'operazione vera e propria di riconoscimento. Proviamo a simulare il comportamento della macchina M. Ci serve una macchina universale che simuli M su input $\langle \lceil M \rceil, x \rangle$. Dovremmo quindi portare avanti la computazione e, nel caso la macchina M rifiuti, accettare, altrimenti rifiutare.

Abbiamo due problemi: la macchina simulata potrebbe non terminare su quell'input e non sappiamo che complessita' abbia la nostra simulazione, percio' potrebbe sforare il limite di f. Dobbiamo fare una simulazione bound, limitata da una quantita' data in input.

Il nostro bound ce lo abbiamo, dalla seconda parte della definizione di $\mathcal{L}$. Possiamo pensare di ritagliarci uno spazio di dimensione data nei nastri della macchina. Se, durante la computazione, la nostra simulazione cerca di sforare il limite possiamo interropompere la computazione e dire che la stringa non appartiene nel linguaggio. Siccome diamo bound f al simulatore riusciremo ad utilizzarlo con quello spazio, non di piu'. Abbiamo quindi limitato la complessita' in spazio.

Questo concluderebbe la dimostrazione.

La parte cruciale della dimostrazione e' la simulazione bound della macchina M. Nell'emulare non abbiamo un problema legato alla memorizzazione della macchina da emulare, poiche' questo richiede una quantita' lineare nella dimensione dell'input. Per dare l'upper bound alla macchina simulata e' sufficiente dare questo upper bound allo spazio dedicato alla memorizzazione dei nastri simulati nell'emulatore.

Resta una piccola problematica. Possiamo si' dare un upper bound allo spazio, ma cio' non implica che la macchina che stiamo simulando termini. Il problema sorge perche' se la macchina emulata non termina noi dovremmo accettare, non dovremmo divergere. La limitazione in spazio non ci aiuta, perche' la nostra macchina puo' divergere anche senza sforare il limite.

Dobbiamo utilizzare il teorema sulla relazione tra tempo e spazio per dare un upper bound anche al tempo. Se la macchina simulata non termina la sua computazione entro il tempo dato allora rifiuta l'input, e quindi l'emulatore accetta. Dobbiamo quindi memorizzare un timer e decrementarlo ad ogni ope-

razione della macchina simulata. Se questo raggiunge lo 0 allora la macchina simulata e' in loop e noi dobbiamo accettare.

Quanto occupa il timer? Il timer ha valore esponenziale rispetto allo spazio, ma noi lo rappresentiamo come un numero. Di conseguenza avra' un'occupazione logaritmica rispetto al suo valore. Ci serve quindi un'occupazione di memoria dell'ordine $O(f)$. □

Nella dimostrazione abbiamo omesso un dettaglio: f deve essere costruibile in spazio. Noi prendiamo l'input e, in base alla sua dimensione e attraverso f, allochiamo la memoria che fa da bound. Ma questo calcolo per l'allocazione richiede uno spazio limitato da f? Non possiamo avere questa garanzia in generale, da cui la necessita' che f sia costruibile in spazio.

Definizione 13.1. Una funzione $f : \mathbb{N} \to \mathbb{N}$ e' detta costruibile in tempo se esiste una MdT M sull'alfabeto $\Sigma = \{0\}$ che calcola una funzione f_M per cui:

- per ogni n, $f_M(0^n) = 0^{f(n)}$
- $t_M \in O(f)$.

Analogamente, f e' costruibile in spazio se valgolo le stesse condizioni con s_M al posto di t_M.

Questo significa che il calcolo di f deve poter essere fatto in $O(f)$. Tutte le funzioni semplici sono tipicamente costruibili: un polinomio lo costruiamo in tempo polinomiale, un esponenziale in tempo esponenziale, ecc. Ci sono pero' funzioni f che hanno delle complicazioni tali che f richiede piu' di $O(f)$ spazio per essere calcolata.

Di solito quando parliamo di classi di complessita' come $\text{DSPACE}(f)$ e $\text{DTIME}(f)$, affinche' abbiano senso, richiediamo almeno che la f sia costruibile, per non creare delle situazioni strane. Si puo' dimostrare che senza questa ultima ipotesi il teorema sarebbe falso. Ovvero, si puo' dimostrare che se abbiamo due funzioni non costruibili f, g, con $g \in o(f)$, possiamo avere un gap con nessun linguaggio in mezzo.

Ad esempio il logaritmo non e' una funzione costruibile in tempo: non riusciamo a calcolare il logaritmo in tempo logaritmico, e' una funzione troppo complicata.

Se usiamo una funzione f per definire una classe di complessita', ovvero vogliamo raggruppare tutti i problemi risolvibili con una complessita' che ha f come upper bound, ci aspettiamo che il calcolo di f faccia parte della classe. Ad esempio, $\text{DTIME}(n^2)$ e' una classe ragionevole perche la funzione n^2 e' calcolabile in tempo n^2. La costruibilita' di una funzione e' un sanity check che possiamo fare sulle classi di complessita' che definiamo. E' un po' il motivo per cui non ha senso considerare classi di complessita' sublineari in tempo.

13.1.2 Il teorema della gerarchia in tempo

Il teorema della gerarchia vale anche per il tempo. Tuttavia per il tempo la cosa non e' cosi' semplice.

Teorema 13.2. *Siano $f, f' : \mathbb{N} \to \mathbb{N}$. Si ha che:*

$$f' \in o\left(\frac{f}{\log(f)}\right) \implies \text{DTIME}(f') \subset \text{DTIME}(f)$$

In questo teorema richiediamo inoltre che $n \in O(f')$, ovvero f' sia maggiore o uguale all'identita'.

