Automi, Calcolabilità e Complessità

Alessandro Savioli

Ottobre 2024

Contents

1 Prefazione

La teoria della computazione inizia con una semplice domanda, **" Cos'è un computer? "**, al giorno d'oggi chiunque potrebbe rispondere, ma i computer reali sono piuttosto complessi per permetterci di sviluppare una teoria matematica direttamente su di essi, per questo serve avvalersi di un " Computer Ideale ", chiamato Modello Di Computazione.

Come per ogni modello scientifico un modello di computazione può essere accurato in alcuni aspetti ma non rispetto ad altri, per questo lungo il corso ne useremo diversi, a seconda delle caratteristiche su cui ci vogliamo soffermare.

Iniziamo quindi con il modello più semplice, chiamato **Macchina a stati finiti** oppure **Automa finito**.

2 Automi Finiti

Gli automi finiti sono un buon modello per computer con una quantità estremamente limitata di memoria. Essi sono alla base di vari dispositivi elettromeccanici.

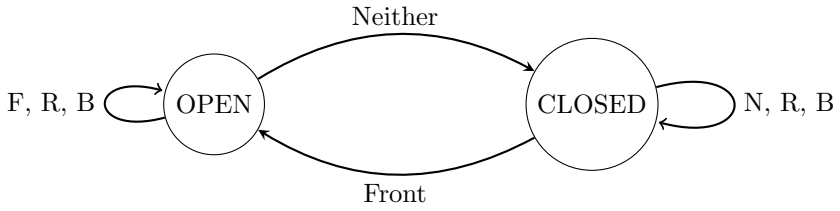
Il sistema di controllo per una porta automatica è un esempio di tale dispositivo, esse si aprono quando il sistema di controllo avverte che una persona si sta avvicinando.

Questo tipo di porte, che si possono trovare spesso alle entrate/uscite dei supermercati, ha un sensore davanti per rilevare la presenza di una persona che sta per attraversare la soglia, mentre un altro sensore è collocato dall'altro lato in modo che il sistema possa mantenere la porta aperta finchè la persona possa attraversarla in tranquillità.

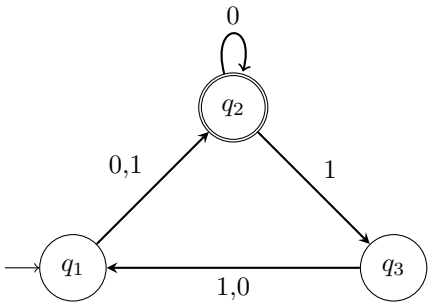
Possiamo raffigurare la disposizione in questo modo:

Ricordati di mettere l'immagine (porta supermercato)

Questo sistema di controllo può essere in 2 diversi stati, **OPEN** o **CLOSED**, che rappresentano la condizione corrispondente alla porta, mentre possono esserci quattro condizioni di input: **'FRONT', 'REAR', 'BOTH', 'NEITHER'**



Esaminiamo ora gli automi finiti da una prospettiva matematica. Svilupperemo una definizione precisa di automa finito, una terminologia per descriverli e usarli, e risultati teorici per descrivere il loro potere e i loro limiti.



Questo tipo di schema viene chiamato **Diagramma di stato**, esso ha tre stati, etichettati q1 q2 e q3, lo **Stato iniziale** q1 è indicato dall'arco entrante in esso e non uscente da un altro stato. Lo **Stato accettante** q2 è indicato da un doppio cerchio. Gli archi che vanno da uno stato all'altro vengono detti **Transizioni**.

Quando questo automa riceve una stringa in input come 1100, esso la elabora e produce un output, che può essere **Accetta** o **Rifiuta**, a seconda se alla fine della lettura dell'input, la macchina si trovi o meno in uno stato accettante, considerando per ora solo questo tipo di output sì/no.

2.1 Definizione Formale di Automa Finito

Definizione Un automa finito è una quintupla (Q, Σ, δ, q_0, F), dove

1. Q è un insieme finito chiamato insieme degli **stati**
2. Σ è un insieme finito chiamato **alfabeto**
3. δ : Q x $\Sigma \rightarrow$ Q è la **funzione di transizione**
4. $q_0 \in$ Q è lo **stato iniziale**
5. F $\subseteq$ Q è l'**insieme degli stati accettanti**

Successivamente diremo che se A è l'insieme di tutte le stringe che una macchina M accetta, allora chiameremo A il **Linguaggio della macchina M** e scriveremo

$$\mathbf{L(M) = A}$$

2.2 Definizione Formale di Computazione

Definizione Sia M = (Q, Σ, δ, q_0, F) un automa finito e sia w = $w_1 w_2 \cdots w_n$ una stringa dove ogni w_i è un elemento dell'alfabeto Σ.

allora M **accetta** w se esiste una sequenza di stati $r_0, r_1, ..., r_n$ in Q con tre condizioni:

1. $r_0 = q_0$

2. $\delta(r_i, w_i + 1) = r_i + 1$, per i = 0, . . . , n-1

3. $r_n \in F$

Definizione Un linguaggio viene chiamato **Linguaggio Regolare** se esiste un automa finito che lo riconosce

2.3 Le Operazioni Regolari

In aritmetica, gli oggetti di base sono i numeri e gli strumenti sono le operazioni per trattarli, come + e ×. Nella teoria della computazione, gli oggetti sono i linguaggi e gli strumenti includono operazioni specificatamente progettate per trattarli. Definiamo tre operazioni sui linguaggi, chiamate **Operazioni Regolari**, e le usiamo per studiare le proprietà dei linguaggi regolari.

