Teoria dei Linguaggi

Indice

1. Lezione 01 [26/02]

1.1. Cosa faremo

In questo corso studieremo dei sistemi formali che possiamo quindi descrivere a livello matematico. Questi sistemi descrivono dei linguaggi. Ci chiediamo giustamente cosa sono in grado di fare questi sistemi, ovvero cosa sono in grado di descrivere in termini di linguaggi.

Ci occuperemo anche delle risorse utilizzate dal sistema o delle risorse necessarie per descrivere il linguaggio. Per le prime citate, ci occuperemo del tempo come numero di mosse eseguite da una macchina riconoscitrice oppure del numero di stati per descrivere, ad esempio, una macchina a stati finiti oppure dello spazio utilizzato da una macchina di Turing. Queste ultime due questioni rientrano più nella complessità descrizionale di una macchina.

1.2. Storia

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/e 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 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, quindi linguaggi con strutture di controllo, la teoria dei linguaggi formali è diventata sempre più importante.

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

1.3. Ripasso

Un **alfabeto** è un insieme non vuoto e finito di simboli, di solito indicato con Σ o Γ.

Una **stringa** x (o **parola**) è una sequenza finita $x = a_1 ... a_n$ di simboli appartenenti a Σ.

Data una parola w, possiamo definire:
- $|w|$ numero di caratteri di w;
- $|w|_a$ numero di occorrenze della lettera $a \in \Sigma$ in w.

Una parola molto importante è la **parola vuota** ε o λ, che, come dice il nome, ha simboli, ovvero $|\varepsilon| = |\lambda| = 0$ (ogni tanto è Λ).

L'insieme di tutte le possibili parole su Σ è detto Σ^*, ed è un insieme infinito.

Un'importante operazione sulle parole è la **concatenazione** (o prodotto), ovvero se $x, y \in \Sigma^*$ allora la concatenazione w è la parola $w = xy$.

Questo operatore di concatenazione:
- non è commutativo, infatti $w_1 = xy \neq yz = w_2$ in generale;
- è associativo, infatti $(xy)z = x(yz)$.

La struttura $(\Sigma^*, \cdot, \varepsilon)$ è un **monoide** libero generato da Σ.

Vediamo ora alcune proprietà delle parole:

- **prefisso**: x si dice prefisso di w se esiste $y \in \Sigma^*$ tale che $xy = w$;
  - **prefisso proprio** se $y \neq \varepsilon$;
  - **prefisso non banale** se $x \neq \varepsilon$;
  - il numero di prefissi è uguale a $|w| + 1$.
- **suffisso**: y si dice suffisso di w se esiste $x \in \Sigma^*$ tale che $xy = w$;
  - **suffisso proprio** se $x \neq \varepsilon$;
  - **suffisso non banale** se $y \neq \varepsilon$;
  - il numero di suffissi è uguale a $|w| + 1$.
- **fattore**: y si dice fattore di w se esistono $x, z \in \Sigma^*$ tali che $xyz = w$;
  - il numero di fattori è al massimo $\frac{|w||w+1|}{2} + 1$, visti i doppioni.
- **sottosequenza**: x si dice sottosequenza di w se x è ottenuta eliminando 0 o più caratteri da w; in poche parole, x si ottiene da w scegliendo dei simboli IN ORDINE; non devono essere caratteri contigui, basta che una volta scelti i caratteri essi siano mantenuti nell'ordine di apparizione della stringa iniziale;
  - un fattore è una sottosequenza contigua.

Un **linguaggio** L definito su un alfabeto Σ è un qualunque sottoinsieme di Σ^*.

1.4. Gerarchia di Chomsky

Vogliamo rappresentare in maniera finita un oggetto infinito come un linguaggio.

Abbiamo a nostra disposizione due modelli molto potenti:

- **generativo**: date delle regole, si parte da un certo punto e si generano tutte le parole di quel linguaggio con le regole date; parleremo di questi modelli tramite le grammatiche;
- **riconoscitivo**: si usano dei modelli di calcolo che prendono in input una parola e dicono se appartiene o meno al linguaggio.

Considerando il linguaggio sull'alfabeto $\{(,)\}$ delle parole ben bilanciate, proviamo a dare due modelli:

- generativo: a partire da una sorgente S devo applicare delle regole per derivate tutte le parole appartenenti a questo linguaggio;
  - la parola vuota ε è ben bilanciata;
  - se x è ben bilanciata, allora anche (x) è ben bilanciata;
  - se x, y sono ben bilanciate, allora anche xy è ben bilanciata.
- riconoscitivo: abbiamo una black-box che prende una parola e ci dice se appartiene o meno al linguaggio (in realtà potrebbe non terminare mai la sua esecuzione);
  - $\#(\; = \; \#)$;
  - per ogni prefisso, $\#(\; \geq \; \#)$.

2. Lezione 02 [28/02]

2.1. Grammatiche

Una **grammatica** è una tupla (V, Σ, P, S), con:
- V insieme finito e non vuoto delle **variabili**; queste ultime sono anche dette simboli non terminali e sono usate durante il processo di generazione delle parole del linguaggio; sono anche detti meta-simboli;
- Σ insieme finito e non vuoto dei **simboli terminali**; questi ultimi appaiono nelle parole generate, a differenza delle variabili che invece non possono essere presenti;
- P insieme finito e non vuoto delle **regole di produzione**;
- $S \in V$ **simbolo iniziale** o **assioma**, è il punto di partenza della generazione.

2.1.1. Regole di produzione

Soffermiamoci sulle regole di produzione: la forma di queste ultime è $\alpha \longrightarrow \beta$, con $\alpha \in (V \cup \Sigma)^+$ e $\beta \in (V \cup \Sigma)^*$. Non l'abbiamo detto la scorsa volta, ma la notazione con il $+$ è praticamente Σ^* 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.

Diciamo che x deriva y in un passo, con $x, y \in (V \cup \Sigma)^*$, se e solo se $\exists(\alpha \longrightarrow \beta) \in P$ e $\exists \eta, \delta \in (V \cup \Sigma)^*$ tali che $x = \eta\alpha\delta$ e $y = \eta\beta\delta$.

Il passo di derivazione lo indichiamo con $x \Rightarrow y$.

La versione estesa afferma che x deriva y in $k \geq 0$ passi, e lo indichiamo con $x \overset{k}{\Rightarrow} y$, se e solo se $\exists x_0, ..., x_k \in (V \cup \Sigma)^*$ tali che $x = x_0$, $x_k = y$ e $x_{i-1} \Rightarrow x_i \quad \forall i \in [1, k]$.

Teniamo anche il caso $k = 0$ per dire che da x derivo x stesso, ma è solo per comodità.

Se non ho indicazioni sul numero di passi k posso scrivere:
- $x \overset{*}{\Rightarrow} y$ per indicare un numero generico di passi, e questo vale se e solo se $\exists k \geq 0$ tale che $x \overset{k}{\Rightarrow} y$;
- $x \overset{+}{\Rightarrow} y$ per indicare che serve almeno un passo, e questo vale se e solo se $\exists k > 0$ tale che $x \overset{k}{\Rightarrow} y$.

2.1.2. Linguaggio generato da una grammatica

Indichiamo con $L(G)$ il linguaggio generato dalla grammatica G, ed è l'insieme $\left\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\right\}$. In poche parole, è l'insieme di tutte le stringhe di non terminali che si possono ottenere tramite **derivazioni** a partire dall'assioma S della grammatica.

In questo insieme abbiamo solo stringhe di non terminali che otteniamo tramite derivazioni. Le stringhe intermedie che otteniamo nei vari passi di derivazioni sono dette **forme sintattiche**.

Due grammatiche G_1, G_2 sono **equivalenti** se e solo se $L(G_1) = L(G_2)$.

Se consideriamo l'esempio delle parentesi ben bilanciate, possiamo definire una grammatica per questo linguaggio con le seguenti regole di produzione:
- $S \longrightarrow \varepsilon$;
- $S \longrightarrow (S)$;
- $S \longrightarrow SS$.

Vediamo un esempio più complesso. Siano:
- $\Sigma = \{a, b\}$;
- $V = \{S, A, B\}$;
- $P = \{S \longrightarrow aB \mid bA, A \longrightarrow a \mid aS \mid bAA, B \longrightarrow b \mid bS \mid aBB\}$.

Questa grammatica genera il linguaggio $L(G) = \{w \in \Sigma^* \mid \#_a(w) = \#_b(w)\}$: infatti, ogni volta che inserisco una a inserisco anche una B per permettere poi di inserire una b. Il discorso vale lo stesso a lettere invertite.

Vediamo un esempio ancora più complesso. Siano:
- $\Sigma = \{a, b\}$;
- $V = \{S, A, B, C, D, E\}$;
- $P = \{S \longrightarrow ABC, AB \longrightarrow \varepsilon \mid aAD \mid bAE, DC \longrightarrow BaC, EC \longrightarrow BbC, Da \longrightarrow aD, Db \longrightarrow bD, Ea \longrightarrow aE, Eb \longrightarrow bE, C \longrightarrow \varepsilon, aB \longrightarrow Ba, bB \longrightarrow Bb\}$.

Questa grammatica genera il linguaggio pappagallo $L(G) = \{ww \mid w \in \Sigma^*\}$: infatti, eseguendo un paio di derivazioni si nota questo pattern.

2.2. Gerarchia di Chomsky

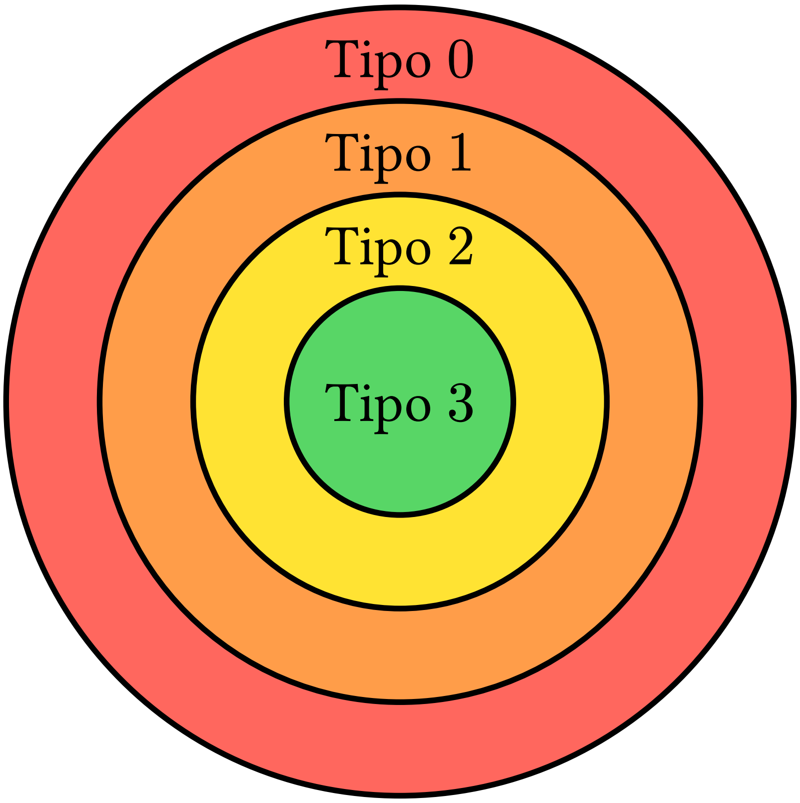
Negli anni $'50$ Noam Chomsky studia la generazione dei linguaggi formali e crea una **gerarchia di grammatiche formali**. La classificazione delle grammatiche viene fatta in base alle regole di produzione che definiscono la grammatica.

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.	Se $(\alpha \longrightarrow \beta) \in P$ allora $\lvert\beta\rvert \geq \lvert\alpha\rvert$, ovvero devo generare parole che non siano più corte di quella di partenza. Sono dette dipendenti dal contesto perché ogni regola $(\alpha \longrightarrow \beta) \in P$ può essere riscritta come $\alpha_1 A \alpha_2 \longrightarrow \alpha_1 B \alpha_2$, con $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ che rappresentano il contesto, $A \in V$ e $B \in (V \cup \Sigma)^+$	**Automi limitati linearmente**
Tipo 2, dette **context-free** o libere dal contesto	Le regole in P sono del tipo $\alpha \longrightarrow \beta$, 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 $A \longrightarrow aB$ oppure $A \longrightarrow a$, con $A, B \in V$ e $a \in \Sigma$. Vale anche il simmetrico.	**Automi a stati finiti**

Nella figura successiva vediamo una rappresentazione grafica della gerarchia di Chomsky: notiamo come sia una gerarchia propria, ovvero

$$L_3 \subset L_2 \subset L_1 \subset L_0,$$

ma questa gerarchia non esaurisce comunque tutti i linguaggi possibili. Esistono infatti linguaggi che non sono descrivibili in maniera finita con le grammatiche.



Sia $L \subseteq \Sigma^*$, allora L è di tipo i, con $i \in [0, 3]$, se e solo se esiste una grammatica G di tipo i tale che $L = L(G)$, ovvero posso generare L a partire dalla grammatica di tipo i.

Se una grammatica é 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**.

3. Esercizi lezione 01 e 02 [28/02]

3.1. Esercizio 01

Esercizio 3.1.1: Considerate l'alfabeto $\Sigma = \{a, b\}$.

Richiesta 3.1.1.1: Fornite una grammatica CF per il linguaggio delle stringhe palindrome di lunghezza pari su Σ, cioè per l'insieme $\text{PAL}_{\text{pari}} = \{ww^R \mid w \in \Sigma^*\}$.

Soluzione 3.1.1.1: Definisco G tale che $V = \{S\}$ e

$$P = \{S \longrightarrow \varepsilon \mid aSa \mid bSb\}.$$

Richiesta 3.1.1.2: Modificate la grammatica precedente per generare l'insieme PAL di tutte le stringhe palindrome su Σ.

Soluzione 3.1.1.2: Definisco G tale che $V = \{S\}$ e

$$P = \{S \longrightarrow \varepsilon \mid aSa \mid bSb \mid a \mid b\}.$$

Richiesta 3.1.1.3: Per ogni $k \in \{0, ..., 3\}$, rispondete alla domanda «Il linguaggio PAL è di tipo k?» giustificando la risposta.

Soluzione 3.1.1.3: Non è di tipo 3 per le produzioni $S \longrightarrow aSa \mid bSb$ ma è di tipo 2 visto che rispetta le restrizioni sulle produzioni. Di conseguenza, è anche di tipo 1 e di tipo 0.

Richiesta 3.1.1.4: Se sostituiamo l'alfabeto con $\Sigma = \{a, b, c\}$, le risposte al punto precedente cambiano? E se sostituiamo con $\Sigma = \{a\}$?

