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

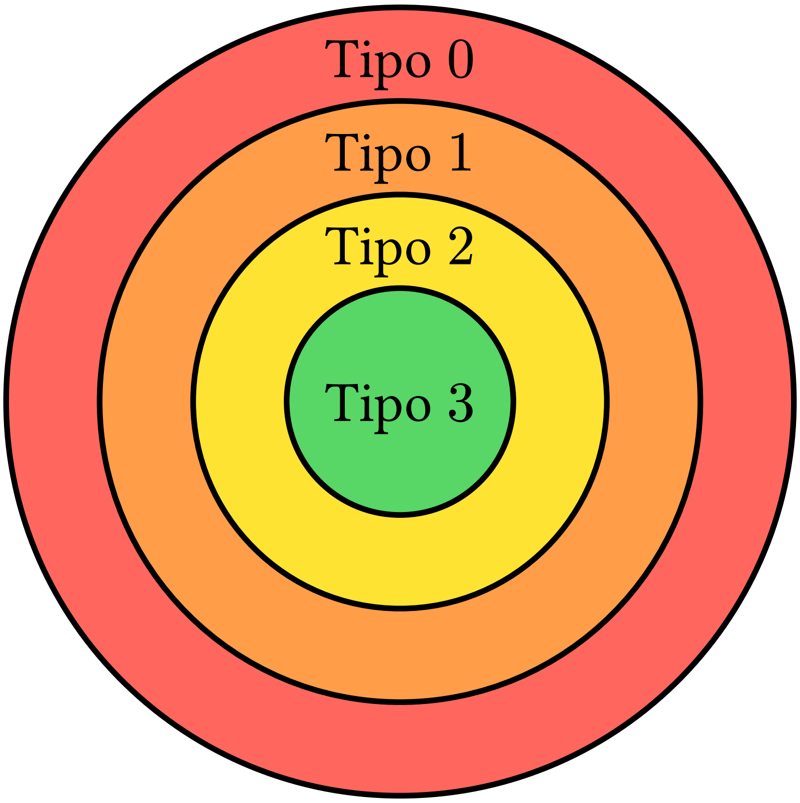
Negli anni $'50$ Noam Chomsky studia la generazione dei linguaggi formali e crea una **gerarchia di grammatiche formali**. La classificazione delle grammatiche viene fatta in base alle regole di produzione che definiscono la grammatica.

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

$$\begin{aligned} S &\longrightarrow \varepsilon \mid [S] \mid SS \mid T \\ T &\longrightarrow \varepsilon \mid (T) \mid TT. \end{aligned}$$

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 $L(G) = \{a^{2^n} \mid n \geq 0\}$.

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

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

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. $\blacksquare$

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 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.

Esempio 5.1.1.1: Sia $A = (Q, \Sigma, \delta, q_0, F)$ tale che:

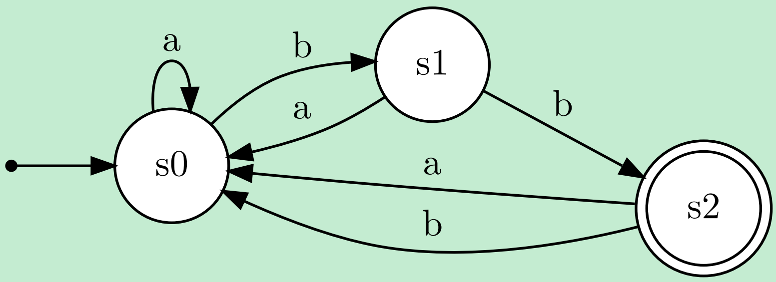
- $Q = \{s_0, s_1, s_2\}$;

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

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

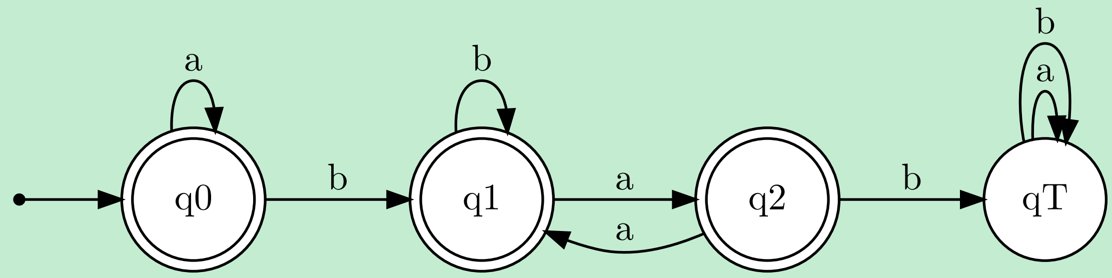
$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}.$$

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.



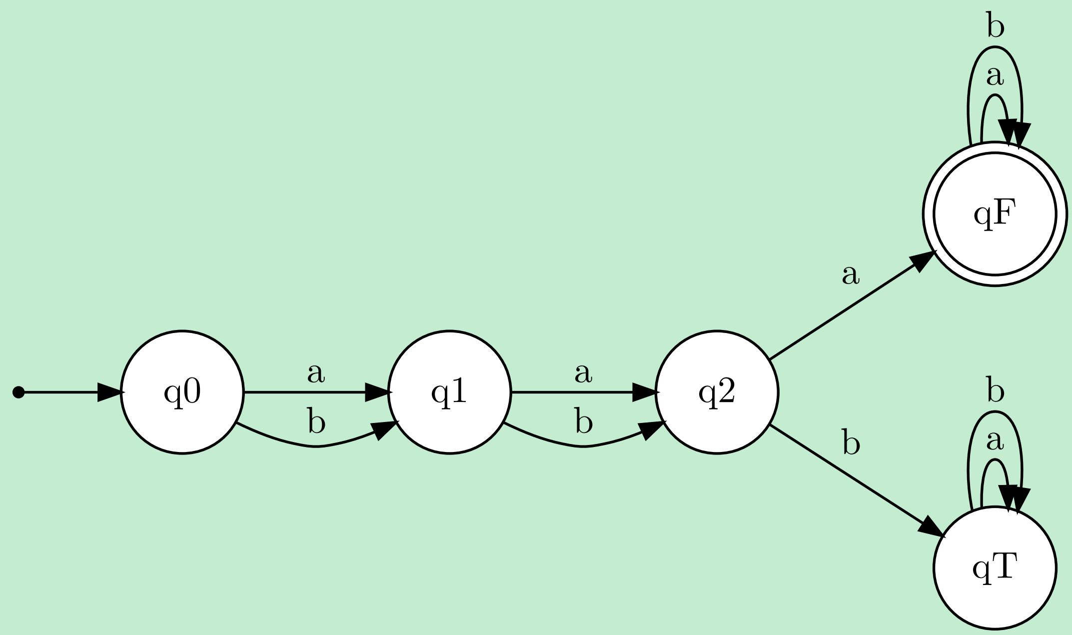
Come vediamo dall'esempio precedente, abbiamo uno stato particolare q_T che è detto **stato trappola**: esso viene utilizzato come «punto di arrivo» per esaurire la lettura dell'input e non accettare la stringa data in input. Finiamo in questo stato se, in uno stato q, leggiamo un carattere che rende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

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

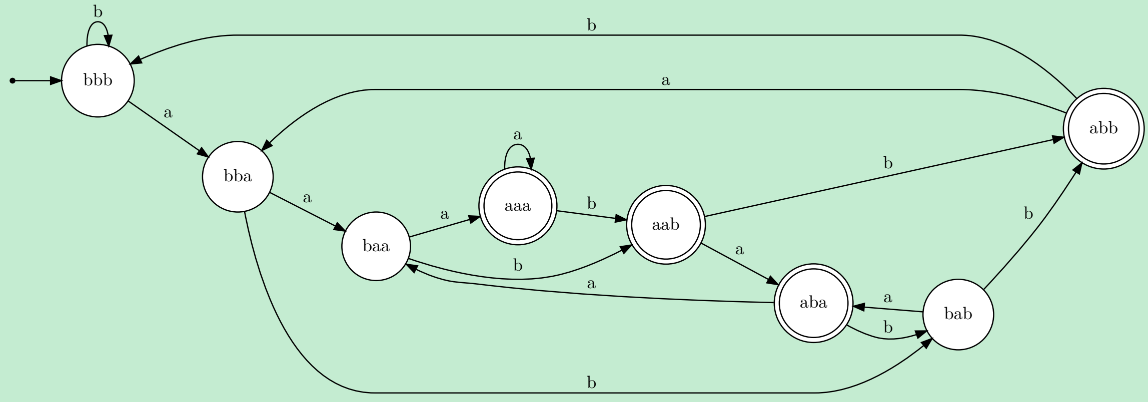
Costruiamo un automa deterministico per L.