Definizione Siano A e B linguaggi. Definiamo le operazioni regolari **Unione, Concatenazione e Star** come segue:

- **Unione**: A ∪ B = {x | x ∈ A o x ∈ B}
- **Concatenazione** : A ∘ B = {xy | x ∈ A e y ∈ B}
- **Star** $A^* = \{x_1 x_2 \cdots x_k \mid \text{k} \geq 0 \text{ e ogni } x_i \in A\}$

Teorema

La classe dei linguaggi regolari (REG) è chiusa rispetto all'operazione di unione e concatenazione.

In altre parole,

1. Se $A_1 e A_2$ sono linguaggi regolari, anche $A_1 \cup A_2$ lo sarà
2. Se $A_1 e A_2$ sono linguaggi regolari, anche $A_1 \circ A_2$ lo sarà

2.4 Non Determinismo

Il non determinismo è un concetto utile che ha avuto un grande impatto sulla teoria della computazione.

Finora ogni passo della computazione seguiva univocamente dal passo precedente, facendoci parlare così di macchina **deterministica**.

In una macchina **non deterministica**, invece, possono esistere diverse scelte per lo stato successivo. Il non determinismo è una generalizzazione del determinismo, quindi OGNI AUTOMA FINITO DETERMINISTICO È AUTOMATICAMENTE UN AUTOMA FINITO NON DETERMINISTICO.

In un DFA le etichette sugli archi di transizione sono simboli dell'alfabeto, mentre in un NFA si possono avere archi etichettati con simboli dell'alfabeto e un nuovo tipo di arco, etichettato con 'ε', chiamato epsilon-arco.

Come computa un NFA?

Supponiamo di eseguire un NFA su una stringa di input e di giungere in uno stato con più modi di procedere, dopo aver letto il simbolo in input, la macchina si divide in **più copie di sé stessa e segue tutte le possibilità**

in parallelo, se ci sono scelte successive, la macchina si divide di nuovo. Se invece ci imbattiamo in un epsilon-arco, accade qualcosa di simile, senza leggere nessun input la macchina si divide in più copie, una che segue ciascun arco uscente etichettato con ε e una che resta nelllo stato corrente.

Definizione formale di automa finito non deterministico

Un DFA e un NFA hanno una definizione formale molto simile, essi infatti differiscono solo nella funzione di transizione, in un DFA quest'ultima prende uno stato e un simbolo in input e produce lo stato successivo, mentre in un NFA prende uno stato e un simbolo in input (o ε) e produce l'insieme dei possibili stati successivi.

Un automa finito non deterministico è una quintupla (Q, Σ, δ, q_0, F), dove

1. Q è un insieme finito chiamato insieme degli **stati**
2. Σ è un insieme finito chiamato **alfabeto**
3. δ : Q x $\Sigma_\varepsilon \to \mathcal{P}(Q)$ è la **funzione di transizione**
4. $q_0 \in$ Q è lo **stato iniziale**
5. F $\subseteq$ Q è l'**insieme degli stati accettanti**

teorema Per ogni automa finito non deterministico esiste un automa finito deterministico equivalente (due macchine sono **equivalenti** se esse riconoscono lo stesso linguaggio).

2.5 Espressioni Regolari

In aritmetica possiamo usare i simboli + e $\times$ per costruire espressioni del tipo:

$$(5 + 3) \times 4$$

Analogamente, possiamo usare le nostre operazioni regolari per costruire espressioni che descrivono linguaggi, chiamate **Espressioni Regolari**, come:

$$(0 \cup 1)\ 0^*$$

Il valore dell'espressione aritmetica scritta sopra è 32, mentre il valore dell'espressione regolare è un linguaggio, in questo caso un linguaggio che consiste in **tutte le stringhe che iniziano con uno 0 o un 1 seguito da qualsiasi numero di simboli uguali a 0**.

Definizione Formale di Espressione Regolare

Diciamo che R è un espressione regolare se è:

1. a per qualche a nell'alfabeto Σ;
2. ε;
3. $\emptyset$;

4. $(R_1 \cup R_2)$, dove R_1 ed R_2 sono espressioni regolari;
5. $(R_1 \circ R_2)$, dove R_1 ed R_2 sono espressioni regolari;
6. (R_1^*), dove R_1 è un'espressione regolare.

Osservazione Non bisogna confondere le espressioni regolari ε e $\emptyset$, in quanto la prima rappresenta il linguaggio che contiene una sola stringa, ovvero la stringa vuota, mentre la seconda rappresenta il linguaggio che non contiene nessuna stringa.
L'ordine in una espressione regolare risulta essere il seguente:

Star, Concatenazione, Unione

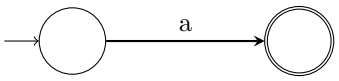
Teorema
Un linguaggio è regolare se e solo se qualche espressione regolare lo descrive

Questo teorema deve essere dimostrato in entrambe le direzioni, quindi enunciamo ogni direzione in un lemma separato.

Lemma
Se un linguaggio è descritto da un'espressione regolare, allora esso è regolare

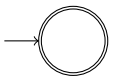
Dimostrazione →
Supponiamo di avere un'espressione regolare R che descrive un linguaggio A. Mostriamo come trasformare R in un NFA che riconosce A, in quanto se un NFA riconosce A allora A è regolare.
Per farlo, consideriamo i 6 casi nella definizione di espressione regolare.

