

Teoria dei Linguaggi

Indice

Introduzione	3
--------------------	---

Parte I — Gerarchia di Chomsky 4

1. Breve ripasso	5
2. Gerarchia di Chomsky	8
2.1. Grammatiche	8
2.2. Gerarchia di Chomsky	11
2.3. Decidibilità	13
2.4. Introduzione della parola vuota	15
2.5. Linguaggi non esprimibili tramite grammatiche finite	16

Parte II — Linguaggi regolari 18

1. Automi a stati finiti deterministici	19
1.1. Definizione	19
1.2. Esempi	20
2. Automi a stati finiti non deterministici	23
2.1. Definizione	23
2.2. Confronto tra DFA e NFA	24
2.3. Forme di non determinismo	25
3. Numero minimo di stati	28
3.1. Distinguibilità	28
3.1.1. Automa di Meyer-Fischer	32
3.1.2. Esempi	35
3.2. Fooling set	38
3.2.1. Esempi	39

Introduzione

In questo corso studieremo dei **sistemi formali** che descrivono dei linguaggi: ci chiederemo cosa sono in grado di fare, ovvero cosa sono in grado di descrivere in termini di **linguaggi**.

Ci occuperemo anche delle **risorse utilizzate**, come il **numero di mosse** eseguite da una macchina che deve riconoscere un linguaggio, oppure il **numero di stati** che sono necessari per descrivere una macchina a stati finiti, oppure ancora lo **spazio utilizzato** da una macchina di Turing.

Un **linguaggio** è «uno strumento di comunicazione usato da membri di una stessa comunità», ed è composto da due elementi:

- **sintassi**: insieme di simboli (o parole) che devono essere combinati con una serie di regole;
- **semantica**: associazione frase-significato.

Per i linguaggi naturali è difficile dare delle regole sintattiche: vista questa difficoltà, nel 1956 **Noam Chomsky** introduce il concetto di **grammatiche formali**, che si servono di regole matematiche per la definizione della sintassi di un linguaggio.

Il primo utilizzo dei linguaggi formali risale agli stessi anni con il **compilatore Fortran**. Anche se ci hanno messo l'equivalente di 18 anni/uomo, questa è la prima applicazione dei linguaggi formali. Con l'avvento, negli anni successivi, dei linguaggi Algol, ovvero linguaggi con strutture di controllo, la teoria dei linguaggi formali è diventata sempre più importante.

Oggi la teoria dei linguaggi formali è usata nei compilatori di compilatori, dei tool usati per generare dei compilatori per un dato linguaggio fornendo la descrizione di quest'ultimo.

Parte I — Gerarchia di Chomsky

1. Breve ripasso

Prima di addentrarci nello studio della gerarchia di Chomsky facciamo un breve ripasso delle basi che ci serviranno durante lo studio dei linguaggi formali.

Definizione 1.1 (Alfabeto): Un **alfabeto** è un **insieme non vuoto e finito di simboli**, di solito indicato con le lettere greche maiuscole

$$\Sigma \quad | \quad \Gamma.$$

Definizione 1.2 (Stringa): Una **stringa**, o **parola**, è una **sequenza finita** di simboli appartenenti all'alfabeto Σ . Viene indicata con la lettera x e la possiamo scrivere come

$$x = a_1 \dots a_n \quad | \quad a_i \in \Sigma.$$

Data una stringa x , indichiamo il **numero di caratteri** di x con la notazione

$$|x|$$

e il **numero di occorrenze di un carattere** di Σ in x con la notazione

$$|x|_a \quad | \quad a \in \Sigma.$$

Una stringa/parola molto importante è la **parola vuota**, che possiamo indicare in vari modi:

$$\varepsilon \quad | \quad \lambda \quad | \quad \Lambda.$$

Come dice il nome, questa parola non ha simboli, ovvero è l'unica parola tale che

$$|\varepsilon| = 0.$$

L'insieme di tutte le possibili parole che possiamo formare usando l'alfabeto Σ si indica con Σ^* . Questo insieme, ovviamente, è un **insieme infinito**.

Con le parole possiamo definire una serie di operazioni, ma la più importante tra tutte è la **concatenazione**, o **prodotto**. Date due stringhe

$$x, y \in \Sigma^* \quad | \quad x = x_1 \dots x_n \wedge y = y_1 \dots y_m$$

allora la concatenazione di x e y è la stringa

$$w = x \cdot y = x_1 \dots x_n y_1 \dots y_m.$$

L'operazione di concatenazione **non è commutativa** ma è **associativa**, quindi la struttura

$$(\Sigma^*, \cdot, \varepsilon)$$

è un **monoide libero** generato da Σ .

Vediamo, per (quasi) finire, alcune proprietà che possiamo dare alle stringhe/parole.

Definizione 1.3 (Prefisso): La stringa $x \in \Sigma^*$ si dice **prefisso** di w se

$$\exists y \in \Sigma^* \mid w = xy.$$

In poche parole, x è prefisso di w se riusciamo a scomporre la stringa w in due parti, dove x è la prima di queste due. Abbiamo due tipi di prefisso:

- **proprio** se $y \neq \varepsilon$;
- **non banale** se $x \neq \varepsilon$.

Il **numero** di prefissi di una stringa w è $|w| + 1$.

Definizione 1.4 (Suffisso): La stringa $y \in \Sigma^*$ si dice **suffisso** di w se

$$\exists x \in \Sigma^* \mid w = xy.$$

In poche parole, vale quanto scritto prima, ma in questo caso y è la seconda delle due parti. Anche qui abbiamo tipi di suffisso:

- **proprio** se $x \neq \varepsilon$;
- **non banale** se $y \neq \varepsilon$.

Anche il **numero** di suffissi di una stringa w è $|w| + 1$.

Definizione 1.5 (Fattore): La stringa $y \in \Sigma^*$ si dice **fattore** di w se

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

In poche parole, vale quanto scritto prima, ma in questo caso dividiamo la stringa w in tre parti e y è la centrale di queste.

Il **numero** di fattori di una stringa w è

$$\leq \frac{|w||w+1|}{2} + 1$$

per via dei possibili doppioni che possiamo trovare.

Definizione 1.6 (Sottosequenza): La stringa $x \in \Sigma^*$ si dice **sottosequenza** di w se x è ottenuta eliminando 0 o più caratteri da w . L'eliminazione può avvenire in maniera non contigua: posso eliminare qualsiasi carattere, ma la stringa risultante che leggiamo deve contenere i caratteri nello stesso ordine di partenza.

Possiamo dire che un **fattore** è una **sottosequenza contigua**.

Per finire veramente, diamo forse la definizione più importante, quella di **linguaggio**.

Definizione 1.7 (Linguaggio): Un **linguaggio** L , definito su un alfabeto Σ , è un qualunque sottoinsieme di Σ^* , ovvero

$$L \subseteq \Sigma^*.$$

Ora che abbiamo fatto un ripasso siamo pronti per vedere la gerarchia di Chomsky, per poi addentrarci nello studio di alcune classi di questa gerarchia.

2. Gerarchia di Chomsky

Vogliamo cercare di rappresentare in maniera **finita** un oggetto potenzialmente **infinito**, come ad esempio un linguaggio. Per fare ciò, abbiamo a nostra disposizione due modelli potenti:

- il **modello generativo** fornisce delle regole che, applicate da un certo punto di partenza, generano tutte le parole di un linguaggio;
- il **modello riconoscitivo** utilizza un modello di calcolo che prende in input una parola e ci dice se appartiene o meno al linguaggio.

Esempio 2.1: Consideriamo il linguaggio sull'alfabeto $\Sigma = \{ (,) \}$ delle parole ben bilanciate.

Un **modello generativo** per questo linguaggio deve applicare delle regole a partire da una **sorgente** S per derivare tutte le parole del linguaggio. Le regole potrebbero essere:

- la parola vuota ε è ben bilanciata;
- se x è ben bilanciata, allora anche (x) è ben bilanciata;
- se x e y sono ben bilanciate, allora anche xy è ben bilanciata.

Un **modello riconoscitivo** per questo linguaggio è una black-box che, presa una parola, ci dice se essa appartiene o meno al linguaggio. In realtà questa macchina non potrebbe terminare mai, ma ne parleremo più in fondo in questo capitolo. Una macchina per questo linguaggio deve verificare i seguenti fatti:

- il numero di parentesi aperte è uguale al numero di parentesi chiuse, quindi

$$\#_((x) = \#_)(x);$$

- considerato ogni prefisso, il numero di parentesi aperte non deve superare il numero di parentesi chiuse, quindi

$$\forall y \in \Sigma^* \mid x = yz \quad \#_((y) \leq \#_)(y).$$

2.1. Grammatiche

Le **grammatiche** sono un modello generativo molto potente: vediamo come sono definite.

Definizione 2.1.1 (Grammatica): Una **grammatica** è una quadrupla (V, Σ, P, S) definita da:

- V **insieme finito e non vuoto di variabili**. Sono anche dette **simboli non terminali** (o meta-simboli) e sono usate durante il processo di generazione delle parole del linguaggio;
- Σ **insieme finito e non vuoto di simboli terminali**. Sono chiamati così perché appaiono nelle parole generate, a differenza delle variabili che invece non possono essere presenti;
- P **insieme finito e non vuoto di regole di produzione**;
- $S \in V$ **simbolo iniziale o assioma**, il punto di partenza della generazione.

Soffermiamoci brevemente sulle **regole di produzione**: esse sono nella forma

$$\alpha \longrightarrow \beta \mid \alpha \in (V \cup \Sigma)^+ \wedge \beta \in (V \cup \Sigma)^*.$$

La notazione $()^+$ è praticamente l'insieme $()^*$ senza la parola vuota.

Una regola di produzione viene letta come «se ho α allora posso sostituirlo con β ».

L'applicazione delle regole di produzione è alla base del **processo di derivazione**: esso è formato infatti da una serie di **passi di derivazione**, che permettono di generare una parola del linguaggio.

Definizione 2.1.2 (Derivazione in un passo): Date le stringhe $x, y \in (V \cup \Sigma)^*$ diciamo che x **deriva y in un passo**, e si indica con

$$x \Rightarrow y$$

se e solo se

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

Possiamo estendere questa definizione ad un numero definito o arbitrario di passi.

Definizione 2.1.3 (Derivazione in k passi): Date le stringhe $x, y \in (V \cup \Sigma)^*$ diciamo che x **deriva y in $k \geq 0$ passi**, e si indica con

$$x \xRightarrow{k} y$$

se e solo se

$$\exists x_0, \dots, x_k \in (V \cup \Sigma)^* \mid x = x_0 \wedge y = x_k \wedge (\forall i \in \{1, \dots, k\} \quad x_{i-1} \Rightarrow x_i).$$

Con questa definizione, se contiamo $k = 0$ andiamo a derivare x da sé stessa, ma questo caso lo usiamo solo per comodità, non ha una vera e propria applicazione pratica.

Quando **non abbiamo indicazioni** sul numero di passi possiamo:

- usare la notazione

$$x \xRightarrow{*} y$$

per indicare un processo di derivazione che avviene in un **numero generico di passi**, e questo vale se e solo se

$$\exists k \geq 0 \mid x \xRightarrow{k} y;$$

- usare la notazione

$$x \xRightarrow{+} y$$

per indicare un processo di derivazione che avviene in **almeno un passo**, e questo vale se e solo se

$$\exists k > 0 \mid x \xRightarrow{k} y.$$

Definizione 2.1.4 (Linguaggio generato da una grammatica): Il **linguaggio generato** dalla grammatica G è l'insieme

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

In poche parole, $L(G)$ è l'insieme di tutte le stringhe w che si possono ottenere in un certo numero di passi di derivazione a partire dall'assioma S della grammatica. Notiamo che le stringhe che otteniamo sono formate dai soli **caratteri terminali**. Le stringhe intermedie che utilizziamo invece nei vari passi di derivazione sono dette **forme sentenziali**.

