Indice

Capitolo 1

Intro

Studieremo dei sistemi particolari (macchine/grammatiche) e il loro comportamento (il linguaggio descritto) e le risorse utilizzate (risorse utilizzate, risorse per trasformazioni da una macchina e l'altra, …).

1.1 Terminologia

Quando si parla di un linguagio ci sono due aspetti fondamentali:

- sintassi, formata da
  - i segni o simboli utilizzati
  - le regole che ci permettono di combinarli
- semantica, il processo che associa alle frasi del linguaggio un significato

Noi ci occuperemo principalmente di sintassi.

In ambito linguistico nel 1956 Chomsky introduce le grammatiche formali. Questi sono degli oggetti matematici, il cui scopo originale era descrivere in modo rigoroso la grammatica dell'inglese. Nello stesso periodo si stava sviluppando l'idea di compilatore, specificamente il compilatore Fortran. Si vide che il modello proposto da Chomksy era adatto per formalizzare i linguaggi di programmazione.

Definiamo

- l'alfabeto (Σ) è un insieme finito non vuoto di simboli, e.g.

$$\Sigma = \{a, b, c\}$$

- una stringa o parola è una sequenza di simboli sull'alfabeto. La sua lunghezza è definita come il numero di simboli che la compongono

  $$|abc| = 3$$

  Tra tutte le parole abbiamo una parola speciale, che è la parola vuota costituita da zero simboli. Questa è indicata con ϵ (alcuni la indicano con λ o Λ). La lunghezza di questa è 0.

  Sia $a \in \Sigma$ e $x \in \Sigma^*$, definiamo

  $$\#_a(x) : \Sigma \times \Sigma^* \mapsto \mathbb{N}$$

  come il numero di a in x.

- indichiamo l'insieme di tutte le possibili stringhe sull'alfabeto Σ con Σ^*. Queste sono enumerabili.

  Chiamiamo Σ^n l'insieme di tutte le stringhe di n lettere sull'alfabeto Σ.

- definiamo un'operazione

  $$- \cdot - : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$$

  come la concatenazione o prodotto di due stringhe. Ad esempio

  $$\begin{aligned} x &= abbac \\ y &= ca \\ xy &= abbacca \\ yx &= caabbac \end{aligned}$$

  La parola vuota ϵ è l'identità sia destra che sinistra di questa operazione. L'operazione è associativa ma non commutativa. La struttura $\langle \Sigma^*, \cdot, \epsilon \rangle$ è detto monoide libero generato dall'alfabeto Σ.

- data una stringa $x \in \Sigma^*$ diciamo che y è prefisso di x se

  $$\exists z \in \Sigma^* \mid x = yz$$

  Una stringa ha un numero di prefissi pari alla sua lunghezza + 1. y è prefisso proprio di x se è prefisso e non è ϵ. Similmente definiamo il suffisso di una parola x, quindi y è suffisso di x se

  $$\exists x \in \Sigma^* \mid x = zy$$

Infine diciamo che y è fattore di x se

$$\exists z, w \in \Sigma^* \mid x = zyw$$

I prefissi e i suffissi sono particolari tipi di fattori. Ogni parola ha all'incirca $\frac{n^2}{2}$ fattori, infatti è facile notare che $x = aaaa$ ha diversi fattori uguali.

Una sottosequenza di solito è una serie di elementi scelti da una parola, quindi un fattore può essere definito come una sottosequenza contigua. Ad esmepio una sottosequenza di $x = abbac$ può essere abc. Sottostringa e sottoparola di solito sono sinonimi di fattore, ma a volte possono essere utilizzate come sinonimo di sottosequenza.

- un linguaggio L è definito come un qualunque sottoinsieme di Σ^*. Abbiamo alcuni linguaggi particolari:
  - il linguaggio vuoto $L = \varnothing$
  - il linguaggio della parola vuota $L = \{\epsilon\}$
  - il linguaggio pieno $L = \Sigma^*$

Ad esempio definiamo il linguaggio delle parole che finiscono con due a come

$$L = w \in \Sigma^* \mid \exists y \in \Sigma^* w = yaa$$

quindi con un insieme finito di simboli siamo risuciti a rappresentare un insieme infinito. Questo tipo di descrizione è detta *dichiarativa*.

In altri casi utilizzeremo delle descrizioni:

- *generative*: si fornisce un insieme di regole che permette di costruire tutte le possibili stringhe del linguaggio, ad esempio le grammatiche
- *riconoscitive*: si fornisce un riconoscitore che riceve una stringa x e risponde alla domanda $x \stackrel{?}{\in} L$, in alcuni casi queste macchine si limitano a dire sì se la stringa appartiene o non terminare in altro caso

Fatto 1. *Non tutti i linguaggi si possono descrivere in maniera finita, infatti le descrizioni finite sono numerabili, mentre i linguaggi no.*

Prendiamo come alfabeto $\Sigma = \{(,)\}$, vogliamo descrivere L l'insieme delle sequenze di parentesi bilanciate correttamente. Definiamo la rappresentazione generativa che genera L_G:

1. $\epsilon \in L_G$
2. se $x \in L_G$ allora $(x) \in L_G$
3. se $x, y \in L_G$ allora $xy \in L_G$

E definiamo un riconoscitore, che riconosca L_R:

- il numero di aperte deve essere uguale al numero di chiuse, quindi $\#_(x) = \#_)(x)$
- per ogni prefisso il numero di parentesi aperte è maggiore o uguale del numero di chiuse, quindi per y prefisso $\#_(y) \geq \#_)(y)$.

Questo può essere generato come un automa con contatore.
Si può dimostrare che questi due definiscono lo stesso linguaggio.

Ora prendiamo l'alfabeo $\Sigma = \{(,), [,]\}$, e definiamo L come il linguaggio delle sequenze di parentesi bilanciate correttamente. Definiamo la rappresentazione generativa che genera L_G:

1. $\epsilon \in L_G$
2. se $x \in L_G$ allora $(x) \in L_G$ e $[x] \in L_G$
3. se $x, y \in L_G$ allora $xy \in L_G$

E definiamo un riconoscitore, che riconosca L_R:

- il numero di aperte deve essere uguale al numero di chiuse, quindi $\#_(x) = \#_)(x)$ e $\#_[(x) = \#_](x)$
- si utilizza una pila in cui si mette ogni parentesi aperta e si toglie quando si trova una parentesi chiusa nel caso corrisponda

Quindi un automa a pila.
In alcuni casi è più facile definire come si genera, e in altri come riconoscere.

Capitolo 2

Grammatiche

Una grammatica è una quadrupla $\langle V, \Sigma, P, S \rangle$, dove

- V è l'insieme finito e non vuoto delle variabili, questi vengono anche chiamate non terminali. Questo lo possiamo chiamare anche l'alfabeto delle variabili.
- Σ è l'insieme finito e non vuoto dei termianli, o anche alfabeto dei terminali.
- P è l'insieme finito delle regole di produzione. Queste sono della forma

  $$\alpha \to \beta$$

  con $\alpha \in (V \cup \Sigma)^+$ e $\beta \in (V \cup \Sigma)^*$.
- S è un elemento di V che chiamiamo simbolo iniziale o assioma.