1. R = a per qualche a $\in \Sigma$. Allora L(R) = $\{a\}$ e il seguente NFA riconosce L(R).



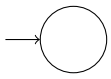
   Formalmente, N = $(\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$

2. R = ε. Allora L(R) = $\{\varepsilon\}$ e il seguente NFA riconosce L(R).



   Formalmente, N = $(\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$

3. R = $\emptyset$. Allora R = $\{\emptyset\}$, e il seguente NFA riconosce L(R).



Formalmente, N = ($\{q\}$, Σ, δ, q, $\{\emptyset\}$)

4. R = $R_1 \cup R_2$.
5. R = $R_1 \circ R_2$.
6. R = R_1^*.

Per gli ultimi tre casi, sappiamo che la classe dei linguaggi regolari è chiusa rispetto alle operazioni regolari, quindi non dovremo far altro che costruire l'NFA a partire da R visti nei primi 3 punti.

Lemma

Se un linguaggio è regolare, allora è descritto da un'espressione regolare.

Dimostrazione ←

Dobbiamo dimostrare che se un linguaggio A è regolare, allora un'espressione regolare lo descrive. Poichè A è regolare, esso è accettato da un DFA. Descriviamo quindi una procedura che permette di trasformare i DFA in espressioni regolari equivalenti.

Dividiamo questa procedura in 2 parti, usando un nuovo tipo di automa finito chiamato **Automa Finito non Deterministico Generalizzato** GNFA.

Prima mostriamo come trasformare un DFA in un GNFA e poi come trasformare un GNFA in un'espresione regolare.

Un GNFA legge blocchi di simboli dall'input, non necessariamente solo un simbolo alla volta come in un comune NFA.Per comodità, richiediamo che ogni GNFA abbia sempre una forma speciale che soddisfi le seguenti condizioni:

1. Lo stato iniziale ha archi di transizione uscenti verso un qualsiasi altro stato ma nessun arco entrante proveniente da altri stati.
2. Esiste un solo stato accettante, ed esso ha archi entranti provenienti da un qualsiasi altro stato ma nessun arco uscente verso altri stati, inoltre, lo stato accettante è diverso dallo stato iniziale.
3. Eccetto che per lo stato iniziale e lo stato accettante, un arco va da ogni stato ad ogni altro stato e anche da ogni stato in sé stesso.

Possiamo facilmente trasformare un DFA in un GNFA aggiungendo semplicemente un nuovo stato iniziale con un ε-arco che entra nel vecchio stato iniziale e un nuovo stato accettante con ε-archi entranti provenienti dai vecchi stati accettanti.

Se alcuni archi hanno più etichette, sostituiamo ognuno di essi con un solo arco la cui etichetta è l'unione delle precedenti etichette. Infine, aggiungiamo archi con etichetta $\emptyset$ tra stati che non hanno archi.

Successivamente, mostriamo come trasformare un GNFA in un'espressione regolare. Supponiamo che il GNFA abbia k stati. Allora, poiché un GNFA deve avere uno stato iniziale ed uno finale distinti, sappiamo che k $\geq$ 2.

Se k $>$ 2, costruiamo un GNFA equivalente con $k-1$ stati. Questo passo può essere svolto fino a quando il nostro GNFA avrà solo due stati, quindi stato iniziale e stato accettante, e l'etichetta dell'arco di transizione che li collega sarà proprio la nostra **Espressione Regolare**.

MANCA FINE DIMOSTRAZIONE

2.6 Linguaggi Non Regolari

Possiamo dire con certezza se tutti i linguaggi sono regolari oppure no? Si, ad esempio sappiamo che:

$$\text{L} = \{0^n 1^n : n \geq 0\}$$
NON è un linguaggio regolare.

Possiamo facilmente vedere che in un DFA con numero di stati = n, ricevendo un input w con $|w| > n$, ci dovranno essere degli stati ripetuti.

2.6.1 Pumping Lemma

La nostra tecnica per provare se un linguaggio sia regolare o meno deriva da un teorema sui linguaggi regolari, chiamato Pumping Lemma. Questo teorema afferma che tutti i linguaggi regolari hanno una proprietà speciale.

La proprietà afferma che tutte le stringhe nel linguaggio possono essere 'replicate' se la loro lunghezza raggiunge almeno uno specifico valore, chiamato **Lunghezza del Pumping**. Questo significa che ogni tale stringa contiene una parte che può essere ripetuta un numero qualsiasi di volte ottenendo una stringa che appartiene ancora al linguaggio.

Teorema: Pumping Lemma

Se A è un linguaggio regolare, allora esiste un numero p (la lunghezza del pumping) tale che se s è una qualsiasi stringa in A di lunghezza almeno p, allora s può essere scomposta in tre parti, s = xyz, che soddisfano le seguenti condizioni:

1. per ogni i $\geq$ 0, $xy^iz \in A$,
2. $|y| > 0$,
3. $|xy| \leq p$.

Dimostrazione

Sia M = (Q, Σ, δ, q_1, F) un DFA che riconosce A e sia p il numero di stati di M.

Sia s = $s_1 s_2 \cdots s_n$ un stringa in A di lunghezza n dove n $\geq$ p. Sia $r_1, ..., r_n + 1$ la sequenza di stati attraversati da M mentre elabora s, quindi $\delta(r_i, s_i) = r_i + 1$ per i compreso tra 1 ed n. Questa sequenza ha lunghezza n + 1, che è almeno p + 1. Due tra i primi p + 1 elementi nella sequenza devono essere lo stesso stato, per il **principio della piccionaia**.

Chiamiamo il primo di questi r_i e il secondo r_j. Poichè r_j si presenta tra le prime p + 1 posizioni in una sequenza che inizia in r_1, abbiamo j $\leq$ p + 1.

Ora sia x = $s_1 ... s_i - 1$, y = $s_i ... s_j - 1$ e z = $s_j ... s_n$.