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

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

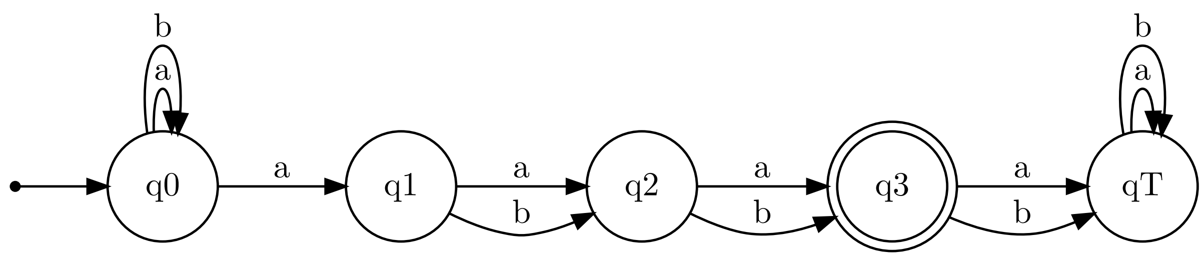
Costruiamo un automa deterministico per L. Qua l'idea è ricordarsi una finestra di 3 simboli e grazie a questa vediamo se il primo carattere che definisce lo stato è una a.



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.

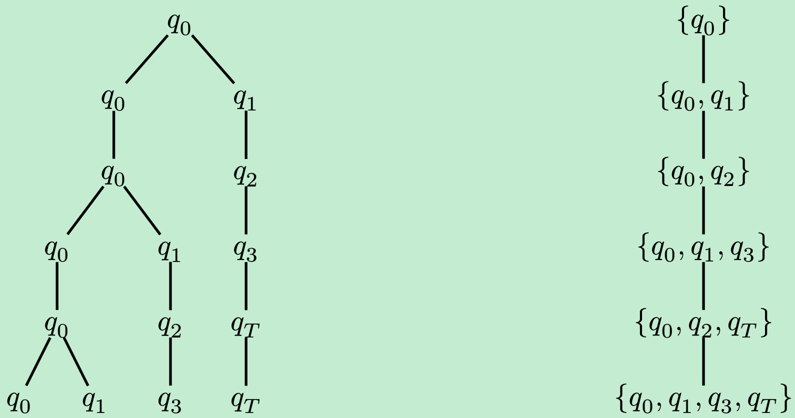
Gli **automi non deterministici**, o **NFA**, sono definiti da una quintupla $A = (Q, \Sigma, \delta, q_0, F)$ definita allo stesso modo dei DFA tranne la funzione di transizione. Essa è la funzione

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

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

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

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

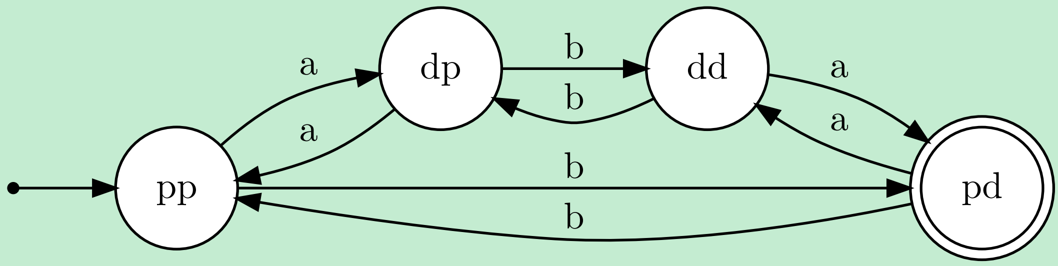
Una ulteriore forma di non determinismo, oltre a quella sulle molteplici transizioni con lo stesso carattere in uno stato, è quella di avere **molteplici stati iniziali**.

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

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

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

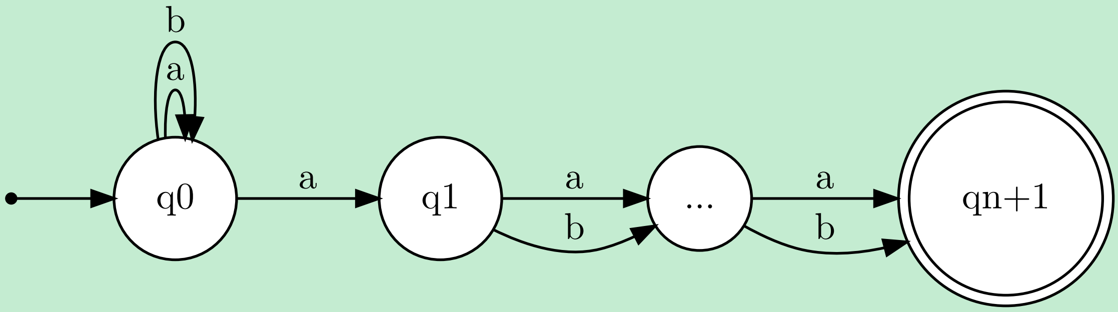
$$z \in \Sigma^{i-1}$$

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.

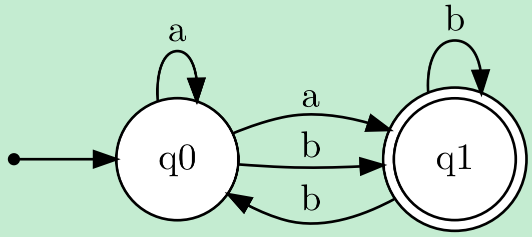


Il numero totale di stati è $n + 1$.