Per $x, y \in (V \cup \Sigma)^*$, diciamo che x deriva y, indicato con $x \Rightarrow y$ (opzionalmente indicando la grammatica $x \underset{G}{\Rightarrow} y$), se e solo se

$$\exists (\alpha \to \beta) \in P, \exists \eta, \delta \in (V \cup \Sigma)^* \mid x = \eta\alpha\delta \wedge y = \eta\beta\delta$$

Inoltre diciamo che x deriva y in k passi, con $k \in \mathbb{N}$, indicato $x \overset{k}{\Rightarrow} y$, se e solo se

$$\exists x_0, x_1, \ldots, x_k \mid x_0 = x \wedge x_k = y \wedge \forall i \in 1, \ldots, k x_{i-1} \Rightarrow x_i$$

Infine diciamo che x deriva y in un certo numero di passi, indicato con $x \overset{*}{\Rightarrow} y$, se e solo se

$$\exists k \geq 0 \mid x \overset{k}{\Rightarrow} y$$

e che x deriva y in almeno un passo, indicato con $x \overset{+}{\Rightarrow} y$, se e solo se

$$\exists k > 0 \mid x \overset{k}{\Rightarrow} y$$

Definiamo il linguaggio generato dalla grammatica G

$$L(G) = \{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$$

Una stringa di terminali e non terminali è detta *forma sentenziale*.
Due grammatiche G_1 e G_2 sono equivalenti se e solo se $L(G_1) = L(G_2)$.

Sia $\Sigma = \{(,)\}$. Costruiamo V, inizialmente contentente solo S. Ora la sequenza vuota è bilanciata, quindi $S \to \epsilon$. Una sequenza di parentesi bilanciata rimane bilanciata se inserita tra due parentesi, quindi $S \to (S)$. Infine la concatenazione di due sequenze bilanciate è ancora bilanciata, quindi $S \to SS$.
Ad esempio

$$\begin{aligned} S &\Rightarrow (S) \\ &\Rightarrow ((S)) \\ &\Rightarrow ((SS)) \\ &\Rightarrow (((S)S)) \\ &\Rightarrow (((S)(S))) \\ &\Rightarrow ((()(S))) \\ &\Rightarrow ((()())) \end{aligned}$$

Data la grammatica $\langle \{S, B\}, \{a, b, c\}, P, S \rangle$, con P

$$
\begin{aligned}
S &\to aBSc \\
S &\to abc \\
Ba &\to aB \\
Bb &\to bb
\end{aligned}
$$

Eseguiamo una derivazione

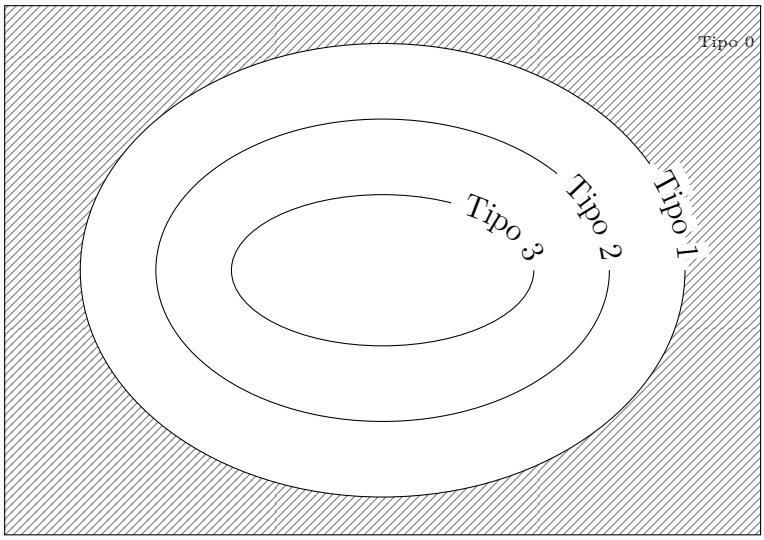
$$
\begin{aligned}
S &\Rightarrow aBSc \\
&\Rightarrow aBabcc \\
&\Rightarrow aaBbcc \\
&\Rightarrow aabbcc \\
S &\Rightarrow aBSc \\
&\Rightarrow aBaBScc \\
&\Rightarrow aaBBScc \\
&\Rightarrow aaBBabccc \\
&\Rightarrow aaBaBbccc \\
&\Rightarrow aaaBBbccc \\
&\Rightarrow aaaBbbccc \\
&\Rightarrow aaabbbccc
\end{aligned}
$$

Si può dimostrare che il linguaggio generato da questa grammatica è $L = a^n b^n c^n \mid n > 0$.

Le grammatiche sono classificate in base alla forma delle produzioni, possiamo definire quattro tipi di grammatiche:

tipo 0	nessuna restrizione	
tipo 1	Se $\alpha \to \beta$ è una produzione, allora $\lvert\beta\rvert \geq \lvert\alpha\rvert$. Equivalentemente una grammatica in cui tutte le produzioni sono della forma $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$ dove $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*, A \in V, \beta \in (V \cup \Sigma)^+$	anche dette context sensitive o dipendenti da contensto
tipo 2	Se $\alpha \to \beta \in P$, allora $\alpha \in V$ e $\beta \in (V \cup \Sigma)^+$	contenxt free o libere da contesto
tipo 3	Possiamo avere produzioni solo della forma $A \to \alpha B$ o $A \to \alpha$, con $A, B \in V, \alpha \in \Sigma$	grammatiche regolari

Ogni grammatica restringe il livello precedente



2.1 Classificazione dei linguaggi

Sia $\mathscr{L} \subseteq \Sigma^*$, diciamo che $\mathscr{L}$ è di tipo $0 \leq i \leq 3$, se e solo se $\exists G$ di tipo i tale che $\mathscr{L} = L(G)$. Questa classificazione genera una gerachia dei linguaggi, detta gerarchia di Chomksy, e chiamiamo

- i linguaggi di tipo 3 come linguaggi regolari
- i linguaggi di tipo 2 come linguaggi conxtext free o CF

- i linguaggi di tipo 1 come linguaggi context sensitive o CS
- i linguaggi di tipo 0 come linguaggi ricorsivamente enumerabili (RE) o semidecibili

E per ognuna di queste classi esiste un modello di macchina riconoscitrice:

- per i linguaggi di tipo 3 sono gli automi a stati finiti
- per i linguaggi di tipo 2 sono gli automi a pila
- per i linguaggi di tipo 1 sono gli automi limitati linearmente, questi hanno un nastro finito che possono modificare
- per i linguaggi di tipo 0 sono le macchine di Turing, nella variante più semplice queste sono macchine con un nastro infinito su cui inizialmente c'è scritto l'input. Per questi linguaggi se la stringa appartiene al linguaggio il riconoscitore termina sempre con sì, mentre se la stringa non appartiene al linguaggio il riconoscitore può anche non terminale ($\perp$)

Teorema 1. *I linguaggi di tipo 1, 2 e 3 sono anche detti decidibili o ricorsivi. (Un insieme ricorsivo è un insieme per cui posso avere una macchina che può rispondere sì o no alla domanda se un insieme appartenga)*

Dimostrazione. Sia G una grammatica di tipo 1 e w una stringa di terminali di lunghezza $n = |w|$, voglio sapere se w appartiene a G. Siccome la grammatica di tipo 1 è monotonica (le regole forme sentenziali non possono decrescere), queste non possono mai superare n durante la derivazione di w. Chiamiamo

$$T_i = \gamma \in (V \cup \Sigma)^{\leq n} \mid S \overset{\leq i}{\Rightarrow} \gamma$$

Costruiamo gli insiemi T_i per $i = 0, 1, \ldots$

$$\begin{aligned} T_0 &= S \\ &\vdots \\ T_i &= T_{i-1} \cup \gamma \in (V \cup \Sigma)^{\leq n} \mid \exists \beta \in T_{i-1}.\beta \Rightarrow \gamma \end{aligned}$$

Allora

$$T_0 \subseteq T_1 \subseteq \cdots \subseteq T_{i-1} \subseteq T_i \subseteq \cdots \subseteq (V \cup \Sigma)^{\leq n}$$

Dato che $(V \cup \Sigma)^{\leq n}$ è un insieme finito, quindi esiste un certo i tale che $T_i = T_{i-1}$. A questo punto ho trovato tutte le forme sentenziali di lunghezza n, e per vedere se w appartiene al linguaggio basta controllare che $w \in T_i$.