Poichè x porta M da r_1 a r_i, y porta M da r_i a r_i e z porta M da r_i a $r_n + 1$, che è uno stato accettante, M deve accettare $xy^i z$ per i $\geq$ 0. Sappiamo che i $\neq$ j, perciò $|y| > 0$

e j $\leq$ p + 1, perciò $|xy| \leq p$. Quindi tutte le condizioni del pumping lemma sono rispettate.

3 Linguaggi Context-Free

Presentiamo ora le **Grammatiche Context-Free**, un metodo più potente per scrivere linguaggi. Queste grammatiche possono descrivere alcuni aspetti che hanno una struttura ricorsiva, che le rende utili in una varietà di applicazioni.

Un'importante applicazione delle grammatiche si trova nella specifica e nella compilazione dei linguaggi di programmazione, infatti la maggior parte dei compilatori e degli interpreti contiene una componente chiamata **Parser** che estrae il significato di un programma prima di generare il codice compilato o di eseguire il programma interpretato.

I linguaggi associati alle grammatiche context-free sono chiamati **Linguaggi Context-Free**. La classe di questi linguaggi include tutti i linguaggi regolari e molti ulteriori linguaggi.

In questo capitolo diamo una definizione delle grammatiche context-free e studiamo le proprietà dei linguaggi context-free. Introduciamo anche gli **Automi a Pila** (pushdown automata), una classe di macchine che riconoscono i linguaggi enunciati poco fa.

3.1 Grammatiche Context-Free

Un esempio di grammatica context-free, che chiamiamo G_1, è il seguente:

$$\begin{aligned} \text{A} &\rightarrow \text{0A1} \\ \text{A} &\rightarrow \text{B} \\ \text{B} &\rightarrow \# \end{aligned}$$

Una grammatica consiste in una serie di regole di **sostituzione**, anche chiamate **Produzioni**. Ogni regola appare come una linea nella grammatica, costituita da un simbolo e una stringa separati da una freccia.

Il simbolo è chiamato **Variabile**, mentre la stringa consiste di variabili e altri simboli chiamati **Terminali**.

Le variabili sono spesso rappresentate da lettere maiuscole, mentre i terminali sono rappresentati da lettere minuscole, numeri o simboli speciali.

Una delle variabili è chiamata **Variabile Iniziale** (solitamente è quella al lato sinistro della regola più in alto)

Una grammatica può essere usata per descrivere un linguaggio generando ogni stringa del linguaggio nel seguente modo:

1. Scrivi la variabile iniziale.
2. Trova una variabile che è stata scritta e una regola che inizia con quella variabile, sostituisci la variabile scritta con il lato destro di quella regola.
3. Ripeti il passo 2 fino a quando non ci sono più variabili.

Una sequenza di sostituzioni per ottenere una stringa è chiamata **Derivazione**.

Tutte le stringhe generate in questo modo costituiscono il **Linguaggio della Grammatica**.

Denoteremo con L(G_1) il linguaggio della grammatica G_1, che risulta essere $\{0^n\#1^n \mid n \geq 0\}$. Ogni linguaggio che può essere generato da una grammatica context-free è chiamato **Linguaggio Context-Free** (CFL).

Definizione Formale di Grammatica Context-Free

Una grammatica context-free è una quadrupla (V, Σ, R, S), dove

1. **V** è un insieme finito di variabili,
2. Σ è un insieme finito, disgiunto da V, i cui elementi sono chiamati terminali,
3. **R** è un insieme finito di regole,
4. **S** ∈ V, ovvero la variabile iniziale.

---

se u, v e w sono stringhe di variabili e terminali e A → w è una regola della grammatica, diciamo che uAv **Produce** uwv, e lo denotiamo come uAv → uwv.

Diciamo invece che u **Deriva** da v, e lo denotiamo con u →* v, se u = v o se esiste una sequenza $u_1, u_2, \ldots, u_k$ con k ≥ 0 e

$$u \to u_1 \to u_2 \to \ldots \to u_k \to v$$

Qualche volta una grammatica può generare la stessa stringa in più modi diversi. Tale stringa avrà dunque diversi alberi sintattici e quindi diversi significati. Questo risultato può essere indesiderabile per alcune applicazioni, come i **linguaggi di programmazione**, che dovrebbero avere un'unica interpretazione.

Se una grammatica genera la stessa stringa in più modi diversi, diremo che la stringa è derivata **ambiguamente** in quella grammatica.

Se una grammatica genera alcune stringhe ambiguamente, diremo che la grammatica è **ambigua**.

Formalizziamo quindi la nozione di ambiguità. Quando diciamo che una grammatica genera ambiguamente una stringa, intendiamo che la stringa ha almeno due diversi alberi sintattici, non due diverse derivazioni, in quanto quest'ultime possono differire solo nell'ordine in cui esse sostituiscono le variabili ma non nella loro struttura complessiva.

Per concentrarci sulla struttura, definiamo un tipo di derivazione che sostituisce le variabili con un ordine stabilito: la **Derivazione a Sinistra** (leftmost derivation), che ad ogni passo sostituisce la variabile più a sinistra.

3.2 Forma Normale di Chomsky

Quando si lavora con le grammatiche context-free, è spesso conveniente averle in forma semplificata. Una delle forme più semplici e utili si chiama forma normale di Chomsky.

Definizione

Una grammatica context-free è in **forma normale di Chomsky** se ogni regola è della forma

$$A \to BC$$
$$A \to a$$

dove 'a' è un terminale e 'A, B, C' sono variabili qualsiasi, tranne che B e C non possono essere la variabile iniziale.