Cosa cambia col tempo? La prima parte della dimostrazione non cambia, dato che nella prima parte non abbiamo posto molta attenzione sul tipo di risorsa usata. Il fatto che usassimo lo spazio piuttosto che il tempo non entrava praticamente in gioco. Il problema e' nella seconda parte, ovvero nella simulazione con un bound in tempo. Determinare il costo della simulazione comporta qualche complicazione ulteriore.

In una simulazione bound in tempo abbiamo una macchina e una limitazione in tempo e vogliamo che la macchina venga simulata con il vincolo in tempo dato. Abbiamo una complicazione dovuta alla gestione del bound. E' una computazione piu' difficile da gestire di quella di una macchina universale, che data la macchina e un input esegue la computazione senza alcun vincolo.

Abbiamo quindi un rallentamento nella simulazione bound in tempo, ma di che tipo? Abbiamo sicuramente un rallentamento vistoso dovuto al fatto che con l'emulatore dobbiamo andare a cercare la prossima istruzione della macchina emulata. In effetti questo mostra un aspetto un po' criticabile della teoria della complessita', in cui una macchina di Turing consuma con costo 1 il programma e, "magicamente", trova istantaneamente la quintupla corrispondente alla prossima istruzione da eseguire ad ogni passo della computazione. La ricerca delle istruzioni da eseguire in memoria fatta dall'emulatore corrisponde ad una situazione gia' piu' realistica.

Questa ricerca e' limitata in tempo dalla dimensione del programma. Possiamo considerare questa costante, che dipende da programma a programma, ma sicuramente non dipende dall'input. Avremo quindi un costo aggiuntivo M, ma dal punto di vista della complessita' non ci cambia niente.

Tuttavia nella simulazione bound dobbiamo anche gestire un timer da decrementare ad ogni operazione. Questo ha una dimensione logaritmica rispetto al valore del timer. Di conseguenza dobbiamo tenere conto, nel programma simulato, di un rallentamento logaritmico dovuto alla gestione del timer. Se il nostro bound era $f(x)$ avremo una complessita' $f(x)\log(f(x))$. Sebbene il timer decresca col tempo questo non cambia l'ordine di grandezza del rallentamento.

Di conseguenza non riusciamo a garantire la seconda parte della definizione di $\mathcal{L}$, dobbiamo aumentare il gap. Abbiamo bisogno che $O(t') \subseteq O(\frac{t}{\log(t)})$. Poiche' abbiamo un rallentamento logaritmico avremo che invece di operare in tempo t la nostra simulazione operera' in tempo $t\log(t)$, se t era il bound che diamo alla nostra simulazione. Se alla simulazione diamo un buond $\frac{t}{\log(t)}$ avremo che la simulazione avra' costo $\frac{t}{\log(t)} \cdot \log\left(\frac{t}{\log(t)}\right) \leq \frac{t}{\log(t)} \cdot \log(t) \leq t$, e riusciremo a farla in un tempo $O(t)$.

Allo stato attuale della conoscenza non riusciamo ad essere piu' precisi al riguardo. Tuttavia non e' stato dimostrato che non si possa essere piu' precisi

in questa classificazione. Questo pero' non ci cambia molto dal punto di vista delle gerarchie: rimane l'inclusione stretta tra classi di complessita' come, ad esempio, DTIME(n) e DTIME(n^2), e $\mathbb{P}$ e EXP.

Anche in questo caso richiediamo la costruibilita' in tempo della funzione che calcola il valore del timer che fa da upper bound.

In tempo non riusciamo a fare classi cosi' fini come riusciamo in spazio.

Abbiamo che se $f' \in o(\frac{f}{\log(f)})$ allora abbiamo un linguaggio $\mathcal{L}$ riconoscibile in tempo $O(f)$ ma non in tempo $O(f')$.

13.2 Alcune gerarchie di complessita'

Definiamo alcune classi di complessita' come segue:

- $\mathbb{P} = \bigcup_{c \in \mathbb{N}} \text{DTIME}(n^c)$
- $\text{EXP} = \bigcup_{c \in \mathbb{N}} \text{DTIME}(2^{cn})$
- $\text{LOGSPACE} = \text{DSPACE}(log)$
- $\text{PSPACE} = \bigcup_{c \in \mathbb{N}} \text{DSPACE}(n^c)$
- $\text{EXPSPACE} = \bigcup_{c \in \mathbb{N}} \text{DSPACE}(2^{cn})$

Sappiamo che DTIME(2^n) e' diversa da DTIME(2^{2n}), per il teorema della gerarchia in tempo. Un'altra possibile definizione di EXP e' data dall'unione, su tutti i polinomi p, di DTIME(2^p). Sono due definizioni diverse, quest'ultima contiene la prima.

Abbiamo le seguenti relazioni tra queste classi di complessita':

- $\text{LOGSPACE} \subseteq \mathbb{P} \subseteq \text{PSPACE} \subset \text{EXPSPACE}$
- $\text{LOGSPACE} \subset \text{PSPACE}$
- $\mathbb{P} \subset \text{EXP} \subseteq \text{EXPSPACE}$

Per le inclusioni strette ci sono delle dimostrazioni, per quelle non strette non sappiamo se siano strette.

Perche' LOGSPACE e' incluso in $\mathbb{P}$? I teoremi della gerarchia si applicano a classi uniformi di complessita' (spazio vs spazio, tempo vs tempo). Con $\mathbb{P}$ e LOGSPACE stiamo confrontando risorse diverse, e percio' il teorema della gerarchia non si applica. Abbiamo che LOGSPACE contiene tutti linguaggi riconoscibili in spazio $O(\log(n))$. Per il teorema tempo-spazio abbiamo che un linguaggio con quella complessita' in spazio e' riconoscibile con una complessita' in tempo $O(2^{c \cdot (\log(n) + \log(n))}) = O(2^{c' \log(n)})$, per qualche c'. Ma $O(2^{c' \log(n)}) = O(n^{c'})$, quindi abbiamo una complessita' poliomiale in tempo, e quindi il nostro linguaggio fa anche parte di $\mathbb{P}$.