Quello che posso fare è considerare l'insieme

$$U_i = \gamma \in (V \cup \Sigma)^* \mid S \overset{\leq i}{\Rightarrow} \gamma$$

Quindi ancora

$$\begin{aligned} U_0 &= S \\ &\vdots \\ U_i &= U_{i-1} \cup \gamma \in (V \cup \Sigma)^* \mid \exists \beta \in U_{i-1}.\beta \Rightarrow \gamma \\ U_0 &\subseteq U_1 \subseteq \cdots \subseteq U_{i-1} \subseteq U_i \subseteq \cdots \subseteq (V \cup \Sigma)^* \end{aligned}$$

Quindi che nei linguaggi di tipo 0 perdo la monotonicità delle derivazioni, non posso più trovare una i tale che $U_i = U_{i-1}$. Però per ogni U_i posso controllare se $w \in U_i$, infatti se la grammatica genera w, prima o poi questa la troverò, se non la genera non la troverò mai. ■

Il motivo per cui si dicono ricorsivamente enumerabili è perché si può costruire un programma che elenchi tutti gli elementi del linguaggio.

La gerarchia di Chomksy vale per alfabeti di almeno due lettere.

2.2 Parola vuota ed ϵ-produzioni

Supponendo di avere una grammatica G di tipo 1, supponiamo di voler aggiungere la parola vuota al linguaggio generato. Se si aggiunge banalmente la regola per la parola vuota

$$S \to \epsilon$$

non ci sono garanzie che noi non abbiamo aggiunto anche altre stringhe al linguaggio, ad esempio nel caso

$$\alpha \Rightarrow \ldots S \ldots$$

inoltre perdiamo la proprietà della monotonicità delle forme sentenziali.

Quindi bisogna essere leggermente più delicati, dato $L = L(G)$, voglio costruire G' tale che $L' = L(G') = L(G) \cup \{\epsilon\}$. Per gare ciò definisco $G' = \langle V \cup \{S'\}, \Sigma, P \cup \{S' \to \epsilon \mid S\}, S' \rangle$.

Dal simbolo iniziale posso utilizzare $S \to \epsilon$ purché S non appaia su lato destro di nessuna produzione. Questo ovviamente vale anche per grammatiche di tipo 2 e 3.

Data una grammatica di tipo 2 che utilizza ϵ-produzioni

$$A \to \epsilon$$

è possibile trovare una grammatica di tipo 2 equivalente che genera lo stesso linguaggio, meno la parola vuota, che non utilizza ϵ-produzioni.

Anche nelle gramamtiche di tipo 3 possiamo ammettere ϵ-produzioni

$$A \to \epsilon$$

2.3 Automa a stati finiti

Un automa è un dispositivo che ha

- un nastro diviso in celle che contiene l'input, ogni cella contiene un simbolo dell'alfabeto Questo non può essere modificato.
- una testina che passa il nastro da sinistra a destra, per questo vengono chiamati anche automi $one-way$. Quando questa finisce il nastro deve poter rispondere se la parola appartiene o no al linguaggio. La testina contiene una macchina che ha un insieme finito di stati.

Un automa è quindi una quintupla $\mathscr{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$, dove

- Q è l'insieme degli stati
- Σ è l'alfabeto finito di input
- δ è il programma dell'automa, o funzione di transizione
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali

Ad ogni passo l'automa legge un simbolo, ed in base allo stato attuale e il simbolo in input sceglie il prossimo stato con δ

$$\delta : Q \times \Sigma \to Q$$

Definiamo

$$\delta^* : Q \times \Sigma^* \to Q$$

per induzione sulla lunghezza delle stringhe in input:

- $\forall q \in Q \mid \delta^*(q, \epsilon) = q$
- $\forall q \in Q, x \in \Sigma^*, a \in \Sigma \mid \delta^*(q, xa) = \delta(\delta^*(q, x), a)$

Visto che δ^* è effettivamente una estensione di δ, chiameremo δ^* δ.

Ora possiamo definire il linguaggio riconosciuto dall'automa $\mathscr{A}$ come

$$L(\mathscr{A}) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$$

Ad esempio $\mathscr{A} = \langle Q = \{q_0, q_1\}, \Sigma = \{a, b\}, \delta, q_0, F = \{q_1\}\rangle$, con δ definita come

Per $aabb$ abbiamo

$$q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{b} q_0$$

quindi la stringa non è accettata perché $q_0 \notin F$. Questo è linguaggoi di tutte le stringhe con un numero dispari di b.

Invece che rappresentare δ come una tabella, di solito è comodo rappresentarla come un diagramma di transizione:

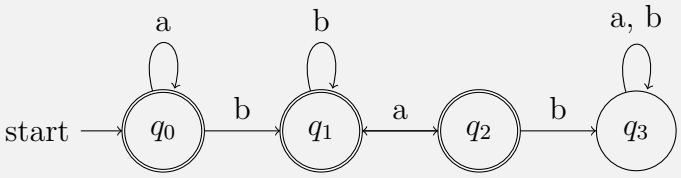
- Lo stato iniziale è rappresentato da una freccia entrante
- gli stati finali da un doppio cerchio.

Quindi una derivazione non è altro che un cammino in questo grafo, e la parola è accettata se si finisce in uno stato finale.

Dato $\Sigma = \{a, b\}$, e il linguaggio

$$\mathscr{L} = \{x \in \Sigma^* \mid P(x)\}$$

con P definita come la proprietà per cui tra ogni coppia di b successive in x c'è un numero di a pari.



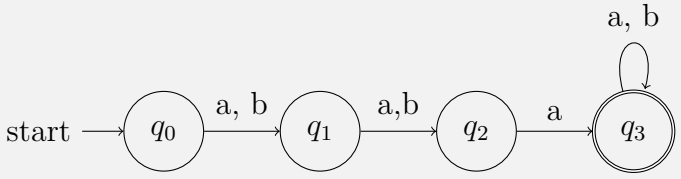
Visto che

$$\delta : Q \times \Sigma \to Q$$

nello stato q_2 bisogna aggiungere una freccia ad uno stato da cui non si può uscire, infatti se siamo in quello stato vuol dire che abbiamo trovato un numero dispari di a e una b, quindi la parola non è valida. Questo tipo di stati è detto stato trappola, e di solito vengono lasciati impliciti.

Dato $\Sigma = \{a, b\}$, e il linguaggio

$$\mathscr{L} = \{x \in \Sigma^* \mid \text{Il terzo simbolo di } x \text{ è una } a\}$$



Dato $\Sigma = \{a, b\}$, e il linguaggio

$$\mathscr{L} = \{x \in \Sigma^* \mid \text{ Il terzultimo simbolo di } x \text{ è una } a\}$$

Teniamo uno stato per ogni tripla di a, b, quindi $2^3 = 8$ stati

Una operazione che può interessare per le stringhe è costruire il linguaggio che riconosce le stringhe roversciate di uno. Dato un linguaggio $\mathscr{L}$, il linguaggio che riconosce le sue stringhe inverse è detto il reversal di $\mathscr{L}$. Nel caso dei due linguaggi precedenti se prendiamo l'automa del primo e lo invertiamo ottieniamo un automa valido (non deterministico) per il secondo.