Soluzione 3.1.1.4: Se $\Sigma = \{a, b, c\}$ vanno aggiunte due produzioni che però sono nella forma di quelle precedenti, quindi le risposte non cambiano.

Se $\Sigma = \{a\}$ le uniche produzioni che abbiamo sono

$$S \longrightarrow \varepsilon \mid aS \mid a$$

e quindi la grammatica è di tipo 3. Di conseguenza, è anche di tipo 2, tipo 1 e tipo 0.

3.2. Esercizio 02

Esercizio 3.2.1: Considerate l'alfabeto $\Sigma = \{a, b\}$.

Richiesta 3.2.1.1: Scrivete una grammatica per generare il complemento di PAL.

Soluzione 3.2.1.1: Sia G tale che $V = \{S, D, B\}$ e P formato da

$$\begin{aligned} S &\longrightarrow aSa \mid bSb \mid D \\ D &\longrightarrow aDb \mid bDa \mid B \\ B &\longrightarrow \varepsilon \mid a \mid b \mid S. \end{aligned}$$

3.3. Esercizio 03

Esercizio 3.3.1: Sia $\Sigma = \{(,)\}$ un alfabeto i cui simboli sono la parentesi aperta e la parentesi chiusa.

Richiesta 3.3.1.1: Scrivete una grammatica CF che generi il linguaggio formato da tutte le sequenze di parentesi correttamente bilanciate, come ad esempio $(()(()))()$.

Soluzione 3.3.1.1: Sia G una grammatica con $V = \{S\}$ e P formato da

$$S \longrightarrow \varepsilon \mid (S) \mid SS.$$

Richiesta 3.3.1.2: Risolvete il punto precedente per un alfabeto con due tipi di parentesi, come $\Sigma = \{(,), [,]\}$, nel caso non vi siano vincoli tra i tipi di parentesi (le tonde possono essere contenute tra quadre e viceversa). Esempio $[()([])[]]$ ma non $[[][(])()]$.

Soluzione 3.3.1.2: Sia G una grammatica con $V = \{S\}$ e P formato da

$$S \longrightarrow \varepsilon \mid (S) \mid [S] \mid SS.$$

Richiesta 3.3.1.3: Risolvete il punto precedente con $\Sigma = \{(,),[,]\}$, con il vincolo che le parentesi quadre non possano apparire all'interno di parentesi tonde. Esempio $[()(())[][]](()())$, ma non $[()([])[]]$.

Soluzione 3.3.1.3: Sia G una grammatica con $V = \{S, T\}$ e P formato da

$$S \longrightarrow \varepsilon \mid [S] \mid SS \mid T$$
$$T \longrightarrow \varepsilon \mid (T) \mid TT.$$

3.4. Esercizio 04

Esercizio 3.4.1: Sia $G = (V, \Sigma, P, S)$ la grammatica con $V = \{S, B, C\}$, $\Sigma = \{a, b, c\}$ e P contenente le seguenti produzioni:

$$\begin{aligned} S &\longrightarrow aSBC \mid aBC \\ CB &\longrightarrow BC \\ aB &\longrightarrow ab \\ bB &\longrightarrow bb \\ bC &\longrightarrow bc \\ cC &\longrightarrow cc. \end{aligned}$$

Richiesta 3.4.1.1: Dopo aver stabilito di che tipo è G, provate a derivare alcune stringhe. Riuscite a dire da quali stringhe è formato il linguaggio generato da G?

Soluzione 3.4.1.1: La grammatica G è di tipo 1.

Prima derivazione:

$$S \longrightarrow aBC \longrightarrow abC \longrightarrow abc.$$

Seconda derivazione:

$$\begin{aligned} S &\longrightarrow aSBC \longrightarrow aaBCBC \longrightarrow aabCBC \\ &\longrightarrow aabBCC \longrightarrow aabbCC \longrightarrow aabbcC \longrightarrow aabbcc. \end{aligned}$$

Possiamo dire che $L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

3.5. Esercizio 05

Esercizio 3.5.1: Sia $G = (V, \Sigma, P, S)$ la grammatica con $V = \{S, B, C\}$, $\Sigma = \{a, b, c\}$ e P contenente le seguenti produzioni:

$$S \longrightarrow aBSc \mid abc$$
$$Ba \longrightarrow aB$$
$$Bb \longrightarrow bb.$$

Richiesta 3.5.1.1: Dopo aver stabilito di che tipo è G, provate a derivare alcune stringhe. Riuscite a dire da quali stringhe è formato il linguaggio generato da G?

Soluzione 3.5.1.1: La grammatica G è di tipo 1.

Prima derivazione:

$$S \longrightarrow abc.$$

Seconda derivazione:

$$S \longrightarrow aBSc \longrightarrow aBabcc \longrightarrow aaBbcc \longrightarrow aabbcc.$$

Come prima, possiamo dire che $L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

3.6. Esercizio 06

Esercizio 3.6.1: Sia $G = (V, \Sigma, P, S)$ la grammatica con $V = \{S, A, B, C, D, E\}$, $\Sigma = \{a, b\}$ e P contenente le seguenti produzioni:

$$S \longrightarrow ABC$$
$$AB \longrightarrow aAD \mid bAE \mid \varepsilon$$
$$DC \longrightarrow BaC$$
$$EC \longrightarrow BbC$$
$$Da \longrightarrow aD$$
$$Db \longrightarrow bD$$
$$Ea \longrightarrow aE$$
$$Eb \longrightarrow bE$$
$$C \longrightarrow \varepsilon$$
$$aB \longrightarrow Ba$$
$$bB \longrightarrow Bb.$$

Richiesta 3.6.1.1: Dopo aver stabilito di che tipo è G, provate a derivare alcune stringhe. Riuscite a dire da quali stringhe è formato il linguaggio generato da G?

Suggerimento. Per ogni $w \in \{a, b\}^*$ è possibile costruire una derivazione $S \overset{*}{\Rightarrow} wABwC$ (provate a procedere per induzione sulla lunghezza di w cercando di capire il ruolo di ciascuna delle variabile nel processo di derivazione).

Soluzione 3.6.1.1: La grammatica G è di tipo 1.

Prima derivazione:

$$S \longrightarrow ABC \longrightarrow C \longrightarrow \varepsilon.$$

Seconda derivazione:

$$S \longrightarrow ABC \longrightarrow aADC \longrightarrow aABaC \longrightarrow aaC \longrightarrow aa.$$

Terza derivazione:

$$\begin{aligned} S &\longrightarrow ABC \longrightarrow aADC \longrightarrow aABaC \longrightarrow abAEaC \longrightarrow abAaEC \\ &\longrightarrow abAaBbC \longrightarrow abABabC \longrightarrow ababC \longrightarrow abab. \end{aligned}$$

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

3.7. Esercizio 07

Esercizio 3.7.1: Sia $G = (V, \Sigma, P, S)$ la grammatica con $V = \{S, A, B, C, X, Y, L, R\}$, $\Sigma = \{a\}$ e P contenente le seguenti produzioni:

$$\begin{gathered} S \longrightarrow LXR \\ LX \longrightarrow LYYA \mid aC \\ AX \longrightarrow YYA \\ AR \longrightarrow BR \\ YB \longrightarrow BX \\ LB \longrightarrow L \\ CX \longrightarrow aC \\ CR \longrightarrow \varepsilon. \end{gathered}$$

Richiesta 3.7.1.1: Riuscite a stabilire da quali stringhe è formato il linguaggio generato da G?

Suggerimento. Si può osservare che $LX^iR \overset{*}{\Rightarrow} LY^{2i}AR \Rightarrow LX^{2i}R$ per ogni $i > 0$. Inoltre dal simbolo iniziale si ottiene la forma LXR. Le ultime tre produzioni sono utili per sostituire variabili in una forma sentenziale con occorrenze di terminali.

Soluzione 3.7.1.1: La grammatica G è di tipo 0.

Prima derivazione:

$$S \longrightarrow LXR \longrightarrow aCR \longrightarrow a.$$

Seconda derivazione:

$$\begin{aligned} S &\longrightarrow LXR \longrightarrow LYYAR \longrightarrow LYYBR \longrightarrow LYBXR \\ &\longrightarrow LBXXR \longrightarrow LXXR \longrightarrow aCXR \longrightarrow aaCR \longrightarrow aa. \end{aligned}$$

Terza derivazione:

$$\begin{aligned} S &\longrightarrow LXR \longrightarrow LYYAR \longrightarrow LYYBR \longrightarrow LYBXR \longrightarrow LBXXR \longrightarrow LXXR \\ &\longrightarrow LYYAXR \longrightarrow LYYYYAR \longrightarrow LYYYYBR \longrightarrow LYYYBXR \\ &\longrightarrow LYYBXXR \longrightarrow LYBXXXR \longrightarrow LBXXXXR \longrightarrow LXXXXR \\ &\longrightarrow aCXXXR \longrightarrow aaCXXR \longrightarrow aaaCXR \longrightarrow aaaaCR \longrightarrow aaaa. \end{aligned}$$

Possiamo dire che $L(G) = \left\{a^{2^n} \mid n \geq 0\right\}$.

3.8. Esercizio 08

Esercizio 3.8.1:

Richiesta 3.8.1.1: Modificate la grammatica dell'esercizio 7 in modo da ottenere una grammatica di tipo 1 che generi lo stesso linguaggio.

Soluzione 3.8.1.1: La produzione che dà problemi è $LB \longrightarrow L$. La facciamo diventare

$$LB \longrightarrow CRL.$$

In questo modo rispettiamo tutti i vincoli delle grammatiche di tipo 1 e non modifichiamo la grammatica, visto che CR non genera problemi con la L o con la a quando facciamo le sostituzioni finali.

3.9. Esercizio 09

Esercizio 3.9.1:

Richiesta 3.9.1.1: Dimostrate che la grammatica $G = (\{A, B, S\}, \{a, b\}, P, S)$, con l'insieme delle produzioni P elencate sotto, genera il linguaggio $\{w \in \{a, b\}^* \mid \forall x \in \{a, b\}^* \quad w \neq xx\}$.

$$S \longrightarrow AB \mid BA \longrightarrow A \mid B$$
$$A \longrightarrow aAa \mid aAb \mid bAa \mid bAb \mid a$$
$$B \longrightarrow aBa \mid aBb \mid bBa \mid bBb \mid b$$

Soluzione 3.9.1.1: Eseguendo come prima produzione $S \longrightarrow A \mid B$ si ottengono delle stringhe di lunghezza dispari, che quindi non possono essere scritte come concatenazione di due stringhe uguali.

Eseguendo invece come prima produzione $S \longrightarrow AB \mid BA$ e facendo un numero di sostituzioni uguali per entrambe le parti, le due stringhe risultanti di ugual lunghezza avranno almeno una posizione differente, generate dall'ultimo cambio di A e B.

Eseguendo invece come prima produzione $S \longrightarrow AB \mid BA$ e facendo un numero di sostituzioni diverso per le due parti, non lo so dimostrare, secondo Martino lo dobbiamo fare ma io non lo farò, baci.

4. Lezione 03 [05/03]

4.1. Gerarchia

Come si modifica la gerarchia di Chomsky considerando il non determinismo? Abbiamo che:
- le tipo 3 ha i modelli equivalenti, con un costo in termini della descrizione;
- le tipo 2 ha un cambiamento nei modelli, con quello non deterministico strettamente più potente;
- le tipo 1 sono complicate;
- le tipo 0 ha i modelli equivalenti.

Il non determinismo è una nozione del **riconoscitore** che uso per riconoscere: nel determinismo il riconoscitore può fare una cosa alla volta, nel non determinismo può fare più cose contemporaneamente. Nelle grammatiche è difficile catturare questa nozione, perché esse lo hanno intrinsecamente, perché le derivazioni le applico tutte per ottenere le stringhe del linguaggio.

4.2. Decidibilità

Teorema 4.2.1 (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 4.2.1.1: 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. 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.

Definiamo quindi gli insiemi

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

Calcoliamo induttivamente questi insiemi.

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

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

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

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

Noi partendo da T_0 calcoliamo tutti i vari insiemi ottenendo una serie di T_i.

Per come abbiamo definito gli insiemi, sappiamo che

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

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

Ma allora G è decidibile. ■

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

Teorema 4.2.2 (Semi-decidibilità dei linguaggi di tipo 0): I linguaggi di tipo 0 sono ricorsivamente enumerabili.

Dimostrazione 4.2.2.1: 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. 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 n e poi sperare di tornare indietro in qualche modo.

Definiamo quindi gli insiemi

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

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 facciamo a rispondere 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.

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. ■

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

4.3. Parola vuota

Vediamo il problema della **parola vuota**: nelle grammatiche di tipo 2 abbiamo ho messo il $+$ per evitare la parola vuota nelle derivazioni, ma ogni tanto potrebbe servirmi la parola vuota nel linguaggio di quella grammatica. La mossa di mettere $\star$ mi farebbe cadere tutta la gerarchia.

Come risolviamo questo problema?

Partiamo da una grammatica $G = (V, \Sigma, P, S)$ di tipo 1. Creiamo una nuova grammatica $G_1 = (V_1, \Sigma, P_1, S_1)$ tale che $L(G) = L(G_1)$. Vediamo come sono fatte le componenti di G_1:
- $V_1 = V \cup \{S_1\}$;
- per P_1 abbiamo due opzioni:
  - $P_1 = P \cup \{S_1 \to \alpha \mid (S \to \alpha) \in P\} \cup \{S_1 \to \varepsilon\}$;
  - $P_1 = P \cup \{S_1 \longrightarrow S\} \cup \{S_1 \longrightarrow \varepsilon\}$;
- S_1 nuovo assioma che non appare mai nel lato destro delle produzioni.

La gerarchia ora diventa:
- tipo 1 abbiamo $|\alpha| \leq |\beta|$ ed è possibile $S \longrightarrow \varepsilon$ purché S non appaia mai sul lato destro delle produzioni;
- tipo 2 permettiamo direttamente $A \longrightarrow \beta$ con $\beta \in (V \cup \Sigma)^*$ senza costringere ad isolarle. Questo perché non creano problemi, comunque resta decidibile se una stringa appartiene al linguaggio, anche se posso cancellare e ridurre la lunghezza;
- tipo 3 idem delle tipo 2.

Queste produzioni particolari sono dette **ε-produzioni**.

4.4. Linguaggi non esprimibili tramite grammatiche finite

Ora vediamo linguaggi che non possiamo esprimere tramite grammatiche. Utilizzeremo la **dimostrazione per diagonalizzazione**, famosissima e utilizzatissima in tante dimostrazioni.

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 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 $\mathbb{N}$.

Consideriamo ora i sottoinsiemi di $\mathbb{N}$. Sono più questi sottoinsiemi o i numeri interi? In questo caso, sono di più i sottoinsiemi, che hanno la **cardinalità del continuo**. Per dimostrare questo useremo una dimostrazione per diagonalizzazione.