Inoltre, permettiamo la regola $S \to \varepsilon$, dove S è la variabile iniziale.

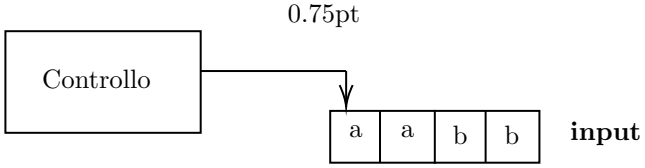
3.3 Automi a Pila

In questa sezione introduciamo un nuovo tipo di modello computazionale, chiamato **automa a pila** (pushdown automata). Questi automi sono come gli automi finiti non deterministici ma hanno una componente in più, chiamata **pila** (stack). La pila fornisce memoria aggiuntiva, che consente a tali automi di riconoscere alcuni linguaggi non regolari.

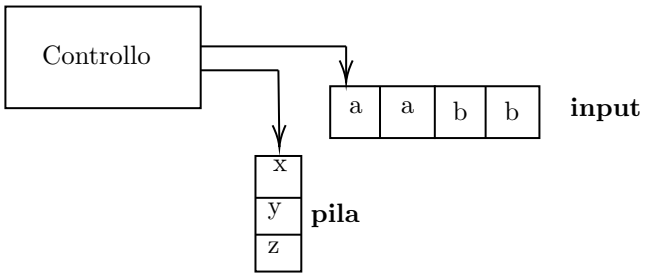
Gli automi a pila sono computazionalmente equivalenti alle grammatiche context-free.

La figura seguente è una rappresentazione schematica di un automa finito. Il controllo rappresenta gli stati e la funzione di transizione, il nastro contiene la stringa di input e la freccia rappresenta la testina sull'input, che indica il successivo simbolo da leggere.

0.75pt



Aggiungendo l'elemento pila, otteniamo una rappresentazione schematica di un automa a pila, come mostrato nella figura seguente.



Un automa a pila (PDA) può scrivere simboli nella pila e rileggerli in seguito. Scrivere un simbolo 'spinge giù' tutti gli altri simboli nella pila. In un qualunque momento il simbolo sulla cima (top) della pila può essere letto e rimosso, facendo risalire tutti gli altri simboli.

L'operazione di scrittura sulla pila viene chiamata **push**, mentre quella di eliminarne uno viene chiamata **pop**. Ricordiamo che ogni operazione, sia di scrittura che di lettura, può essere effettuato solo sulla sommità.

La pila risulta essere molto importante perché permette di memorizzare una quantità non limitata di informazioni. Ricordiamo che un automa finito non può riconoscere il linguaggio $\{0^n1^n | \ n \geq 0\}$ poiché non è in grado di fare ciò che abbiamo appena enunciato.

Definizione Formale di Automa a Pila

un automa a pila è una sestupla (Q, Σ, Γ, δ, q_0, F) dove Q, Σ, Γ ed F sono tutti insiemi finiti e:

1. Q è l'insieme degli stati,
2. Σ è l'alfabeto dell'input,
3. Γ è l'alfabeto della pila,
4. δ : Q $\times\Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$ è la funzione di transizione,
5. $q_0 \in Q$ è lo stato iniziale,
6. F $\subseteq$ Q è l'insieme degli stati accettanti.

3.3.1 Equivalenza con le Grammatiche Context-Free

In questa sezione mostriamo che grammatiche context-free e automi a pila sono computazionalmente equivalenti, mostreremo come trasformare ogni grammatica context-free in un automa a pila che riconosce lo stesso linguaggio e viceversa.

Ricordiamo che abbiamo definito un linguaggio context-free come un linguaggio che può essere descritto con una grammatica context-free, quindi il nostro obiettivo è il teorema seguente.

Teorema

Un linguaggio è context-free se e solo se esiste un automa a pila che lo riconosce.

Andiamo a vedere dunque entrambi i lati del 'se e solo se':

Lemma →

Se è un linguaggio è context-free, allora esiste un automa a pila che lo riconosce.

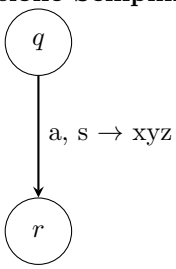
Idea e Dimostrazione →

IDEA. Sia A un CFL. Dalla definizione sappiamo che esiste una CFG, G, che genera A. mostriamo come trasformare G in un PDA equivalente, che chiamiamo P.

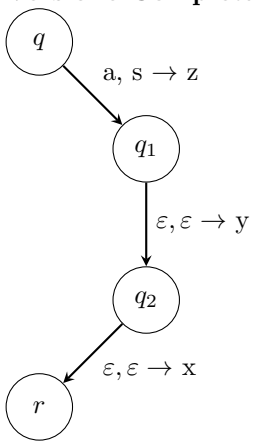
DIMOSTRAZIONE. Ora diamo i dettagli formali della costruzione dell'automa a pila P = (Q, Σ, Γ, δ, q_start, F). Per rendere più chiara la costruzione, usiamo una notazione abbreviata per la funzione di transizione. Questa notazione fornisce un modo per scrivere un'intera stringa sulla pila in un passo della macchina.

questa notazione è (r, u) $\in \delta$(q, a, s), denota che quando q è lo stato in cui si trova l'automa, a è il prossimo simbolo di input ed s è il simbolo sulla cima della pila, il PDA può leggere a ed eliminare s, poi inserire la stringa u nella pila e passare allo stato r, la figura seguente mostra la realizzazione.