2.3.1 Automi non deterministici

Definiamo un automa non deterministico come la quintupla $\mathscr{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$, ma ora δ è definita come

$$\delta : Q \times \Sigma \to 2^Q$$

quindi δ non ritorna più un singolo insieme ma un insieme di possibili stati.

Una stringa viene accettata se esiste almeno un cammino che mi porta in uno stato finale. Estensiamo ancora δ alle stringhe

$$\delta^* : Q \times \Sigma \to 2^Q$$

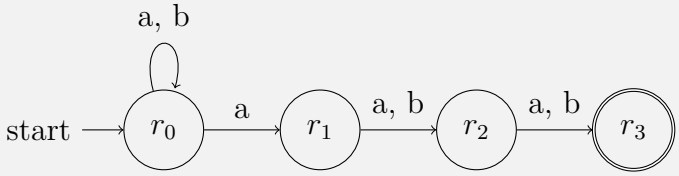
è definita come

- $\forall q \in Q \mid \delta^*(q, \epsilon) = \{q\}$
- $\forall q \in Q, x \in \Sigma^*, a \in \Sigma \mid \delta^*(q, xa) = \bigcup_{p \in \delta^*(q,x)} \delta(p, a)$

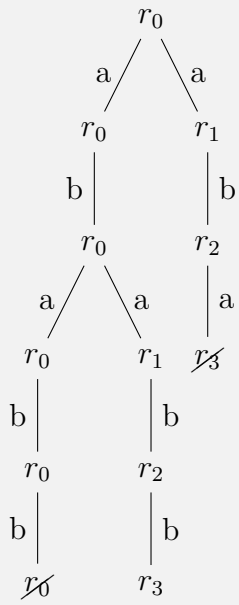
Come prima chiameremo δ^* semplicemente δ.

Ora il linguaggio accettato dall'automa $\mathscr{A}$ non deterministico è $L(\mathscr{A}) = \{w \in \Sigma^* \mid \delta(q_0, w) \cap F \neq \varnothing\}$.

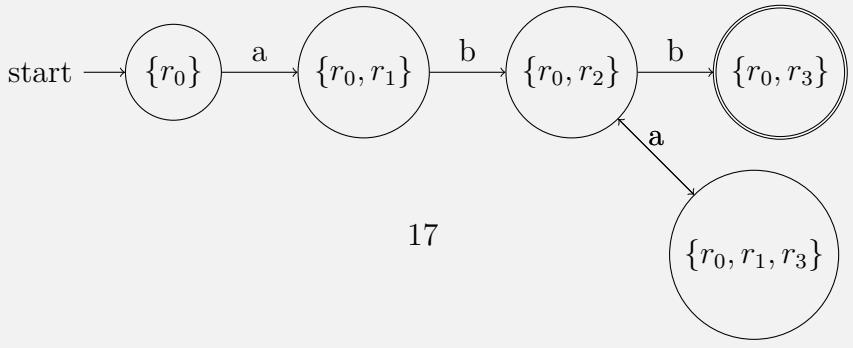
Dato l'automa non deterministico



con la stringha *ababb* mostriamo tutte le possibili strade che si possono prendere



è necessario che esista almeno una strada dell'albero che risponda sì per accettare. Questa struttura è chiamato albero di computazione, e un cammino descrive una computazione.
Bisogna supporre che l'automa riesca a sceglie la strada esatta. Alternativamente si può pensare che l'automa possa visitare tutte le strade in parallelo, quindi possiamo trasformare l'albero in



Questo

Ogni autma non deterministico (NFA) con n stati può sempre essere trasformato in un automa deterministico con al più 2^n stati. Nel caso degli automi a pila invece il modello non deterministico è più potente. Qui invece tra DFA e NFA non cambia la potenza computazionale, ma la semplicità descrittiva.

2.4 Numero di stati

Dato un linguaggio riconoscibile da un automa a stati finiti, quanti stati sono necessari per riconoscerlo?

Nell'ambito degli automi deterministici definiamo il concetto di **distribuibilità** tra stringhe rispetto al linguaggio $L \subseteq \Sigma^*$. Due stringhe qualunque $x, y \in \Sigma^*$ sono distinguibili per L se

$$\exists z \in \Sigma^* (xz \in L \wedge yz \notin L) \vee (xz \notin L \wedge yz \in L)$$

Molto semplicemente se x e y sono distinguibili queste ci porteranno in due stati diversi. Da questi due stati diversi si può costruire una z che in un caso ci porta in uno stato finale e nell'altro no. Quindi in un automa deterministico se due stringhe sono distinguibili, non ci possono mandare nello stesso stato.

Teorema 2. *Sia $L \subseteq \Sigma^*$ e $X \subseteq \Sigma^*$ tale che ogni coppia di stringhe in X è distinguibile in rispetto ad L. Allora ogni DFA che accetta L deve avere almeno $\#X$ stati (cardinalità di X).*

Dimostrazione. Supponiamo che $X = \{x_1, x_2, \ldots, x_k\}$. Sia $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$ un DFA per L. Sia $\forall i \in 1, \ldots, k \mid p_i = \delta(q_0, x_i)$. Se $\#Q < k$ allora esistono due stati uguali

$$\exists i, j \in 1, \ldots, k \mid i \neq j \wedge p_1 = p_j$$

Ma x_i, x_j sono distinguibili, quindi

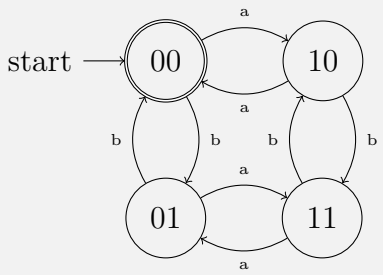
$$\exists z \mid (xz \in L \wedge yz \notin L) \vee (xz \notin L \wedge yz \in L)$$

Ma questo z non può esistere, quindi la $\#Q$ deve per forza essere $\leq k$. ∎

Questo può essere anche utile per determinare se un linguaggio non può essere definito da un automa a stati finiti. Infatti se si trova un insieme X infinito, allora il linguaggio non può essere regolare.

Sia $\Sigma = \{a, b\}$ e

$$L = \{x \in \Sigma^* \mid \#_a(x) \text{ è pari } \wedge \#_b(x) \text{ è pari } \}$$



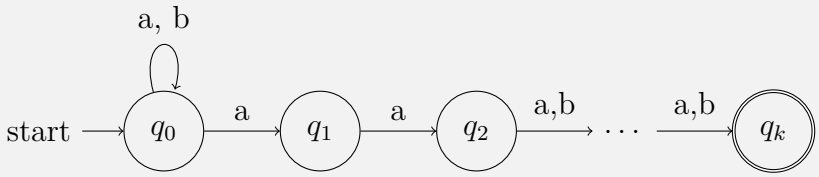
Costruisco $X = \{\epsilon, a, b, ab, \}$, per vedere che sono distinguibili costruiamo

z	ϵ	a	b	ab
ϵ		ϵ	ϵ	ϵ
a			a	a
b				b
ab				

Dato il linguaggio

$$\mathcal{L}_n = \{x \in \{a,b\}^* \mid \text{l'}n\text{-esimo simbolo da destra di } x \text{ è una } a\}$$