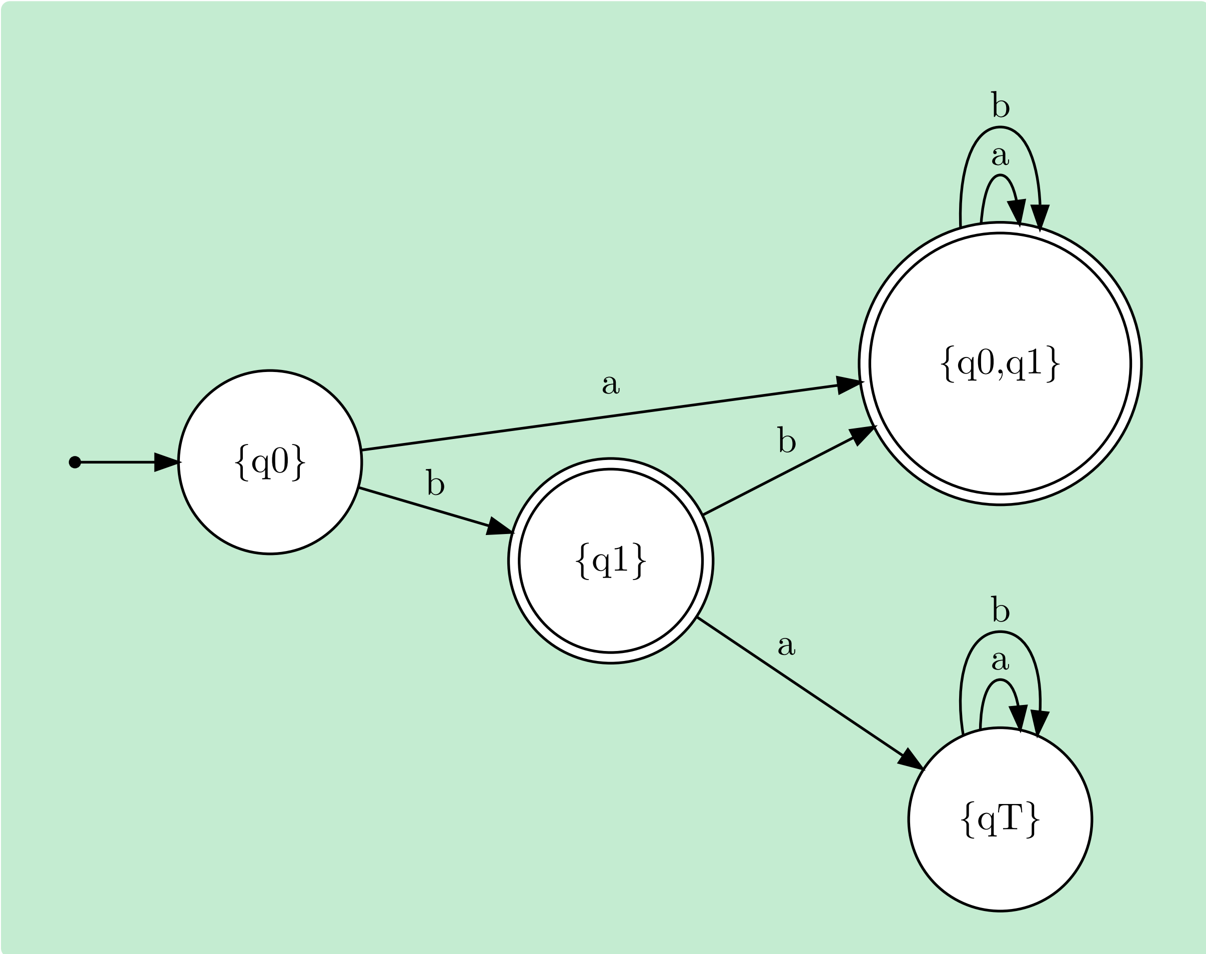
Per L_n abbiamo quindi visto che il numero di stati richiesti per un NFA è $n + 1$, mentre per un DFA è almeno 2^n grazie al teorema sulla distinguibilità. Il salto che abbiamo fatto è quindi **esponenziale**.

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

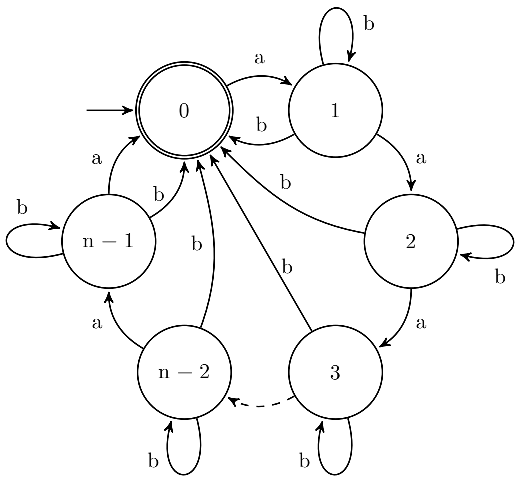
L'**automa di Meyer-Fischer**, ideato da questi due bro nel 1971, sarà il nostro NFA salvatore che ci permetterà di dimostrare quanto detto fino ad adesso.

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

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

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

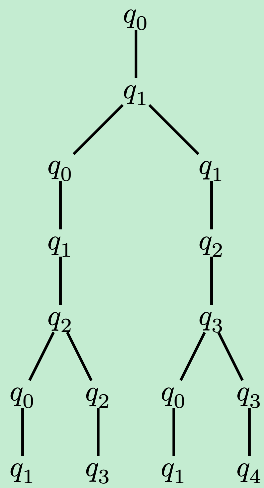
$$\delta(q_0, w_S) = S.$$

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. ■

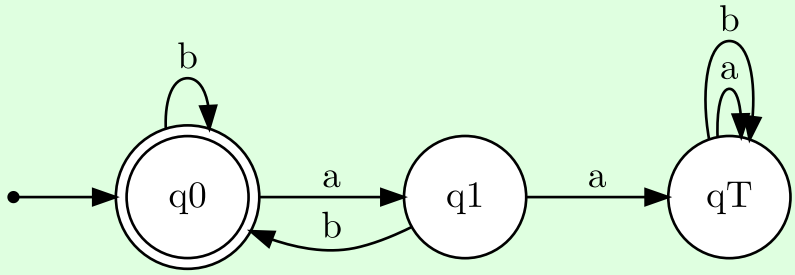
7. Esercizi lezioni 03, 04 e 05 [12/03]

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:



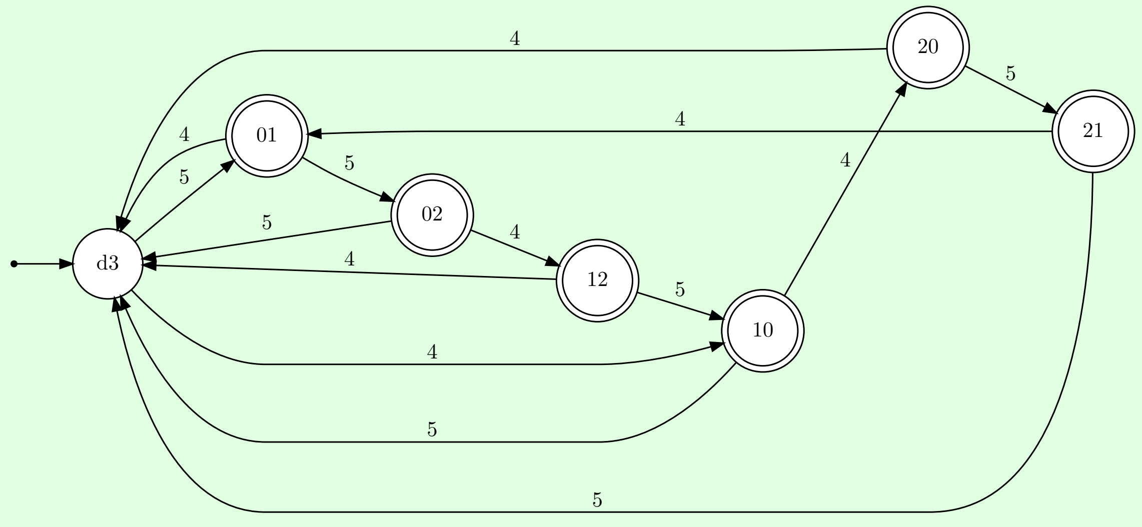
7.2. Esercizio 02

Esercizio 7.2.1:

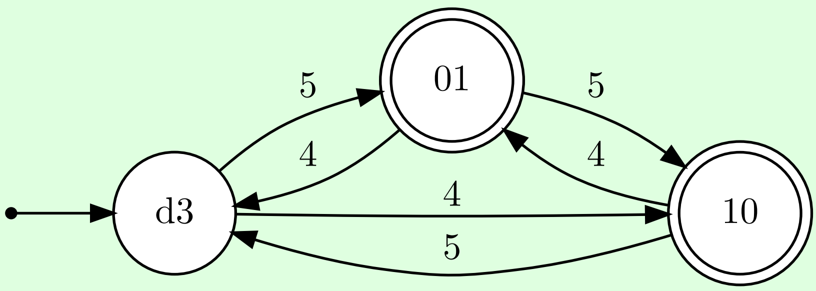
Richiesta 7.2.1.1: Costruite un automa a stati finiti che riconosca il linguaggio formato da tutte le stringhe sull'alfabeto $\{4, 5\}$ che, interpretate come numeri in base 10, rappresentano interi che non sono divisibili per 3.