13.3 Padding

Il padding e' l'unica altra tecnica nota, oltre al teorema della gerarchia, per separare classi di complessita'.

Il padding e' una trasformazione tra linguaggi. Abbiamo $\mathcal{L}$ e lo trasformiamo in $\mathcal{L}^{\text{PAD}}$. Il linguagggio $\mathcal{L}^{\text{PAD}}$, ovvero il linguaggio delle stringhe paddate, e' cosi' definito:

$$\mathcal{L}^{\text{PAD}} = \{x\#^{f(x)} \mid x \in \mathcal{L}\}$$

A seconda della funzione f cambia il padding.

$$x \mapsto x\overbrace{\#\#\cdots\#}^{f(x)}$$

L'obiettivo e' ingrandire la dimensione delle stringhe del linguaggio. Possiamo fare qualunque tipo di padding; di solito si fa un padding quadratico. E' un'operazione di "imballaggio" delle stringhe di $\mathcal{L}$ con un numero di simboli in piu'.

Ad esempio, se vogliamo un linguaggio con padding quadratico avremmo

$$\mathcal{L}^{\text{PAD}} = \{x\#^{|x^2|-|x|} \mid x \in \mathcal{L}\}$$

Supponiamo di avere $\mathcal{L}$ e $\mathcal{L}^{\text{PAD}}$. Qual e' piu' semplice da riconoscere? $\mathcal{L}^{\text{PAD}}$. Per riconoscere $\mathcal{L}^{\text{PAD}}$, ovvero decidere se una stringa paddata w sta in $\mathcal{L}^{\text{PAD}}$, possiamo estrarre x da w e verificare se $x \in \mathcal{L}$ con il riconoscitore di $\mathcal{L}$. La rimozione del padding da w ha costo lineare rispetto a $|w|$, e' un'operazione semplice. Riconoscere x ha una certa complessita' che dipende dalla dimensione di x. Tuttavia noi abbiamo ingrandito l'input: la dimensione di w e' sicuramente maggiore di quella di x. Di conseguenza la complessita' di verificare se $w \in \mathcal{L}^{\text{PAD}}$, che calcoliamo rispetto alla dimensione di w, puo' essere minore della complessita' richiesta per verificare se $x \in \mathcal{L}$, a seconda del tipo di padding che abbiamo fatto.

Abbiamo questo fenomeno: gonfiando opportunamente l'input la complessita' degli algoritmi sembra diminuire.

Se ad esempio usiamo una notazione in base 1 per i numeri allora tutti gli algoritmi sembrano migliorare drasticamente a livello di complessita'. Ma questo miglioramento e' fittizio, ed e' legato all'esplosione esponenziale della dimensione dell'input.

Tipicamente abbiamo uno speedup se calcoliamo la complessita' del riconoscimento del linguaggio paddato, proporzionale alla nuova dimensione dell'input.

Supponiamo, ad esempio, che la complessita' richiesta per riconoscere x sia $O(|x^2|)$. Supponiamo di fare un padding quadratico, ovvero di passare a w con dimensione $|x^2|$. La complessita' dell'algoritmo di riconoscimento diventa $O(|w|)$, e sembra lineare. Tutto questo discorso suppone che estrarre x da w sia un'operazione di costo lineare.

Perche' e' interessante il padding? Perche' ci puo' aiutare a separare classi di complessita'.

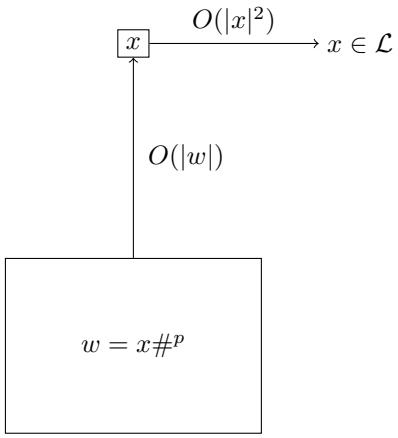


Figura 13.2: Riconoscere $\mathcal{L}^{\text{PAD}}$ costa in genere di meno di riconoscere $\mathcal{L}$

Supponiamo di avere due classi di complessita' che vogliamo separare: $\mathbb{C}$ e $\mathbb{C}'$. Quando vogliamo capire se due classi sono distinte ci possiamo chiedere rispetto a quali operazioni sono chiuse.

Le trasformazioni prendono in input un linguaggio e ne restituiscono in output un altro. Che una trasformazione produca un linguaggio all'interno della classe in cui si trovava il linguaggio di input non e' ovvio, e quando succede per ogni linguaggio della classe diciamo che la classe e' chiusa rispetto a questa trasformazione.

Un modo per separare le classi di complessita' e' prendere un'operazione, ad esempio il padding, e verificare se una delle due e' chiusa rispetto ad essa mentre l'altra non lo e'. Con questo abbiamo la dimostrazione che le due classi sono diverse.

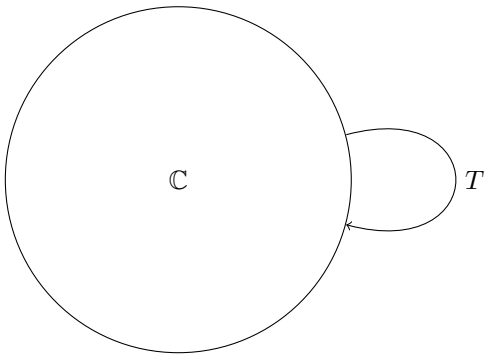


Figura 13.3: La classe $\mathbb{C}$ in figura e' chiusa rispetto alla trasformazione T

Prendiamo PSPACE e EXP. La nostra EXP e l'unione dei DTIME(2^{cn}) per ogni $c \in \mathbb{N}$. Si puo' definire EXP_p come l'unione di DTIME(2^{n^c} per ogni $c \in \mathbb{N}$. Sono due classi diverse. Abbiamo che $\text{EXP} \subset \text{EXP}_p$: lo si dimostra con il teorema della gerarchia.