Definizione 2.1.5 (Grammatiche equivalenti): Due grammatiche G_1 e G_2 sono **equivalenti** se e solo se

$$L(G_1) = L(G_2).$$

Vediamo qualche grammatica come esempio.

Esempio 2.1.1: Riprendiamo il linguaggio delle parentesi tonde ben bilanciate.

Possiamo definire una grammatica che ha le seguenti regole di produzione:

$$S \longrightarrow \varepsilon \qquad S \longrightarrow (S) \qquad S \longrightarrow SS$$

Esempio 2.1.2: Sia $G = (\{S, A, B\}, \{a, b\}, P, S)$ una grammatica con le seguenti regole di produzione:

$$S \longrightarrow aB \mid bA \qquad A \longrightarrow a \mid aS \mid bAA \qquad B \longrightarrow b \mid bS \mid aBB$$

Questa grammatica genera il linguaggio

$$L(G) = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}.$$

Infatti, ogni forma sentenziale w che generiamo è tale che

$$\#_{\{a,A\}}(w) = \#_{\{b,B\}}(w)$$

e questa relazione viene poi mantenuta trasformando tutte le A e B nei relativi simboli terminali.

Esempio 2.1.3: Definiamo ora la grammatica $G = (\{S, A, B, C, D, E\}, \{a, b\}, P, S)$ che contiene le seguenti regole di produzione:

$$\begin{array}{lll}
 S \rightarrow ABC & AB \rightarrow \varepsilon \mid aAD \mid bAE & DC \rightarrow BaC \\
 EC \rightarrow BbC & Da \rightarrow aD & Db \rightarrow bD \\
 Ea \rightarrow aE & Eb \rightarrow bE & C \rightarrow \varepsilon \\
 aB \rightarrow Ba & & bB \rightarrow Bb
 \end{array}$$

Generando qualche parola ci si accorge che questa grammatica genera il **linguaggio pappagallo**, definito come

$$L(G) = \{ww \mid w \in \Sigma^*\}.$$

2.2. Gerarchia di Chomsky

Negli anni '50 **Noam Chomsky** studia la generazione dei linguaggi formali e crea una **gerarchia di grammatiche formali** che si basa sulla forma delle **regole di produzione** che definiscono la grammatica.

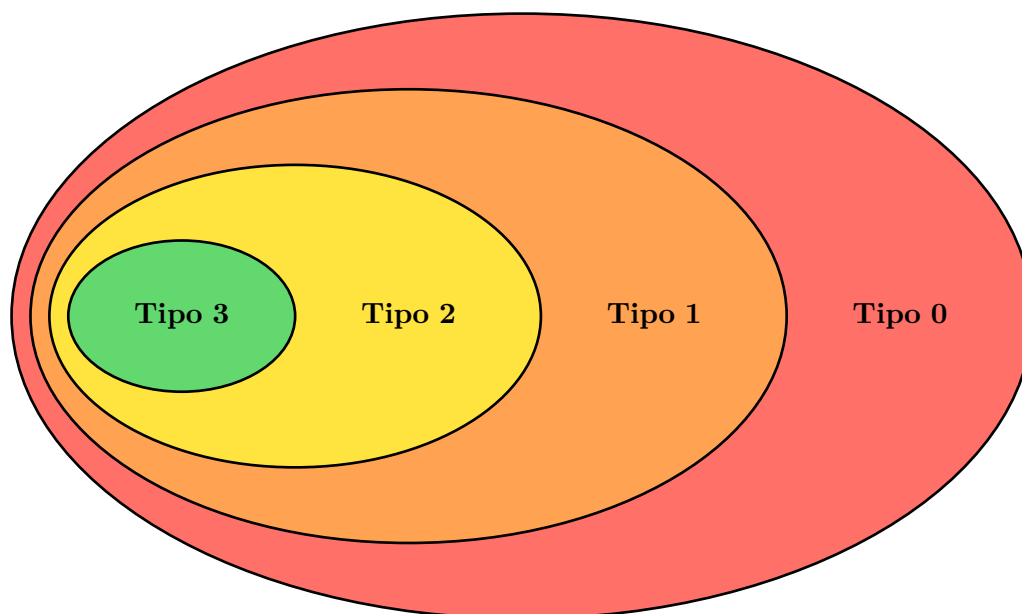
Grammatica	Regole	Modello riconoscitivo
Tipo 0	Nessuna restrizione, sono il tipo più generale	Macchine di Turing
Tipo 1 , dette context-sensitive (o dipendenti dal contesto)	<p>Se $(\alpha \rightarrow \beta) \in P$ allora $\beta \geq \alpha$, ovvero devo generare parole che non sono più corte di quella di partenza</p> <p>Sono dette dipendenti dal contesto perché ogni regola $(A \rightarrow B) \in P$ può essere riscritta come $\alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2$, con $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ che rappresentano il contesto, $A \in V$ e $B \in (V \cup \Sigma)^+$</p>	Automi limitati linearmente
Tipo 2 , dette context-free (o libere dal contesto)	Le regole in P sono del tipo $\alpha \rightarrow \beta$, con $\alpha \in V$ e $\beta \in (V \cup \Sigma)^+$	Automi a pila

Tipo 3 , dette grammatiche regolari	Le regole in P sono del tipo $A \rightarrow aB$ oppure $A \rightarrow a$, con $A, B \in V$ e $a \in \Sigma$	Automi a stati finiti
---	---	------------------------------

La gerarchia che ha definito Chomsky è **propria**, ovvero:

$$L_3 \subset L_2 \subset L_1 \subset L_0.$$

Come vedremo a fine di questo capitolo, questa gerarchia **non esaurisce** tutti i linguaggi possibili: esistono infatti linguaggi che non sono descrivibili in maniera finita con le grammatiche.



Definizione 2.2.1 (Tipo di una grammatica): Sia $L \subseteq \Sigma^*$ un linguaggio. Allora L è di tipo i se e solo se esiste una grammatica G di tipo i tale che

$$L = L(G).$$

La gerarchia data considera dei **modelli deterministici**, ma come cambia considerando invece dei **modelli non deterministici**? Sappiamo che:

- le grammatiche di tipo 3 hanno la stessa potenza computazionale, pagando un costo in termini di descrizione;
- le grammatiche di tipo 2 hanno il modello deterministico strettamente più potente;
- le grammatiche di tipo 1 sono abbastanza complicate, quindi non lo vedremo;
- le grammatiche di tipo 0, come quelle regolari, hanno la stessa potenza computazionale.

Il non determinismo comunque è una nozione del **riconoscitore** che sto usando:

- nel determinismo il riconoscitore può fare una scelta alla volta;
- nel non determinismo può fare più scelte contemporaneamente.

Nelle grammatiche è difficile catturare questa nozione, perché esse lo hanno **intrinsecamente**, perché le derivazioni le applico tutte assieme per ottenere le stringhe del linguaggio.

2.3. Decidibilità

Se una grammatica è di tipo 1 allora possiamo costruire una macchina che sia in grado di dire, in tempo finito, se una parola appartiene o meno al linguaggio generato da quella grammatica. Questa macchina è detta **verificatore**, e si dice che le grammatiche di tipo 1 sono **decidibili**.

Teorema 2.3.1 (Decidibilità dei linguaggi context-sensitive): I linguaggi di tipo 1 sono **ricorsivi**.

Con **ricorsività** non intendiamo le procedure ricorsive, ma si intende una procedura che è calcolabile automaticamente. Nei linguaggi, un qualcosa di ricorsivo intende una macchina che, data una stringa x in input, riesce a rispondere a $x \in L$ terminando sempre dicendo **SI** o **NO**. Si usano i termini **ricorsivo** e **decidibile** come sinonimi.

Dimostrazione 2.3.1.1: In una grammatica di tipo 1 l'unico vincolo è sulla lunghezza delle produzioni, ovvero non possono mai accorciarsi.

In input ho una stringa $w \in \Sigma^*$ la cui lunghezza è $|w| = n$. Ho una grammatica G di tipo 1 e mi chiedo se $w \in L(G)$. Per rispondere a questo, devo cercare w nelle forme sentenziali, ma possiamo limitarci a quelle che non superano la lunghezza n : infatti, visto che la lunghezza aumenta sempre (o al massimo rimane uguale) posso arrivare al massimo alle stringhe di lunghezza n e controllare solo quelle.

Definiamo quindi gli insiemi

$$T_i = \left\{ \gamma \in (V \cup \Sigma)^{\leq n} \mid S \xRightarrow{\leq i} \gamma \right\} \quad \forall i \geq 0.$$

Calcoliamo induttivamente questi insiemi.

Se $i = 0$ non eseguo nessuna derivazione, quindi

$$T_0 = \{S\}.$$

Supponiamo di aver calcolato T_{i-1} . Ma allora

$$T_i = T_{i-1} \cup \left\{ \gamma \in (V \cup \Sigma)^{\leq n} \mid \exists \beta \in T_{i-1} \mid \beta \Rightarrow \gamma \right\}.$$

Noi partendo da T_0 calcoliamo tutti i vari insiemi ottenendo una serie di T_i . Per come abbiamo definito gli insiemi, sappiamo che

$$T_0 \subseteq T_1 \subseteq T_2 \subseteq \dots \subseteq (V \cup \Sigma)^{\leq n}$$

e l'ultima inclusione è vera perché ho fissato la lunghezza massima, non voglio considerare di più perché vogliamo w di lunghezza n .

La grandezza dell'insieme $(V \cup \Sigma)^{\leq n}$ è finita, quindi anche andando molto avanti con le computazioni prima o poi arrivo ad un certo punto dove non posso più aggiungere niente, ovvero vale che

$$\exists i \in \mathbb{N} \mid T_i = T_{i-1}.$$

Ora è inutile andare avanti, questo T_i è l'insieme di tutte le stringhe che riesco a generare nella grammatica. Ora mi chiedo se $w \in T_i$, che posso fare molto facilmente scorrendo l'insieme.

Ma allora G è decidibile. ■

Ci rendiamo conto che questa soluzione è **altamente inefficiente**: infatti, in tempo polinomiale non riusciamo a fare questo nelle tipo 1, ma è una soluzione che ci garantisce la decidibilità, quindi sium.

Teorema 2.3.2 (Semi-decidibilità dei linguaggi di tipo 0): I linguaggi di tipo 0 sono ricorsivamente enumerabili.

Dimostrazione 2.3.2.1: In una grammatica di tipo 0 non abbiamo vincoli da considerare.

In input ho una stringa $w \in \Sigma^*$ la cui lunghezza è $|w| = n$. Ho una grammatica G di tipo 0 e, come prima, mi chiedo se $w \in L(G)$. Per rispondere a questo, devo cercare w nelle forme sentenziali, ma a differenza di prima non possiamo limitarci a quelle che non superano la lunghezza n : infatti, visto che le forme sentenziali si possono accorciare posso anche superare di molto la lunghezza n e poi sperare di tornare indietro in qualche modo.

Definiamo quindi gli insiemi

$$U_i = \left\{ \gamma \in (V \cup \Sigma)^* \mid S \xRightarrow{\leq i} \gamma \right\} \quad \forall i \geq 0.$$

Calcoliamo induttivamente questi insiemi.