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

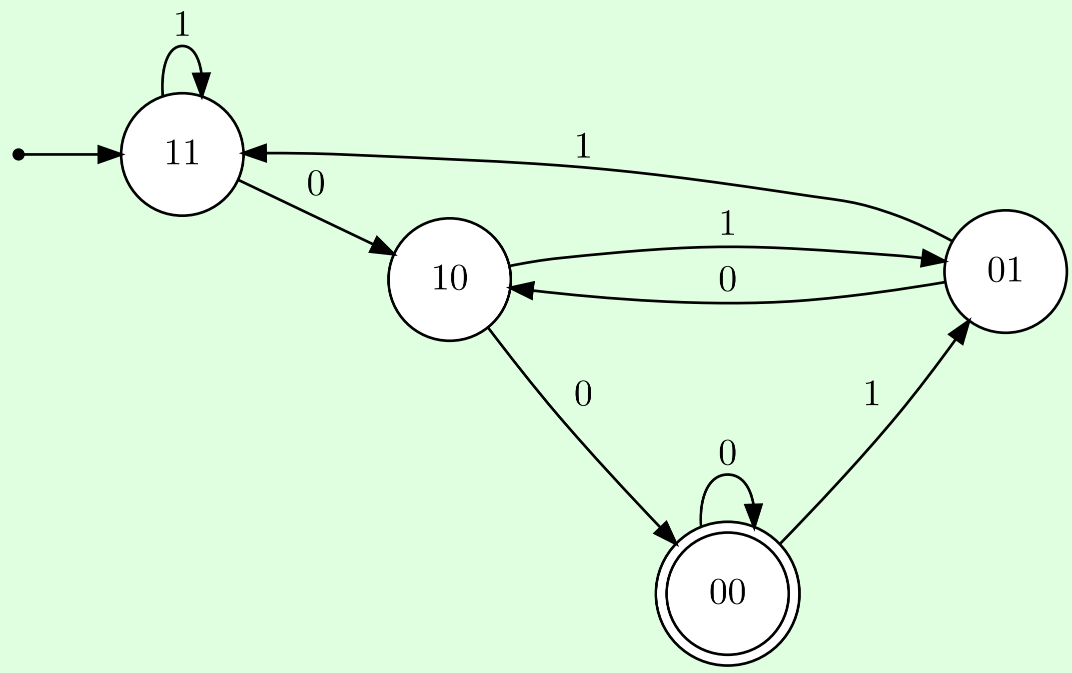


7.3. Esercizio 03

Esercizio 7.3.1:

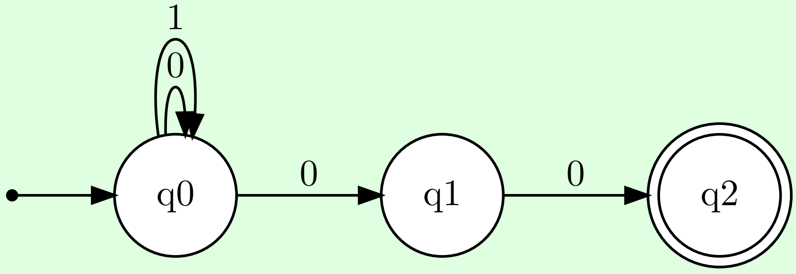
Richiesta 7.3.1.1: Costruite un automa a stati finiti deterministico che riconosca il linguaggio formato da tutte le stringhe sull'alfabeto $\{0, 1\}$ che, interpretate come numeri in notazione binaria, denotano multipli di 4.

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.

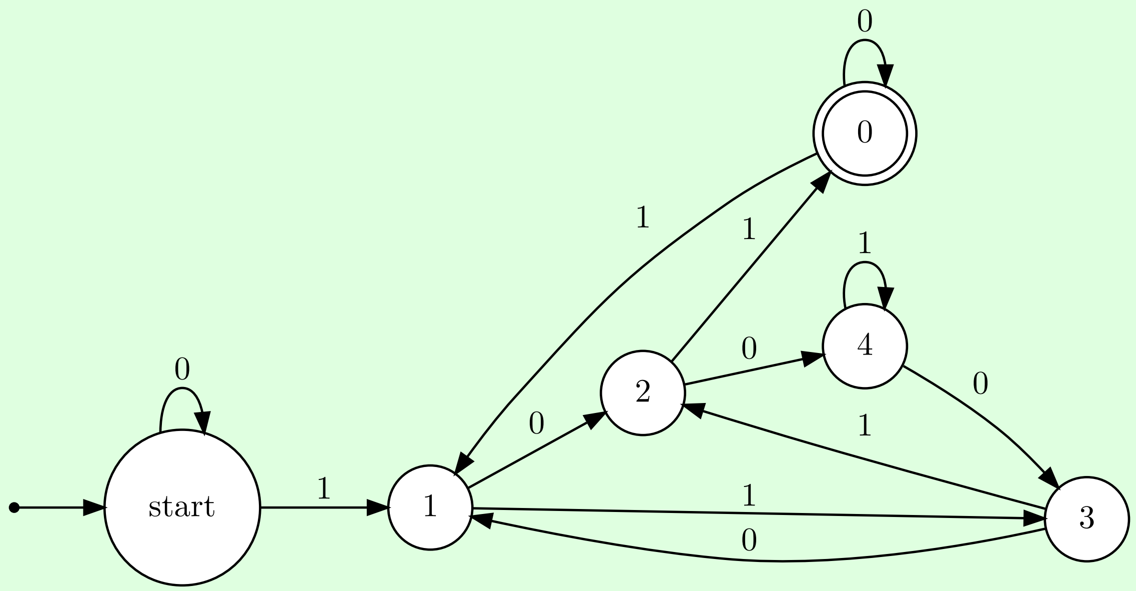
Per gli NFA che riconoscono i multipli di 2^k dobbiamo utilizzare $k + 1$ stati, di cui k leggono gli ultimi k zeri e uno che fa da «stato scommettitore».

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:



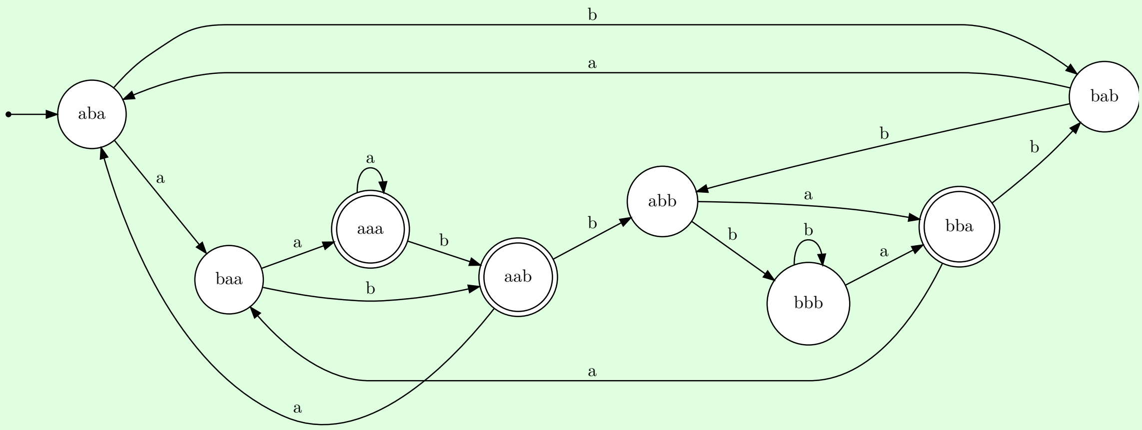
7.5. Esercizio 05

Esercizio 7.5.1: Considerate il seguente linguaggio:

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

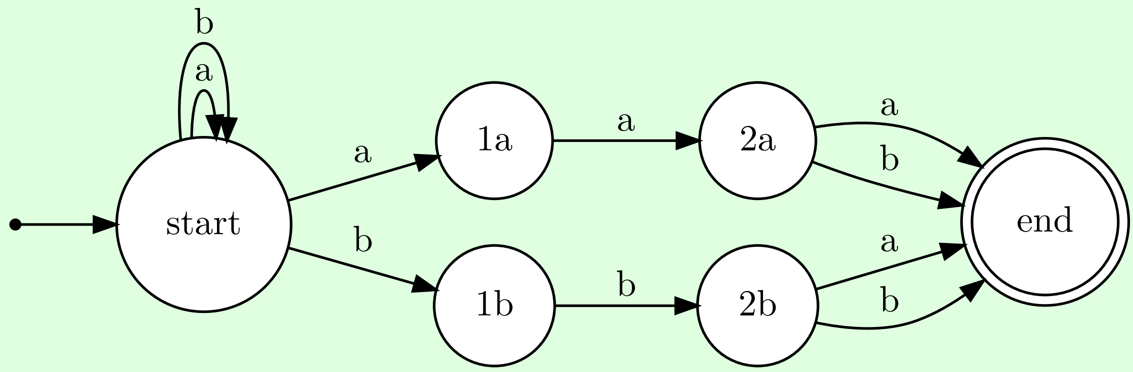
Richiesta 7.5.1.1: Costruite un automa a stati finiti deterministico che accetta L.

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.

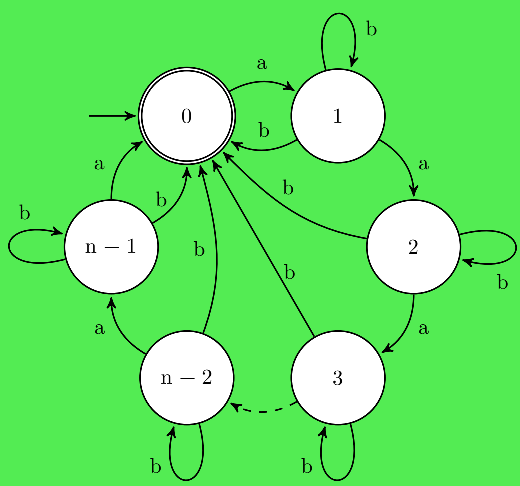
Soluzione 7.6.1.3: $X = \{(ab)^n \mid n \geq 0\}$.

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.

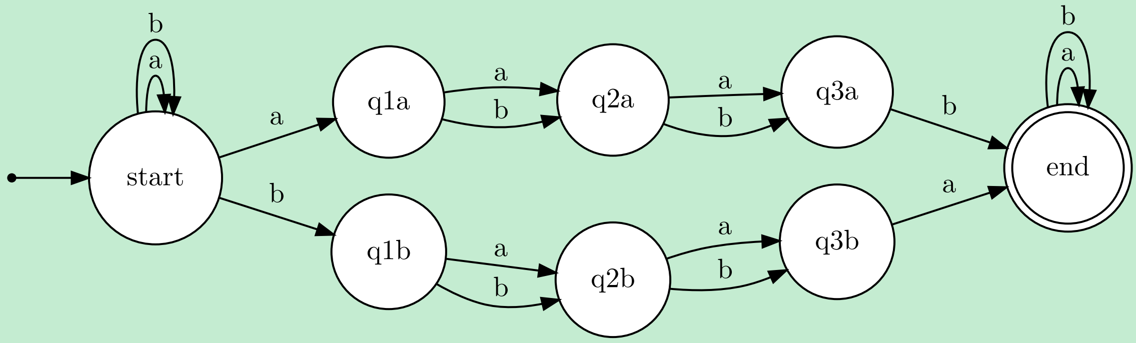
Visto che siamo bravi con le scommesse, andiamo a fare un po' di sano **gambling**.

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

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

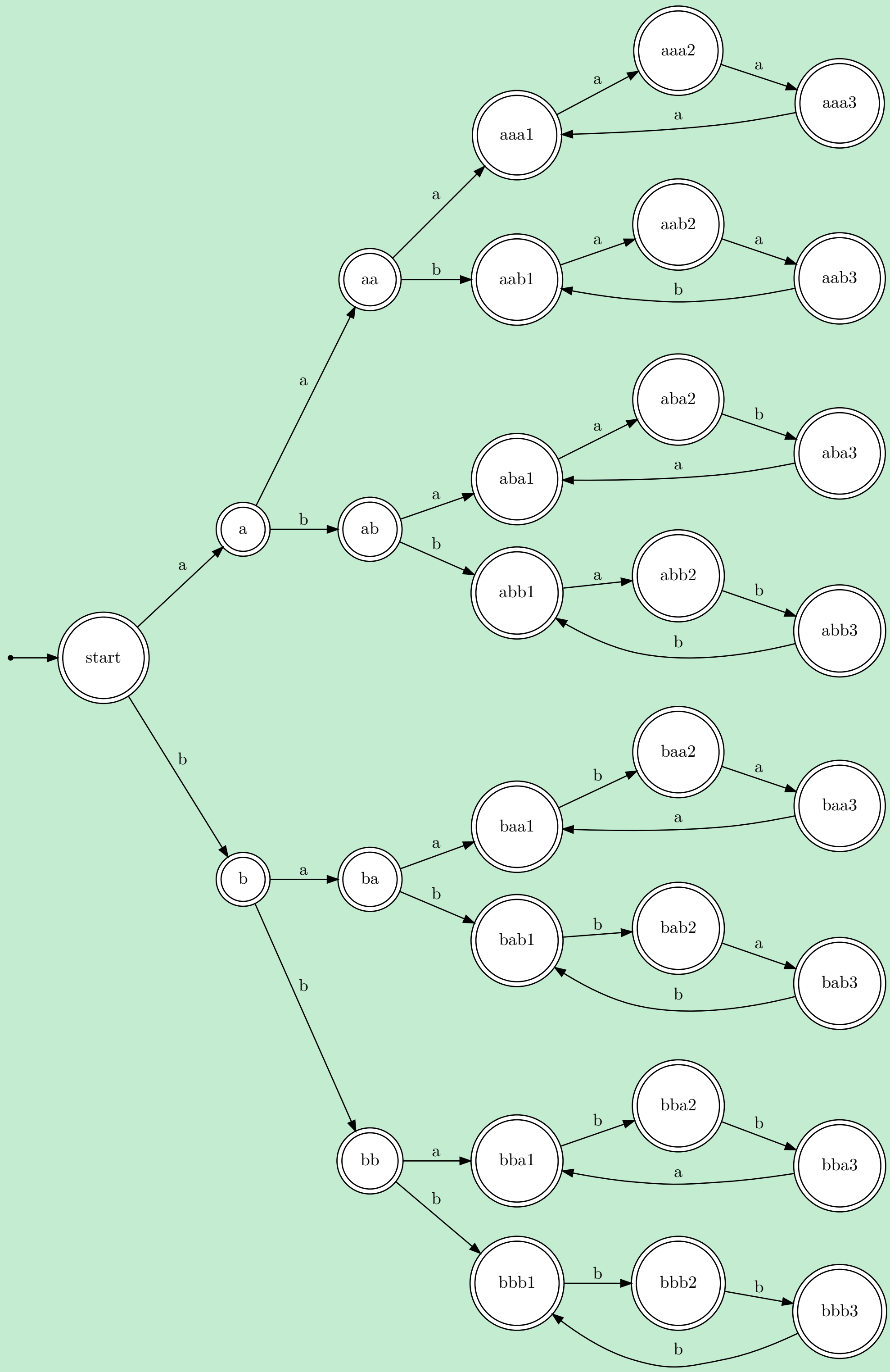
Posso ripetere w quante volte voglio, ma poi la parte finale deve ripetere in parte w.