Ci aspettiamo una relazione tra PSPACE ed EXP? Si'. Per il teorema tempo spazio uno spazio polinomiale implica un tempo esponenziale. Il grado dell'esponente, pero', dipende dal polinomio. Sicuramente $\text{PSPACE} \subseteq \text{EXP}_p$. Riusciamo a dimostrare altro? Non riusciamo a dimostrare $\text{PSPACE} \subset \text{EXP}_p$ allo stato attuale dell'arte, anche se si suppone cio'. Si riesce a dimostrare che $\text{PSPACE} \neq \text{EXP}$. Questo perche' abbiamo che PSPACE e' chiusa rispetto al padding polinomiale, mentre EXP non lo e'. Abbiamo inoltre che EXP_p e' chiusa rispetto ad un certo tipo di padding.

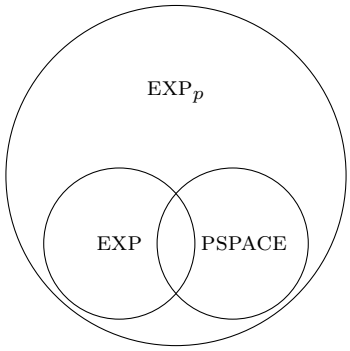


Figura 13.4: Gerarchia congetturata tra le classi PSPACE, EXP e EXP_p

Teorema 13.3. $\text{EXP} \neq \text{PSPACE}$

Dimostrazione. Dimostriamo innanzitutto che $\text{EXP} \subset \text{DTIME}(2^{n^2})$. Infatti abbiamo che $\forall c \in \mathbb{N}, 2^{cn} \in o(2^{n^2})$. L'unione di classi tutte contenute in DTIME(2^{n^2}) sta ancora in DTIME(2^{n^2}). Dato che EXP e' definita come il limite di questa unione potrebbe essere uguale a 2^{n^2}, di conseguenza non ci basta l'ipotesi di prima per concludere $\text{EXP} \subset \text{DTIME}(2^{n^2})$. Ce ne serve una piu' forte: consideriamo DTIME($2^{n^{1.5}}$). Abbiamo che $\forall c, 2^{cn} \in o(2^{n^{1.5}})$, e $\text{DTIME}(2^{n^{1.5}}) \subset \text{DTIME}(2^{n^2})$ per il teorema della gerarchia. Di conseguenza $\text{EXP} \subset \text{DTIME}(2^{n^2})$

Data questa inclusione stretta abbiamo che $\exists \mathcal{L}, \mathcal{L} \in \text{DTIME}(2^{n^2}) \wedge \mathcal{L} \notin \text{EXP}$. Consideriamo ora $\mathcal{L}^{\text{PAD}}$ con padding quadratico:

$$\mathcal{L}^{\text{PAD}} = \{x\#^{|x|^2-|x|} \mid x \in \mathcal{L}\}$$

Supponiamo per assurdo che $\text{PSPACE} = \text{EXP}$. Ci chiediamo ora, qual e' la classe di complessita' di $\mathcal{L}^{\text{PAD}}$? Di quanto decrescera' la complessita' di riconoscere $\mathcal{L}^{\text{PAD}}$ rispetto alla complessita' di riconoscere $\mathcal{L}$? Per riconoscere $\mathcal{L}^{\text{PAD}}$ noi partiamo da una stringa w con una certa dimensione $|w|$. Da questa estraiamo,

con complessita' lineare, x, con lunghezza n (da cui $|w| = n^2$). Cosa dobbiamo fare per vedere se $w \in \mathcal{L}^{\text{PAD}}$? Verificare se $x \in \mathcal{L}$, con costo $O(2^{n^2})$. La complessita' complessiva diventa $O(2^{|w|})$, ovvero con esponente lineare in $|w|$. Abbiamo quindi che $\mathcal{L}^{\text{PAD}} \in \text{DTIME}(2^n)$, il che implica che $\mathcal{L}^{\text{PAD}} \in \text{EXP}$. Ma questo implica $\mathcal{L}^{\text{PAD}} \in \text{PSPACE}$ per la nostra ipotesi di assurdo.

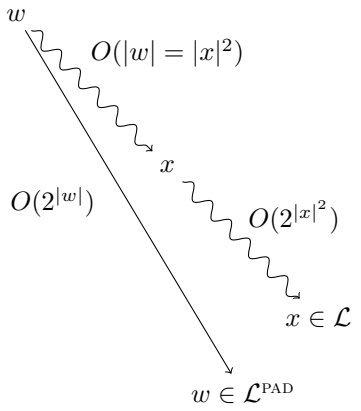


Figura 13.5: Riconoscere $\mathcal{L}^{\text{PAD}}$ ha una complessita' che ci permette di posizionarlo in EXP grazie al padding quadratico

Cosa possiamo dire della complessita' in tempo di riconoscere $\mathcal{L}$ sapendo che $\mathcal{L}^{\text{PAD}} \in \text{PSPACE}$? Cosa dobbiamo fare per riconoscere $\mathcal{L}$ sotto questa ipotesi? Ci basta prendere x, paddarlo e darlo in pasto all'algoritmo di riconoscimento di $\mathcal{L}^{\text{PAD}}$. Il padding richiede tempo quadratico, il riconoscimento di $\mathcal{L}^{\text{PAD}}$ lo facciamo in uno spazio in PSPACE.