Teorema 4.4.1: Vale

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

Dimostrazione 4.4.1.1: Per assurdo sia $\mathbb{N} \sim 2^{\mathbb{N}}$, 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 $\mathbb{N} \nsim 2^{\mathbb{N}}$. ■

Prima dell'ultima parte chiediamoci ancora una cosa: sono più le stringhe o i numeri interi? Questo è facile, basta trasformare ogni stringa in un numero intero con una qualche codifica a nostra scelta.

Teorema 4.4.2: Esistono linguaggi che non sono descrivibili da grammatiche finite.

Dimostrazione 4.4.2.1: 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

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

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 $\mathbb{N} \nsim 2^L$. ■

5. Lezione 04 [07/03]

5.1. Linguaggi regolari

5.1.1. Macchine a stati finiti deterministiche

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

Un FSM 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. Lo 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**.

Formalmente, una FSM 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 nullo, 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.

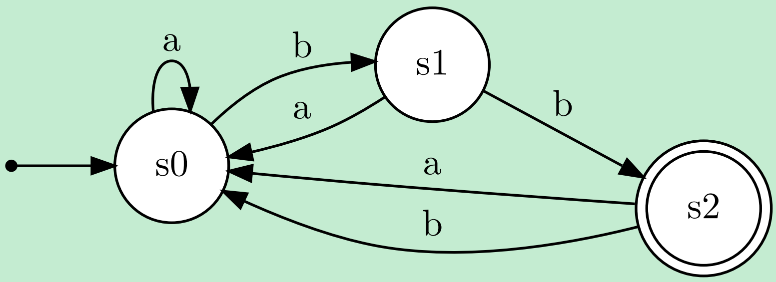
Esempio 5.1.1.1: Sia $A = (Q, \Sigma, \delta, q_0, F)$ tale che:
- $Q = \{s_0, s_1, s_2\}$;

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

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

$$\begin{bmatrix} & a & b \\ \hline s_0 & s_0 & s_1 \\ s_1 & s_0 & s_2 \\ s_2 & s_0 & s_0 \end{bmatrix}.$$

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



Il linguaggio che riconosce questo automa è

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

Dobbiamo modificare leggermente la FDT: 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

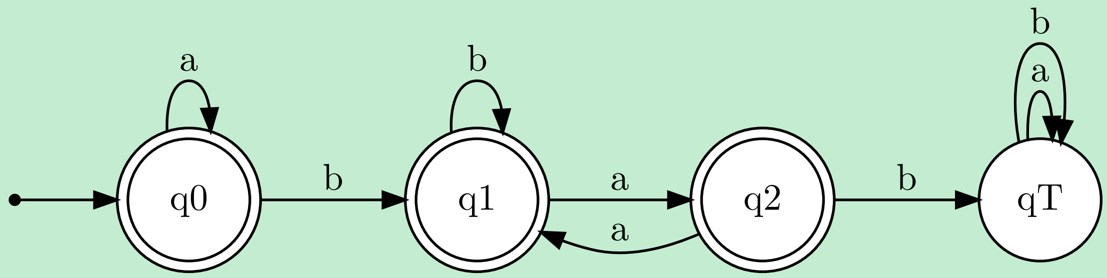
$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}.$$

Nel primo esempio abbiamo visto quello che si definiamo un **problema di analisi**: abbiamo in mano l'automa, dobbiamo descrivere il linguaggio che riconosce. L'altro tipo di problema è il **problema di sintesi**: abbiamo in mano un linguaggio, dobbiamo scrivere un automa per esso.

Esempio 5.1.1.2: 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.



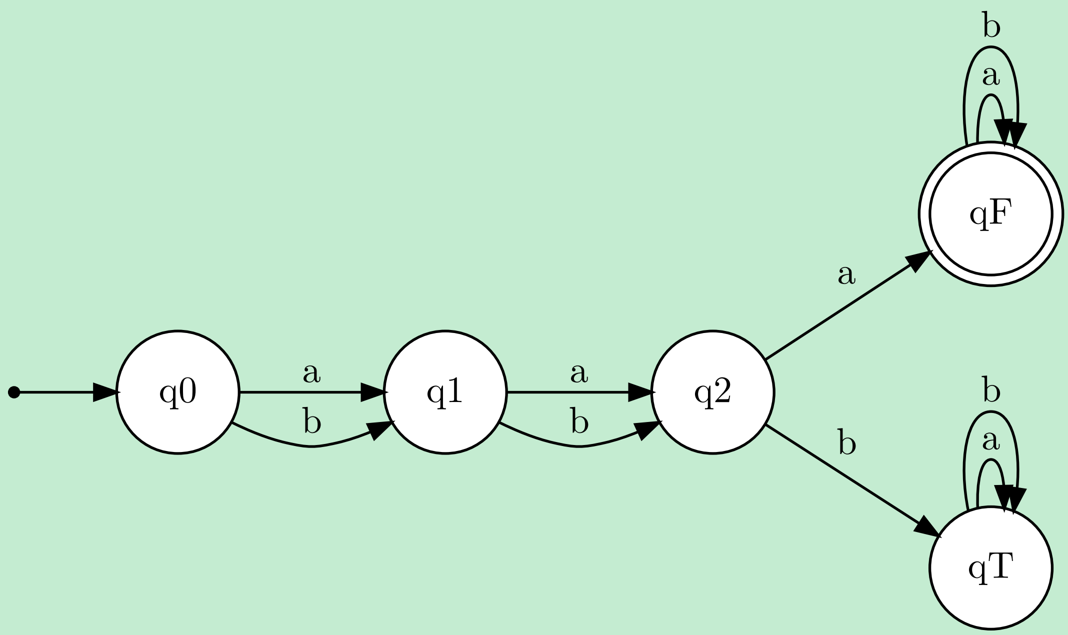
Come vediamo dall'esempio precedente, abbiamo uno stato particolare q_T che è detto **stato trappola**: esso viene utilizzato come «punto di arrivo» per esaurire la lettura dell'input e non accettare la stringa data in input. Finiamo in questo stato se, in uno stato q, leggiamo un carattere che rende la stringa non generabile da 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 5.1.1.3: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

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

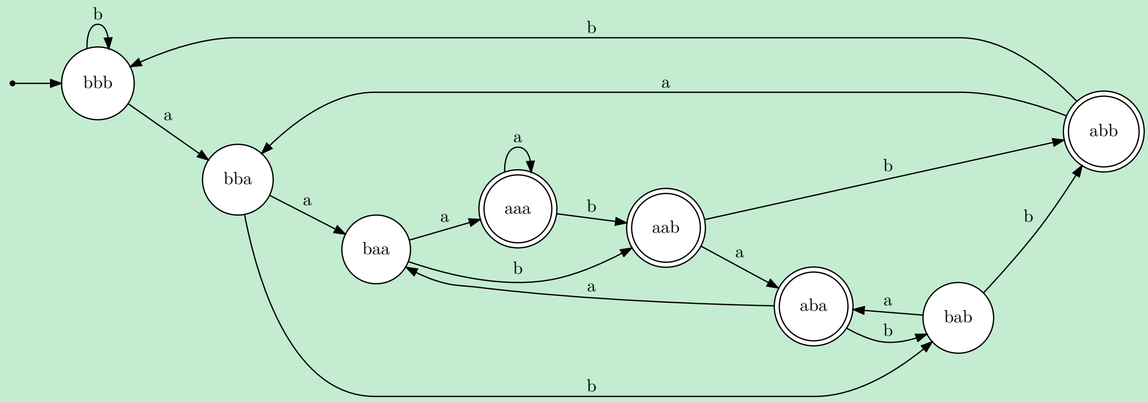
Costruiamo un automa deterministico per L.



Esempio 5.1.1.4: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

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

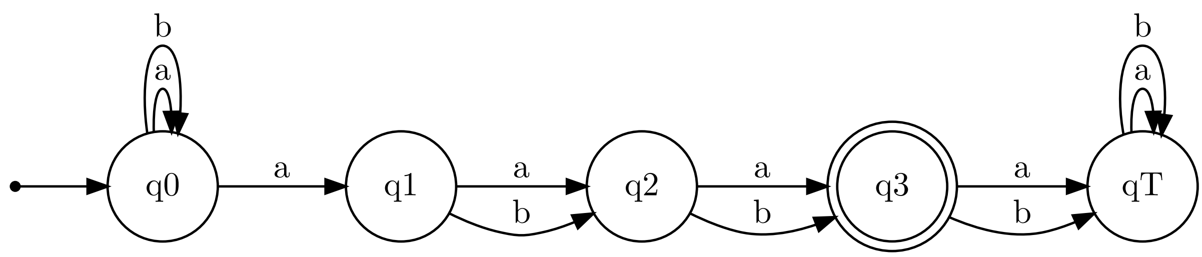
Costruiamo un automa deterministico per L. Qua l'idea è ricordarsi una finestra di 3 simboli e grazie a questa vediamo se il primo carattere che definisce lo stato è una a.



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

5.1.2. Macchine a stati finiti non deterministiche

Vediamo un automa che utilizza meno stati per riconoscere il linguaggio precedente.



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

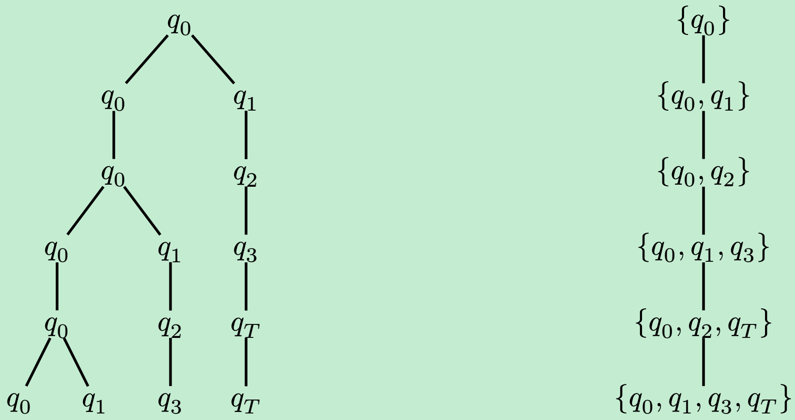
Gli **automi non deterministici**, o **NFA**, sono definiti da una quintupla $A = (Q, \Sigma, \delta, q_0, F)$ definita allo stesso modo dei DFA tranne la funzione di transizione. Essa è la funzione

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

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

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.

Esempio 5.1.2.1: Considerando l'automa precedente, scrivere 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.

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.

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

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

5.1.3. 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;

- $\delta' : Q' \times \Sigma \longrightarrow Q'$ 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;
- $F' = \{\alpha \in Q' \mid \alpha \cap F \neq \varnothing\}$ 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.

5.1.4. Altre forme di non determinismo

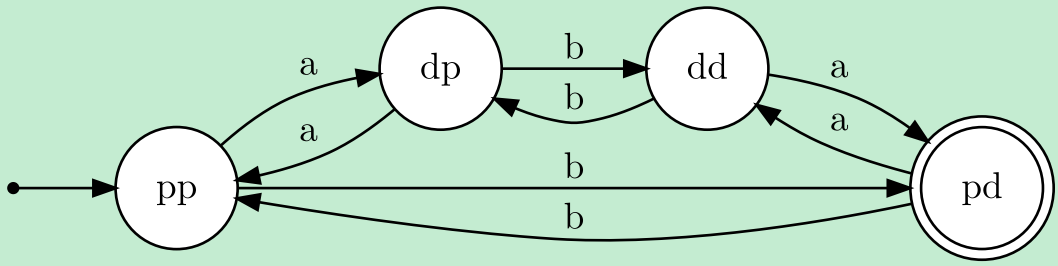
Una ulteriore forma di non determinismo, oltre a quella sulle molteplici transizioni con lo stesso carattere in uno stato, è quella di avere **molteplici stati iniziali**.

6. Lezione 05 [12/03]

6.1. Distinguibilità

Esempio 6.1.1: Sia $\Sigma = \{a, b\}$ e vogliamo un automa che riconosca il linguaggio

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



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

$$(xz \in L \land yz \notin L) \lor (xz \notin L \land yz \in L).$$

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 6.1.1 (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 6.1.1.1: Sia $X = \{x_1, ..., x_n\}$ e sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA che accetta il linguaggio L. Definiamo gli stati

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

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

$$\begin{gathered} q_0 \overset{x_0}{\rightsquigarrow} p_0 \\ \dots \\ q_0 \overset{x_n}{\rightsquigarrow} p_n \end{gathered}$$

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

$$|Q| < n \Longrightarrow |Q| \geq n. \qquad \blacksquare$$

Esempio 6.1.2: 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**.

6.2. Linguaggio L_n

Esempio 6.2.1: Riprendiamo il linguaggio della scorsa lezione 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

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

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 6.2.2: 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 $\sigma = \sigma_1 ... \sigma_n$.

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

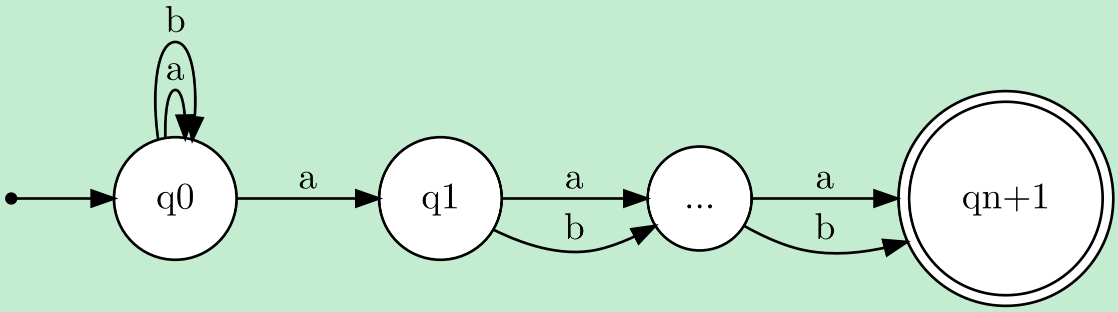
$$z \in \Sigma^{i-1}$$

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 6.2.3: 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.

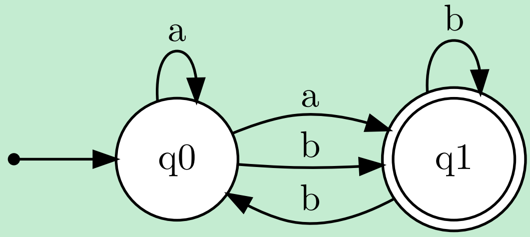


Il numero totale di stati è $n + 1$.