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'automa.

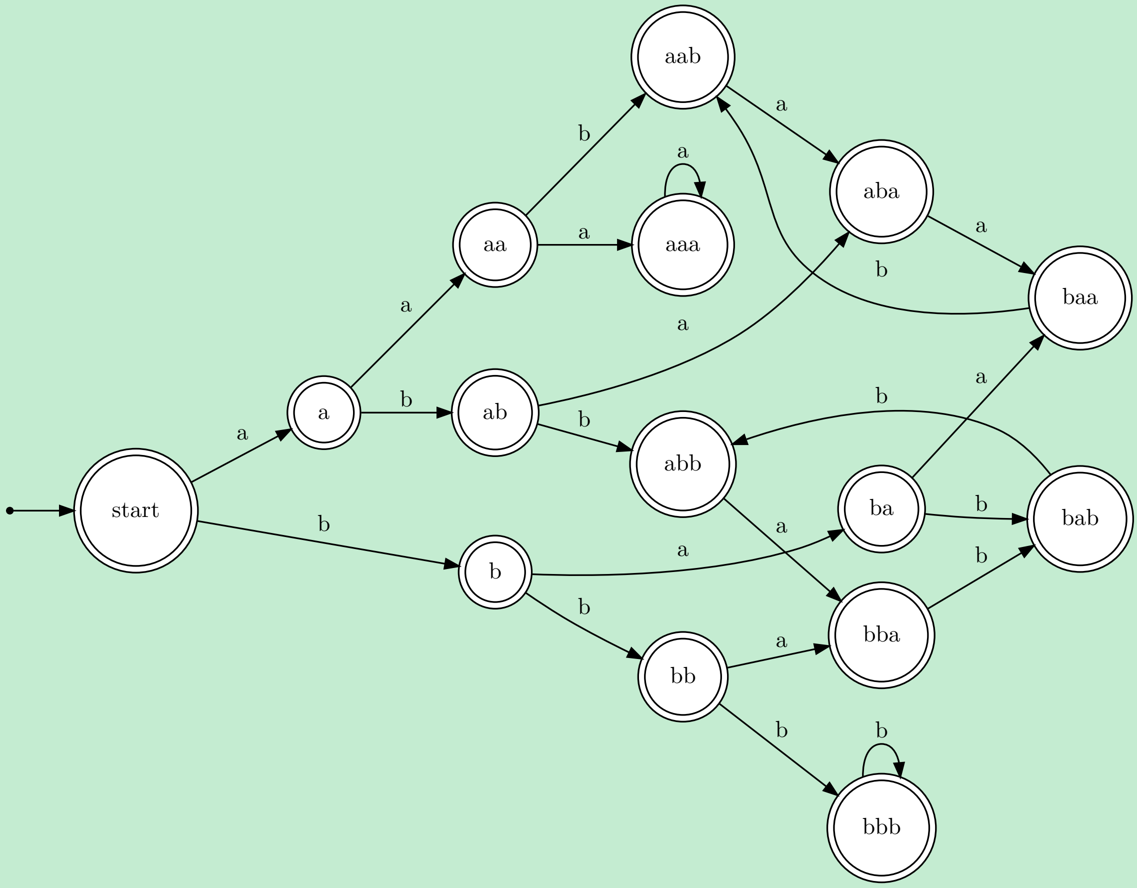


Per il linguaggio generico L'_n, l'albero usa un numero di stati pari a

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

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.

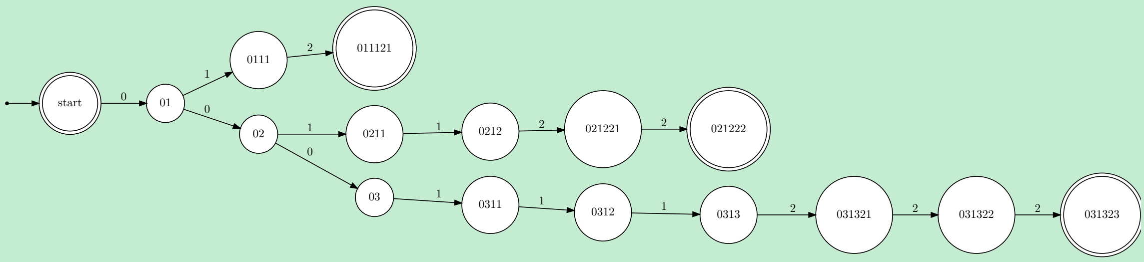
Linguaggio	DFA	NFA
L_n	$\geq 2^n$	$\leq 2n + 2$
L'_n	$\geq 2^n \wedge \leq 2^{n+1}$	$\geq 2^n$

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

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

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.

9. Esercizi lezione 06 [14/03]

9.1. Esercizio 01

Esercizio 9.1.1 : Considerate il linguaggio

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

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

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

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.

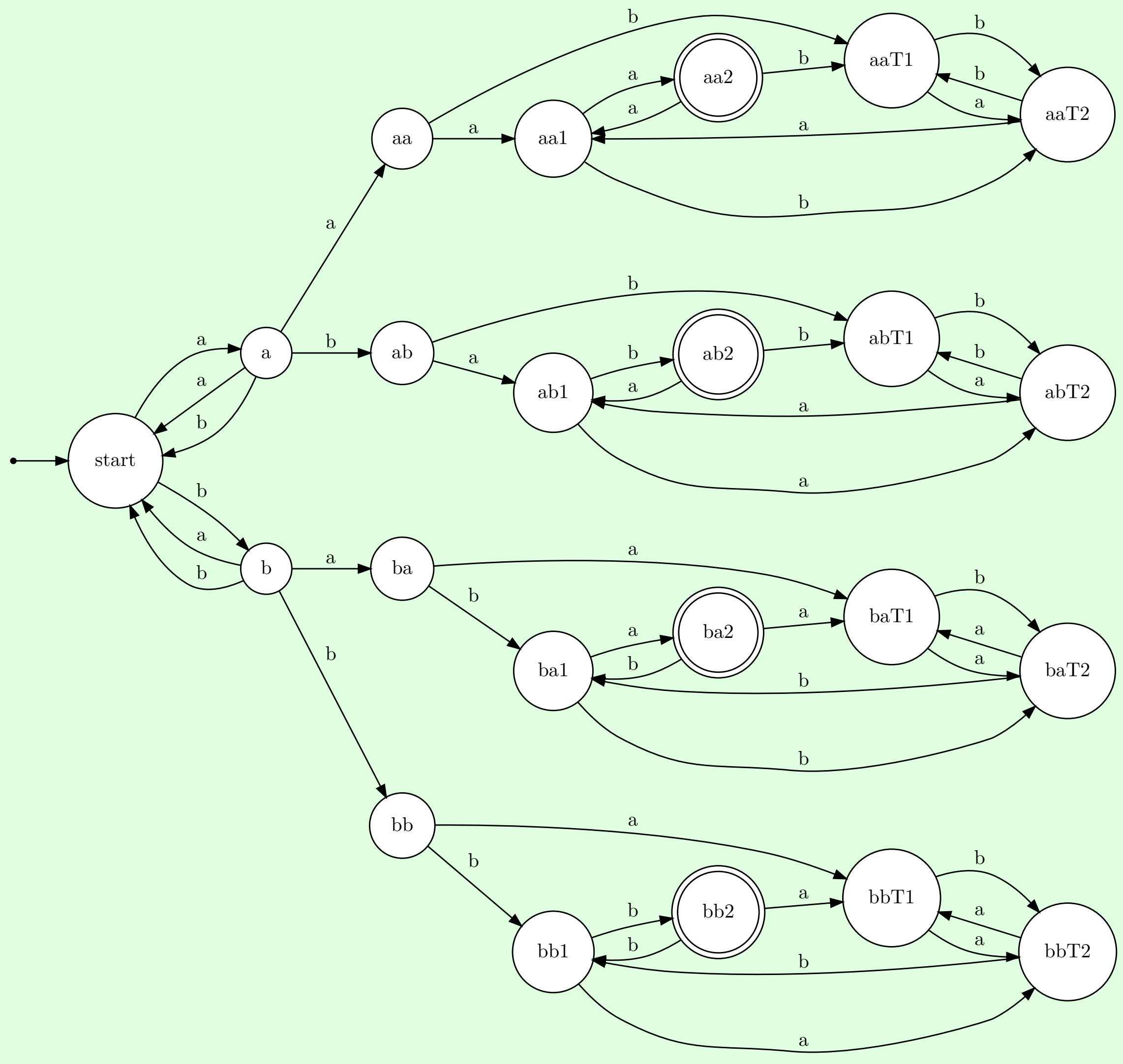
La cardinalità di P è 2^k, quindi ogni NFA per K_k ha almeno 2^k stati.