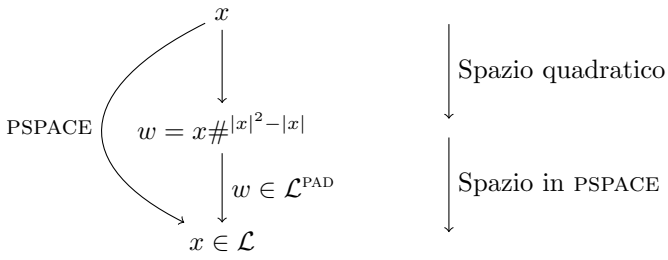


Figura 13.6: Sapendo che $\mathcal{L}^{\text{PAD}} \in \text{PSPACE}$ riusciamo a concludere che anche $\mathcal{L}$ appartiene a PSPACE

Qua non possiamo fare il massimo tra le due operazioni per calcolare la complessita' totale, dato che la stringa da cui partiamo e' piu' piccola di quella paddata, e il padding ci costa. Cio' che conta pero' e' che la composizione di polinomi e' un polinomio, di conseguenza il processo usa uno spazio in PSPACE.

Di conseguenza $\mathcal{L}^{\text{PAD}} \in \text{PSPACE} \implies \mathcal{L} \in \text{PSPACE} \implies \mathcal{L} \in \text{EXP}$, il che contraddice la nostra ipotesi iniziale. □

In questo caso abbiamo dimostrato che EXP $\neq$ PSPACE per assurdo. Si potrebbe anche dimostrare facendo vedere che PSPACE e' chiusa rispetto al padding polinomiale mentre EXP non lo e'.

13.3.1 Complessita' della composizione di funzioni

Sapendo qualcosa su due funzioni f e g sappiamo dire quanto ci costa calcolare $f(g(x))$? Non e' banale, perche' la complessita' la misuriamo rispetto alla dimensione dell'input. Ci chiediamo quindi prima quale l'ordine di grandezza degli output di g. Se lo sappiamo possiamo calcolare la complessita' di $f(g(x))$ usando la complessita' di f, altrimenti no. Se gli output di g fossero costanti in dimensione non avrebbe neanche senso chiedersi quale sia la complessita' di $f \circ g$, dato che sarebbe sempre costante. A noi interessa la complessita' di funzioni che prendono input che possono crescere arbitrariamente.

Ad esempio, g potrebbe produrre liste e f potrebbe ordinarle. Supponiamo che g, data una lista, produca il suo prodotto cartesiano. Supponiamo che g non restituisca risultati ordinati. Avremo un output di g di dimensione n^2. In questo caso avremmo una complessita' $O(n^2 + n^2 \log(n^2)) = O(n^2 \log n)$.

Se noi conosciamo solo la complessita' in tempo possiamo dire qualcosa sulla dimensione degli output? Con il teorema tempo spazio sappiamo che gli output prodotti sono bound dal tempo richiesto dall'algoritmo. Non possiamo dare bound migliori per il caso pessimo.

Bisogna stare attenti a capire se gli algoritmi producono strutture dimaniche. In tal caso c'e' una complicazione nel calcolo della complessita' di una funzione composta. E' un'analisi delicata.

13.4 Conclusioni

Volendo riassumere i risultati noti riguardanti le relazioni tra alcune classi di complessita' possiamo utilizzare un diagramma come quello in figura 13.8.

Nel diagramma le linee vanno intese come inclusioni. Con il $\neq$ abbiamo un'inclusione stretta.

Se abbiamo a che fare con risorse dello stesso tipo possiamo usare il teorema della gerarchia per dire qualcosa riguardo l'inclusione tra due classi di complessita'.

Nel caso abbiamo a che fare con risorse di tipo diverso dobbiamo usare altri strumenti. Ad esempio il teorema tempo spazio ci permette di stabilire inclusioni tra classi di complessita' in tempo e spazio.

Il teorema della gerarchia ci puo' dare inclusioni strette, quello tempo spazio da' solo inclusioni lasche.

Il teorema tempo spazio ci da', se abbiamo un bound polinomiale allo spazio, un bound esponenziale al tempo di cui non conosciamo la costante.

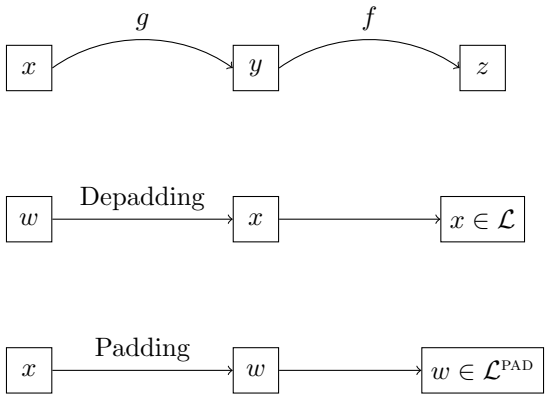


Figura 13.7: Per conoscere la complessita' di $f \circ g$ dobbiamo conoscere l'ordine di grandezza degli output di g. Nel primo esempio abbiamo che l'output del depadding ha una dimensione pari alla radice quadrata dell'input, mentre il riconoscimento ha complessita' $O(2^{|x|^2})$, per una complessita' totale di $O(2^{|w|})$. Nel secondo esempio l'output del padding ha dimensione $O(|x|^2)$, mentre il riconoscimento di $\mathcal{L}^{\text{PAD}}$ ha una complessita' in PSPACE, per una complessita' totale in PSPACE

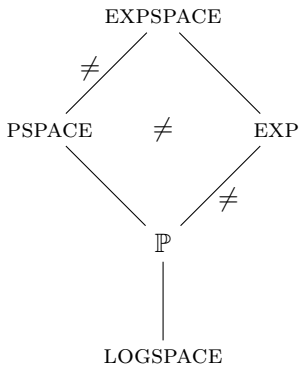


Figura 13.8: Relazioni di inclusioni tra alcune classi di complessita'

Ad esempio, se abbiamo un algoritmo che lavora in spazio logaritmico abbiamo che lavorera' in tempo polinomiale. Di questo polinomio pero' non possiamo sapere, in generale, il grado.