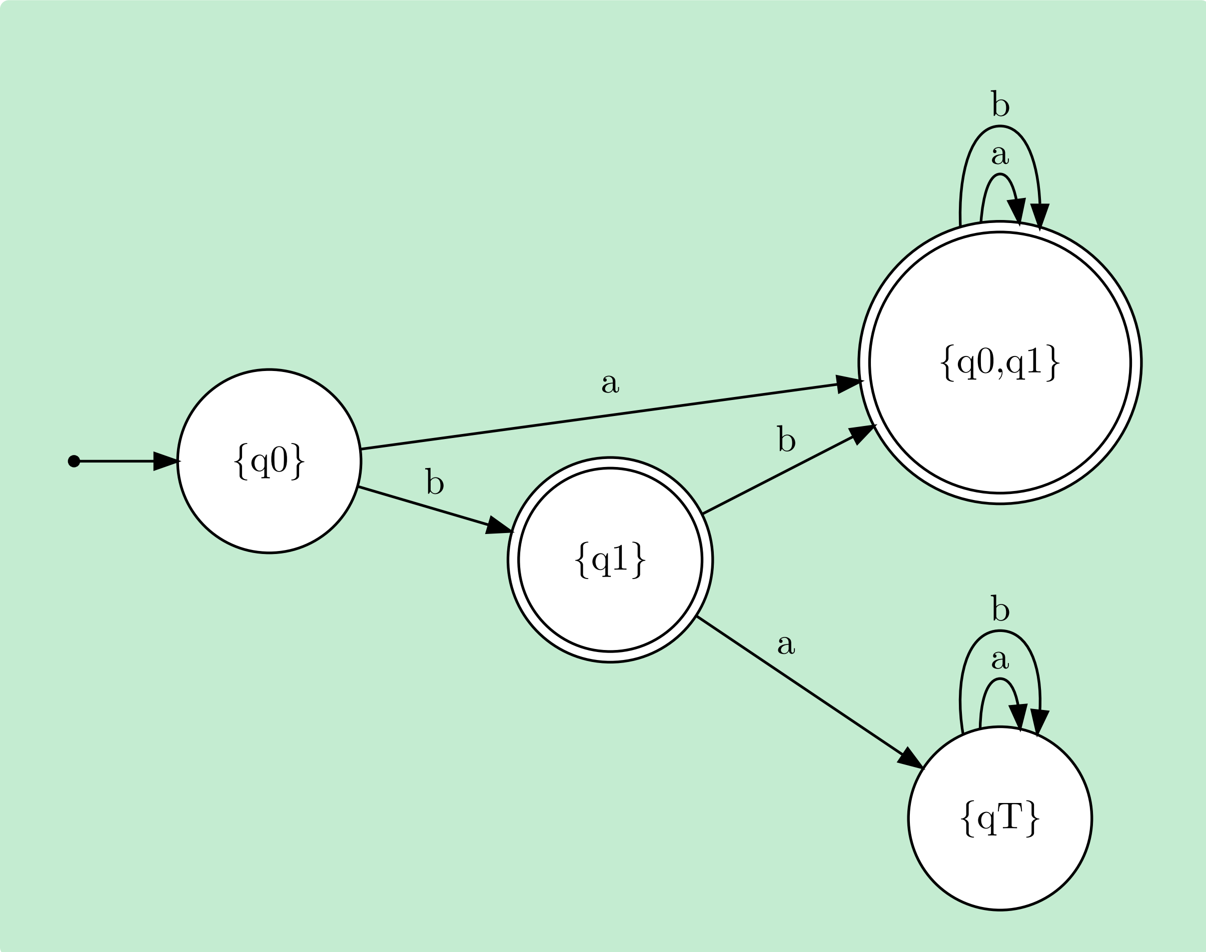
Per L_n abbiamo quindi visto che il numero di stati richiesti per un NFA è $n + 1$, mentre per un DFA è almeno 2^n grazie al teorema sulla distinguibilità. Il salto che abbiamo fatto è quindi **esponenziale**.

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

Esempio 6.2.4: 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 $n \to 2^n$, 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.

6.3. Automa di Meyer-Fischer

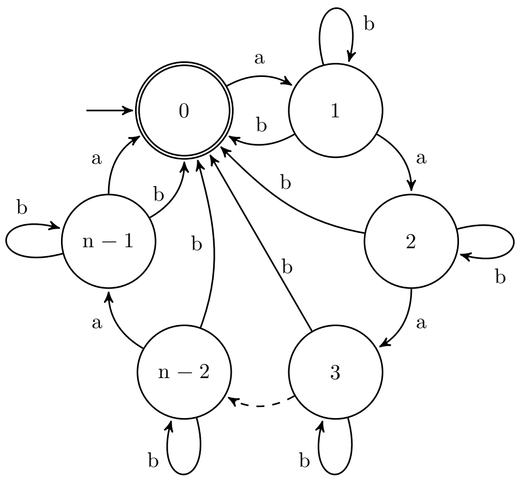
L'**automa di Meyer-Fischer**, ideato da questi due bro nel 1971, sarà il nostro NFA salvatore che ci permetterà di dimostrare quanto detto fino ad adesso.

Sia $M_n = (Q, \Sigma, \delta, q_0, F)$ tali che:
- $Q = \{0, ..., n-1\}$ insieme di n stati;
- $\Sigma = \{a, b\}$;
- $q_0 = 0$ stato iniziale e anche unico stato finale.

La funzione di transizione è tale che

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

L'automa M_n lo possiamo disegnare in questo modo.



Teorema 6.3.1: Ogni DFA equivalente a M_n deve avere almeno 2^n stati.

Dimostrazione 6.3.1.1: Sia $S \subseteq \{0, ..., n-1\}$. Definiamo la stringa

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

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

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

Si può dimostrare inoltre che dati $S, T \subseteq \{0, ..., n-1\}$, 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

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

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

Il numero di stringhe in X dipende dal numero di sottoinsiemi di $\{0, ..., n-1\}$: 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 6.3.1: Per ogni $S \subseteq \{0, ..., n-1\}$ vale

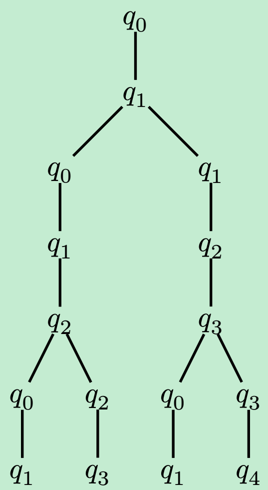
$$\delta(q_0, w_S) = S.$$

Esempio 6.3.1: Sia M_5 una istanza dell'automa di Meyer-Fischer.

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

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

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

Dimostrazione 6.3.2.1: 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. ∎

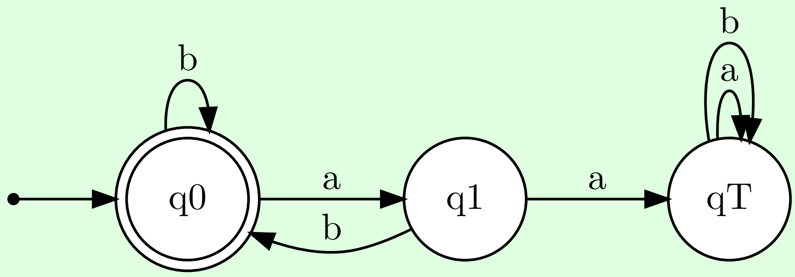
7. Esercizi lezioni 03, 04 e 05 [12/03]

7.1. Esercizio 01

Esercizio 7.1.1:

Richiesta 7.1.1.1: Costruite un automa a stati finiti che riconosca il linguaggio formato da tutte le stringhe sull'alfabeto $\{a, b\}$ nelle quali ogni a è seguita immediatamente da una b.

Soluzione 7.1.1.1:



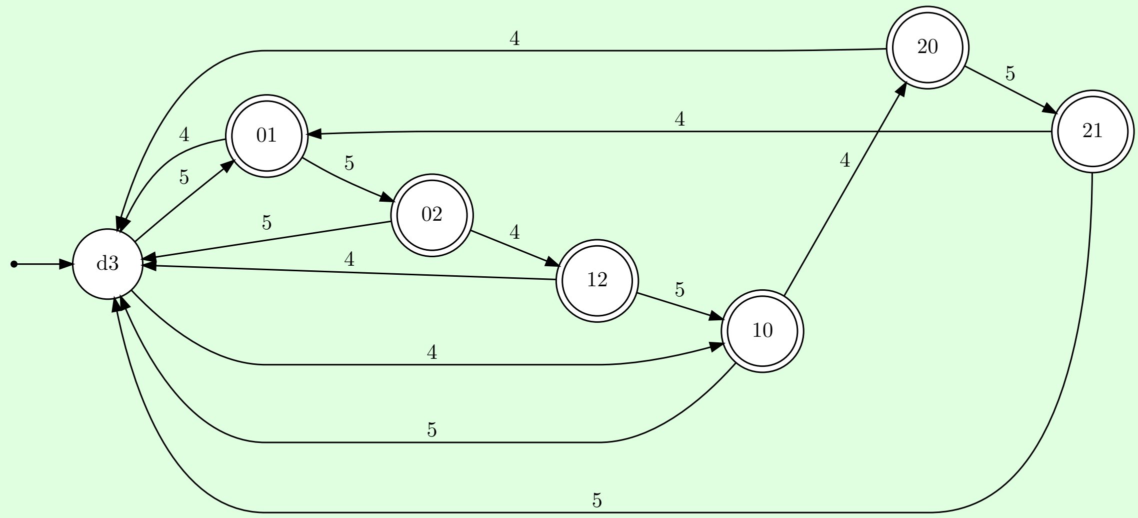
7.2. Esercizio 02

Esercizio 7.2.1:

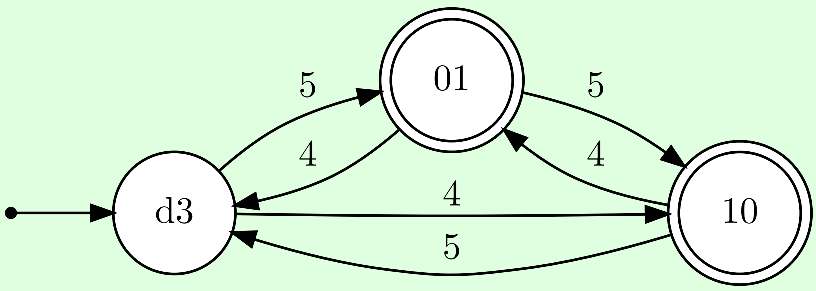
Richiesta 7.2.1.1: Costruite un automa a stati finiti che riconosca il linguaggio formato da tutte le stringhe sull'alfabeto $\{4, 5\}$ che, interpretate come numeri in base 10, rappresentano interi che non sono divisibili per 3.

Suggerimento. Un numero intero è divisibile per 3 se e solo se la somma delle sue cifre in base 10 è divisibile per 3.

Soluzione 7.2.1.1: Prima versione



Soluzione 7.2.1.2: Seconda versione

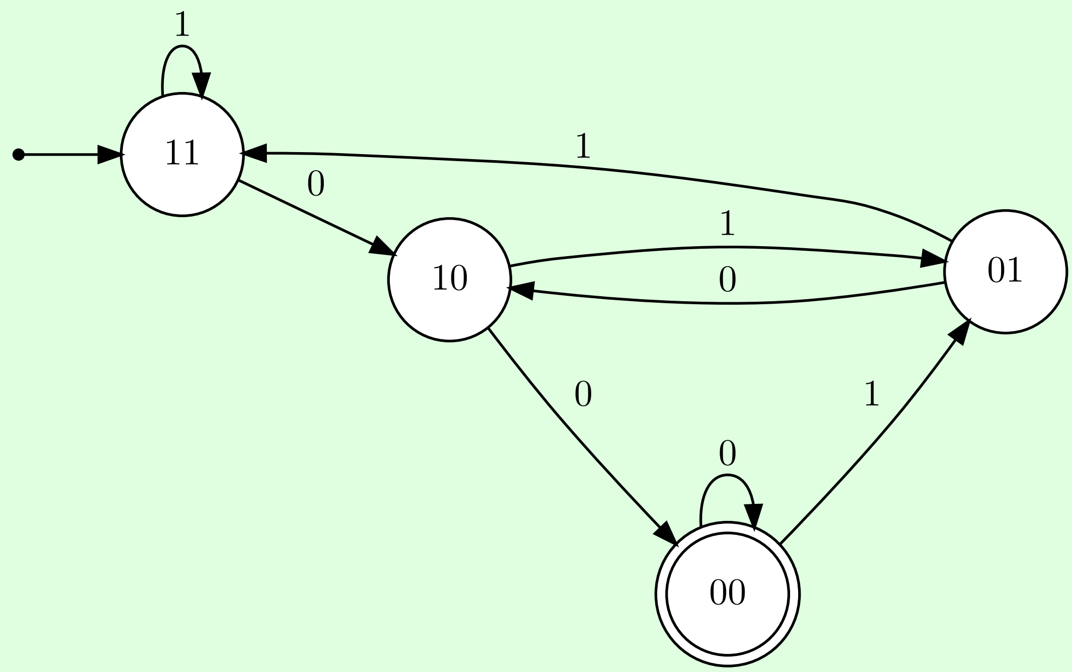


7.3. Esercizio 03

Esercizio 7.3.1:

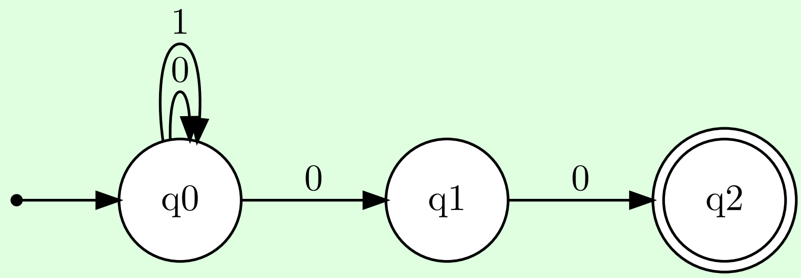
Richiesta 7.3.1.1: Costruite un automa a stati finiti deterministico che riconosca il linguaggio formato da tutte le stringhe sull'alfabeto $\{0, 1\}$ che, interpretate come numeri in notazione binaria, denotano multipli di 4.

Soluzione 7.3.1.1:



Richiesta 7.3.1.2: Utilizzando il non determinismo si riesce a costruire un automa con meno stati?

Soluzione 7.3.1.2: Utilizzando il non determinismo riusciamo ad utilizzare 1 stato in meno, se non inseriamo uno stato trappola per le transizioni dagli stati q_1 e q_2.



Richiesta 7.3.1.3: Generalizzate l'esercizio a multipli di 2^k, dove $k > 0$ è un intero fissato.