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

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

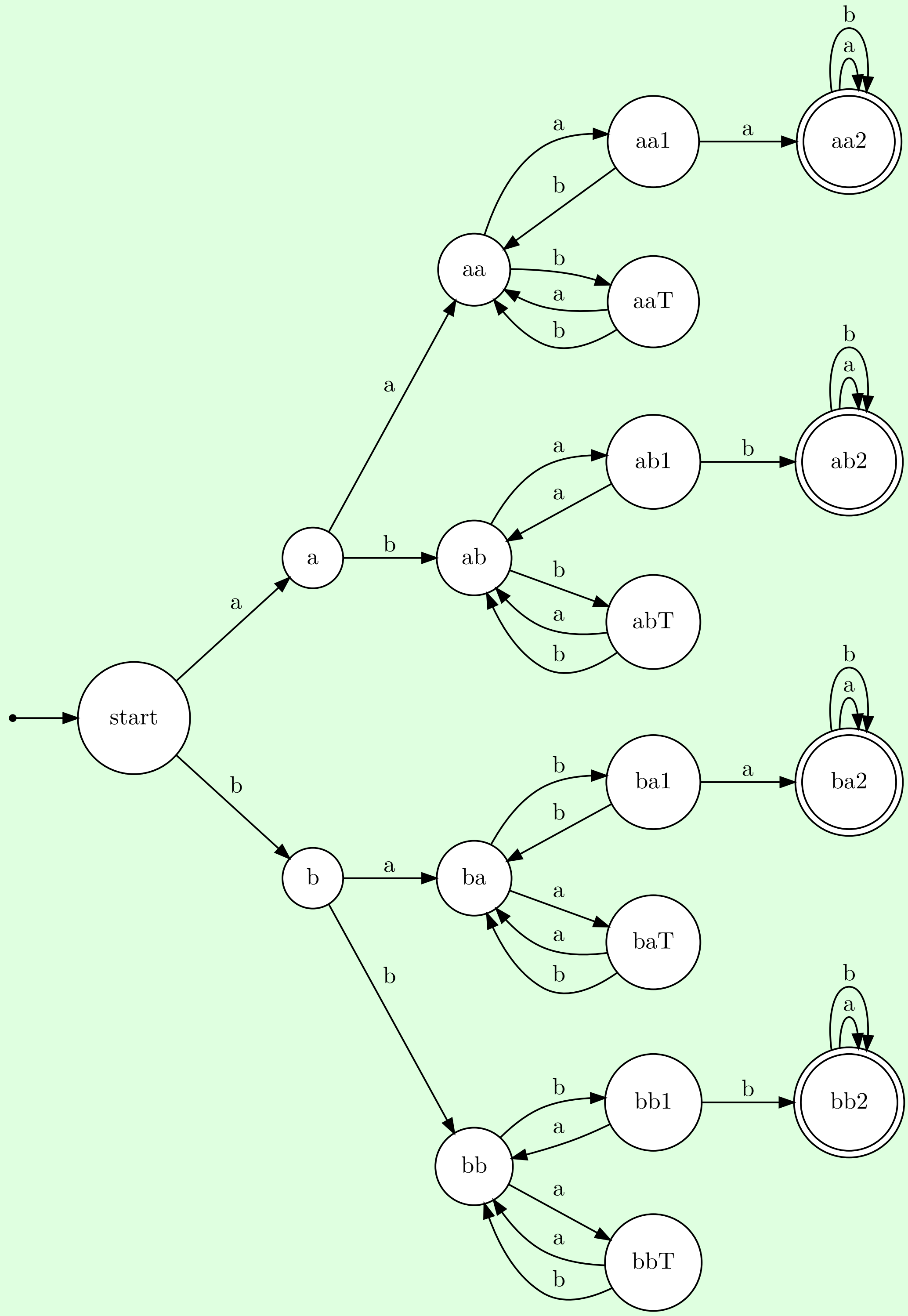
dove k è un intero fissato. Si può osservare che ogni stringa w di questo linguaggio è la concatenazione di blocchi di lunghezza k, in cui il primo blocco coincide con uno dei blocchi successivi; ogni stringa di J_k si ottiene «rovesciando» una stringa del linguaggio K_k dell'esercizio 3.

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** $\forall x, y, z \in S \quad x \, R \, y \wedge y \, R \, z \Longrightarrow x \, R \, z$.

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. ■

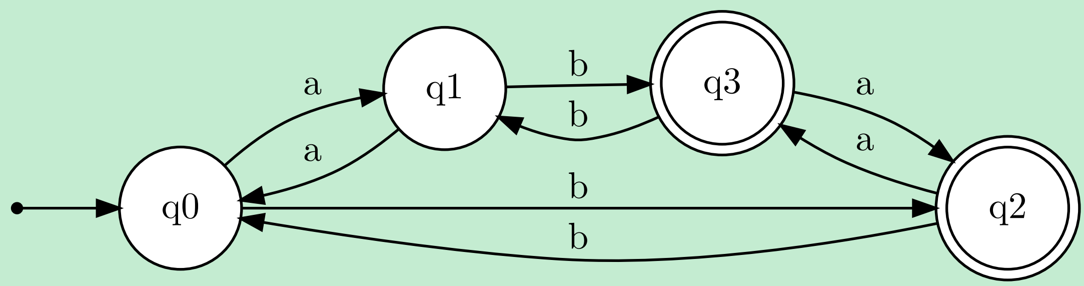
Quante classi di equivalenza abbiamo? Al massimo il numero di stati dell'automa. Come mai diciamo **AL MASSIMO** e non esattamente il numero di stati? Perché in un DFA potremmo avere degli stati che sono irraggiungibili e che quindi non vanno a creare nessuna classe di equivalenza.

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.