Nel parlare di classi "feasible", ovvero con complessita' ragionevole, a volte si tende ad escludere $\mathbb{P}$, perche' risulta troppo grande, includendo problemi con complessita' polinomiale alta. Si preferisce quindi spesso restringersi a LOGSPACE. In realta' anche questa non e' ideale, dato che ci possono essere

algoritmi che lavorano in spazio logaritmico ma in tempo polinomiale con un polinomio di grado alto. Inoltre esistono linguaggi riconoscibili in tempo polinomiale basso ma che usano uno spazio piu' che logaritmico, che in questo caso vengono esclusi. In ogni caso LOGSPACE resta una classe interessante che sta completamente in $\mathbb{P}$. Si congettura che LOGSPACE $\subset \mathbb{P}$, ma non si e' ancora riusciti a dimostrarlo.

PSPACE e' molto piu' grande di $\mathbb{NP}$. Abbiamo infatti che $\mathbb{P} \subseteq \mathbb{NP} \subseteq$ PSPACE, ma non si e' ancora dimostrato $\mathbb{P} \neq$ PSPACE, nonostante lo si congetturi.

Le inclusioni del diagramma che non sono strette sono inclusioni che sono congetturate essere strette ma che non sono ancora state dimostrate esserlo. Le maggior parte delle inclusioni strette vengono dai teoremi della gerarchia. La relazione tra EXP e PSPACE e' invece dimostrabile sfruttando il padding.

Il bound esponenziale dato dallo spazio al tempo e' interessante perche', ad esempio, ci permette di stabilire che LOGSPACE $\subseteq \mathbb{P}$.

Capitolo 14

Complessita' non deterministica

14.1 Macchine di Turing non deterministiche

Vediamo un altro modello di calcolo interessante, quello delle Macchine di Turing non deterministiche.

Una MdT non deterministica e' definita in modo analogo alla versione deterministica, con la sola differenza che la funzione di transizione δ e' multivalore:

$$\delta \subseteq Q \times \Sigma^k \times Q \times (\Sigma \times \{L, R\}^k)^k$$

In altri termini nella MdTN invece di avere una sola quintupla per ogni coppia stato-carattere ne abbiamo un insieme finito. In un certo senso e' come se avessimo un programma che, invece di avere una singola istruzione con cui continuare, ha un insieme finito di istruzioni tra cui scegliere per proseguire la sua computazione.

E' semplice da definire formalmente, basta modificare la definizione della macchina deterministica. La macchina deterministica e' in un certo senso piu' difficile da definire formalmente rispetto alla versione non deterministica, dato che la si puo' definire ponendo dei vincoli alla definizione di quest'ultima.

La macchina deterministica rappresenta un caso particolare della macchina non deterministica dove la scelta della quintupla e' univoca. Se qualcosa e' calcolabile da una macchina deterministica lo e' anche da una non deterministica. Di conseguenza le classi deterministiche sono contenute nelle corripondenti classi non determistiche. Ad esempio, $\mathbb{P} \in \mathbb{NP}$, PSPACE $\in$ NPSPACE, etc.

Definire la MdTN e' semplice. E' piu' complicato definire cosa calcola la MdTN. In una MdTN e' facile capire che ci sono piu' computazioni possibili. Quale sia pero' il risultato calcolato dalla macchina non e' ovvio.

Noi ci restringiamo a considerare solamente macchine decisionali, ovvero che rispondono si'/no. Queste ci sono sufficienti perche' siamo interessati a riconoscere linguaggi. Le nostre macchine terminano con risposta booleana.

Quando possiamo affermare che una stringa e' accettata da una MdTN? Abbiamo due criteri. Una stringa puo' essere accettata dalla MdTN se, qualunque computazione della macchina venga fatta, la stringa viene sempre accettata, oppure puo' essere accettata se esiste una computazione che accetta la stringa. A noi interessa la seconda definizione, quella della "macchina fortunata": ogni volta che abbiamo una scelta la nostra macchina fa quella giusta.

La computazione della macchina puo' essere rappresentata da un grafo.

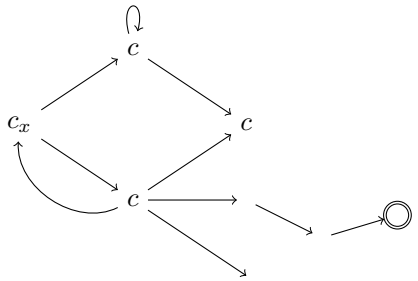


Figura 14.1: Grafo delle computazioni di una MdTN

Il grafo e' diretto non necessariamente aciclico. Il numero di scelte e' finito e definito dalla macchina (dal programma), che e' finita. Il grafo non e' neanche necessariamente finito, dato che possono esistere computazioni che non terminano non cicliche.

Ci porremo piu' avanti il problema di simulare la MdTN, e qui ci tornera' utile questo grafo, detto grafo delle computazioni.

Alcuni cammini nel grafo portano a stati finali accettanti, altri no, altri possono andare avanti indefinitamente. A noi basta che esista un cammino che porti ad una configurazione di accettazione per riconoscere una stringa.

Consideriamo SAT. Come funzionerebbe la MdTN per SAT? Noi dobbiamo prendere una formula proposizionale e decidere se questa e' soddisfacibile. La nostra macchina comincia a leggere la formula e trova come prima variabile proposizionale, ad esempio, A. A e' vera o falsa? La macchina "tira una moneta" e decide il valore di verita' di A. Dopodiche' la macchina va avanti. Questo processo continua fino alla fine della formula. Se la formula e' soddisfacibile esiste almeno una computazione fortunata che indovinera' l'assegnazione giusta di valori di verita' alle variabili proposizionali.