Se $i = 0$ non eseguo nessuna derivazione, quindi

$$U_0 = \{S\}.$$

Supponiamo di aver calcolato U_{i-1} . Vogliamo calcolare

$$U_i = U_{i-1} \cup \{ \gamma \in (V \cup \Sigma)^* \mid \exists \beta \in U_{i-1} : \beta \Rightarrow \gamma \}.$$

Noi partendo da U_0 calcoliamo tutti i vari insiemi ottenendo una serie di U_i . Per come abbiamo definito gli insiemi, sappiamo che

$$U_0 \subseteq U_1 \subseteq U_2 \subseteq \dots \subseteq (V \cup \Sigma)^*.$$

A differenza di prima, la grandezza dell'insieme $(V \cup \Sigma)^*$ è infinita, quindi non ho più l'obbligo di stopparmi ad un certo punto per esaurimento delle stringhe generabili.

Come rispondiamo a $w \in L(G)$? Iniziamo a costruire i vari insiemi U_i e ogni volta che termino la costruzione mi chiedo se $w \in U_i$:

- se questo è vero allora rispondo **SI**;
- in caso contrario vado avanti con la costruzione.

frac(4 pi, lambda) Vista la cardinalità infinita dell'insieme che fa da container, potrei andare avanti all'infinito (a meno di ottenere due insiemi consecutivi identici, in tale caso rispondo NO).

Ma allora G è semi-decidibile. ■

Usiamo la dicitura **ricorsivamente enumerabile** perché ogni volta che costruisco un insieme U_i posso prendere le stringhe $w \in \Sigma^*$ appena generate ed elencarle, quindi enumerarle una per una.

2.4. Introduzione della parola vuota

Introduciamo il **problema della parola vuota**. Dalle grammatiche di tipo 1 a salire abbiamo il vincolo di non poter scendere di lunghezza con le derivazioni, quindi diciamo che la parola vuota ε non la vedremo mai come lato destro di una derivazione. Eppure, ogni tanto la parola vuota dovrebbe appartenere al linguaggio generato da una grammatica. Come possiamo risolvere questo problema, senza far decadere l'intera gerarchia?

Una possibile soluzione è **spezzare le regole di produzione**.

Partiamo da una grammatica $G = (V, \Sigma, P, S)$ di tipo 1 e definiamo una nuova grammatica

$$G' = (V', \Sigma, P', S')$$

tale che $L(G) = L(G')$. Vediamo le componenti di questa grammatica:

- l'**insieme delle variabili** contiene un nuovo elemento, il **nuovo assioma** S' , ovvero

$$V' = V \cup \{S'\};$$

- l'**insieme delle produzioni** mantiene le regole vecchie ma aggiunge due nuove regole, ovvero

$$P' = P \cup \{S' \rightarrow \varepsilon\} \cup \{S' \rightarrow S\}$$

dove:

- ▶ la prima regola permette di generare la parola vuota ε ;
- ▶ la seconda regola permette di far partire la computazione della vecchia grammatica;
- l'**assioma** S' ci permette la generazione della parola vuota ma dobbiamo garantire che non appaia mai nel lato destro delle produzioni.

Con questi accorgimenti ora riusciamo a generare anche la parola vuota: infatti, questo lo possiamo fare all'inizio partendo da S' . Se non ci interessa la parola vuota facciamo partire, sempre da S' , la computazione della vecchia grammatica.

Come cambia la gerarchia considerando anche la parola vuota? Abbiamo che:

- le grammatiche di tipo 1 mantengono la clausola $|\beta| \geq |\alpha|$ ma è possibile ottenere ε da S' purché S' non appaia mai nel lato destro delle produzioni;
- le grammatiche di tipo 2 modificano la notazione $()^+$ in $()^*$ nel lato destro delle produzioni senza isolare in modo specifico ε perché questo non crea problemi;
- le grammatiche di tipo 3 seguono le precedenti.

Queste particolari produzioni che considerano la parole vuota sono dette ε -produzioni.

2.5. Linguaggi non esprimibili tramite grammatiche finite

Vediamo infine dei linguaggi che non possiamo esprimere tramite **grammatiche finite**. Per fare ciò useremo la famosissima **dimostrazione per diagonalizzazione** di Cantor.

Esempio 2.5.1: Sono più i numeri pari o i numeri dispari? Sono più i numeri pari o i numeri interi? Sono più le coppie di numeri naturali o i naturali stessi?

Per rispondere a queste domande si usa la definizione di **cardinalità**, e tutti questi insiemi che abbiamo ce l'hanno uguale. Anzi, diciamo di più: tutti questi insiemi sono grandi quanto i naturali, perché esistono funzioni biettive tra questi insiemi e l'insieme \mathbb{N} .

Esempio 2.5.2: Sono più i sottoinsiemi di naturali o i naturali stessi?

In questo caso, sono di più i sottoinsiemi, che hanno la **cardinalità del continuo**. Per dimostrare questo useremo una dimostrazione per diagonalizzazione.

Teorema 2.5.1: Vale

$$\mathbb{N} \approx 2^{\mathbb{N}}.$$

Dimostrazione 2.5.1.1: Per assurdo sia $\mathbb{N} \approx 2^{\mathbb{N}}$, ovvero ogni elemento di $2^{\mathbb{N}}$ è **listabile**.

Creiamo una tabella booleana M indicizzata sulle righe dai sottoinsiemi di naturali S_i e indicizzata sulle colonne dai numeri naturali. Per ogni insieme S_i abbiamo sulla riga la funzione caratteristica, ovvero

$$M[i, j] = \begin{cases} 1 & \text{se } j \in S_i \\ 0 & \text{se } j \notin S_i \end{cases}.$$

Creiamo l'insieme

$$S = \{x \in \mathbb{N} \mid x \notin S_x\},$$

ovvero l'insieme che prende tutti gli elementi 0 della diagonale di M . Questo insieme non è presente negli insiemi S_i listati perché esso è diverso da ogni S_i in almeno una posizione, ovvero la diagonale.

Abbiamo ottenuto un assurdo, ma allora $\mathbb{N} \approx 2^{\mathbb{N}}$. ■

Esempio 2.5.3: Sono più le stringhe o i numeri naturali?

Questo è facile: basta trasformare ogni stringa in un numero naturale con una qualche codifica a nostra scelta.

Ora che abbiamo a disposizione queste nozioni possiamo dimostrare il seguente teorema.

Teorema 2.5.2: Esistono linguaggi che non sono descrivibili da grammatiche finite.

Dimostrazione 2.5.2.1: Prendiamo una grammatica $G = (V, \Sigma, P, S)$.

Per descriverla devo dire come sono formati i vari campi della tupla. Cosa uso per descriverla? Sto usando dei simboli come lettere, numeri, parentesi, eccetera, quindi la grammatica è una descrizione che possiamo fare sotto forma di stringa. Visto quello che abbiamo da poco dimostrato, ogni grammatica la possiamo descrivere come stringa, e quindi come un numero intero. Siano G_i tutte queste grammatiche, che sono appunto listabili.

Consideriamo ora, per ogni grammatica G_i , l'insieme $L(G_i)$ delle parole generate dalla grammatica G_i , ovvero il linguaggio generato da G_i . Mettiamo dentro L tutti questi linguaggi.

Per assurdo, siano tutti questi linguaggi listabili, ovvero $\mathbb{N} \sim 2^L$.

Come prima, creiamo una tabella M indicizzata sulle righe dai linguaggi $L(G_i)$ e indicizzata sulle colonne dalle stringhe x_i che possiamo però considerare come naturali. La matrice M ha sulla riga i -esima la funzione caratteristica di $L(G_i)$, ovvero

$$M[i, j] = \begin{cases} 1 & \text{se } x_j \in L(G_i) \\ 0 & \text{se } x_j \notin L(G_i) \end{cases}.$$

In poche parole, abbiamo 1 nella cella $M[i, j]$ se e solo se la stringa x_j viene generata da G_i .

Costruiamo ora l'insieme

$$LG = \{x_i \in \mathbb{N} \mid x_i \notin L(G_i)\},$$

ovvero l'insieme di tutte le stringhe x_i che non sono generate dalla grammatica G_i con lo stesso indice i . Come prima, questo insieme non è presente in L perché differisce da ogni insieme presente in almeno una posizione, ovvero quello sulla diagonale.

Siamo ad un assurdo, ma allora $\mathbb{N} \not\sim 2^L$. ■

Parte II — Linguaggi regolari

1. Automi a stati finiti deterministici

Nel contesto delle grammatiche di tipo 3 andiamo ad utilizzare le **macchine a stati finiti** per stabilire se, data una stringa x , essa appartiene o meno ad un dato linguaggio. Le macchine a stati finiti da ora le chiameremo anche **FSM** (Finite State Machine) o, nel caso delle macchine deterministiche, **DFA** (Deterministic Finite Automata).

1.1. Definizione

Un DFA è un dispositivo formato da un **nastro**, che contiene l'input x da esaminare disposto carattere per carattere uno per cella del nastro da sinistra verso destra. Abbiamo anche una **testina** read-only che punta alle celle del nastro e un **controllo a stati finiti**. Il numero di stati, come si capisce, sono in numero finito, e soprattutto sono **fissati**, ovvero non dipendono dalla grandezza dell'input. Infine, il modello base che usiamo per ora è quello delle FSM **one-way**, ovvero quello che usa una testina che va sinistra verso destra senza poter tornare indietro.

All'accensione della macchina il controllo si trova nello **stato iniziale** q_0 con la testina sul primo carattere dell'input. Ad ogni passo della computazione la testina legge un carattere e, in base a questo e allo stato corrente, calcola lo stato prossimo. Questo spostamento avviene grazie alla **funzione di transizione**, che vedremo dopo. Arrivati alla fine dell'input grazie alla funzione di transizione, la macchina deve rispondere **SI** o **NO**.

Definizione 1.1.1 (DFA): Un DFA è una **quintupla**

$$A = (Q, \Sigma, \delta, q_0, F)$$

formata da:

- Q **insieme finito di stati**;
- Σ **alfabeto** di input;
- δ **funzione di transizione**;
- $q_0 \in Q$ **stato iniziale**;
- $F \subseteq Q$ **insiemi degli stati finali**.

La funzione di transizione, che non abbiamo ancora definito formalmente, è il **programma dell'automa**, il **motore** che ci manda avanti. Essa è una funzione

$$\delta : Q \times \Sigma \longrightarrow Q$$

che, dati il simbolo letto dalla testina e lo stato corrente, mi dice in che stato muovermi.

La funzione di transizione spesso è comodo scriverla in **forma tabellare**, con le righe indicizzate dagli stati, le colonne indicizzate dai simboli e nelle celle inseriamo gli stati prossimi.

Può essere comodo anche **disegnare** l'automa. Esso è un **grafo orientato**, con i **vertici** che rappresentano gli stati e gli **archi** che rappresentano le transizioni. Gli archi sono **etichettati** dai simboli di Σ che causano una certa transizione. Lo **stato iniziale** è indicato con una freccia che arriva dal nulla, mentre gli **stati finali** sono indicati con un doppio cerchio o con una freccia che va nel nulla, ma quest'ultima convenzione è francese e noi non lo siamo, viva le lumache.

Dobbiamo modificare leggermente la funzione di transizione: a noi piacerebbe averla definita sulle stringhe e non sui caratteri. Definiamo quindi l'**estensione** di δ come la funzione

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

definita induttivamente come

$$\begin{aligned}\delta^*(q, \varepsilon) &= q \\ \delta^*(q, xa) &= \delta(\delta^*(q, x), a) \mid x \in \Sigma^* \wedge a \in \Sigma.\end{aligned}$$

Per non avere in giro troppo nomi usiamo δ^* con il nome δ anche per le stringhe, è la stessa cosa.

Noi **accettiamo** se finiamo in uno stato finale. Il **linguaggio accettato** da A è l'insieme

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

1.2. Esempi

Vediamo un po' di esempi che ci permettono di introdurre anche una serie di situazioni particolari.