L'automa si ricorda gli ultimi n simboli, quindi ha 2^n stati. Mentre l'NFA corrispondente è



questo ha $n+1$ stati.
Possiamo fare meglio per il deterministico? Costruiamo $X = \{a,b\}^n = \{x \in \{a,n\}^* \mid |x| = n\}$. Siano $x, y \in X$, con

$$x = x_1 x_2 \dots x_i \dots x_n$$
$$y = y_1 y_2 \dots y_i \dots y_n$$

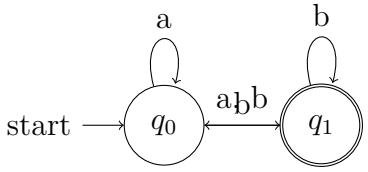
Visto che $x \neq y$, $\exists i \mid x_i \neq y_i$. Supponiamo che $x_i = a$, ed $y_i = b$, quindi

$$x = x_1 x_2 \dots a \dots x_n$$
$$y = y_1 y_2 \dots b \dots y_n$$

Ora per distinguere le tue parole basta scegliere $z = \{a,b\}^{i-1}$, ad esempio $z = a^{i-1}$, con questa z $xz \in \mathcal{L}$ e $yz \notin \mathcal{L}$. Visto che la cardinalità di X è di 2^n allora non possiamo costruire un automa di meno di 2^n stati.
Questo mostra come gli automi non deterministici possano essere molto più compatti.

Se definiamo $\mathcal{L} = \{w \in \{a,b\}^* \mid \#_a(w) = \#_b(w)\}$ ha X infinito

2.4.1 Costruzione coi sottoinsiemi



$\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F\rangle$ NFA definisco $\mathcal{A}' = \langle Q', \Sigma, \delta', q_0', F'\rangle$, con $Q' = 2^Q$ e $q_0' = \{q_0\}$. Quindi

$$\forall a \in Q', a \in \Sigma \mid \delta'(\alpha, a) = \bigcup_{q \in \alpha} \delta(q, a)$$

ed

$$F' = \{\alpha \in Q' \mid \alpha \cap F \neq \varnothing\}$$

Se l'NFA ha n stati, il DFA ottenuto ha 2^n stati.

Costruiamo ora l'automa M_n, o automa di Meyer&Fisher (1971), non deterministico tale per cui l'automa deterministico corrispondente ha necessariamente 2^n stati. Quindi è l'automa $\mathcal{A} = \langle Q = \{0, 1, \ldots, n-1\}, \{a, b\}, \delta, q_0 = 0, F = \{0\}\rangle$ con δ definita come

$$\delta(i, a) = (i + 1) \mod n$$

$$\delta(i, b) = \begin{cases} 0 & \text{se } i = 0 \\ \{i, 0\} & \text{altrimenti} \end{cases}$$

Sia $S \subseteq \{0, \ldots, n-1\}$, definisco w_s come

$$w_s = \begin{cases} b & \text{se } S = \varnothing \\ a^i & \text{se } S = \{i\} \\ a^{e_k - e_{k-1}} b a^{e_{k-1} - e_{k-2}} b \ldots b a^{e_2 - e_1} b a^{e_1} & \text{se } S = \{e_1, e_2, \ldots, e_k\} \text{ con } k \geq 2, e_1 < e_2 < \cdots < e_k \end{cases}$$

Ad esempio prendiamo $S = \{1, 3, 4\}$, la w_S corrispondente è

$$w_S = aba^2ba = abaaba$$

Mostriamo che l'insieme di stati raggiunto da questa stringa è esattamente $\{1, 3, 4\} = S$. Mostriamo anche per $S = \{0, 2\}$, $w_S = a^2b$. O ancora $S = \varnothing$, $w_S = b$.

Proprietà 1. *È possibile dimostrare che* $\forall S \subseteq \{0, \ldots, n-1\} \mid \delta(0, w_S) = S$.

Proprietà 2. *Siano* $S, T \subseteq \{0, \ldots, n-1\}$*, se* $S \neq T$ *allora* w_S *e* w_T *sono distinguibili.*

Dimostrazione. Se $S \neq T$ esiste un numero che appartiene a S ma non a T, chiamiamolo $x \in S$ T Abbiamo che

$$\delta(0, w_S) = S$$
$$\delta(0, w_T) = T$$

Dato uno stato y ci si mettono a^{n-y} per arrivare allo stato finale. Quindi da w^S ci si mettono a^{n-x} per arrivare ad uno stato finale.

$$0 \overset{w_S}{\rightsquigarrow} x \overset{a^{n-x}}{\rightsquigarrow} 0$$

Sia $y \in T$, se $y \neq x$, allora

$$0 \overset{w_T}{\rightsquigarrow} y \overset{a^{n-x}}{\not\rightsquigarrow} 0$$

Allora a^{n-x} distingue S e T.

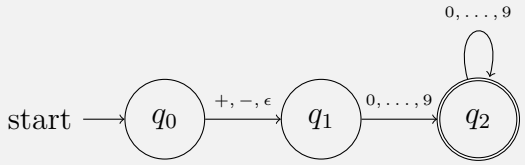
Ed abbiamo che $X = \{w_S | S \subseteq \{0, \ldots, n-1\}\}$ è foramto da stringhe a coppie di stringhe Quindi ogni DFA ha almeno 2^n stati. ■

Questo è il linguaggio (più o meno) delle stringhe che hanno un suffisso della forma bx dove $\#_a(x)$ è multiplo di n.

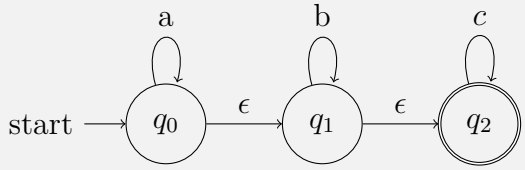
2.4.2 ϵ-mosse

Possiamo, in certi casi, introdurre delle transizioni sulle parole vuote.

Supponiamo di volere l'automa che riconosce i numeri con segno (opzionale)



Oppure supponiamo di volere l'automa per $a^l b^m c^n$



Generalmente se ho

$$q_0 \xrightarrow{\epsilon} q_1 \xrightarrow{a} q_2 \xrightarrow{\epsilon} q_3$$

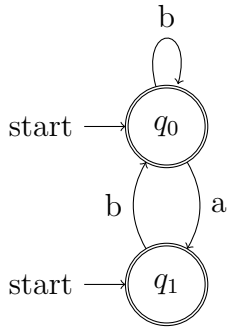
questo diventa

$$q \xrightarrow{a} q_3$$

Inoltre se uno stato ha un cammino formato da ϵ-mosse fino ad uno stato finale, esso stesso è finale. Questo necessariamente introduce non determinismo.

2.4.3 Stati iniziali multipli

Questa è un'altra estensione che genera non determinismo.



Per renderli deterministici si crea lo stato iniziale che è l'insieme dei diversi stati iniziali.

Se si prende un automa con stati iniziali multipli, ma transizioni deterministiche, si può comunque avere un gap esponenziale tra il numero di stati del'NFA e del DFA.

Capitolo 3

Espressioni regolari

3.1 Operazioni sui linguaggi

Visto che un linguaggio $L \subseteq \Sigma^*$ è un sottoinsieme di un insieme, possiamo fare tutte le operazioni insiemistiche su un linguaggio:

- intersezione $L \cap L'$
- unione $L \cup L'$
- complemento L^C. Si può pensare di calcolare il complemento di un linguaggio $L \subseteq \Sigma^*$ rispetto ad un alfabeto più grande $\Sigma \subseteq \Gamma$. Il complemento è esattamente $\Gamma^* \setminus L$.