Soluzione 7.3.1.3: Per i DFA che riconoscono i multipli di 2^k dobbiamo ricordarci una finestra di k caratteri. Tutte le possibili combinazioni di queste finestre sono 2^k, quindi anche il DFA che riconosce quel linguaggio ha 2^k stati.

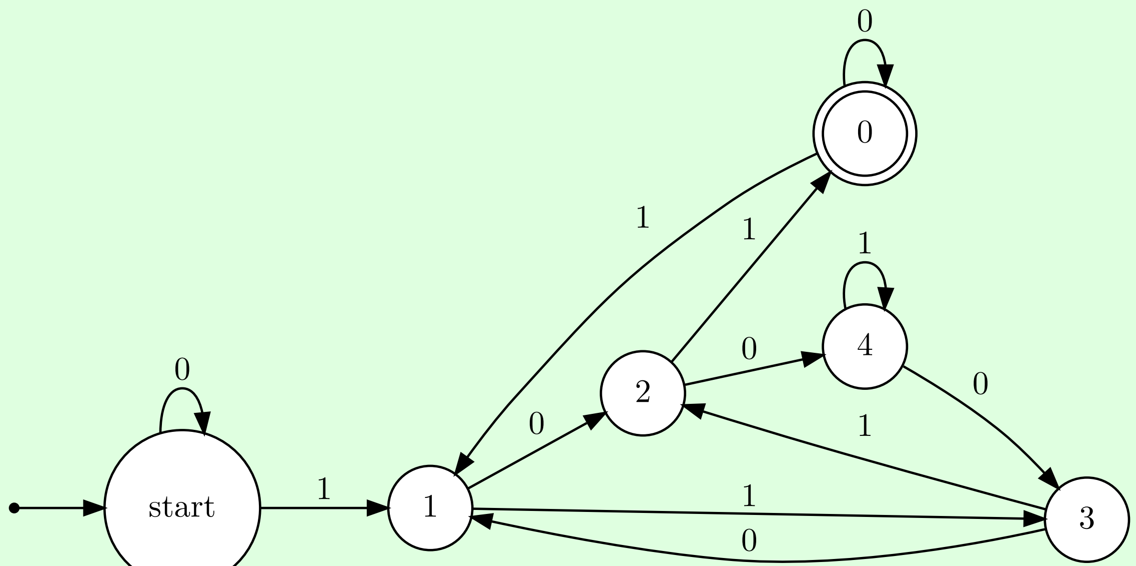
Per gli NFA che riconoscono i multipli di 2^k dobbiamo utilizzare $k + 1$ stati, di cui k leggono gli ultimi k zeri e uno che fa da «stato scommettitore».

7.4. Esercizio 04

Esercizio 7.4.1:

Richiesta 7.4.1.1: Costruite un automa a stati finiti che riconosca il linguaggio formato da tutte le stringhe sull'alfabeto $\{0, 1\}$ che, interpretate come numeri in notazione binaria, rappresentano multipli di 5.

Soluzione 7.4.1.1:



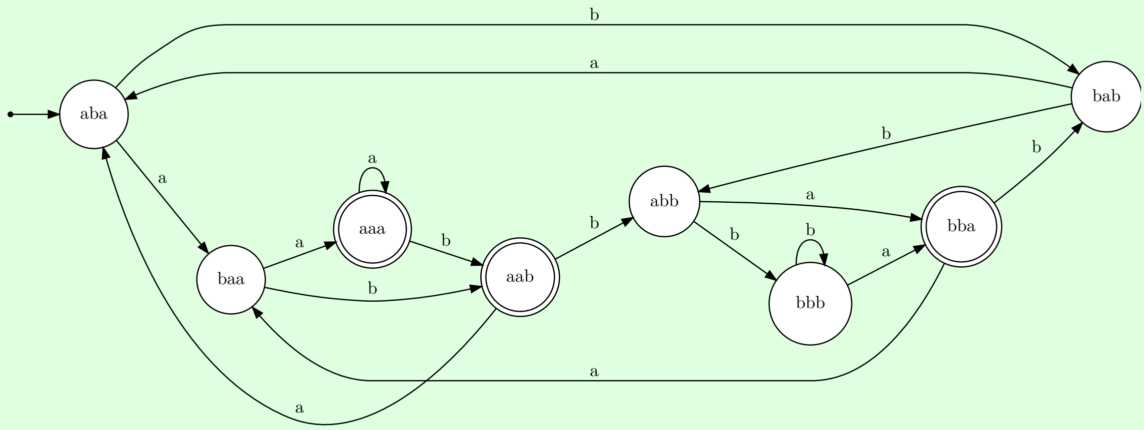
7.5. Esercizio 05

Esercizio 7.5.1: Considerate il seguente linguaggio:

$$L = \{w \in \{a, b\}^* \mid \text{il penultimo e il terzultimo simbolo di } w \text{ sono uguali}\}.$$

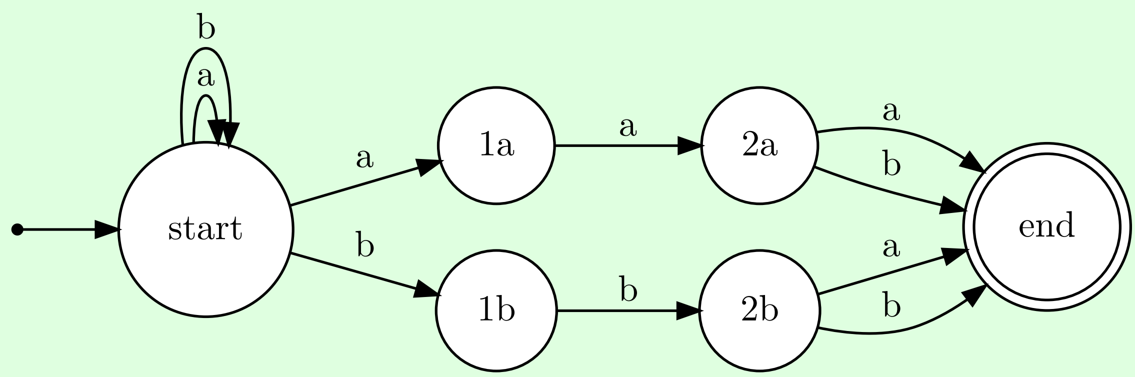
Richiesta 7.5.1.1: Costruite un automa a stati finiti deterministico che accetta L.

Soluzione 7.5.1.1: Secondo me andrebbero usati più stati per le stringhe di 1 e 2 caratteri, oppure si dovrebbe imporre il riconoscimento di stringhe lunghe almeno 3 caratteri.



Richiesta 7.5.1.2: Costruite un automa a stati finiti non deterministico che accetta L.

Soluzione 7.5.1.2:



Richiesta 7.5.1.3: Dimostrare che per il linguaggio L:

- tutte le stringhe di lunghezza 3 sono distinguibili tra loro;
- la parola vuota è distinguibile da tutte le stringhe di lunghezza 3.

Soluzione 7.5.1.3: Sia $X = \{a, b\}^3$. Date due stringhe $\sigma, \gamma \in X$ esse possono avere un carattere diverso in 3 posizioni:

- se $\sigma_1 \neq \gamma_1$:
  - se $\sigma_2 = \gamma_2$ usiamo $z = \varepsilon$;
  - se $\sigma_2 \neq \gamma_2$ usiamo $z = a^2$ oppure $z = b^2$;
- se $\sigma_2 \neq \gamma_2$:
  - se $\sigma_3 = \gamma_3$ usiamo $z \in \{a, b\}$;
  - se $\sigma_3 \neq \gamma_3$ usiamo $z = a^2$ oppure $z = b^2$;
- se $\sigma_3 \neq \gamma_3$ usiamo $z = a^2$ oppure $z = b^2$.

Non voglio dimostrare perché funziona, ma funziona, fidatevi di me.

Inoltre, la stringa vuota è distinguibile da ogni stringa di lunghezza 3 perché basta aggiungere una stringa z formata dall'ultimo carattere della stringa σ che stiamo considerando ripetuto due volte.

Richiesta 7.5.1.4: Utilizzando i risultati precedenti, ricavate un limite inferiore per il numero di stati di ogni automa deterministico che accetta L.

Soluzione 7.5.1.4: Grazie al teorema sulla distinguibilità, ogni DFA per il linguaggio L deve usare almeno 8 stati.

7.6. Esercizio 06

Esercizio 7.6.1: Costruite un insieme di stringhe distinguibili tra loro per ognuno dei seguenti linguaggi.

Richiesta 7.6.1.1: $\{w \in \{a, b\}^* \mid \#_a(w) = \#_b(w)\}$.

Soluzione 7.6.1.1: $X = \{a^n \mid n \geq 0\}$.

Richiesta 7.6.1.2: $\{a^n b^n \mid n \geq 0\}$.

Soluzione 7.6.1.2: $X = \{a^n \mid n \geq 0\}$.

Richiesta 7.6.1.3: $\{ww^R \mid w \in \{a,b\}^*\}$ dove, per ogni stringa w, w^R indica la stringa w scritta al contrario.

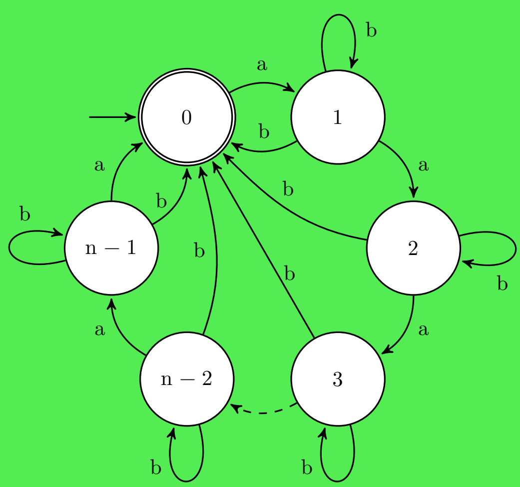
Soluzione 7.6.1.3: $X = \{(ab)^n \mid n \geq 0\}$.

Richiesta 7.6.1.4: Per alcuni di questi linguaggi riuscite ad ottenere insiemi di stringhe distinguibili di cardinalità infinita? Cosa significa ciò?

Soluzione 7.6.1.4: Significa che non sono dei linguaggi di tipo 3.

7.7. Esercizio 07

Esercizio 7.7.1: Considerate l'automa di Meyer e Fischer M_n presentato nella Lezione 5 (caso peggiore della costruzione per sottoinsiemi) e mostrato nella seguente figura:



Richiesta 7.7.1.1: Descrivete a parole la proprietà che deve soddisfare una stringa per essere accettata da M_n. Riuscite a costruire un automa non deterministico, diverso da

M_n, per lo stesso linguaggio, basandovi su tale proprietà? (Potete usare un numero di stati diverso da n, ma non esponenziale, e stati iniziali multipli.)

Soluzione 7.7.1.1: No non ci riesco.

8. Lezione 06 [14/03]

8.1. Molti 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 8.1.1: 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.

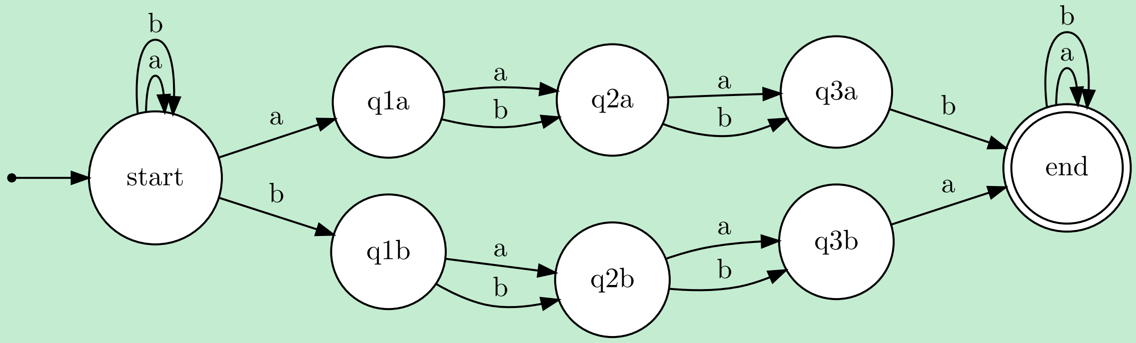
Visto che siamo bravi con le scommesse, andiamo a fare un po' di sano **gambling**.

Esempio 8.1.2: 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 8.1.3: Dato L_n il linguaggio di prima, sia $X = \Sigma^n$.

Prendiamo le stringhe $\sigma = \sigma_1 ... \sigma_n$ e $\gamma = \gamma_1 ... \gamma_n$ 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 $\sigma_1 ... \sigma_{i-1}$: con questa scelta otteniamo le stringhe

$$\sigma z = \sigma_1 ... \sigma_{i-1} \sigma_i \sigma_{i+1} ... \sigma_n \sigma_1 ... \sigma_{i-1} \{a, b\}$$
$$\gamma z = \gamma_1 ... \gamma_{i-1} \gamma_i \gamma_{i+1} ... \gamma_n \gamma_1 ... \gamma_{i-1} \{a, b\}$$