Diamo subito una distinzione dei problemi che abbiamo sugli automi:

- se abbiamo in mano un automa e dobbiamo descrivere il linguaggio che riconosce, siamo davanti ad un **problema di analisi**;
- se abbiamo in mano un linguaggio e dobbiamo scrivere un automa che lo riconosce, siamo davanti ad un **problema di sintesi**.

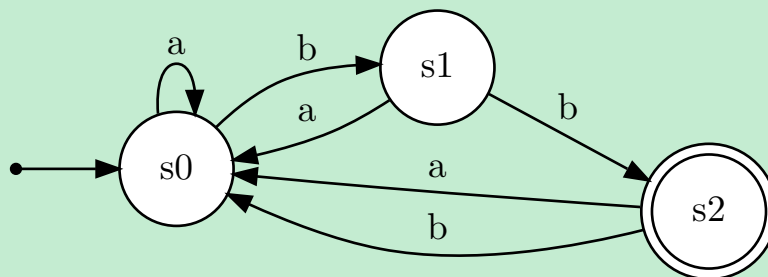
Esempio 1.2.1: Sia $A = (Q, \Sigma, \delta, q_0, F)$ tale che:

- $Q = \{s_0, s_1, s_2\}$;
- $\Sigma = \{a, b\}$;
- $q_0 = s_0$;
- $F = s_2$.

Diamo una **rappresentazione tabellare** della funzione di transizione δ . Essa è

	a	b
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_0

Disegniamo anche l'automa A avendo a disposizione la rappresentazione di δ .



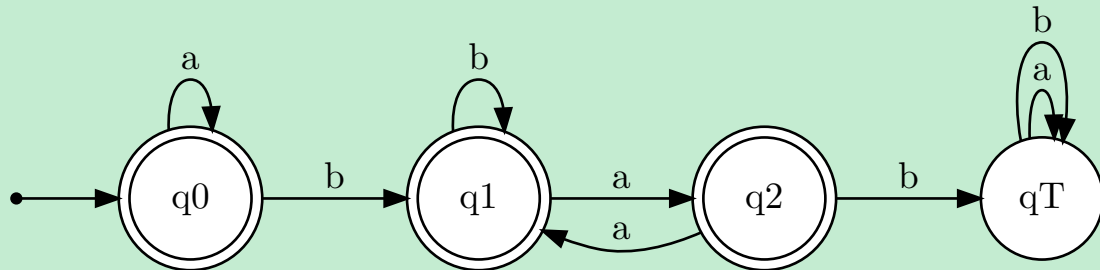
Il linguaggio che riconosce questo automa è

$$L = \{x \in \Sigma^* \mid \text{il più lungo suffisso di } x \text{ formato solo da } b \text{ è lungo } 3k + 2 \mid k \geq 0\}.$$

Esempio 1.2.2: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

$$L = \{x \in \Sigma^* \mid \text{tra ogni coppia di } b \text{ successive vi è un numero di } a \text{ pari}\}.$$

Costruiamo un automa **deterministico** per L .



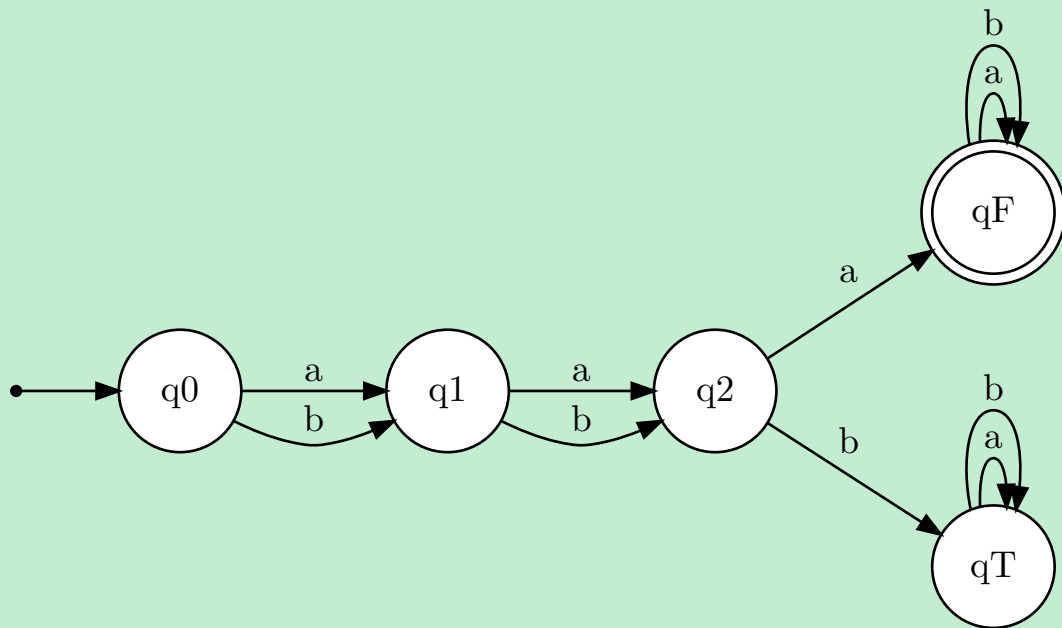
Come vediamo dall'esempio precedente, abbiamo uno stato particolare q_T che è detto **stato trappola**: esso viene utilizzato come «punto di arrivo» per esaurire la lettura dell'input e non accettare la stringa data in input. Finiamo in questo stato se, in uno stato q , leggiamo un carattere che rende la stringa non presente in L .

Lo stato trappola è opzionale: per semplicità, quando un automa **non è completo**, ovvero uno stato non ha un arco per un carattere, si assume che quell'arco vada a finire in uno stato trappola. Questa semplificazione permette di disegnare automi molto più compatti, ma io sono un precisino e devo avere tutti gli stati disegnati.

Esempio 1.2.3: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

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

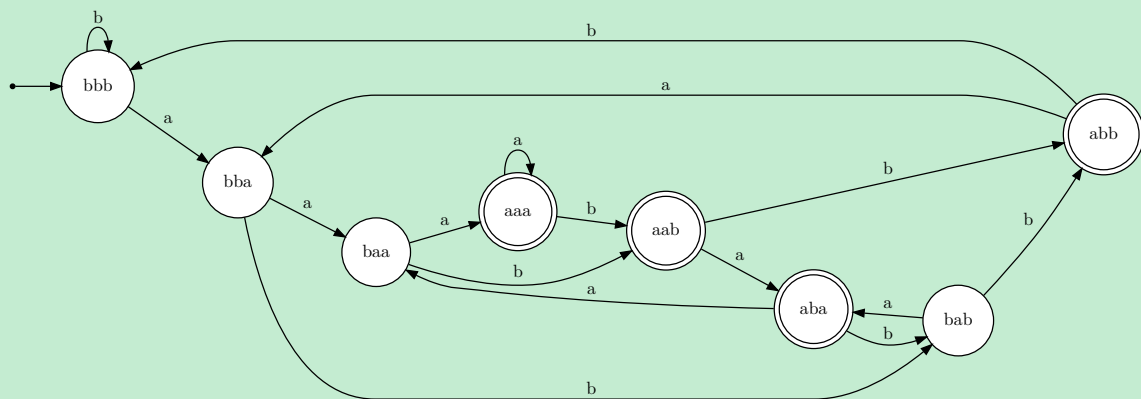
Costruiamo un automa deterministico per L .



Esempio 1.2.4: Sia $\Sigma = \{a, b\}$, vogliamo ora trovare un automa per il linguaggio

$$L = \{x \in \Sigma^* \mid \text{il terzo simbolo di } x \text{ da destra è una } a\}.$$

Costruiamo un automa deterministico per L . Qua l'idea è ricordarsi una finestra di 3 simboli e grazie a questa vediamo se il primo carattere che definisce lo stato è una a .



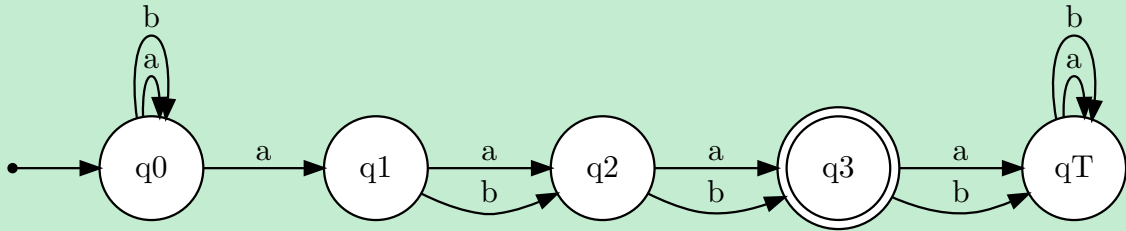
Ci servono per forza 8 stati o possiamo fare meglio? Abbiamo trovato la strada migliore?

Quando introdurremo il concetto di **distinguibilità** vedremo che questo è l'automa migliore che possiamo costruire per questo linguaggio.

2. Automi a stati finiti non deterministici

L'ultimo esempio del capitolo precedente ci ha richiesto 2^n stati. Abbiamo poi detto che con la nozione di **distinguibilità** dimostreremo che non ci sono DFA con meno stati di quello che abbiamo costruito. Ma se invece utilizzassimo degli **automi non deterministici**?

Esempio 2.1: Vediamo un automa non deterministico per il linguaggio dell'ultimo esempio del capitolo precedente.



Abbiamo usato un numero di stati uguale a $n + 1$ (escluso quello trappola), dove n è la posizione da destra del carattere richiesto.

2.1. Definizione

Nello scorso esempio abbiamo generato un **automa non deterministico**. Infatti, dallo stato q_0 noi abbiamo la possibilità di scegliere se restare in q_0 o andare in q_1 , ovvero abbiamo più scelte di transizioni in uno stesso stato. Che significato diamo a questo? Noi non sappiamo a che punto siamo della stringa, quindi usiamo il non determinismo come una **scommessa**: scommetto che, quando sono in q_0 , io sia nel terzultimo carattere, e che quindi riuscirò a finire nello stato q_3 .

Gli **automi non deterministici**, o **NFA**, sono definiti dalla quintupla

$$A = (Q, \Sigma, \delta, q_0, F)$$

che differisce da quella dei DFA solo nella **funzione di transizione**. Essa è la funzione

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

che, dati lo stato corrente e il carattere letto dalla testina, mi manda in un insieme di stati possibili.

Prima di definire formalmente l'accettazione di una stringa da parte di un automa non deterministico, definiamo l'**estensione** di δ come la funzione

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

definita induttivamente come

$$\begin{aligned} \delta^*(q, \varepsilon) &= \{q\} \\ \delta^*(q, xa) &= \bigcup_{p \in \delta^*(q, x)} \delta(p, a) \mid x \in \Sigma^* \wedge a \in \Sigma. \end{aligned}$$

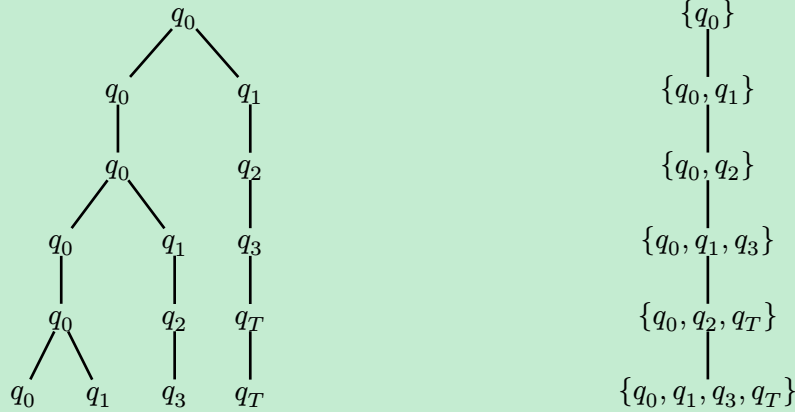
Come prima, per non avere in giro troppo nomi, usiamo δ^* con il nome δ anche per le stringhe.