Definiamo il prodotto di due linguaggi $L', L'' \subseteq \Sigma^*$

$$L' \cdot L'' = \{z \mid \exists x \in L' \exists y \in L''.z = xy\}$$

E se $L' \subseteq \Sigma^{*\prime}, L'' \subseteq \Sigma^{*\prime\prime}$, allora $L' \cdot L'' \subseteq \Sigma' \cup \Sigma''$. Salvo casi particolari (e.g. alfabeto da una lettera sola) questa operazione non è commutativa. E vale $\{\epsilon\}$ è l'indentità destra e sinistra, e $\varnothing$ è l'elemento nullo destro e sinistro.

Definiamo la potenza di un linguaggio

$$L^k = \underbrace{L \cdot L \ldots}_{k \text{ volte}}$$

e vale che

$$L^0 = \{\epsilon\}$$

ovvero il linguaggio ottenuto concatenando zero parole di L. Il prodotto e la potenza di un linguaggio finito sono ancora finiti.

Definiamo la chiusura di Kleene di un linguaggio

$$L^* = \bigcup_{k \geq 0} L^k$$

La chiusura di Kleene anche di un linguaggio finito è finito.

$$\begin{aligned} L &= \{a, bb\}^* \\ &= \{x \in \{a, b\}^* \mid \text{ Ogni fattore massimale di } b \text{ è di lunghezza pari } \} \end{aligned}$$

Dove massimale indica che non può essere allungato.

Vale inoltre che

$$\{\epsilon\}^* = \{\epsilon\}$$

e che

$$\varnothing^* = \{\epsilon\}$$

Definiamo la chiusura di Kleene positiva di un linguaggio

$$L^+ = \bigcup_{k \geq 1} L^k$$

Inoltre vale che L non contiene la parola vuota, allora L^+ a sua volta non la contiene, e questo è uguale a $L^+ = L^* \setminus \{\epsilon\}$. Nota bene, non vale che generalmente $L^+ = L^* \setminus \{\epsilon\}$.

Fatto 2.

$$L \cdot L^* = L \bigcup_{k \geq 0} L^k = \bigcup_{k \geq 0} LL^k = \bigcup_{k \geq 1} L^k = L^+ = L^* \cdot L$$

3.2 Espressioni regolari

Le espressioni regolari sono in un certo senso una descrizione dichiarativa di un linguaggio. Dato un alfabeto Σ, definiamo ricorsivamente le espressioni regolari come

- $\varnothing$, rappresenta il linguaggio vuoto
- ϵ, rappresenta il linguaggio della parola vuota
- $a \in \Sigma$, rappresenta il linguaggio di solo a

ora definiamo

- $E_1 + E_2$, rappresenta il linguaggio $L(E_1) \cup L(E_2)$
- $E_1 \cdot E_2$, rappresenta il linguaggio $L(E_1) \cdot L(E_2)$
- E^*, rappresenta il linguaggio $L(E)^*$

L'espressione regolare

$$(a + bb)^*$$

rappresenta il linguaggio $\{a, bb\}^*$.

Il linguaggio dove il terzultimo simbolo è una a è rappresentato dall'espressione regolare

$$(a + b)^* a (a + b)(a + b)$$

E volendo generalizzare al linguaggio il cui n-esimo simbolo da destra è una a

$$(a + b)^* a (a + b)^n$$

Dove le potenze sulle espressioni regolari rappresentano una serie di prodotti.

Il linguaggio dove due simboli a distanza n sono uguali

$$((a + b)^* a (a + b)^{n-1} a (a + b)^*) + ((a + b)^* b (a + b)^{n-1} b (a + b)^*)$$

Questo può essere semplificato in

$$(a + b)^* ((a(a + b)^{n-1} a) + (b(a + b)^{n-1} b))(a + b)^*$$

Definizione 1 (State complexity)**.** *Definiamo la complessità di stati o state complexity di un linguaggio $L \subseteq \Sigma^*$, indicata* $\mathrm{sc}(L)$*, come il minimo numero di stati del DFA che accetta L. E definiamo la non-deterministic state complexity, indicata con* $\mathrm{nsc}(L)$*, come il minimo numero di stati del NFA che accetta L.*

Fatto 3.

$$\mathrm{sc}(L) \leq 2^{\mathrm{nsc}(L)}$$

Dato $L_n = (a+b)^* a (a+b)^{n-1}$, vale che $\mathrm{nsc}(L_n) \leq n+1$, mentre $\mathrm{sc}(L_n) \geq 2^n$, e visto che per questo linguaggio sappiamo costruire un automa da 2^n stati, allora $\mathrm{sc}(L_n) = 2^n$.
La stringa più corta di questa linguaggio è la stringa che inizia con a ed ha n simboli. Supponendo di avere meno di n stati, allora ce ne deve essere almeno uno che si ripete; quindi c'è un loop. Se io elimino il loop potrei riconoscere una stringa più breve di quella più breve.
Quindi abbiamo che per forza $\mathrm{nsc}(L_n) = n+1$.
Alternativamente si può costruire il fooling set composto da tutte le possibili scomposizioni in due stringhe di ab^{n-1}.

Proposizione 1. *Presa la stringa più corta del linguaggio, ogni NFA deve per forza avere $n+1$ stati, dove n sono i simboli della stringa.*

Teorema 3 (Teorema di Kleene o Teorema fondamentale degli automi a stati finiti)**.** *La classe dei linguaggi accettati da automi a stati finiti è la più piccola sottoinsieme di Σ^* che contiene i linguaggi finiti ed è chiusa rispetto alle operazioni di unione, prodotto e chiusura di Kleene.*

Quindi la classe degli automi a stati finiti coincide con la classe esprimibile dalle espressioni regolari.

Dimostrazione. Dimostriamo il primo lato, passandro da un automa ad una regex. Dato un automa

$$\mathcal{A} = \langle Q, \Sigma, \delta, q_1, F \rangle$$

e supponiamo che gli stati siano numerati da uno ad n

$$Q = \{q_1, q_2, \ldots, q_n\}$$

possiamo vedere come il linguaggio come tutti i cammini dallo stato iniziale agli stati finali, e che concantenando le etichette del cammino otteniamo un

parola accettata dal percorso.le etichette del cammino otteniamo un parola accettata dal percorso.

Costruiamo un algoritmo simile a quello di Floyd-Wharshall che trova tutti i camminini minimi. Chiamiamo $R_{ij}^{(k)}$ l'insieme di tutti i percorsi da q_i a q_j che passano solo per stati con indice minore di k. Quando metteremo $k = n$ avrò trovato tutte le stringhe da i a j. Andando per induzione definiamo l'insieme come

- $R_{ij}^{(0)}$ è il caso in cui c'è una transizione tra lo stato i e lo stato j.

$$R_{ij}^{(0)} = \begin{cases} \{\alpha \in \Sigma \mid \delta(q_i, \alpha) = q_j\} & \text{se } i \neq j \\ \{\epsilon\} \cup \{\alpha \in \Sigma \mid \delta(q_i, \alpha) = q_i\} & \text{se } i = j \end{cases}$$

- supponiamo di avere $R_{ij}^{(k-1)}$, voglio trovare $R_{ij}^{(k)}$. Cerco tutti i punti per cui il nuovo cammino passa per q_k, nelle sezioni tra i q_k gli stati intermedi sono tutti minori di k. Quindi

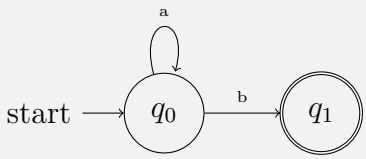
$$R_{ij}^{(k)} = R_{ij}^{(k-1)} \cup R_{ik}^{(k-1)} \left(R_{kk}^{(k-1)}\right)^* R_{kj}^{(k-1)}$$

Con $k = n$ abbiamo tutti i percorsi e vale che

$$L = \bigcup_{q_i \in F} R_{1i}^{(n)}$$

Dimostriamo il secondo lato, passandro da una regex ad un automa. ■

Mostriamo una tecnica diversa per generare la regex da un automa.