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

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

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

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

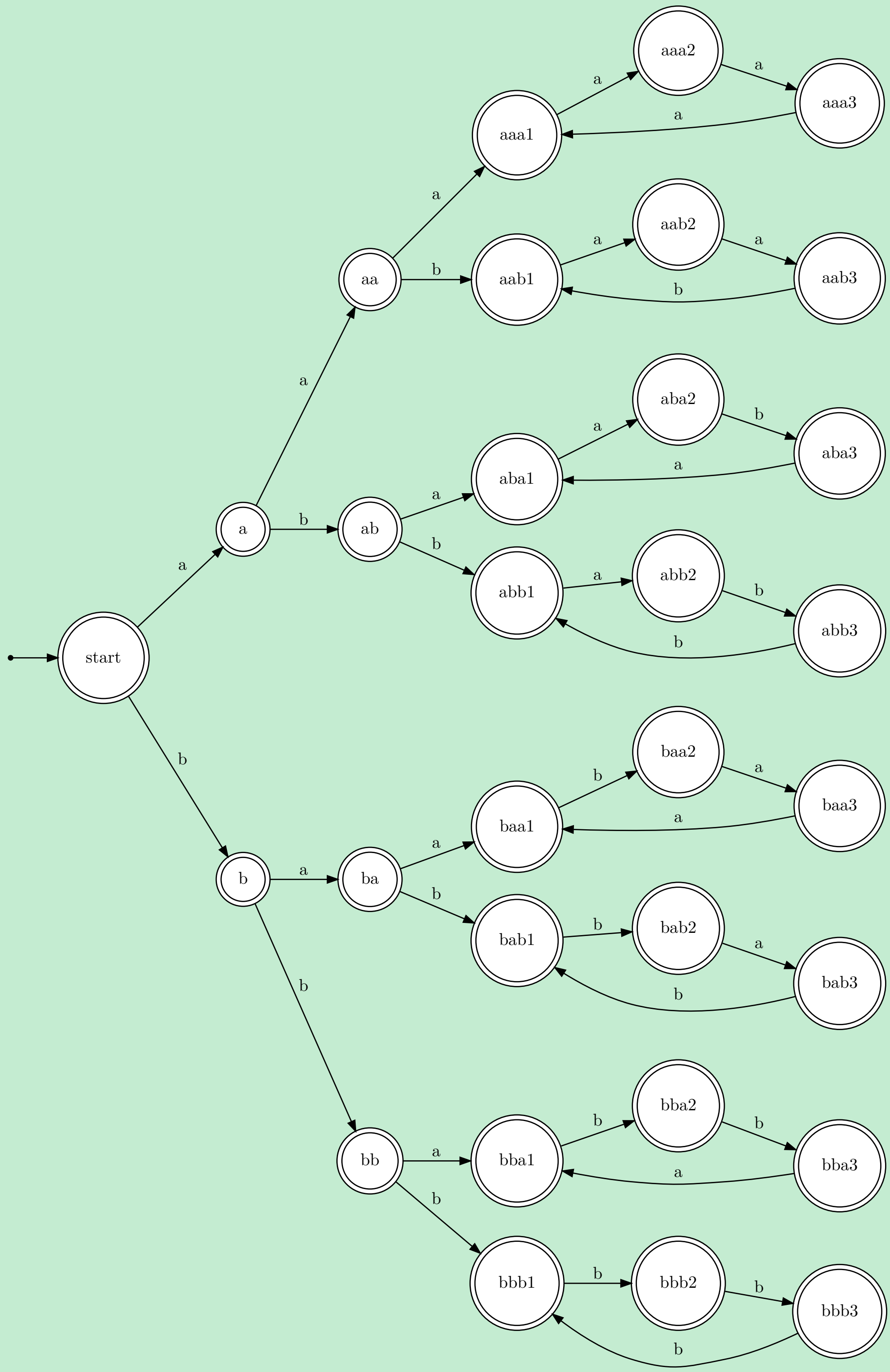
Posso ripetere w quante volte voglio, ma poi la parte finale deve ripetere in parte w.

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'automa.

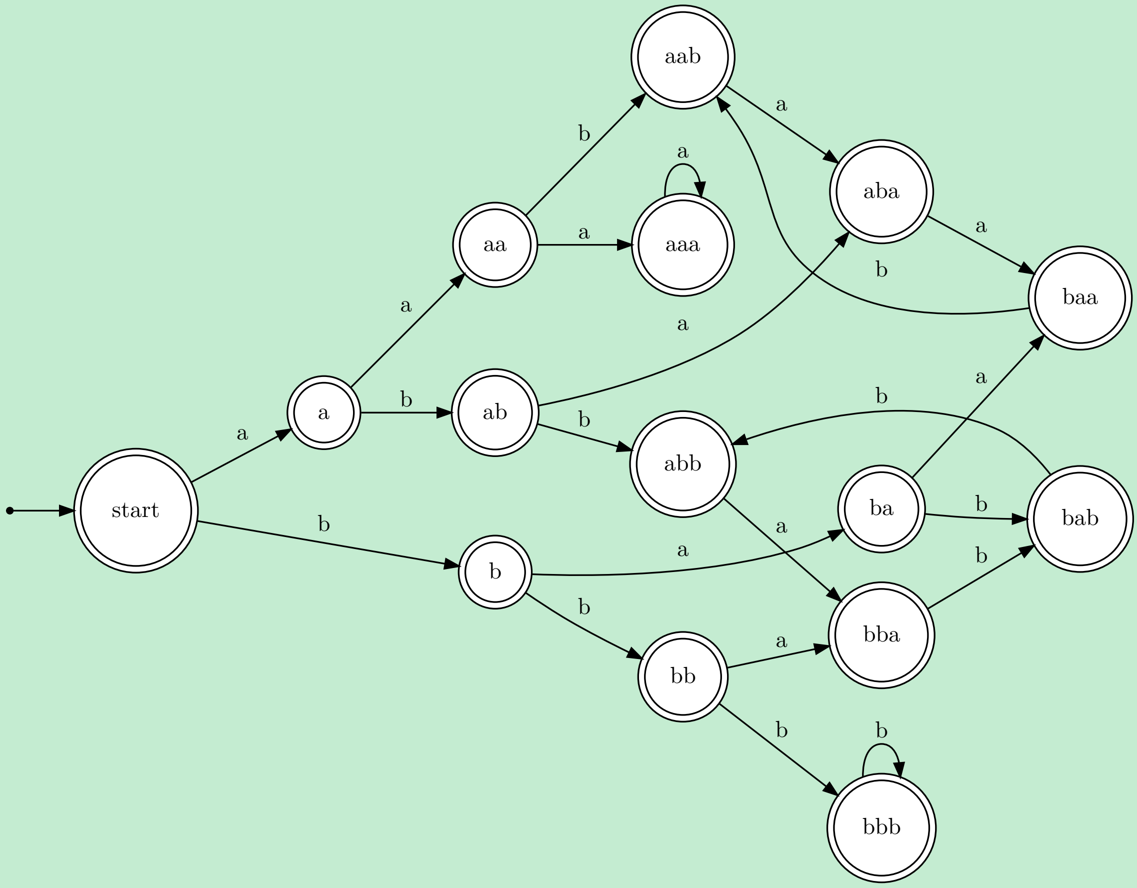


Per il linguaggio generico L'_n, l'albero usa un numero di stati pari a

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

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.

Pesante questo pezzo, ma rendiamolo ancora più pesante: se volessimo fare un NFA? Questa domanda è un po' pallosa perché il non determinismo è buono quando la scommessa da fare è una sola, non quando le scommesse sono da fare sempre, come in questo caso che abbiamo un «per ogni».

8.2. Fooling set

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

Definizione 8.2.1 (Fooling set): Sia $L \subseteq \Sigma^*$. Definiamo

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

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

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

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

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

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

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 8.2.1 (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 8.2.1.1: 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:

$$q_0 \overset{x_1}{\rightsquigarrow} p_1 \overset{y_1}{\rightsquigarrow} f_1$$
$$\vdots$$
$$q_0 \overset{x_N}{\rightsquigarrow} p_N \overset{y_N}{\rightsquigarrow} f_N$$

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. ■

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

Esempio 8.2.1: 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.

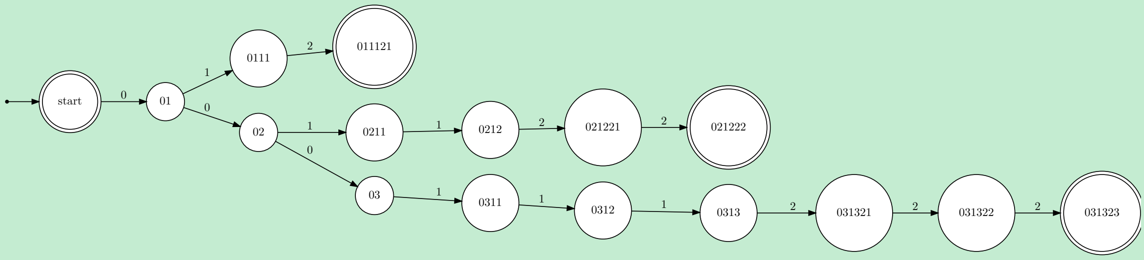
Linguaggio	**DFA**	**NFA**
L_n	$\geq 2^n$	$\leq 2n + 2$
L'_n	$\geq 2^n \wedge \leq 2^{n+1}$	$\geq 2^n$

Finiamo con un ultimo esempio.

Esempio 8.2.2: 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

$$P = \left\{\left(0^i 1^j, 1^{i-j} 2^i\right) \mid i = 1, ..., n \land j = 1, ..., i\right\}.$$

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.

9. Esercizi lezione 06 [14/03]

9.1. Esercizio 01

Esercizio 9.1.1: Considerate il linguaggio

$$\mathrm{DOUBLE}_k = \{ww \mid w \in \{a,b\}^k\},$$

dove $k > 0$ è un intero fissato.

Richiesta 9.1.1.1: Trovare un fooling set di cardinalità 2^k per questo linguaggio. Riuscite a trovare un fooling set o un extended fooling set di cardinalità maggiore?

Soluzione 9.1.1.1: Definiamo il fooling set

$$P = \left\{(x,x) \mid x \in \{a,b\}^k\right\}.$$

Esso è un fooling set per DOUBLE_k:
- la stringa $z = xx$ appartiene al linguaggio;
- presi due elementi di lunghezza k da due coppie diverse, essi saranno diversi in almeno una posizione e quindi la stringa risultate non appartiene al linguaggio.

Non so se riusciamo a trovare un fooling set o un extended set di cardinalità maggiore, non ti bastava questo qua?

9.2. Esercizio 02

Esercizio 9.2.1: Considerate il linguaggio

$$\mathrm{PAL}_k = \{w \in \{a,b\}^k \mid w = w^R\},$$

dove k è un intero fissato.

Richiesta 9.2.1.1: Qual è l'extended fooling set per PAL_k di cardinalità maggiore che riuscite a trovare?

Soluzione 9.2.1.1: Definiamo l'insieme

$$P = \left\{(w, w^R) \mid w \in \{a,b\}^{\frac{k}{2}}\right\}.$$

Esso è un fooling set per il linguaggio PAL_k:

- la coppia (w, w^R) appartiene al linguaggio;
- ogni altra stringa formata da elementi di due coppie diverse non appartiene al linguaggio perché esiste almeno una coppia di caratteri con la stessa distanza dagli estremi che sono diversi.

P ha cardinalità $2^{\frac{k}{2}}$, quindi ogni NFA per PAL_k ha almeno $2^{\frac{k}{2}}$ stati.

9.3. Esercizio 03

Esercizio 9.3.1: Considerate il linguaggio

$$K_k = \{w \mid w = x_1 x_2 \cdots x_m x \mid m > 0, x_1, \ldots x_m \in \{a, b\}^k, \exists i \in \{1, \ldots, m\} \mid x_i = x\},$$

dove k è un intero fissato. Si può osservare che ogni stringa w di questo linguaggio è la concatenazione di blocchi di lunghezza k, in cui l'ultimo blocco coincide con uno dei blocchi precedenti.

Richiesta 9.3.1.1: Riuscite a costruire un (extended) fooling set di cardinalità 2^k o maggiore per il linguaggio K_k?

Suggerimento. Ispiratevi all'esercizio 1.

Soluzione 9.3.1.1: Definiamo il fooling set

$$P = \{(x, x) \mid x \in \{a, b\}^k\}.$$

Esso è un fooling set per K_k:
- la stringa $z = xx$ appartiene al linguaggio;
- presi due elementi di lunghezza k da due coppie diverse, essi rappresentano due blocchi diversi perché diversi in almeno una posizione, quindi la stringa risultate non appartiene al linguaggio.

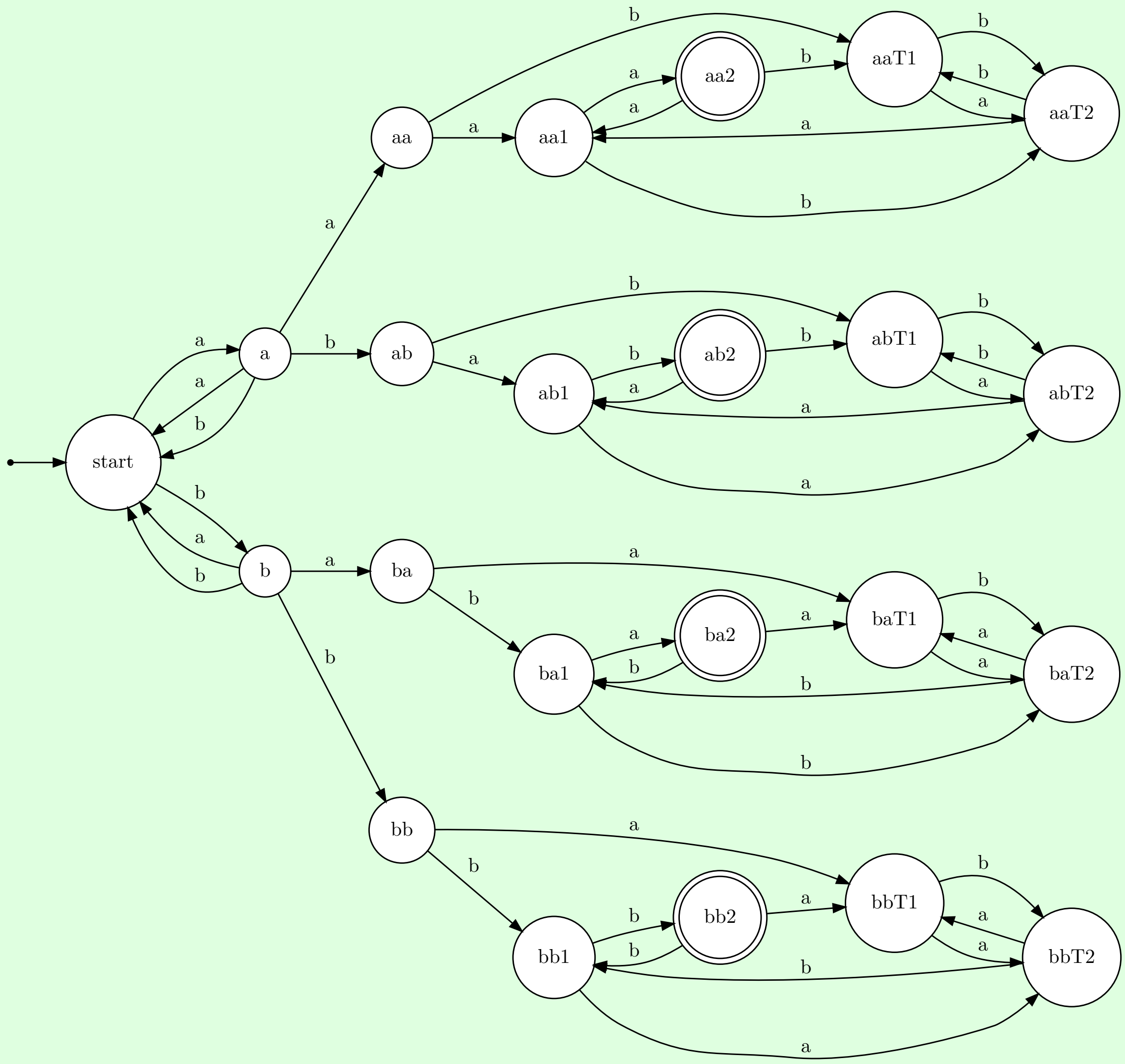
La cardinalità di P è 2^k, quindi ogni NFA per K_k ha almeno 2^k stati.