Quando **accettiamo** una stringa? Avendo teoricamente la possibilità di fare **infinite computazioni parallele**, visto che ad ogni passo posso sdoppiare la mia computazione, ci basta avere almeno un percorso che finisce in uno stato finale.

Il **linguaggio riconosciuto** dall'automa A non deterministico è

$$L(A) = \{w \in \Sigma^* \mid \delta^*(q_0, w) \cap F \neq \emptyset\}.$$

Esempio 2.1.1: Considerando l'automa precedente, scriviamo l'albero di computazione che viene generato dall'automa mentre cerca di riconoscere la stringa $x = ababa$.



Visto che raggiungiamo, all'ultimo livello dell'albero, almeno una volta lo stato finale q_3 , la stringa x viene accettata dall'automa.

2.2. Confronto tra DFA e NFA

Banalmente, ogni automa deterministico è anche un automa non deterministico nel quale abbiamo, per ogni stato, al massimo un arco uscente etichettato con lo stesso carattere. In poche parole, abbiamo sempre una sola scelta. Ma allora la classe dei linguaggi riconosciuti da DFA è inclusa nella classe dei linguaggi riconosciuti da NFA.

Ma vale anche il viceversa: ogni automa non deterministico può essere trasformato in un automa deterministico con una costruzione particolare, detta **costruzione per sottoinsiemi**.

Dato $A = (Q, \Sigma, \delta, q_0, F)$ un NFA, e costruisco

$$A' = (Q', \Sigma, \delta', q'_0, F')$$

un DFA tale che:

- $Q' = 2^Q$, ovvero gli **stati** sono tutti i possibili sottoinsiemi;
- $\delta' : Q' \times \Sigma \rightarrow Q'$ è la nuova **funzione di transizione** che ci permette di navigare tra i possibili sottoinsiemi, ed è tale che

$$\delta'(\alpha, a) = \bigcup_{q \in \alpha} \delta(q, a);$$

- $q'_0 = \{q_0\}$ nuovo **stato iniziale**;
- $F' = \{\alpha \in Q' \mid \alpha \cap F \neq \emptyset\}$ nuovo **insieme degli stati finali**.

Come vediamo, il non determinismo è estremamente comodo, perché ci permette di rendere molto compatta la rappresentazione degli automi, ma è irrealistico pensare di fare sempre la scelta giusta nelle scommesse.

2.3. Forme di non determinismo

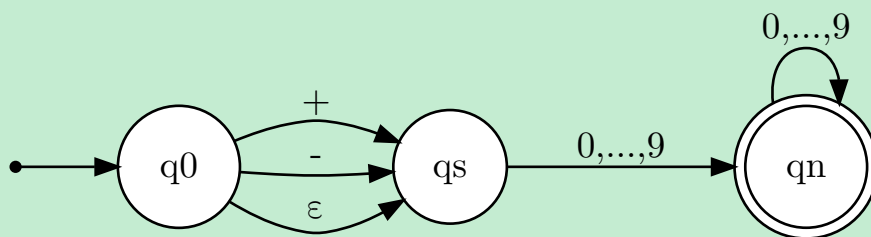
Il non determinismo sulle **transizioni**, ovvero avere più opzioni per la stessa lettera a partire da uno stato, non è l'unica forma di non determinismo che abbiamo.

Infatti, un'altra forma di non determinismo è quella di avere **stati iniziali multipli**, ovvero poter scegliere più punti di partenza. Come potenza siamo uguali agli automi deterministici: basta fare una **costruzione per sottoinsiemi** e abbiamo sistemato tutto.

L'ultima forma di non determinismo che abbiamo è quella delle ε -**produzioni**: esse sono transizioni di stato etichettate dalla ε che permettono di spostarsi da uno stato all'altro senza leggere un carattere della stringa da riconoscere.

Che applicazioni può avere una forma del genere? Nei **compilatori** questo approccio è comodissimo per riconoscere dei numeri che possono essere indicati con o senza segno.

Esempio 2.3.1: Se $\Sigma = \{0, \dots, 9, +, -\}$ definiamo un numero come una sequenza non vuota di cifre, con un segno iniziale opzionale.



La epsilon mossa indica una opzionalità: potremmo leggere il prossimo carattere stando nello stato q_0 oppure nello stato q_s .

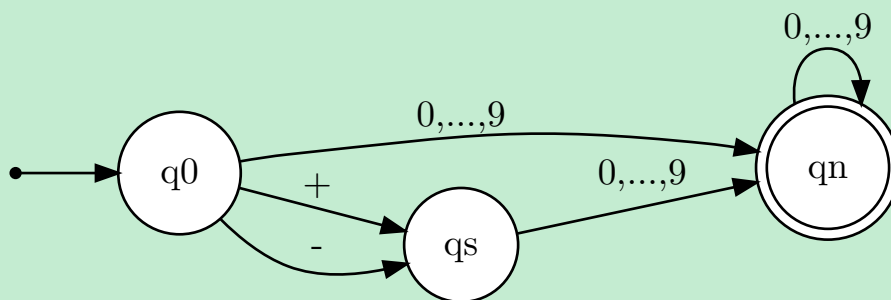
Questa soluzione aumenta la potenza dell'automa? **NO**: ogni sequenza nella forma

$$p \xrightarrow{\varepsilon} p' \xrightarrow{a} q' \xrightarrow{\varepsilon} q$$

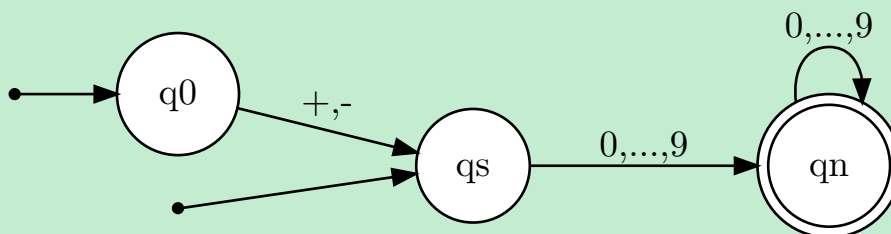
può essere tradotta nella transizione

$$p \xrightarrow{a} q.$$

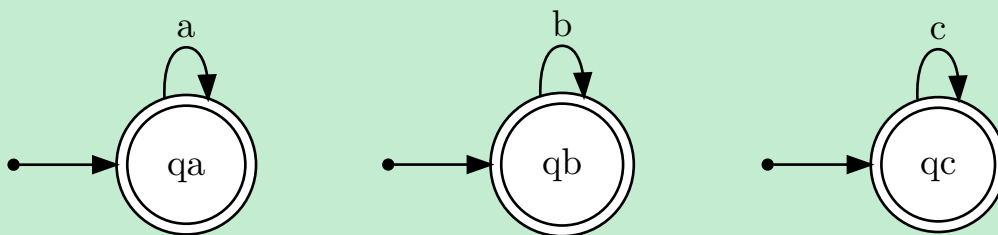
Esempio 2.3.2: Andiamo a rimuovere la ε -transizione usando le sequenze appena descritte.



Una soluzione analoga rimuove le ε -transizioni inserendo degli stati iniziali multipli, ma questo mantiene ancora la forma di non determinismo dell'automa e non migliora la potenza, visto che basta trasformare l'NFA in un DFA con la costruzione per sottoinsiemi e come stato iniziale si avranno più di due elementi.

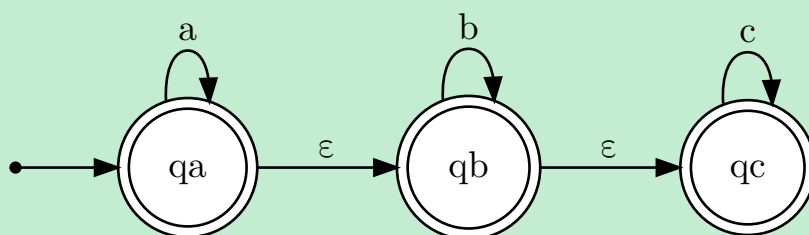


Esempio 2.3.3: Ci vengono dati tre automi, che riconoscono sequenze di a , b e c arbitrarie.

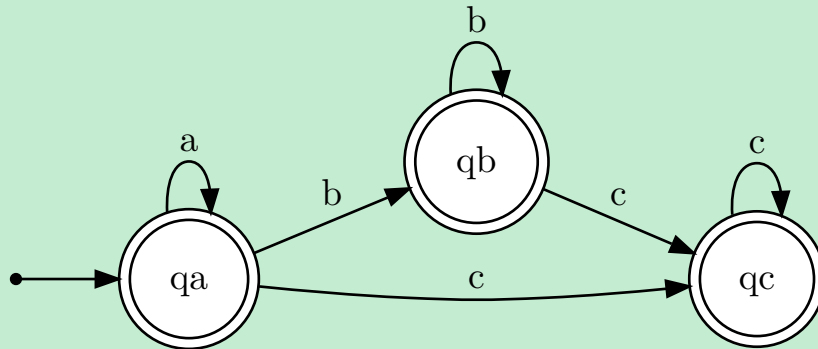


Vogliamo costruire un automa che utilizzi le ε -transizioni usando questi tre moduli per riconoscere il linguaggio

$$L = \{a^n b^m c^h \mid m, n, h \geq 0\}.$$

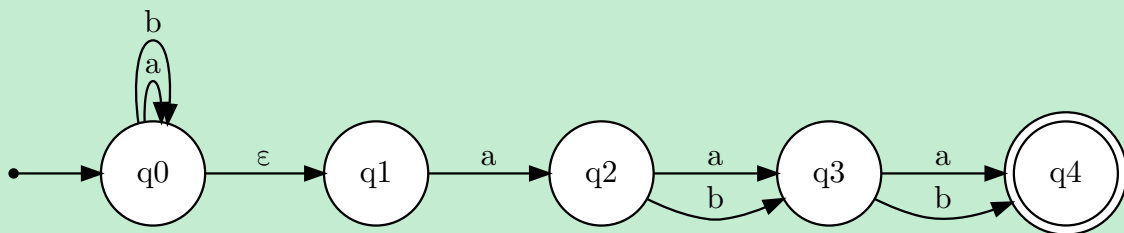


Come lo rendiamo deterministico? Sicuramente non andiamo ad utilizzare gli stati iniziali multipli, che qui ci starebbero molto bene, ma appunto vogliamo un comportamento deterministico.



Siamo nel deterministico, ma l'automa di prima è molto più leggibile di questo.

Esempio 2.3.4: Riprendiamo il linguaggio L_n delle stringhe con l' n -esimo carattere da destra uguale ad una a . Avevamo visto un NFA sulle transizioni, vediamo uno non deterministico sulle ε -transizioni fissando il valore a $n = 3$.



La scommessa qua l'abbiamo messa nel primo stato, che cerca di indovinare se sia arrivato o meno al terzultimo carattere. Il numero di stati, per L_n generico, è $n + 2$.