Versione Semplificata

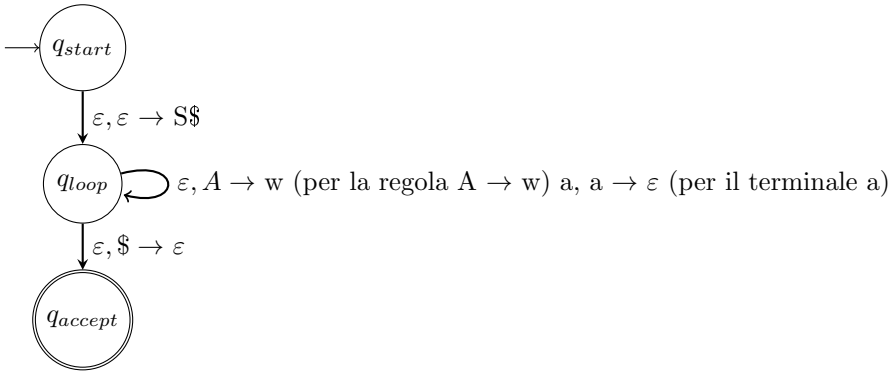


Versione Completa



Gli stati di P sono Q = $\{q_{start}, q_{loop}, q_{accept}\} \cup$ E, dove E è l'insieme degli **stati intermedi** (come q_1, q_2 nel disegno sopra).

Il suo diagramma di stato è mostrato nella figura seguente.



Questo prova come un PDA può riconoscere un linguaggio context-free, passiamo ora alla seconda implicazione.

Lemma ←

Se un linguaggio è riconosciuto da un automa a pila, allora esso è context-free.

Idea e Dimostrazione ←

IDEA. Abbiamo un PDA P e vogliamo costruire una CFG G che genera tutte le stringhe che P accetta. In altre parole, G dovrebbe generare una stringa se quella stringa fa andare il PDA dal suo stato iniziale ad uno stato accettante.

Per raggiungere questo risultato, progettiamo una grammatica che fa un po' in più. Per ciascuna coppia di stati p e q in P, la grammatica avrà una variabile

A_{pq}. Questa variabile genera tutte le stringhe che possono portare P da p con pila vuota a q con pila vuota.

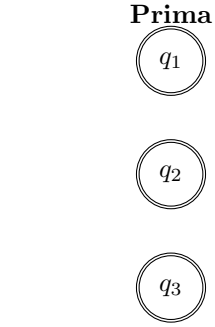
Innanzitutto, semplifichiamo il nostro compito modificando leggermente P per munirlo delle seguenti 3 caratteristiche:

1. Ha un unico stato accettante, q_{accept},
2. Svuota la sua pila prima di accettare,
3. Ciascuna transizione inserisce un simbolo sulla pila o ne elimina uno dalla pila, ma non può fare entrambe le cose.

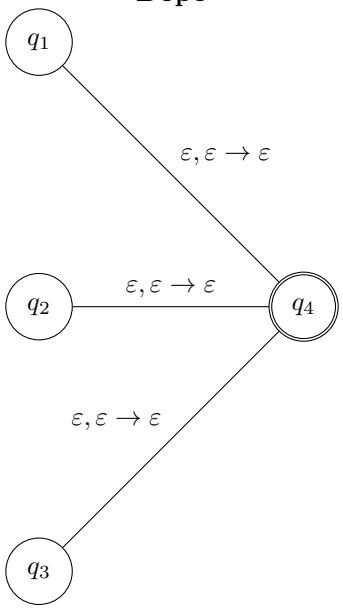
Per dare a P queste caratteristiche faremo come rappresentato nelle immagini seguenti:

Caratteristica 1

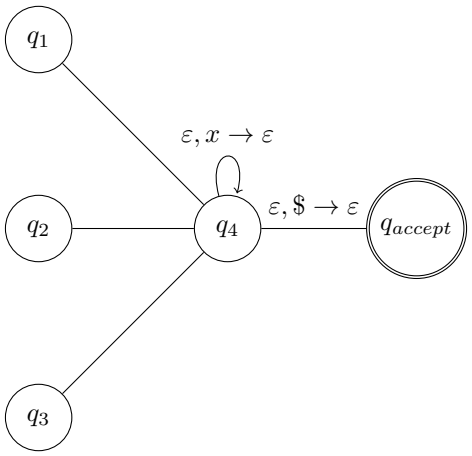
Prima



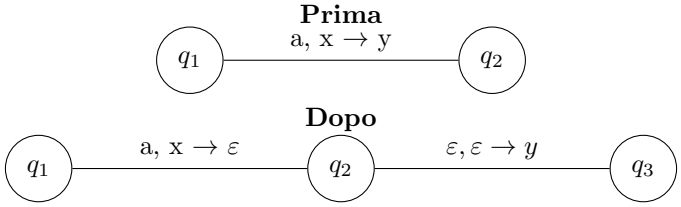
Dopo



Caratteristica 2



Caratteristica 3



Successivamente, per progettare G in modo tale che A_{pq} generi tutte le stringhe che portano P da p a q, iniziando e terminando con una pila vuota, dobbiamo capire come P agisce su queste stringhe. Per ognuna di queste stringhe x, la prima mossa di P su x deve essere un push, poiché P non può eliminare da una pila vuota. Analogamente, l'ultima mossa di P sarà un pop perché alla fine la pila deve essere vuota.

Durante la computazione di P su x si presentano due eventualità:

1. il simbolo eliminato alla fine è il simbolo inserito all'inizio, quindi la pila non si è svuotata nel mezzo;
2. il simbolo eliminato alla fine **non** è il simbolo inserito all'inizio, quindi la pila si è svuotata nel mezzo.