Richiesta 9.3.1.2: Quale è l'informazione principale che un automa non deterministico può scegliere di ricordare nel proprio controllo a stati finiti durante la lettura di una stringa per riuscire a riconoscere K_k?

Suggerimento. Che «scommessa» può fare l'automa mentre legge la stringa in ingresso e come può verificare tale scommessa leggendo l'ultimo blocco?

Soluzione 9.3.1.2: Un automa non deterministico dovrebbe scommettere che ha appena finito di leggere il blocco che poi sarà ripetuto alla fine. Il controllo che fa l'automa alla fine è quello di uguaglianza con il blocco scommesso, che viene fatta in maniera deterministica ma richiede un grande numero di stati. Va creato quindi un albero che in maniera non deterministica mi manda indietro allo start prima dell'ultimo nodo.

Ad esempio, se $k = 2$ l'automa non deterministico ha questa forma.



Richiesta 9.3.1.3: Supponete di costruire un automa deterministico per riconoscere K_k. Cosa ha necessità di ricordare l'automa nel proprio controllo a stati finiti mentre legge la stringa in input?

Soluzione 9.3.1.3: Un automa deterministico si deve ricordare i blocchi di lunghezza k che ha incontrato fino a quel momento. Questo però fa esplodere il numero di stati, perché dobbiamo calcolare praticamente ogni combinazione possibile.

Richiesta 9.3.1.4: Utilizzando il concetto di distinguibilità, dimostrate che ogni automa deterministico che riconosce K_k deve avere almeno 2^{2^k} stati.

Soluzione 9.3.1.4: Sia $T = \{a, b\}^k$ insieme di tutte le stringhe di lunghezza k costruite sull'alfabeto $\{a, b\}$. Definiamo l'insieme

$$X = \mathcal{P}(T)$$

insieme di tutti i sottoinsiemi di T, ovvero l'insieme delle parti di T. Supponiamo che ogni sottoinsieme venga rappresentato come una stringa ottenuta dalla concatenazione dei suoi elementi.

Questo insieme è formato da stringhe distinguibili tra loro: infatti, prese due stringhe di X, esse hanno almeno un blocco lungo k che appartiene a una delle due ma non all'altra. Ma allora usando quell'elemento come stringa z che distingue noi otteniamo l'appartenenza per la stringa che contiene quell'elemento e la non appartenenza per l'altra.

La cardinalità di X è $2^{|T|}$, ovvero 2^{2^k}, quindi ogni DFA per K_k ha almeno 2^{2^k} stati.

9.4. Esercizio 04

Esercizio 9.4.1: Considerate il linguaggio

$$J_k = \{w \mid w = xx_1 \cdots x_m \mid m > 0, x_1, \ldots x_m, x \in \{a, b\}^k, \exists i \in \{1, \ldots, m\} \mid x_i = x\},$$

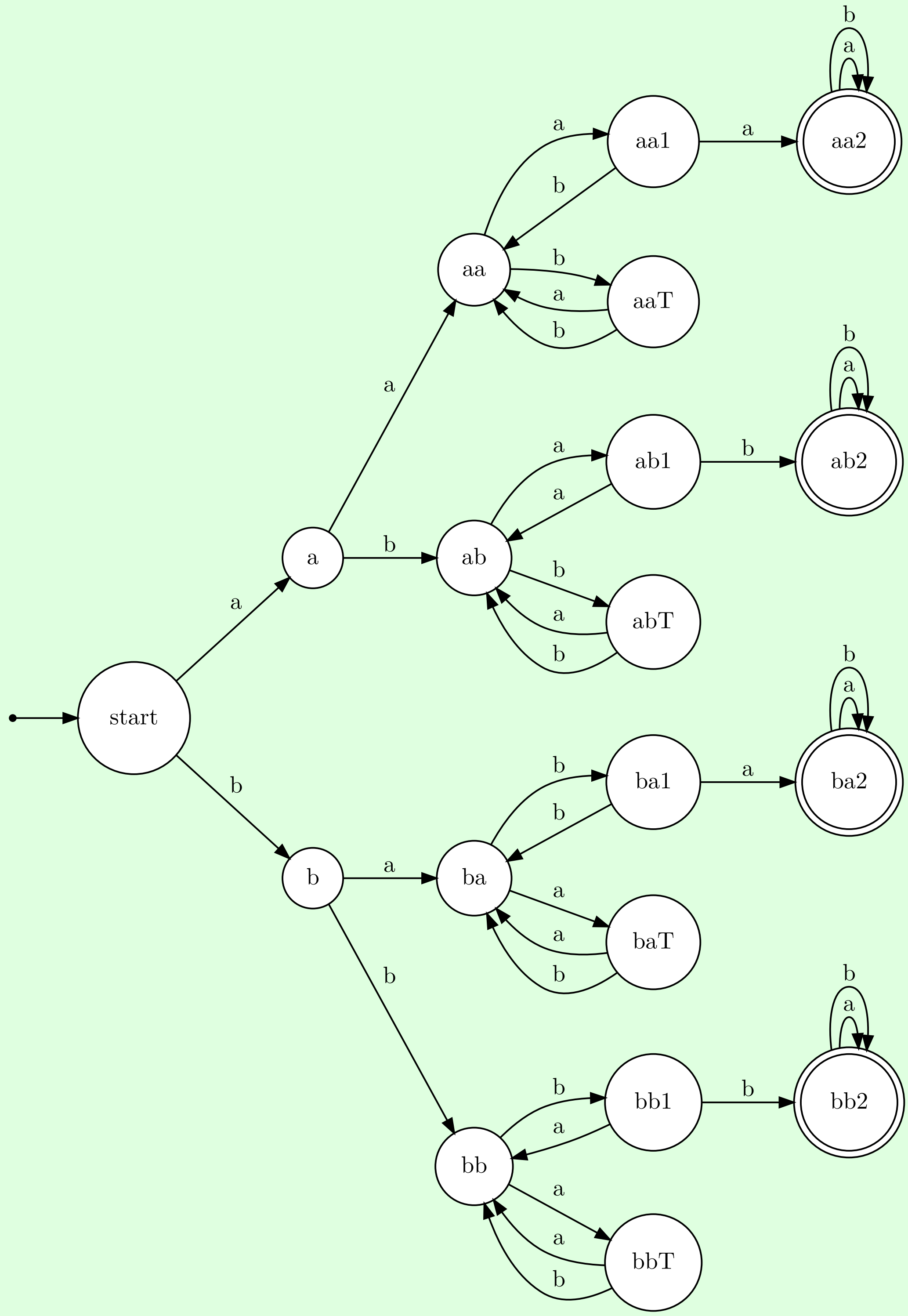
dove k è un intero fissato. Si può osservare che ogni stringa w di questo linguaggio è la concatenazione di blocchi di lunghezza k, in cui il primo blocco coincide con uno dei blocchi successivi; ogni stringa di J_k si ottiene «rovesciando» una stringa del linguaggio K_k dell'esercizio 3.

Richiesta 9.4.1.1: Supponete di costruire automi a stati finiti per J_k. Valgono ancora gli stessi limiti inferiori ottenuti per K_k o si riescono a costruire automi più piccoli? Rispondete sia nel caso di automi deterministici sia in quello di automi non deterministici.

Soluzione 9.4.1.1: Un DFA per J_k usa molti meno stati di un DFA per K_k: infatti, un DFA per J_k deve fare un albero per vedere quale stringa lunga k viene letta all'inizio e poi deve eseguire un controllo con altri $2k - 1$ stati per ogni stato foglia. Il numero di stati è quindi

$$2^{k+1} - 1 + 2^k(2k - 1) = 2^{k+1} - 1 + k2^{k+1} - 2^k = O\big(k2^k\big).$$

Vediamo un esempio con $k = 2$ per semplicità.



Per il caso non deterministico, ogni NFA deve comunque generare l'albero iniziale per vedere le possibili combinazioni. Nei nodi delle combinazioni potremmo inserire la scommessa, ma questo ci farebbe accettare più stringhe di quelle che vorremmo accettare.

Quindi non lo so RIP, ci penserò più avanti, magari c'è un fooling set da trovare.

9.5. Esercizio 05

Esercizio 9.5.1:

Richiesta 9.5.1.1: Ispirandovi all'esercizio 3, fornite limiti inferiori per il numero di stati degli automi che riconoscono il seguente linguaggio:

$$E_k = \{w \mid w = x_1 \cdots x_m \mid m > 0, x_1, ..., x_m \in \{a, b\}^k, \exists i < j \in \{1, ..., m\} \mid x_i = x_j\},$$

dove k è un intero fissato. Considerate sia il caso deterministico che quello non deterministico.

Soluzione 9.5.1.1: Definiamo l'insieme

$$P = \{(x, x) \mid x \in \{a, b\}^k\}.$$

Questo è un fooling set per il linguaggio E_k:
- la stringa $z = xx$ appartiene a al linguaggio;
- la stringa $z = xy$ non appartiene al linguaggio.

Allora ogni NFA per E_k ha almeno $|P| = 2^k$ stati.

Per i DFA, come per K_k, essi si devono ricordare i blocchi lunghi k che hanno incontrato fino a quel momento, e questo fa esplodere il numero di stati. Infatti, definiamo l'insieme

$$X = \mathcal{P}(\{a, b\}^k).$$

Esso è un insieme di stringhe distinguibili per E_k: presi due sottoinsiemi A e B, allora prendiamo un elemento $x \in A/B$ e usiamolo per distinguere le due stringhe (generate dalla concatenazione di tutte le stringhe contenute in un sottoinsieme).

Ma allora ogni DFA per E_k ha almeno $|X| = 2^{2^k}$ stati.

10. Lezione 07 [19/03]

10.1. Introduzione matematica

Abbiamo visto due criteri che limitavano il numero di stati di DFA e NFA per un certo linguaggio. Oggi vediamo un criterio che lavora direttamente sugli automi e non sui linguaggi.

Sia S un insieme, una **relazione binaria** sull'insieme S è definita come l'insieme

$$R \subseteq S \times S.$$

Come notazione useremo $x\,R\,y$ oppure $(x, y) \in R$, molto di più la prima che la seconda.

Ci interessiamo ad un tipo molto particolare di relazioni.

Definizione 10.1.1: La relazione R è una **relazione di equivalenza** se e solo se R è:
- **riflessiva**, ovvero $\forall x \in S \quad x\,R\,x$;
- **simmetrica**, ovvero $\forall x, y \in S \quad x\,R\,y \Longrightarrow y\,R\,x$;
- **transitiva** $\forall x, y, z \in S \quad x\,R\,y \land y\,R\,z \Longrightarrow x\,R\,z$.

Una relazione di equivalenza **induce** sull'insieme S una **partizione** formata da **classi di equivalenza**. Queste classi sono formate da elementi che sono equivalenti tra di loro. Le classi di equivalenza le indichiamo con $[x]_R$, dove $x \in S$ è detto **rappresentante** (credo).

Se R è una relazione di equivalenza, l'**indice** di R è il numero di classi di equivalenza.

Esempio 10.1.1: Sia $S = \mathbb{N}$. Definiamo la relazione $R \subseteq \mathbb{N} \times \mathbb{N}$ tale che

$$x\,R\,y \Longleftrightarrow x \bmod 3 = y \bmod 3.$$

Questa è una relazione di equivalenza (non lo dimostriamo) che ha tre classi di equivalenza:
- $[0]_R$ formata da tutti i multipli di 3;
- $[1]_R$ formata da tutti i multipli di 3 sommati a 1;
- $[2]_R$ formata da tutti i multipli di 3 sommati a 2.

L'indice di questa relazione è quindi 3.

Definizione 10.1.2: Sia $\cdot$ un'operazione sull'insieme S. La relazione R è **invariante a destra** rispetto a $\cdot$ se presi due elementi nella relazione R, e applicando $\cdot$ con uno stesso elemento, otteniamo ancora due elementi in relazione, ovvero

$$x\,R\,y \Longrightarrow \forall z \in S \quad (x \cdot z)\,R\,(y \cdot z).$$

Esempio 10.1.2: Sia R la relazione dell'esempio precedente. Definiamo $\cdot = +$ l'operazione di somma. La relazione R è invariante a destra?

Dobbiamo verificare se

$$x \, R \, y \Longrightarrow \forall z \in \mathbb{N} \quad (x+z) \, R \, (y+z),$$

ovvero se

$$x \bmod 3 = y \bmod 3 \Longrightarrow \forall z \in \mathbb{N} \quad (x+z) \bmod 3 = (y+z) \bmod 3.$$

Questo è vero perché ce lo dice l'algebra modulare, quindi R è invariante a destra.

Ora vediamo una definizione che va contro la semantica italiana.

Definizione 10.1.3: Sia S un insieme e siano $R_1, R_2 \subseteq S \times S$ due relazioni di equivalenza su S.

Diciamo che R_1 è un **raffinamento** di R_2 se e solo se:
1. ogni classe di equivalenza di R_1 è contenuta in una classe di equivalenza di R_2 OPPURE
2. ogni classe di R_2 è l'unione di alcune classi di R_1 OPPURE
3. vale

$$\forall x, y \in S \quad (x,y) \in R_1 \Longrightarrow (x,y) \in R_2.$$

Il primo punto è la definizione, le altre sono solo conseguenze.

Perché non rispecchia molto la semantica italiana? Perché un raffinamento di solito è qualcosa di migliore, in questo caso invece è il contrario: se R_1 è un raffinamento di R_2 allora R_1 è peggiore di R_2 in termini di classi di equivalenza.

Esempio 10.1.3: Data la relazione R di prima, definiamo ora la relazione R' tale che

$$x \, R' \, y \Longleftrightarrow x \bmod 2 = y \bmod 2.$$

Le classi di equivalenza di questa relazione sono $[0]_{R'}$ e $[1]_{R'}$.

Come è messa R rispetto a R'? E R' rispetto a R?

Nessuna delle due è un raffinamento dell'altra: ci sono elementi sparsi un po' qua e là quindi non riusciamo a unire le classi di una nelle classi dell'altra.

Sia invece R'' la relazione tale che

$$x \, R'' \, y \Longleftrightarrow x \bmod 6 = y \bmod 6.$$

La relazione R'' ha 6 classi di equivalenza con le varie classi di resto da 0 a 5.

Come è messa R' rispetto a R''? E R'' rispetto a R'?