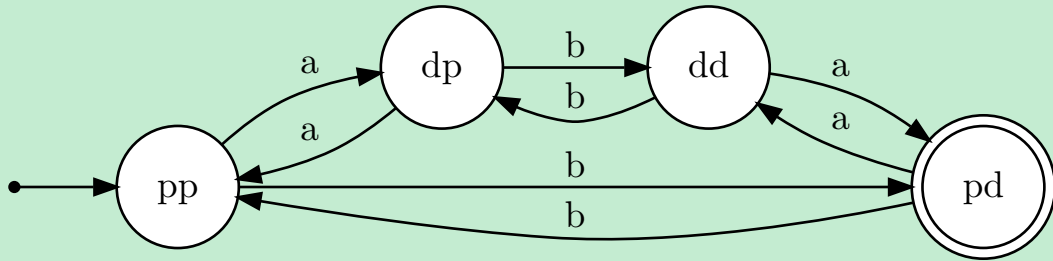
3. Numero minimo di stati

3.1. Distinguibilità

Vediamo un esempio per introdurre il concetto di questa sezione.

Esempio 3.1.1: Sia $\Sigma = \{a, b\}$ e vogliamo un automa che riconosca il linguaggio

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



Ogni stato si ricorda il numero di a e b modulo 2 che ha incontrato.

Possiamo usare meno stati per scrivere un automa per questo linguaggio? Sembra di no, ma non siamo rigorosi. Vediamo un criterio per dire ciò. Ragioniamo sui linguaggi e non sugli automi.

Definizione 3.1.1 (Distinguibilità): Sia $L \subseteq \Sigma^*$ un linguaggio. Date $x, y \in \Sigma^*$, allora esse sono **distinguibili** per L se

$$(xz \in L \wedge yz \notin L) \vee (xz \notin L \wedge yz \in L).$$

In poche parole, riesco a trovare una stringa $z \in \Sigma^*$ tale che, se attacco z alle due stringhe x e y , da una parte mi trovo in L , dall'altra sono fuori L .

Teorema 3.1.1 (Teorema della distinguibilità): Sia $L \subseteq \Sigma^*$ e sia $X \subseteq \Sigma^*$ un insieme tale che tutte le coppie di stringhe $x, y \in X$, con $x \neq y$, sono distinguibili. Allora ogni automa deterministico che accetta L ha almeno $|X|$ stati.

Dimostrazione 3.1.1.1: Sia $X = \{x_1, \dots, x_n\}$ e sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA che accetta il linguaggio L . Definiamo gli stati

$$p_i = \delta(q_0, x_i) \quad \forall i = 1, \dots, n$$

che raggiungiamo dallo stato iniziale usando gli stati x_i di X . In poche parole,

$$\begin{array}{c}
x_0 \\
q_0 \rightsquigarrow p_0 \\
\vdots \\
x_n \\
q_0 \rightsquigarrow p_n
\end{array}$$

Per assurdo, supponiamo che $|Q| < n$. Ma allora esistono due stati tra i vari p_i che sono raggiunti da due stringhe diverse, ovvero

$$\exists i \neq j \mid p_i = p_j.$$

Per ipotesi x_i e x_j sono due stringhe distinguibili, quindi esiste una stringa $z \in \Sigma^*$ che le distingue. Ma partendo dallo stesso stato $p_i = p_j$ e applicando z vado per entrambe le stringhe in uno stato finale o in uno stato non finale.

Ma questo è un assurdo perché va contro la definizione di distinguibilità, quindi non può succedere che

$$|Q| < n \implies |Q| \geq n. \quad \blacksquare$$

Esempio 3.1.2: Trovare un insieme di stringhe distinguibili per il linguaggio precedente.

	ε	a	b	ab
ε	—	b	b	b
a	b	—	ab	ab
b	b	ab	—	ε
ab	b	ab	ε	—

È comodo usare una stringa per ogni stato dell'automa.

Come vediamo, questo teorema è un'arma molto potente: oltre alla possibilità di dare dei **lower bound** al numero di stati di un automa, questo ci permette anche di dire se un linguaggio è di tipo 3 o meno. Infatti, se riusciamo a trovare un insieme X per un linguaggio L che ha un numero infinito di stringhe distinguibili, allora L non può essere riconosciuto da un automa a **STATI FINITI**.

Esempio 3.1.3: Riprendiamo il linguaggio della a in terza posizione da destra e diamogli un nome. Dato l'alfabeto $\Sigma = \{a, b\}$, sia

$$L_3 = \{x \in \Sigma^* \mid \text{il terzo simbolo di } x \text{ da destra è una } a\}.$$

Avevamo visto un DFA per L che prendeva una finestra di 3 simboli, usando 8 stati. Possiamo farlo con meno di 8 stati? Usiamo il teorema precedente e vediamo che succede.

Se scegliamo $X = \Sigma^3$, date due stringhe $\sigma, \gamma \in X$ tali che

$$\sigma = \sigma_1 \sigma_2 \sigma_3 \quad | \quad \gamma = \gamma_0 \gamma_1 \gamma_2$$

allora queste due stringhe le riusciamo a distinguere in base ad una delle posizioni nelle quali hanno un carattere diverso. Infatti, visto che

$$\exists i \mid \sigma_i \neq \gamma_i$$

possiamo affermare che:

- se $i = 1$ allora scelgo $z = \varepsilon$;
- se $i = 2$ allora scelgo $z \in \{a, b\}$;
- se $i = 3$ allora scelgo $z \in \{a, b\}^2$.

Con questa costruzione, noi «rimuoviamo» i caratteri prima della posizione i e aggiungiamo in fondo una qualsiasi sequenza della stessa lunghezza. Abbiamo ottenuto una stringa della stessa lunghezza che però ora ha in prima posizione i due caratteri diversi esattamente nella posizione dove dovremmo avere una a .

Cerchiamo di generalizzare questo concetto.

Esempio 3.1.4: Dato l'alfabeto $\Sigma = \{a, b\}$, chiamiamo

$$L_n = \{x \in \Sigma^* \mid \text{l}'n\text{-esimo simbolo di } x \text{ da destra è una } a\}.$$

Come prima, definisco $X = \Sigma^n$ insieme di stringhe nella forma $\sigma = \sigma_1 \dots \sigma_n$.

Date due stringhe $\sigma, \gamma \in \Sigma^n$ allora

$$\exists i \mid \sigma_i \neq \gamma_i.$$

Questa posizione può essere la prima o una a caso, è totalmente indifferente.

Scelgo di attaccare una stringa

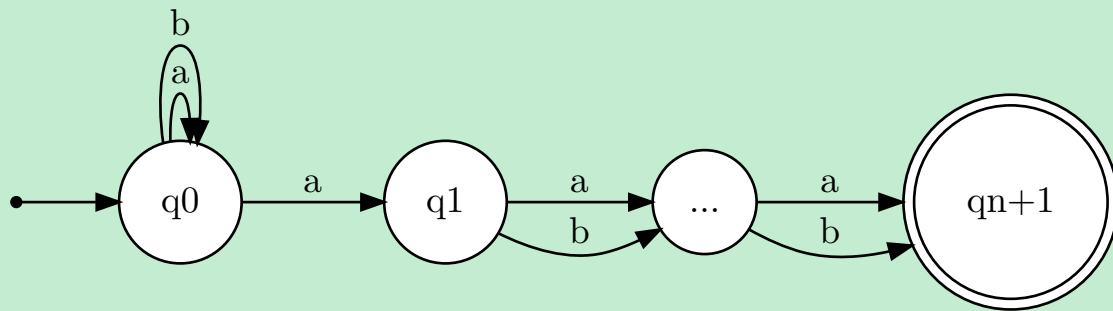
$$z \in \Sigma^{i-1}$$

che mi permette di distinguere: infatti, come prima, «isoliamo» i primi $i - 1$ caratteri, li «spostiamo» alla fine in un'altra forma e consideriamo solo gli n caratteri di destra. In questa nuova «configurazione» abbiamo l' n esimo carattere della stringa che è quello che era in posizione i , che in una stringa vale a e in una vale b , quindi le due stringhe sono distinguibili.

Ma allora ogni DFA per L_n usa almeno $2^{|X|} = 2^n$ stati.

Cosa cambia se invece utilizziamo un NFA per L_n ?

Esempio 3.1.5: Per il linguaggio L_n usiamo uno stato che fa la scommessa di essere arrivati all' n -esimo carattere da destra e uno stato che si ricorda di aver letto una a . Servono poi $n - 1$ stati per leggere i restanti $n - 1$ caratteri della stringa.

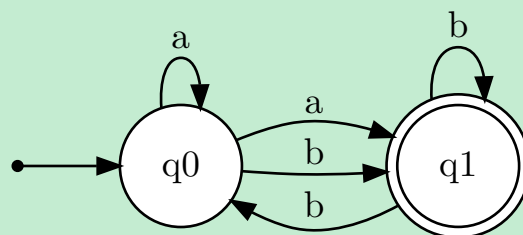


Il numero totale di stati è $n + 1$.

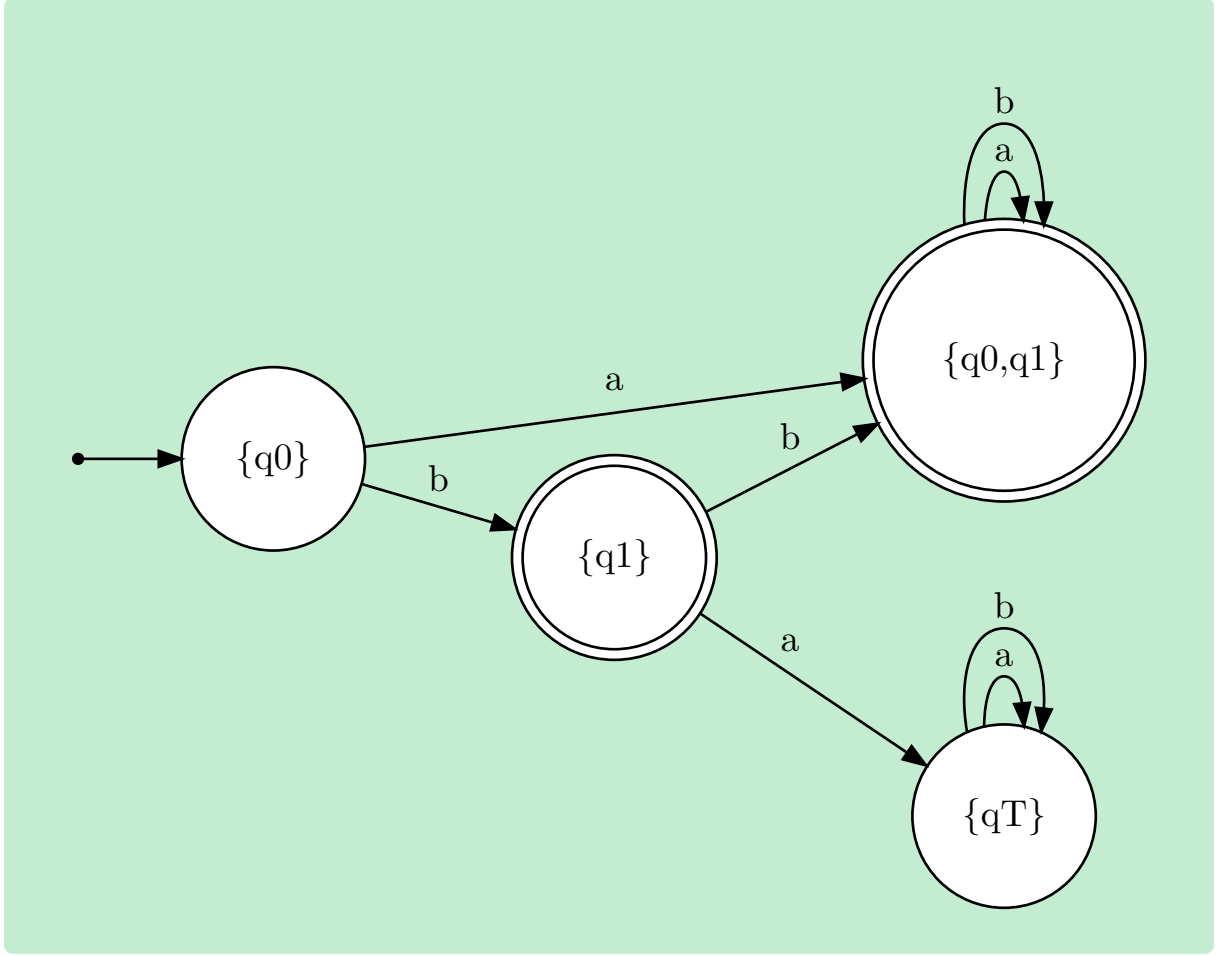
Per L_n abbiamo quindi visto che il numero di stati richiesti per un NFA è $n + 1$, mentre per un DFA è almeno 2^n grazie al teorema sulla distinguibilità. Il salto che abbiamo fatto è quindi **esponenziale**.