Simuliamo la prima possibilità con la regola

$$A_{pq} \to aA_{rs}b$$

Dove a è l'input letto nella prima mossa, b è l'input letto nell'ultima mossa, r è lo stato che segue p e s è lo stato che precede q.

Simuliamo la seconda possibilità con la regola

$$A_{pq} \to A_{pr}A_{rq}$$

Dove r è lo stato in cui la pila si svuota.
DIMOSTRAZIONE.
RICORDA DI FINIRE LA DIMOSTRAZIONE

3.4 I Linguaggi Non Context-Free

4 Calcolabilità La tesi di Church-Turing

Finora nello sviluppare la teoria della computazione abbiamo presentato vari modelli di dispositivi di calcolo. Gli automi finiti sono un buon modello per quei dispositivi che hanno poca memoria, gli automi a pila costituiscono un buon modello per quei dispositivi che hanno una memoria illimitata, ma utilizzabile solo nelle modalità di una pila. Abbiamo dimostrato però che alcuni calcoli vanno oltre le capacità di questi modelli, quindi serve trovare un modello 'più universale'.

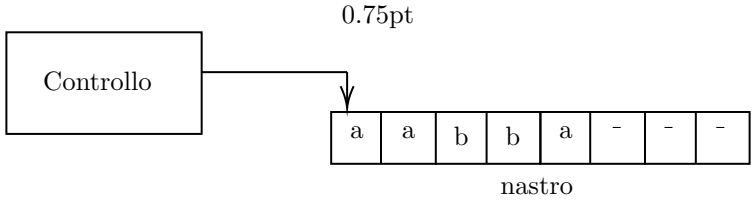
4.1 Macchine di Turing

Introduciamo ora un modello molto più potente, proposto per la prima volta nel 1936 da Alan Turing, chiamato **macchina di Turing**. Simile ad un automa finito, ma con una memoria illimitata e senza restrizioni, una macchina di Turing è un modello molto più preciso di un computer. Questo modello è in grado di fare tutto ciò che un computer reale può fare. Tuttavia, anche una macchina di Turing non è in grado di risolvere alcuni problemi, poiché essi vanno oltre i limiti teorici della computazione.

L'elenco seguente riassume le differenze tra automi finiti e macchine di Turing:

1. Una macchina di Turing può sia leggere che scrivere sul nastro.
2. La testina di lettura-scrittura può muoversi sia verso sinistra che verso desra.
3. Il nastro è infinito.
4. Gli stati di accettazione e rifiuto hanno effetto immediato.

Ecco uno schema di una macchina di turing:



Il cuore della definizione di una macchina di Turing è la funzione di transizione δ perché essa ci dice come la macchina effettua un passo. Per una macchina di Turing, δ prende la forma

$$Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$

Cioè, quando la macchina occupa un certo stato q e la testina punta alla casella del nastro contenente un simbolo a e se δ(q, a) = (r, b, L), la macchina scriverà il simbolo b al posto di a, passerà allo stato r e la testina si sposterà verso sinistra dopo la scrittura.

Definizione Formale di Macchina di Turing

Una **Macchina di Turing** è una 7-upla (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}) dove Q, Σ e Γ sono tutti insiemi finiti e:

1. Q è l'insieme degli stati,
2. Σ è l'alfabeto di input non contenente il simbolo **blank**,
3. Γ è l'alfabeto di nastro, con **blank** $\in \Gamma$ e $\Sigma \subseteq \Gamma$,
4. δ: $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ è la funzione di transizione,
5. q_0 è lo stato iniziale,
6. $q_{accept} \in Q$ è lo stato di accettazione,
7. $q_{reject} \in Q$ è lo stato di rifiuto, con $q_{rej} \neq q_{acc}$.

4.2 Computazione e Configurazione di una Turing Machine

Una macchina di Turing M = (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}) **computa** nel seguente modo: inizialmente M riceve il suo input $w_1 w_2 ... w_n \in \Sigma^*$ sulle n celle più a sinistra del nastro, mentre il resto del nastro contiene tutti simboli **blank**. La testina parte dalla posizione più a sinistra del nastro. Si noti che Σ non contiene il simbolo blank, così il primo di questi simboli segna la fine dell'input.

La computazione finisce quando M entra in uno stato accettante o in uno di rifiuto, altrimenti M entra in **loop**. Durante la computazione di una macchina M si verificano cambiamenti dello stato corrente, del contenuto corrente del nastro e della posizione corrente della testina. Un'impostazione di questi tre elementi è chiamata **configurazione** della macchina di Turing. Per uno stato q e due stringhe u, v sull'alfabeto Γ del nastro, scriviamo 'u q v' per indicare la configurazione, dove:

1. lo stato corrente è q,

2. il contenuto corrente del nastro è u v,

3. la posizione corrente della testina è il primo simbolo di v.

Dopo l'ultimo simbolo di v, il nastro contiene solo simboli blank.

Formalizziamo ora come una macchina di Turing **computa**. Si dice che la configurazione C_1 **produce** C_2 se la TM può passare da C_1 a C_2 in un unico passo. Definiamo formalmente questa nozione nel seguente modo.

Supponiamo di avere a, b e c $\in \Gamma$, così come u e v $\in \Gamma^*$ e gli stati $q_i e q_j$.

In tal caso 'u a q_1 b v' e 'u q_j a c v' sono due configurazioni. Diciamo che

$$\underline{\text{u a } q_1 \text{ b v}} \textbf{ produce } \underline{\text{u } q_j \text{ a c v}}$$

se nella funzione di transizione $\delta(q_i, \text{b}) = (q_j, \text{c}, \text{L})$. Questo nel caso in cui la macchina di Turing effettua uno spostamento verso sinistra.