Alla macchina non deterministica basta una scansione lineare della formula, e quindi riesce a risolvere SAT in tempo lineare, a patto di prendere la strada giusta. SAT e' risolvibile in tempo polinomiale non deterministico da una MdTN. Questa sara' anche il modo in cui definiremo la classe $\mathbb{NP}$.

14.2 Complessita' non deterministica

Dobbiamo definire quali sono il tempo e lo spazio consumati dalla MdTN durante una computazione. Se la macchina accetta l'input esiste una computazione accettante. Se la risorsa che consideriamo e' il tempo noi prendiamo il tempo della computazione accettante piu' corta. In termini del grafo delle computazioni prendiamo il piu' corto cammino accettante della MdTN e la lunghezza di quel cammino e' il numero di passi richiesto per riconoscere una stringa x. Questo numero rappresenta quindi il tempo richiesto dalla macchina. Indichiamo il tempo richiesto dalla MdTN M su input x con $time_M(x)$.

Analogamente per lo spazio prendiamo lo spazio minimo tra gli spazi richiesti dalle computazioni accettanti. Tuttavia bisogna fare attenzione al come calcoliamo lo spazio richiesto da una computazione. Questo corrisponde al massimo spazio usato in una configurazione della computazione. Dobbiamo considerare il "minimo dei massimi". Indichiamo lo spazio richiesto dalla MdTN M su input x con $space_M(x)$.

Le funzioni t_M e s_M sono definite identicamente a come lo erano nel caso deterministico, a patto di usare le nozioni di *time* e *space* appena definite. Un discorso analogo di applica alle definizioni di $\text{NTIME}(f)$ e $\text{NSPACE}(f)$.

Possiamo definire le classi $\mathbb{NP}$, NEXP, NLOGSPACE e NPSPACE in maniera identica a come abbiamo definito le corrispondenti classi deterministiche, avendo cura di usare NTIME al posto di DTIME e analogamente per lo spazio.

Possiamo formalizzare quanto detto prima sulla relazione tra macchine di Turing deterministiche e non deterministiche con il seguente teorema.

Teorema 14.1. *Per ogni* $f : \mathbb{N} \to \mathbb{N}$

$$\text{DTIME}(f) \subseteq \text{NTIME}(f) \wedge \text{DSPACE}(f) \subseteq \text{NSPACE}(f)$$

La dimostrazione e' ovvia essendo la macchina di Turing determinstica un caso particolare della macchina non deterministica.

14.3 Simulazione del non determinismo

Vogliamo stabilire delle relazioni tra la MdTN e la corrispondente macchina deterministica. Per stabilire queste relazioni una tecnica che risulta utile e' pensare in termini di simulazione. Siamo quindi interessati a teoremi che riguardano la simulazione del nondeterminismo.

Immaginiamo di avere una macchina non deterministica che riconosce un linguaggio con una certa complessita' e immaginiamo di simularla con una macchina deterministica. Se riusciamo a determinare la complessita' della simulazione rispetto alla complessita' della macchina di partenza riusciamo a dire qualcosa sulla complessita' del riconoscimento del linguaggio in modo deterministico.

C'e' un modo interessante di vedere la simulazione della MdTN: consideriamo il grafo delle computazioni di una MdTN. Per simulare la macchina in modo deterministico ci basta trovare un algoritmo deterministico di visita dei cammini

del grafo. Dobbiamo fare un'esplorazione completa, dato che possono esistere in generale cammini di riconoscimento migliori di uno gia' trovato. Dobbiamo fare una qualche visita astuta per effettuare la simulazione.

In generale quando vogliamo stabilire delle relazioni tra classi deterministiche e classi non deterministiche usiamo classi di risorse diverse. Ad esempio, avendo un bound di tempo alla complessita' della MdTN diciamo qualcosa riguardo alla complessita' in spazio della simulazione deterministica. Il primo teorema che vediamo mostra questo.

Supponiamo di conoscere la complessita' $O(f)$ in tempo della macchina non deterministica e ci chiediamo quale sia la complessita' in spazio della simulazione deterministica, cercando di minimizzare l'occupazione di memoria.

Quello che dobbiamo fare e' occupare il minor spazio possibile durante la nostra visita del grafo delle computazioni. La visita in ampiezza non e' la migliore scelta in questo caso, dato che in generale potremmo avere una frontiera attiva dei nodi da visitare molto ampia che non ci e' necessaria.

Abbiamo che la MdTN lavora in $\text{NTIME}(f)$. Abbiamo che f rappresenta la lunghezza massima dei cammini da esplorare, dato che sappiamo che nostra macchina non deterministica riconosce il nostro linguaggio in tempo $O(f)$. Rispetto a questa lunghezza dei cammini il numero di nodi che possiamo avere cresce in generale esponenzialmente: se avessimo, ad esempio, per ogni nodo, una scelta binaria avremmo 2^n nodi.

Consideriamo una visita in profondita'. Esploriamo il primo cammino, con lunghezza $O(f(|x|))$. Quante configurazioni incontriamo? $O(f(|x|))$. Possiamo dire qualcosa sulla dimensione di queste configurazioni?

Abbiamo che anche per la MdTN il tempo fa da bound al tempo. Per ogni configurazione l'aspetto che influisce di piu' sull'occupazione di spazio e' la dimensione dei nastri. I nastri crescono, al massimo, in modo lineare rispetto al tempo. La dimensione dei nastri e' bound da $O(f(|x|))$. Per memorizzare la catena della configurazioni ci serve $O((f(|x|))^2)$ spazio.