Tutto bello, ma questo salto esponenziale è evitabile? Possiamo fare di meglio? Possiamo cioè migliorare questa costruzione?

Esempio 3.1.6: Dato il seguente NFA, costruire il DFA associato.



Usando la costruzione per sottoinsiemi otteniamo il seguente DFA.



Escludendo lo stato trappola siamo riusciti ad usare meno stati di quelli del salto $n \rightarrow 2^n$, quindi vuol dire che forse si riesce a fare meglio. E invece **NO**. Esiste un caso peggiore, un automa che esegue un salto preciso da n a 2^n preciso preciso.

Come per la teoria della complessità, dobbiamo considerare sempre il caso peggiore, quindi vedremo un salto da n a 2^n esaurendo completamente tutti i possibili sottoinsiemi di n . Poi si può fare di meglio, ma in generale si fa tutto il salto visto che esiste un controesempio.

3.1.1. Automa di Meyer-Fischer

L'**automa di Meyer-Fischer**, ideato da questi due bro nel 1971, sarà il nostro NFA salvatore che ci permetterà di dimostrare quanto detto fino ad adesso.

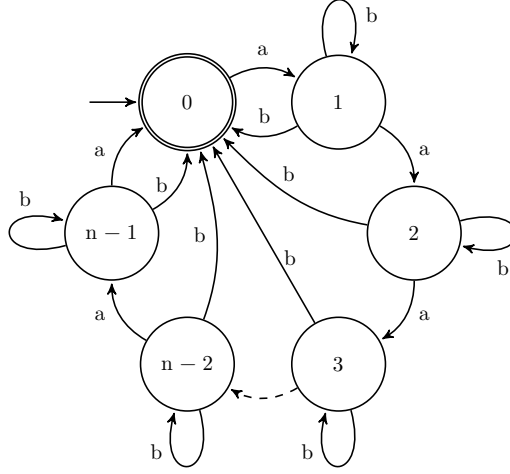
Sia $M_n = (Q, \Sigma, \delta, q_0, F)$ tali che:

- $Q = \{0, \dots, n-1\}$ insieme di n stati;
- $\Sigma = \{a, b\}$;
- $q_0 = 0$ stato iniziale e anche unico stato finale.

La funzione di transizione è tale che

$$\delta(i, x) = \begin{cases} \{(i+1) \bmod n\} & \text{se } x = a \\ \{i, 0\} & \text{se } x = b \\ \emptyset & \text{se } x = b \wedge i = 0 \end{cases}.$$

L'automa M_n lo possiamo disegnare in questo modo.



Teorema 3.1.1.1: Ogni DFA equivalente a M_n deve avere almeno 2^n stati.

Dimostrazione 3.1.1.1.1: Sia $S \subseteq \{0, \dots, n-1\}$. Definiamo la stringa

$$w_S = \begin{cases} b & \text{se } S = \emptyset \\ a^i & \text{se } S = \{i\} \\ a^{e_k - e_{k-1}} b a^{e_{k-1} - e_{k-2}} b \dots b a^{e_2 - e_1} b a^{e_1} & \text{se } S = \{e_1, \dots, e_k\} \mid k > 1 \wedge e_1 < \dots < e_k \end{cases}.$$

Si può dimostrare che per ogni $S \subseteq \{0, \dots, n-1\}$ vale

$$\delta(q_0, w_S) = S.$$

Si può dimostrare inoltre che dati $S, T \subseteq \{0, \dots, n-1\}$, se $S \neq T$ allora w_S e w_T sono distinguibili per il linguaggio $L(M_n)$.

Viste queste due proprietà, l'insieme di tutte le stringhe w_S associate ai vari insiemi S è formato da stringhe indistinguibili tra loro a coppie. Definiamo quindi

$$X = \{w_S \mid S \subseteq \{0, \dots, n-1\}\}$$

insieme di stringhe distinguibili tra loro per $L(M_n)$.

Il numero di stringhe in X dipende dal numero di sottoinsiemi di $\{0, \dots, n-1\}$: questi sono esattamente 2^n , quindi anche $|X| = 2^n$. Ma allora, per il teorema sulla distinguibilità, ogni DFA per M_n deve usare almeno 2^n stati. ■

Formalizziamo un attimo le due proprietà utilizzate. Vediamo la prima.

Lemma 3.1.1.1: Per ogni $S \subseteq \{0, \dots, n-1\}$ vale

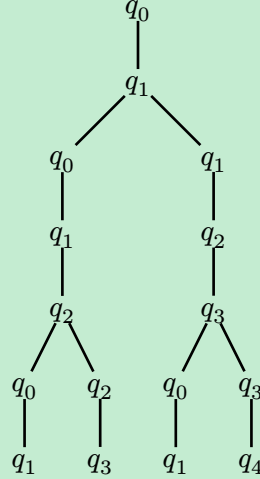
$$\delta(q_0, w_S) = S.$$

Esempio 3.1.1.1: Sia M_5 una istanza dell'automa di Meyer-Fischer.

Se scegliamo $S = \{1, 3, 4\}$ allora

$$w_S = a^{4-3}ba^{3-1}ba^1 = abaaba.$$

Facciamo girare l'automa M_5 sulla stringa w_S . Visto che Cetz fa cagare e non funziona niente, ogni stato i viene trasformato nello stato q_i .



Notiamo come l'insieme degli stati finali possibili sia esattamente S .

E ora vediamo la seconda e ultima proprietà.

Lemma 3.1.1.2: Dati $S, T \subseteq \{0, \dots, n-1\}$, se $S \neq T$ allora w_S e w_T sono distinguibili per il linguaggio $L(M_n)$.

Dimostrazione 3.1.1.2.1: Se $S \neq T$ allora sia $x \in S/T$ uno degli elementi che sta in S ma non in T . Vale anche il simmetrico, quindi consideriamo questo caso per ora.

Per il lemma precedente, sappiamo che

$$\delta(q_0, w_S) = S \quad | \quad \delta(q_0, w_T) = T.$$

Se siamo nello stato x , se vogliamo finire nello stato finale basta leggere la stringa a^{n-x} . Infatti, dato l'insieme S che contiene x , allora

$$w_S a^{n-x} \in L(M_n)$$

perché lo stato x finisce nello stato finale.

Ora, visto che $x \notin T$, allora $w_T a^{n-x} \notin L(M_n)$ perché l'unico modo per finire in 0 leggendo a^{n-x} è essere nello stato x , come visto poco fa.

Ma allora w_S e w_T sono distinguibili. ■

3.1.2. Esempi

Il teorema sulla distinguibilità che abbiamo visto la scorsa lezione è molto potente e ci permette di dimostrare che un linguaggio non è accettato da un automa a stati finiti se troviamo un insieme X con un numero infinito di stringhe.

Esempio 3.1.2.1: Sia

$$L = \{a^n b^n \mid n \geq 0\}.$$

Se scegliamo $X = \{a^n \mid n \geq 0\}$, esso è un insieme di stringhe tutte distinguibili tra loro.

Infatti, prendendo $x = a^i$ e $y = a^j$, con $i \neq j$, basta scegliere

$$z = b^i$$

per avere xz accettata e yz non accettata.

Ma allora L non può essere riconosciuto da un automa a stati finiti.

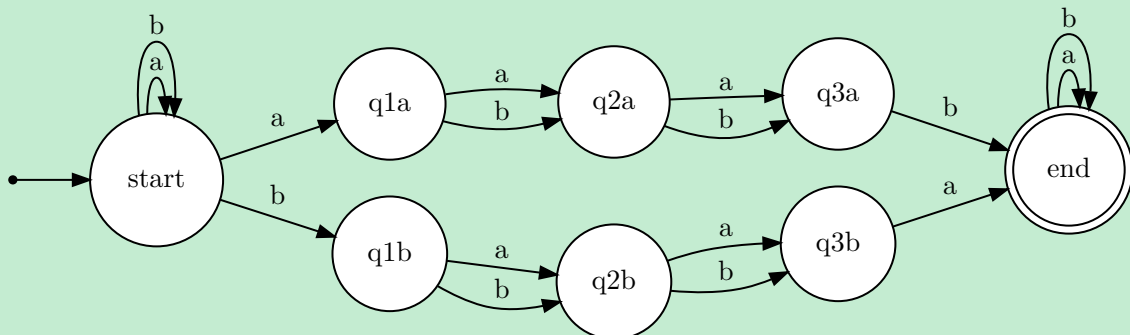
Visto che siamo bravi con le scommesse, andiamo a fare un po' di sano **gambling**.

Esempio 3.1.2.2: Definiamo

$$L_n = \{x \in \{a, b\}^* \mid \exists \text{ due simboli di } x \text{ a distanza } n \text{ che sono diversi}\}.$$

Usiamo anche per questo linguaggio la notazione L_n ma sono due linguaggi diversi.

Vediamo un NFA per L_3 , dove appunto viene fissato $n = 3$.



Una NFA per L_n utilizza $2n + 2$ stati, più un eventuale stato trappola.

Per il DFA riusciamo a trovare un bound al numero di stati?

Esempio 3.1.2.3: Dato L_n il linguaggio di prima, sia $X = \Sigma^n$.

Prendiamo le stringhe $\sigma = \sigma_1 \dots \sigma_n$ e $\gamma = \gamma_1 \dots \gamma_n$ di X , e sia i la prima posizione nella quale le due stringhe sono diverse, ovvero $\sigma_i \neq \gamma_i$. Come stringa z scelgo $\sigma_1 \dots \sigma_{i-1}$: con questa scelta otteniamo le stringhe

$$\sigma z = \sigma_1 \dots \sigma_{i-1} \sigma_i \sigma_{i+1} \dots \sigma_n \sigma_1 \dots \sigma_{i-1} \{a, b\}$$

$$\gamma z = \gamma_1 \dots \gamma_{i-1} \gamma_i \gamma_{i+1} \dots \gamma_n \gamma_1 \dots \gamma_{i-1} \{a, b\}$$

Notiamo come le prime coppie di caratteri sono tutte uguali, nel primo caso perché sono esattamente la stessa lettera, nel secondo caso perché avevamo imposto la prima diversità in i . In base poi al valore di σ_i e γ_i , e al valore scelto in fondo alla stringa, verrà accettata la prima o la seconda stringa.

Ma allora ogni DFA per L_n richiede almeno 2^n stati.

Vediamo ancora un esempio, ma teniamo a mente il linguaggio L_n che abbiamo appena visto.

Esempio 3.1.2.4: Dato l'alfabeto $\Sigma = \{a, b\}$, definiamo

$$L'_n = \{x \in \Sigma^* \mid \text{ogni coppia di simboli di } x \text{ a distanza } n \text{ è formata dallo stesso simbolo}\}.$$

Notiamo che dopo che ho letto n simboli essi si iniziano a ripetere fino alla fine, ma allora

$$x \in L'_n \iff \exists w \in \Sigma^n \wedge \exists y \in \Sigma^{\leq n} \mid x = w^{m \geq 0} y \wedge y \text{ suffisso di } w.$$

Posso ripetere w quante volte voglio, ma poi la parte finale deve ripetere in parte w .