4.3 Linguaggi Turing-Riconoscibili e Turing-Decidibili

L'insieme di stringhe che M accetta rappresenta il **linguaggio di M**, o il **linguaggio riconosciuto da M**, denotato come L(M).

Definizione

Un linguaggio si dice **Turing-riconoscibile** se esiste una macchina di Turing che lo riconosce.

Quando attiviamo una TM su un input, tre risultati sono possibili. La macchina può:

1. accettare,

2. rifiutare,

3. andare in loop.

A volte distinguere una macchina che è entrata in un loop da una che sta semplicemente impiegando tanto tempo risulta difficile. Per questo si preferiscono macchine di Turing che che si fermano su ogni singolo input, ovvero che non vanno mai in loop.

Una tale macchina viene chiamata **decisore** perché in ogni caso prende una decisione, sia essa di accettare o rifiutare l'input. Diciamo che un decisore **decide** un certo linguaggio se riconosce tale linguaggio.

Definizione

Un linguaggio si dice **Turing-decidibile** o semplicemente **decidibile** se esiste una TM che lo decide.

4.4 Varianti di Macchine di Turing

Esistono molte definizioni alternative di macchine di Turing, comprese le versioni multinastro o non-deterministiche. Esse sono dette **varianti** del modello macchina di Turing. Il modello originale e le sue varianti hanno tutte **lo stesso potere computazionale**, ovvero riconoscono la stessa classe di linguaggi.

In questa sezione descriviamo alcune varianti e le relative prove di equivalenza in termini di potenzialità.
Un primo esempio può essere una variante in cui, ad ogni passo computazionale, la macchina può non dover muovere la testina, modificando così la funzione di transizione da

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$$
in
$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

Questa caratteristica non permette in nessuno modo alle macchine di Turing create in questo modo di riconoscere nuovi linguaggi.

Questo perché possiamo dividere in due transizioni (prima muovo la testina verso sinistra, poi la riporto a destra qualsiasi simbolo io legga e senza scrivere nulla) la transizione in cui la testina deve rimanere ferma.

Questo esempio contiene la chiave per poter mostrare l'equivalenza di varianti di TM. Per dimostrare che due macchine sono equivalenti, abbiamo bisogno di dimostrare che **ognuno dei due modelli può simulare l'altro**.

4.4.1 Macchina di Turing Multinastro

Una macchina di Turing multinastro è essenzialmente una generalizzazione della macchina di Turing a nastro singolo. Ogni nastro ha la sua testina per la lettura/scrittura.

Inizialmente l'input si trova sul nastro 1, mentre gli altri nastri sono vuoti. La funzione di transizione viene modificata per consentire la scrittura, la lettura e lo spostamento della testina contemporaneamente su alcuni o tutti i nastri, formalmente:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$$

dove k è il numero di nastri. L'espressione

$$\delta(q_i, a_1, ..., a_k) = (q_j, b_1, ..., b_k, L, R, ..., L)$$

significa che, se la macchina si trova nello stato q_i e le testine da 1 a k leggono i simboli da a_1 a a_k, la macchina va nello stato q_j, scrive i simboli da b_1 a b_k e muove ogni testina a sinistra o a destra, o la fa restare ferma, come specificato.

Teorema

Per ogni macchina di Turing multinastro esiste una macchina di Turing a nastro singolo equivalente.

Dimostriamo ora che questa variante di TM non aggiunge potenza alla TM base, per farlo ricordiamo che due macchine sono equivalenti se riconoscono lo stesso linguaggio.

DIMOSTRAZIONE. Mostriamo come convertire una macchina di Turing multinastro M in una TM S equivalente a nastro singolo. L'idea chiave è di mostrare come **simulare** M con S.

Supponiamo che M abbia k nastri, allora S simula l'effetto di k nastri memorizzando le loro informazioni sul suo singolo nastro. Essa utilizza un nuovo simbolo '#' come delimitatore per separare i contenuti dei diversi nastri. Invece, per tenere traccia delle posizioni delle testine, S marca il simbolo in cui dovrebbe essere posizionata la testina per tutti i contenuti dei 'nastri virtuali'.

4.4.2 Macchina di Turing non Deterministica

Una macchina di Turing non deterministica è definita come ci si aspetta. In ogni passo della computazione la macchina può procedere effettuando varie scelte. Per questo la funzione di transizione va modificata e risulta essere del tipo:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Teorema

Per ogni macchina di Turing non deterministica esiste una macchina di Turing deterministica equivalente.

DIMOSTRAZIONE. La TM D deterministica che simula la TM N non deterministica ha tre nastri. D utilizza questi tre nastri in maniera particolare.

1. Il primo nastro contiene sempre la stringa di input e non viene mai modificato
2. Il secondo nastro mantiene una copia del nastro di N corrispondente a qualche diramazione non deterministica
3. Il terzo nastro tiene traccia della posizione di D nell'albero delle computazioni di N

4.4.3 Enumeratori

RICORDA DI FINIRE QUESTA PARTE

5 Decidibilità

In questo capitolo cominciamo ad investigare la potenza degli algoritmi nella risoluzione di problemi.

Dimostreremo che alcuni problemi possono essere risolti in maniera algoritmica ed altri no. Il nostro obiettivo è esplorare i limiti della risolvibilità algoritmica dei problemi.

5.1 Linguaggi Decidibili

In questa sezione daremo degli esempi di linguaggi decidibili mediante algoritmi.

5.1.1 Problemi decidibili relativi a linguaggi regolari