Questo e' lo spazio richiesto per la visita di un ramo del grafo. Tuttavia il ramo che stiamo visitando potrebbe essere un ramo di fallimento, oppure un ramo che diverge. Nel caso fossimo in uno di questi due casi, una volta raggiunta una configurazione di fallimento o una volta che abbiamo raggiunto il bound in tempo della MdTN senza accettare possiamo tornare indietro lungo il ramo alla prima configurazione dove ci e' ancora possibile fare una scelta e effettuarne una diversa. Procediamo quindi per backtracking. In ogni caso l'occupazione di tutti i cammini che esploriamo ha gli stessi upper bound dati per il primo cammino.

Con una simulazione di questo tipo lo spazio richiesto dalla simulazione deterministica della MdTN e' dell'ordine di $O(f^2)$.

Possiamo pero' fare di meglio. Perche'?

In generale come facciamo a risparmiare spazio? Possiamo rendere implicite le rappresentazioni esplicite dei dati, attraverso una descrizione compatta che prendera' tempo al momento dell'esecuzione ma ci permettera' di risparmiare spazio. Al costo del tempo risparmiamo spazio. Tutto cio' che ci e' oneroso dal punto di vista della memoria lo rendiamo implicito attraverso una sintesi

compatta (ad esempio una compressione). Quando, in fase di esecuzione, avremo bisogno del dato ci bastera' spendere un po' di tempo per estrarlo dalla sua descrizione.

Abbiamo una tecnica analoga per risparmiare tempo a costo dello spazio. Pensiamo ad esempio alla programmazione dinamica con memoization. Manteniamo una tabela per memorizzare le soluzioni dei sottoproblemi. Quando ci serve una soluzione ad un sottoproblema possiamo guardare la tabella e, nel caso il sottoproblema sia gia' stato risolto, ottenere subito il risultato richiesto. Se questo non e' il caso risolviamo il sottoproblema e memorizziamo la soluzione in tabella per usi futuri.

Tempo e spazio sono due risorse "complementari". Non si puo', allo stesso tempo, ottimizzare il tempo e lo spazio. Ottimizzare in tempo richiede spendere in spazio e viceversa. Di solito siamo portati a pensare a ottimizzazioni del tempo, senza considerare lo spazio.

Quello che occupa piu' spazio nella nostra visita dei cammini e' la memorizzazione esplicita della catena di configurazioni. Possiamo dare una descrizione del cammino che prendiamo nel grafo molto piu' compatta, ad esempio come sequenza di scelte fatte dalla configurazione iniziale. Possiamo etichettare ogni scelta nel cammino con un numero e salvarci le etichette del nostro cammino. L'unica informazione che ci serve per definire il cammino e' la sequenza delle scelte prese. Nel caso dovessimo fare backtracking possiamo ripercorrere tutte le scelte del cammino corrente fino alla scelta da rivedere e cambiare scelta. Considerando tutte le possibili scelte avremo considerato tutti i possibili cammini del grafo.

Quanto ci costa memorizzare questa sequenza di scelte? L'unica configurazione che teniamo attiva e' quella finale, che e' bound da $O(f(|x|))$ per il teorema tempo spazio. Le scelte sono in un range finito fissato dalla macchina.

In definitiva ci serviranno $c \cdot O(f(|x|))$ per memorizzare le scelte e $O(f(|x|))$ per la configurazione finale del cammino. L'ordine dell'occupazione di spazio della simulazione fatta in questo modo e' quindi $O(f(|x|))$. Riusciamo a fare la simulazione della MdTN in spazio $O(f(x))$.

Teorema 14.2. *Per ogni $f : \mathbb{N} \to \mathbb{N}$ costruibile in spazio maggiore o uguale all'identita'*

$$\text{NTIME}(f) \subseteq \text{DSPACE}(f)$$

Possiamo vedere la simulazione in un modo piu' operazionale. Dato x inizializziamo due nastri di dimensione $f(|x|)$. In un nastro memorizziamo il tape e in un altro le scelte. All'inizio simuleremo la computazione dove la scelta che facciamo e' sempre la prima. Sul nastro del tape simuliamo la computazione della MdTN. A questo punto arriviamo in fondo in vari modi: o terminiamo il tempo, o cerchiamo di sforare lo spazio, o ci arrestiamo con riconoscimento, o ci arrestiamo con fallimento. In quest'ultimo caso andiamo a cambiare l'ultima scelta per fare backtracking e rifacciamo la computazione. L'occupazione di spazio rimane dell'ordine $O(f)$. Dovremmo memorizzare anche un timer per non sforare in tempo, ma questo ha occupazione logaritmica nel valore di f, di conseguenza l'occupazione di spazio non peggiora.

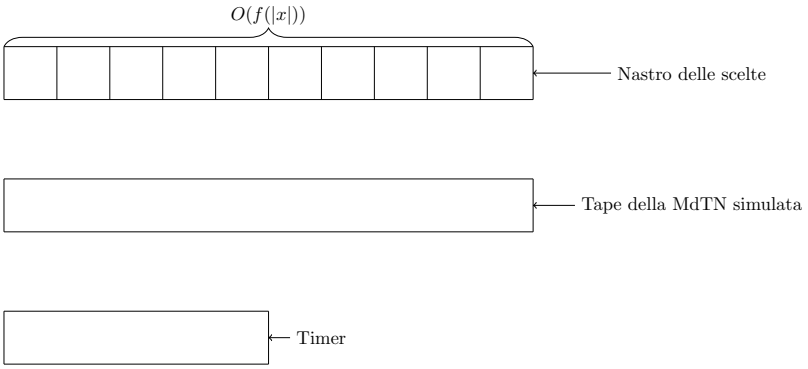


Figura 14.2: Simulazione deterministica di una MdTN con memoriazzazione delle scelte