Notiamo inoltre che questo linguaggio è il complementare del precedente, ovvero

$$L'_n = L_n^C.$$

Vogliamo costruire un DFA per questo linguaggio: posso usare l'insieme X di prima ma cambiare il valore di verità finale. Quindi ci servono ancora 2^n stati per il DFA.

Vediamo un esempio di automa con $n = 3$, un po' grossino, ma fa niente. Non viene inserito lo stato trappola per semplicità, ma ci dovrebbe essere anche quello per ogni transizione «sbagliata» nell'ultima parte dell'automata.

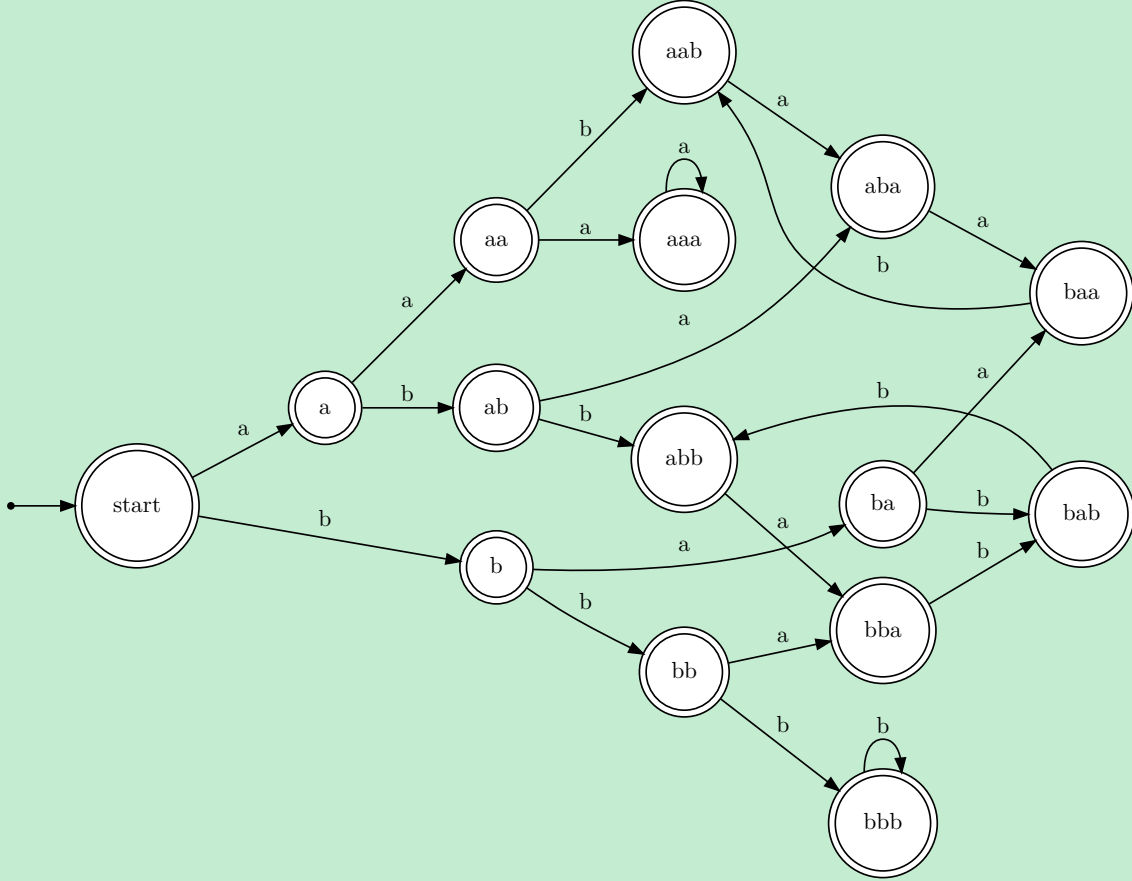


Per il linguaggio generico L'_n , l'albero usa un numero di stati pari a

$$2^{n+1} - 1 + -2^n(n - 1) + 1 = 2^{n+1} + 2^n(n - 1).$$

Una prima versione migliore dell'automa taglia via 4 stati facendo dei cappi negli stati $aaa1$ e $bbb1$, ma il numero rimane sempre esponenziale sotto steroidi.

Una seconda versione ancora migliore taglia tutti i $2^n(n-1)$ stati finali che fanno i cicli. Come mai? Possiamo usare tutte le foglie per mantenere comunque i cicli, abbastanza pesante da vedere però un bro è fortissimo e ha visto sta cosa.



Questa bellissima versione ha un numero di stati pari a

$$2^{n+1} - 1 + 1 = 2^{n+1}.$$

Come vediamo, in entrambi i casi abbiamo un numero esponenziale di stati, ma almeno abbiamo un automa deterministico da utilizzare.

3.2. Fooling set

Avevamo visto un criterio di distinguibilità per i DFA, ma ne esiste uno anche per gli NFA.

Definizione 3.2.1 (Fooling set): Sia $L \subseteq \Sigma^*$. Definiamo

$$P = \{(x_i, y_i) \mid i = 1, \dots, N\} \subseteq \Sigma^* \times \Sigma^*$$

un insieme di N coppie formate da stringhe di Σ^* .

L'insieme P è un **fooling set** per L se:

1. $\forall i \in \{1, \dots, N\} \quad x_i y_i \in L$;
2. $\forall i, j \in \{1, \dots, N\} \mid i \neq j \quad x_i y_j \notin L$.

Cosa ci stanno dicendo queste due proprietà? La prima ci dice che la concatenazione degli elementi della stessa coppia forma una stringa che appartiene al linguaggio, mentre la seconda ci dice che la concatenazione della prima parte di una coppia con la seconda parte di un'altra coppia forma una stringa che non appartiene al linguaggio.

Noi useremo una versione leggermente diversa del fooling set.

Definizione 3.2.2 (Extended fooling set): Un **extended fooling set** è un fooling set nel quale viene modificata la seconda proprietà, ovvero:

1. $\forall i \in \{1, \dots, N\} \quad x_i y_i \in L;$
2. $\forall i, j \in \{1, \dots, N\} \mid i \neq j \quad x_i y_j \notin L \vee x_j y_i \notin L.$

Come vediamo, è una versione un pelo più rilassata: prima chiedevo che, presa ogni prima parte di indice i , ogni concatenazione con seconde parti di indice j mi desse una stringa fuori dal linguaggio. Ora invece me ne basta solo uno dei due versi.

Teorema 3.2.1 (Teorema del fooling set): Se P è un extended fooling set per il linguaggio L allora ogni NFA per L deve avere almeno $|P|$ stati.

Dimostrazione 3.2.1.1: Concentriamoci solo sui cammini accettanti che possiamo avere in un NFA per il linguaggio L . Grazie alla prima proprietà di P , sappiamo che le stringhe $z = x_i y_i$ stanno in L . Calcoliamo i cammini per ogni coppia di P , che sono N :

$$\begin{array}{ccc} q_0 & \xrightarrow{x_1} p_1 & \xrightarrow{y_1} f_1 \\ & \vdots & \\ q_0 & \xrightarrow{x_N} p_N & \xrightarrow{y_N} f_N \end{array}$$

Per assurdo sia A un NFA con meno di N stati. Ma allora esistono due stringhe $x_i \neq x_j$ che mi fanno andare in $p_i = p_j$. Sappiamo che:

- da p_i con y_i vado in uno stato finale;
- da p_j con y_j vado in uno stato finale.

Sappiamo che $p_i = p_j$, ma quindi $x_i y_j$ è una stringa che finisce in uno stato finale, ma questo è un assurdo perché contraddice la seconda proprietà del fooling set.

Quindi ogni NFA deve avere almeno N stati. ■

3.2.1. Esempi

Usiamo questo teorema per valutare un NFA per il linguaggio precedente.

Esempio 3.2.1.1: Dato il linguaggio L'_n definiamo l'insieme

$$P = \{(x, x) \mid x \in \Sigma^n\}$$

extended fooling set per L'_n . Infatti, ogni stringa $z = xx$ appartiene a L'_n , mentre ogni «stringa incrociata» $z = xy$, con $x \neq y$, non appartiene a L'_n perché in almeno una posizione a distanza n ho un carattere diverso.

Il numero di elementi di P è 2^n , che è il numero di configurazioni lunghe n di 2 caratteri, quindi ogni NFA per L'_n ha almeno 2^n stati.

Vediamo un mini **riassunto** dei due linguaggi visti di recente.

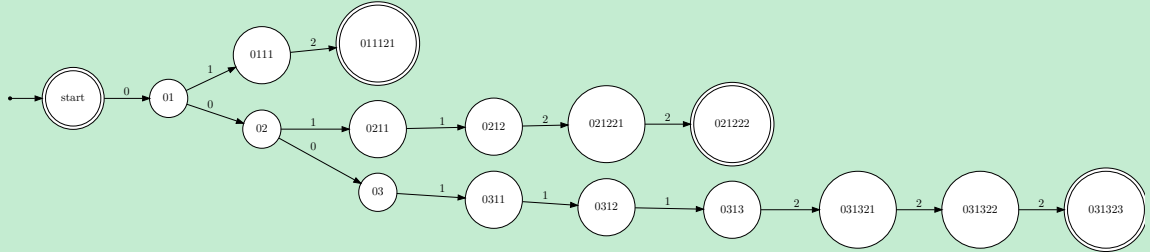
Linguaggio	DFA	NFA
L_n	$\geq 2^n$	$\leq 2n + 2$
L'_n	$\geq 2^n \wedge \leq 2^{n+1}$	$\geq 2^n$

Finiamo con un ultimo esempio.

Esempio 3.2.1.2: Dato il linguaggio $\Sigma = \{0, 1, 2\}$, definiamo il linguaggio

$$L_n = \{0^i 1^i 2^i \mid 0 \leq i \leq n\}.$$

Diamo un DFA per questo linguaggio, fissando $n = 3$.



Il numero di stati del linguaggio L_n generico è

$$\sum_{i=1}^n i + \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2} + \frac{n(n-1)}{2} = \frac{n^2 + n + n^2 - n}{2} = \frac{2n^2}{2} = n^2.$$

Possiamo mangiare qualche stato, facendo rientrare le computazioni più lunghe in quelle più corte quando stiamo leggendo dei 2, ma il numero rimane comunque $O(n^2)$.

Per finire diamo un NFA per il linguaggio L_n . Visto che non sappiamo su cosa scommettere, diamo un lower bound al numero di stati dei nostri NFA.

Creiamo un fooling set

$$P = \{(0^i 1^j, 1^{i-j} 2^i) \mid i = 1, \dots, n \wedge j = 1, \dots, i\}.$$

Questo è un fooling set per L_n :

- una coppia ci dà la stringa $z = 0^i 1^{j+i-j} 2^i = 0^i 1^i 2^i$ che appartiene al linguaggio;
- prendendo due elementi da due coppie diverse:

- se sono diverse le i abbiamo un numero di 0 e 2 diversi;
- se sono uguali le i allora sono diverse le j , ma allora la stringa $0^i 1^{j+i-j'} 2^i$ non appartiene al linguaggio perché $j + i - j' \neq i$.

Il numero di stati di P è ancora una somma di Gauss, quindi

$$\sum_{i=1}^n = \frac{n(n+1)}{2},$$

quindi ogni NFA per L_n ha almeno un numero quadratico di stati.