Costruiamo un sistema di equazioni per ogni stato

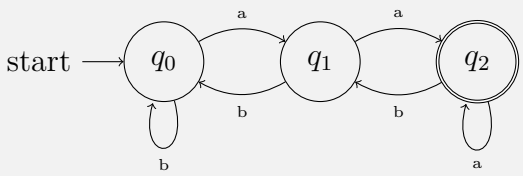
$$\begin{cases} X &= aX_1 + bY \\ Y &= \epsilon \end{cases}$$

e sostituendo

$$\begin{cases} X &= aX + b \\ Y &= \epsilon \end{cases}$$

se ci troviamo in uno stato $X = AX + B$, questo corrisponde alla regex A^*B. Questo è banalmente a^*b.

Un altro esempio più complesso



Costruiamo un sistema di equazioni per ogni stato

$$\begin{cases} X_0 &= aX_1 + bX_0 \\ X_1 &= aX_2 + bX_0 \\ X_2 &= aX_2 + bX_1 + \epsilon \end{cases}$$

ϵ perché X_2 è finale. Sostitendo X_1 abbiamo

$$\begin{aligned} X_0 &= aaX_2 + abX_0 + bX_0 \\ &= aaX_2 + (ab + b)X_0 \end{aligned}$$

e

$$\begin{aligned} X_2 &= aX_2 + baX_2 + bbX_0 + \epsilon \\ &= (a + ba)X_2 + bbX_0 + \epsilon \\ &= (a + ba)^* + bbX_0 + \epsilon \end{aligned}$$

che corrisponde alla regex $(a + ba)^*(bbX_0 + \epsilon)$, prendendo come $A = (a + ba)$ e $B = (bbX_0 + \epsilon)$. E sostituendo ancora

$$\begin{aligned} X_0 &= aaX_2 + (ab + b)X_= \\ &= aa(a + ba)^*(bbX_0 + \epsilon) + (ab + b)X_0 \\ &= (aa(a + ba)^*bb + ab + b)X_0 + aa(a + ba)^* \end{aligned}$$

che quindi è $(aa(a + ba)^*bb + ab + b)^* + aa(a + ba)^*$.

Il complemento di linguaggio riconosciuto da un automa deterministico è banalmente il complemento dell'insieme dei finali.

Date tre espressioni regolari

$$(a * b*)* \equiv (a + b)* \equiv b * (ab*)*$$

Sono tre linguaggi equivalenti.

Ci si può chiedere qual è il numero massimo di star innestate necessarie per generare un linguaggio. Introduciamo l'*altezza di star*, o star height h, come il numero massimo di star innestate. Definita nel seguente modo

$$h(E) = \begin{cases} 0 & \text{se } E = \varnothing \vee E = \epsilon \vee E \in \Sigma^* \\ \max(h(E'), h(E'')) & \text{se } E = E' \cdot E'' \vee E = E' + E'' \\ 1 + h(E') & \text{se } E = E'* \end{cases}$$

Definiamo poi la minima altezza per un linguaggio $L \subseteq \Sigma^*$ come

$$h(L) = \min\{h(E) \mid \ E \text{ denota } L \ \}$$

Teorema 4 (Dejan, Schutzenberger)**.**

$$\forall q > 0 \exists W_q \subseteq \{a, b\}^* \mid h(W_q) = q$$

Equesto è definito come

$$W_q = \{w \in \{a, b\}^* \mid \#_a(w) \equiv \#_b(w) \mod 2^q\}$$

La misura dei cicli è collegata al numero di cicli in un automa.

Per $q = 1, 2, \ldots$, definiamo W_q ed abbiamo che

$$W_2 = ((ab + ba) + (aa + bb)(ab + ba)^*(bb + aa))^*$$

Nota 1. *Se* $|\Sigma| = 1$*, allora* $\forall L \subseteq \Sigma^* \mid h(L) \leq 1$*.*

3.3 Espressioni regolari estese

Supponiamo di avere il linguaggio

$$L = \{w \in \{a, b\}^* \mid \#_a(w) \text{ è pari } \wedge \#_b(w) \text{ è pari } \}$$

Costruiamo prima il linguaggio con solo le a pari

$$(b + ab * a)*$$

e con solo le b pari

$$(a + ba * b)*$$

intuitivamente il linguaggio L sarebbe, se avessimo l'intersezione,

$$(b + ab * a) * \cap (a + ba * b)*$$

Intoduciamo quindi le **espressioni regolari estese**.

Definiamo queste come le espressioni con, oltre alle operazioni di concatenazione, unione e star; l'intersezione ($\cap$) e il complemento (E^-). Introducendo il complemento possiamo ottenere l'intersezione attraverso l'unione.

Dato il linguaggio su $\Sigma = \{a, b\}$ delle stringhe con 3 a consecutive vogliamo riconoscere il suo complemento (senza 3 a consecutive). Questo diventa semplicemente

$$\overline{\overline{\varnothing} aaa \overline{\varnothing}}$$

Quindi ora senza usare la star possiamo esprimere anche linguaggi infiniti.

Si può definire il problema di trovare il l'altezza di star minima (eh) nelle espressioni regolari estese. Questo è un problema aperto. Si sa che valgono

$$\exists L \mid eh(L) = 0$$
$$\exists L \mid eh(L) = 1$$

Capitolo 4

Operazioni sugli automi a stati finiti

In questo capitolo vedremo una serie di operazioni sugli automi a stati finiti e soprattutto il loro impatto sulla dimensione dell'automa. Abbiamo già visto alcune operazioni tra automi nell'ambito delle espressioni regolari, specificamente

op	DFA	NFA
unione	$n' \cdot n''$	$1 + n' + n''$
concatenazione	$n' \cdot 2^{n''}$	$n' + n''$
star	$1 + 2^{n'}$	$n' + 1$
intersezione	$n' \cdot n''$	$n' \cdot n''$
complemento	n'	$2^{n'}$

L'intersezione di automi funziona anche con NFA con la stessa costruzione dei DFA.

4.1 Reversal

Definiamo l'operazione di inversione o reversal. Data una stringa

$$w = a_1 a_2 \dots a_n$$

definiamo

$$w^R = a_n \dots a_2 \dots a_1$$

Per DFA e NFA basta sostituire stati iniziali con i finali e viceversa, e invertire le transizioni. Però può accadere che un DFA invertito diventi non deterministico, infatti

- se l'automa ha diversi stati finali, questi diventeranno stati iniziali multipli
- se uno stato ha più frecce entranti con lo stesso simbolo, queste diventeranno uscenti con lo stesso simbolo

> Ad esempio il linguaggio il cui terzo simbolo è una a è un semplice DFA, mentre il linguaggio il cui terzultimo simbolo è una a abbiamo visto che ha 2^3 stati.
> In generale se chiediamo l'n-esimo simbolo da destra servono $n+1$ stati, mentre se chiediamo l'n-esimo simbolo da sinistra ne servono 2^n.