Possiamo dire che R'' è un raffinamento di R': infatti, la classe $[0]_{R'}$ la possiamo scrivere come

$$\bigcup_{i \text{ pari}} [i]_{R''}$$

mentre la classe $[1]_{R'}$ la possiamo scrivere come

$$\bigcup_{i \text{ dispari}} [i]_{R''}.$$

Infine, come è messa R rispetto a R''? E R'' rispetto a R?

Anche in questo caso, possiamo dire che R'' è un raffinamento di R: infatti, la classe $[0]_R$ la possiamo scrivere come

$$[0]_{R''} \cup [3]_{R''},$$

la classe $[1]_R$ la possiamo scrivere come

$$[1]_{R''} \cup [4]_{R''}$$

mentre la classe $[2]_R$ la possiamo scrivere come

$$[2]_{R''} \cup [5]_{R''}.$$

10.2. Automa minimo

10.2.1. Relazione R_M

Sia $M = (Q, \Sigma, \delta, q_0, F)$ un DFA. Definiamo la relazione

$$R_M \subseteq \Sigma^* \times \Sigma^*$$

tale che

$$x \, R_M \, y \Longleftrightarrow \delta(q_0, x) = \delta(q_0, y).$$

In poche parole, due stringhe sono in relazione se e solo se vanno a finire nello stesso stato.

Lemma 10.2.1.1: La relazione R_M è una relazione di equivalenza.

Dimostrazione 10.2.1.1.1: Facciamo vedere che R_M rispetta RST.

La relazione R_M è riflessiva: banale per la riflessività l'uguale.

La relazione R_M è simmetrica: banale per la simmetria dell'uguale.

La relazione R_M è transitiva: banale per la transitività dell'uguale.

Ma allora R_M è di equivalenza. ∎

Lemma 10.2.1.2: La relazione R_M è invariante a destra rispetto all'operazione di concatenazione.

Dimostrazione 10.2.1.2.1: Dobbiamo dimostrare che

$$x\, R_M\, y \Longrightarrow \forall z \in \Sigma^* \quad (xz)\, R_M\, (yz).$$

Ma questo è vero: con x e y vado nello stesso stato per ipotesi, quindi applicando z ad entrambe le stringhe finiamo nello stesso stato. ■

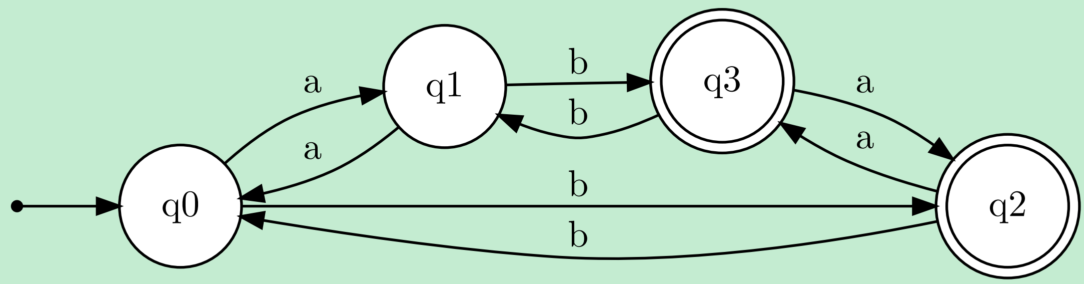
Quante classi di equivalenza abbiamo? Al massimo il numero di stati dell'automa. Come mai diciamo **AL MASSIMO** e non esattamente il numero di stati? Perché in un DFA potremmo avere degli stati che sono irraggiungibili e che quindi non vanno a creare nessuna classe di equivalenza.

In poche parole, R_M è una relazione di equivalenza, invariante a destra e di indice finito limitato dal numero di stati dell'automa M.

Notiamo inoltre che se $(x\, R_M\, y)$ allora x e y sono due stringhe non distinguibili per $L(M)$: infatti, esse vanno nello stato e, aggiungendo qualsiasi stringa $z \in \Sigma^*$ per l'invariante a destra, finisco sempre nello stesso stato. In particolare, se finiamo in uno stato finale accettiamo sia x che y, altrimenti entrambe non sono accettate da M.

Abbiamo appena dimostrato che $L(M)$ è l'**unione** di alcune classi di equivalenza di R_M, ovvero tutte le classi di equivalenza che sono definite da stati finali.

Esempio 10.2.1.1: Dato il seguente automa deterministico, determinare le classi di equivalenza della relazione R_M appena studiata.



Abbiamo 4 classi di equivalenza, che sono tutte le varie combinazioni di a e b pari/dispari.

Questo automa accetta:
- stringhe con a dispari e b dispari;
- stringhe con a pari e b dispari.

Vedremo dopo come migliorare questo automa.

10.2.2. Relazione R_L

Dato un linguaggio $L \subseteq \Sigma^*$, ad esso ci associamo una relazione

$$R_L \subseteq \Sigma^* \times \Sigma^*$$

tale che

$$x\, R_L\, y \Longleftrightarrow \forall z \in \Sigma^* \quad (xz \in L \Longleftrightarrow yz \in L)$$

In poche parole, se a due elementi in relazione attacco una stringa z qualsiasi, allora esse vanno a finire entrambe in stati accettanti oppure no. È praticamente il contrario della distinguibilità.

Lemma 10.2.2.1: La relazione R_L è una relazione di equivalenza.

Dimostrazione 10.2.2.1.1: Facciamo vedere che R_L rispetta RST.

La relazione R_L è riflessiva: banale perché sto valutando la stessa stringa.

La relazione R_L è simmetrica: banale per la simmetria del se e solo se.

La relazione R_L è transitiva: banale per la transitività del se e solo se.

Ma allora R_L è di equivalenza. ■

Lemma 10.2.2.2: La relazione R_L è invariante a destra rispetto all'operazione di concatenazione.

Dimostrazione 10.2.2.2.1: Dobbiamo dimostrare che

$$x \, R_L \, y \Longrightarrow \forall w \in \Sigma^* \quad (xw) \, R_L \, (yw).$$

Se $(x \, R_L \, y)$ allora

$$\forall z \in \Sigma^* \quad (xz \in L \Longleftrightarrow yz \in L).$$

Prendiamo ora una qualsiasi stringa $z \in \Sigma^*$ e aggiungiamola alle due stringhe, ottenendo xwz e ywz. Se chiamiamo $z' = wz$, con un semplice renaming quello che otteniamo è comunque una stringa di Σ^* che mantiene la relazione R_L, ma effettivamente abbiamo aggiunto qualcosa, la stringa z, quindi abbiamo dimostrato che R_L è invariante a destra. ■

Se prendiamo la stringa $z = \varepsilon$, le stringhe x e y che sono nella relazione R_L sono o entrambe dentro o entrambe fuori da L. Ma allora L è l'**unione** di alcune classi di equivalenza di R_L.

Esempio 10.2.2.1: Definiamo il linguaggio

$$L = \{x \in \{a, b\}^* \mid \#_a(x) = \text{dispari}\}.$$

Per questo linguaggio abbiamo due classi di equivalenza rispetto alla relazione R_L: una per le a pari e una per le a dispari.

Non abbiamo ancora parlato di **indice** per R_L. Ci sono linguaggi che hanno un numero di classi di equivalenza infinito: ad esempio il linguaggio

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

ha un numero di classi di equivalenza infinito perché non è un linguaggio di tipo 3.

Se confrontiamo gli ultimi due esempi fatti, notiamo che essi descrivono lo stesso linguaggio, ovvero quello delle stringhe con un numero di a dispari, ma abbiamo due situazioni diverse:
- nel primo esempio la relazione R_M ha 4 classi di equivalenza e il DFA ha 4 stati;
- nel secondo esempio la relazione R_L ha 2 classi di equivalenza e il DFA (non disegnato) ha 2 stati.

Ma allora R_M è un **raffinamento** di R_L. Questa cosa vale solo per questo esempio? **NO**.

Teorema 10.2.2.1 (Teorema di Myhill-Nerode): Sia $L \subseteq \Sigma^*$ un linguaggio.

Le seguenti affermazioni sono equivalenti:
1. L è accettato da un DFA, ovvero L è regolare (lo dobbiamo ancora dimostrare);
2. L è l'unione di alcune classi di equivalenza di una relazione E invariante a destra di indice finito;
3. la relazione R_L associata a L ha indice finito.

Queste relazioni che abbiamo visto fin'ora sono dette **relazioni di Nerode**.

Dimostrazione 10.2.2.1.1: Facciamo vedere $1 \Longrightarrow 2 \Longrightarrow 3 \Longrightarrow 1$.

[$1 \Longrightarrow 2$]

Sia M un DFA per L. Consideriamo la relazione R_M: abbiamo osservato che essa è:
- di equivalenza;
- invariante a destra;
- di indice finito.

Inoltre, rende L unione di alcune classi di equivalenza, quindi è esattamente quello che vogliamo dimostrare.

[$2 \Longrightarrow 3$]

Supponiamo di avere una relazione

$$E \subseteq \Sigma^* \times \Sigma^*$$

di equivalenza, invariante a destra, di indice finito e che L è l'unione di alcune classi di E.

Sia $(x \, E \, y)$. Sappiamo che E è invariante a destra, ovvero vale che

$$\forall z \in \Sigma^* \quad (xz) \, E \, (yz).$$

Inoltre, vale che

$$\forall z \in \Sigma^* \quad (xz \in L \Longleftrightarrow yz \in L)$$

perché L è unione di alcune classi di equivalenza di E.

Ma allora

$$x \, R_L \, y$$

per tutta la catena che abbiamo costruito.

Inoltre, E è un raffinamento di R_L, quindi vuol dire che l'indice di E è maggiore di R_L, ovvero

$$\text{indice}(R_L) \leq \text{indice}(E).$$

Visto che E ha indice finito, anche R_L ha indice finito.

[$3 \Longrightarrow 1$]

Sia R_L di indice finito, costruiamo l'automa M' che deve essere un DFA per L.

Definiamo quindi l'automa $M' = (Q', \Sigma, \delta', q'_0, F')$ tale che:

- Q' insieme degli stati formato delle classi di equivalenza di R_L, ovvero
$$\{[x] \mid x \in \Sigma^*\};$$
- q'_0 stato iniziale che poniamo uguale alla classe di equivalenza che contiene la parola vuota, ovvero
$$q'_0 = [\varepsilon];$$
- δ funzione di transizione tale che
$$\forall \sigma \in \Sigma \quad \delta'([x], \sigma) = [x\sigma];$$
- F insieme degli stati finali formato dalle classi di equivalenza che contengono stringhe del linguaggio, ovvero
$$F' = \{[x] \mid x \in L\}.$$

Ma allora $L(M') = L(M)$ per costruzione. ■

Visto che abbiamo dimostrato questo teorema, possiamo porre E uguale a R_M: otteniamo

$$\text{indice}(R_L) \leq \text{indice}(R_M)$$

se L è una tipo 3, altrimenti partiamo a ∞ con le classi di equivalenza di R_L.

Finiamo con le nozioni di automa minimo.

Con **automa minimo** intendiamo il DFA per L con il minimo numero di stati.

Teorema 10.2.2.2 (Teorema dell'automa minimo): Dato un linguaggio L accettato da automi, il DFA minimo per L è unico. Con unicità intendiamo la non esistenza di una configurazione diversa del grafo.

L'automa minimo contiene anche l'eventuale stato trappola dove mandiamo i pattern non accettanti.

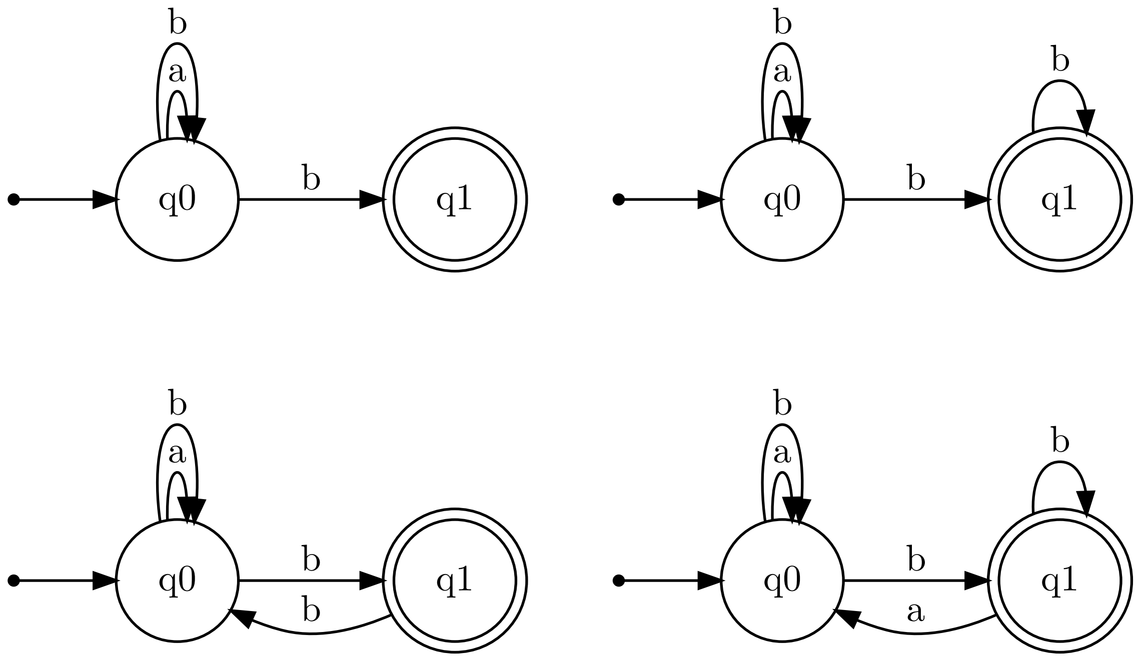
L'automa minimo M' è ottenuto grazie alla relazione R_L.

Per calcolare l'automa minimo abbiamo algoritmi per farlo in modo efficiente, che cercano le stringhe non distinguibili per abbassare il numero di stati. Se troviamo delle stringhe distinguibili siamo arrivati all'automa minimo.

10.2.3. E gli NFA?

Cosa succede se applichiamo tutte questi concetti sugli NFA?

Ad esempio, costruiamo un po' di automi non deterministici per stringhe che finiscono in *b*.



Ovviamente non possiamo andare sotto i 2 stati, almeno un carattere lo dobbiamo leggere, quindi tutti questi sono **automi minimi** ma **non sono unici**.

Inoltre, per i DFA abbiamo algoritmi polinomiali ben studiati negli anni ′60, per gli NFA non abbiamo algoritmi efficienti perché esso è un problema difficile, estremamente difficile, che è ben oltre gli NP-completi, ovvero è un problema PSPACE-completo

Per fare un confronto, un problema NP-completo è CNF-SAT, un problema PSPACE-completo è CNF-SAT con una serie arbitraria di ∀ e ∃ posti davanti alla formula CNF.