Mentre per le espressioni regolari, per la costruzione del reversal, basta solo invertire l'espressione. O più precisamente

- nei casi in cui $E = \varnothing$ o $E = \epsilon$ non cambia niente
- nel caso in cui $E \in \Sigma^*$ questo diventa E^R
- nel caso in cui $E = E' + E''$, abbiamo $E'^R + E''^R$
- nel caso in cui $E = E' \cdot E''$, abbiamo $E''^R \cdot E'^R$
- nel caso in cui $E = E'*$, abbiamo $(E'^R)*$

4.2 Shuffle

Definiamo l'operazione di *shuffle*

$$sh(x, y)$$

che produce ogni possibile interpolazione dei suoi argomenti

$$sh(ab, cd) = \{abcd, acdb, acbd, cabd, \dots\}$$

Cioè vale che se

$$sh(w', w'') = w$$

allora se definiamo che per due simboli $x_i, x_j \in \Sigma$ in posizioni i e j vale $x_i \prec_w x_j$ se $i < j$

$$\forall x, y \in w \mid x \prec_{w'} y \Rightarrow x \prec_w y \wedge x \prec_{w''} y \Rightarrow x \prec_w y$$

Dati due linguaggi, abbiamo che

$$sh(L', L'') = \bigcup_{x \in L', y \in L''} sh(x, y)$$

cioè lo shuffle di due linguaggi è definito come l'unione degli shuffle di tutte le loro stringhe.

4.3 Quoziente

Definiamo l'operazione di quoziente tra due linguaggi

$$L_1 \setminus L_2 = \{x \in \Sigma^* \mid \exists y \in L_2 \; xy \in L_1\}$$

Quindi le stringhe di L_1 in cui ho tolto un suffisso di L_2.

Ad esempio

$$L_1 = a^+bc^+$$
$$L_2 = bc^+$$
$$L_3 = c^+$$

Ho che

$$L_1 \setminus L_2 = a+$$

e che

$$L_1 \setminus L_3 = a + bc*$$

Ad esempio da un linguaggoi su $\Sigma = \{0, 1\}$, voglio togliere tutti gli zeri finali

$$(L \setminus 0*) \cap ((0 + 1) * 1 + \epsilon)$$

Fatto 4. *Dato R linguaggio regolare ed L un linguaggio qualsiasi, allora*

$$R \setminus L$$

è ancora un linguaggio regolare.

Dimostrazione. Dato

$$\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$$

l'automa per R.

Definisco

$$\mathcal{A}' = \langle Q, \Sigma, \delta, q_0, F' = \{q \mid \exists y \in L . \delta(q, y) \in F\} \rangle$$

come l'automa per $R \setminus L$.

Non è detto che sappiamo trovare una $y \in L$. Quindi questa dimostrazione non è costruttiva per tutti i linguaggi. ■

4.3.1 Morfismo

Il linguaggio delle parentesi bilanciate (Dick) non è regolare. Dato un linguaggio di programmazione, lo possiamo ricondurre al linguaggio di Dick, ad esempio

$$\begin{aligned} \{ &\to (\\ \} &\to) \\ a &\to \epsilon \end{aligned}$$

Con questa operazione stiamo definendo un morfismo tra due alfabeti

$$h : \Sigma \to \Delta^*$$

Questa funzione è facilmente estendibile alla stringa

$$\begin{cases} h(\epsilon) = \epsilon \\ h(xa) = h(x)h(a) \end{cases}$$

e al linguaggio

$$h(L) = \{h(w) \mid w \in L\}$$

Dati $\Sigma = \{a, b\}$, e $\Delta = \{0, 1\}$, definiamo il morfismo h come

$$h(a) = 01$$
$$h(b) = 1$$

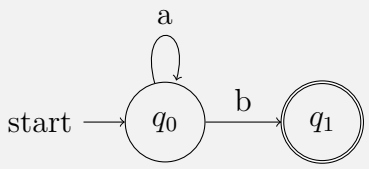
E vale che se abbiamo

$$L = a^*b$$

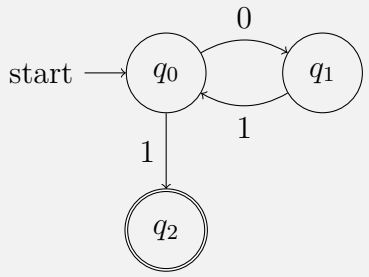
questo corrisponde a

$$(01)^*1$$

E agli automi corrisponde



diventa



Quindi basta sostituire ogni stato che riconosce un simbolo $\sigma \in \Sigma$ con gli stati corrispondenti che riconoscerebbero $h(\sigma) \in \Delta$.

4.3.2 Sostituzione

In questa versione si sostituiscono le lettere di un linguaggio con un altro linguaggio, quindi

$$s : \Sigma \to 2^{\Delta^*}$$

Nel caso di espressioni regolari questo è molto semplice, basta sostituire ogni occorrenza del simbolo $\sigma \in \Sigma$ nell'espressione con $s(\sigma)$.

Ad esempio dato $\Sigma = \{a, b\}$ e $\Delta = \{0, 1\}$, posso dire che

$$s(a) = (01 + 0)^*$$
$$s(b) = 1$$

e

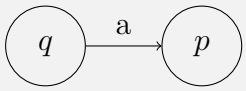
$$L = (ab)^*(a + b)$$

ottengo

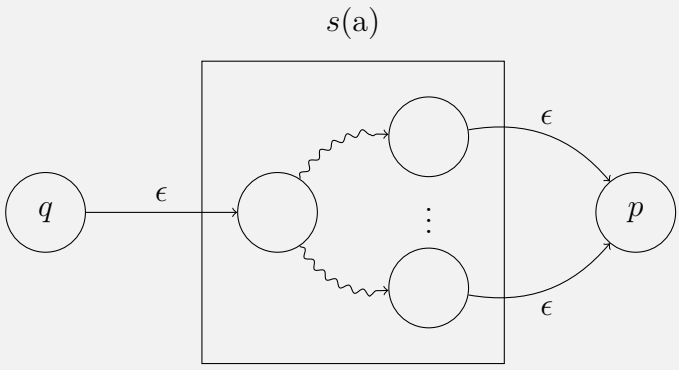
$$((01 + 0)^*1)^*((01 + 0)^* + 1)$$

Nel caso di automi, basta sostituire una transizione con l'intero automa corrispondente a quel simbolo, quindi:

Ad esempio dato l'automa



questo verrebbe trasformato in



Se le sostituzioni sono ancora regolari, allora il linguaggio risultante è regolare.

Dato un linguaggio $L \subseteq \{a, b\}^*$, voglio creare il linguaggio L' in cui ogni a e b è circondata da un numero arbitrario di c, questo è

$$s(a) = c^*ac^*$$
$$s(b) = c^*bc^*$$

Manca però la stringa vuota, quindi abbiamo

$$L' = \begin{cases} s(L) & \text{se } \epsilon \notin L \\ s(L) \cup c^* & \text{altrimenti} \end{cases}$$

Questo povtevamo farlo anche con lo shuffle, quindi

$$L' = sh(L, c^*)$$

4.4 Cycle

Definiamo l'operazione di *cycle*, come

$$cycle(L) = \{yx \mid xy \in L\}$$

ad esempio, se

$$L = a^*b^*$$
$$cycle(L) = a^*b^*a^* + b^*a^*b^*$$

Dato un automa che riconosca L, possiamo costruire un automa che riconosca $cycle(L)$?.