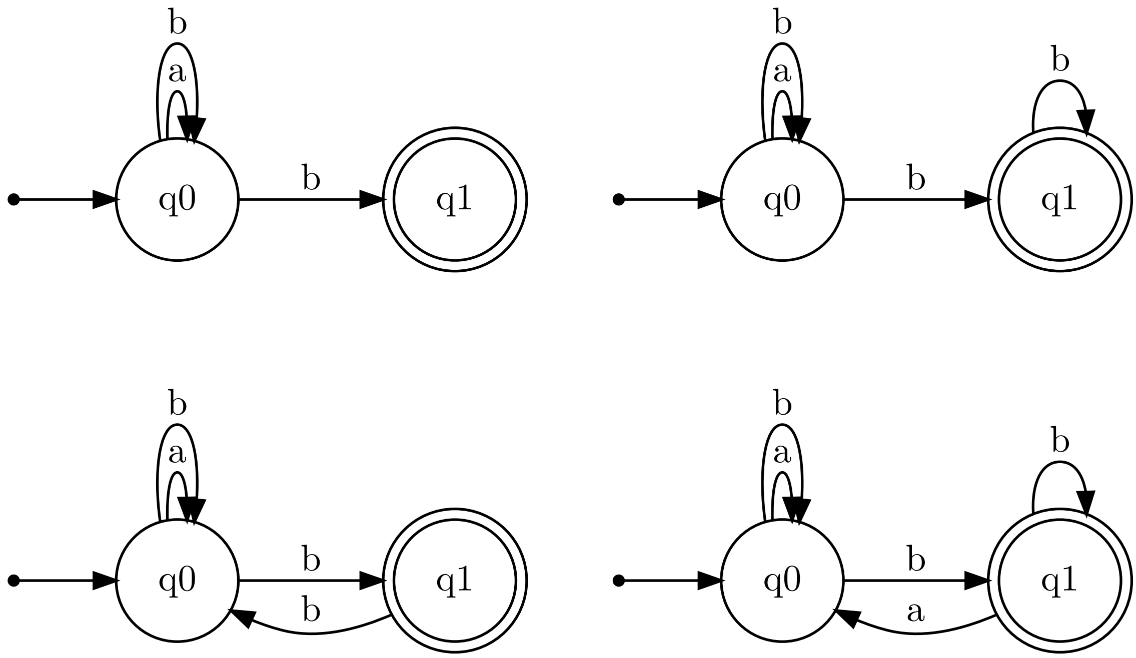
L'automa minimo M' è ottenuto grazie alla relazione R_L.

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 $\forall$ e $\exists$ posti davanti alla formula CNF.

11. Esercizi lezione 07 [19/03]

11.1. Esercizio 01

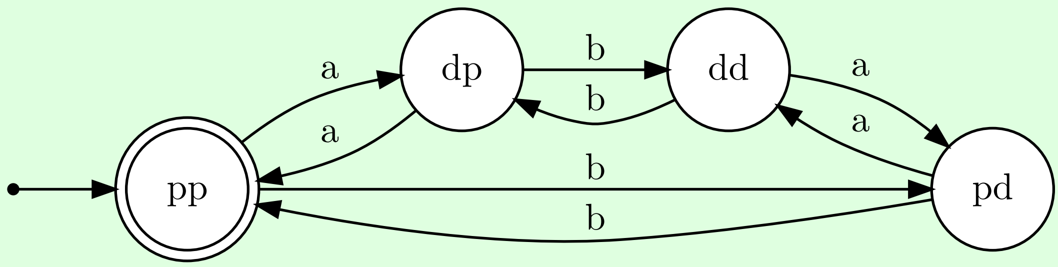
Esercizio 11.1.1: Per tutti i linguaggi di questo esercizio l'alfabeto è $\{a, b\}$.

Richiesta 11.1.1.1: Considerate l'insieme L formato dalle stringhe in cui sia il numero di a che il numero di b sono pari.

Richieste:
- Determinate le classi di equivalenza della relazione R di Myhill-Nerode associata a L.
- Costruite l'automa deterministico minimo corrispondente.
- Determinate un insieme X di cardinalità massima che contenga stringhe tutte distinguibili tra loro.
- La relazione cambia nel caso si chieda che le stringhe del linguaggio abbiano un numero di a pari, un numero di b pari ed entrambi siano positivi? E l'automa?

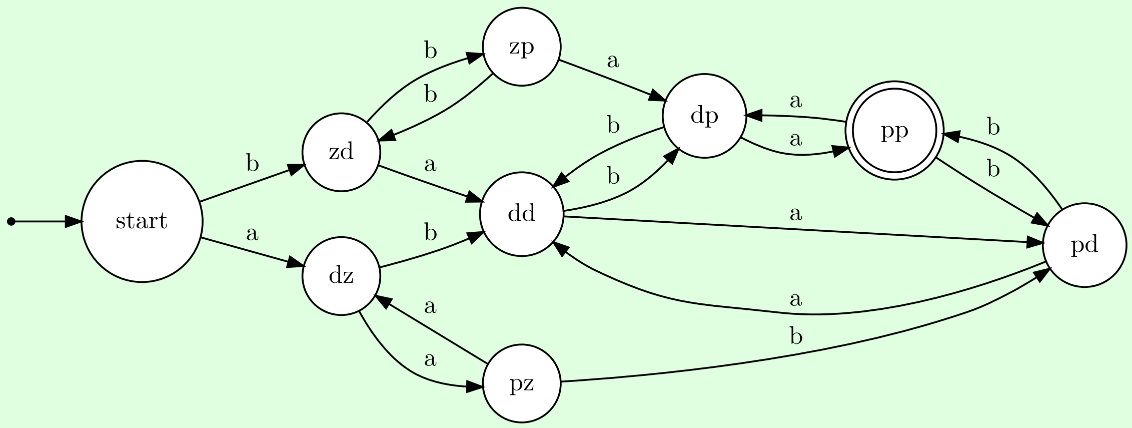
Soluzione 11.1.1.1: Le classi di equivalenza di R sono 4:
- $[pp]_R$ per a e b pari;
- $[dd]_R$ per a e b dispari;
- $[pd]_R$ per a pari e b dispari;
- $[dp]_R$ per a dispari e b pari.

L'automa minimo per L è il seguente.



L'insieme $X = \{\varepsilon, a, b, ab\}$ è un insieme di stringhe distinguibili, non voglio dimostrarlo.

Richiedendo a e b pari in numero positivo la relazione sale a 9 classi di equivalenza, e quindi anche l'automa passa ad avere 9 stati.



Richiesta 11.1.1.2: Considerate ora il linguaggio L' formato dalle stringhe in cui il numero di a sia pari e il numero di b sia dispari.

Richieste:

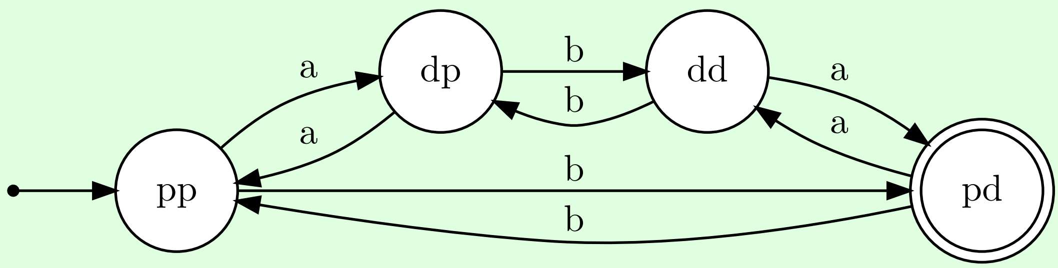
- Verificate che la relazione R' di Myhill-Nerode associata a L' coincide con la relazione R associata a L.
- Cosa hanno in comune e cosa hanno di diverso i rispettivi automi deterministici minimi.

Soluzione 11.1.1.2: Le classi di equivalenza di R' sono 4:

- $[pp]_R$ per a e b pari;
- $[dd]_R$ per a e b dispari;
- $[pd]_R$ per a pari e b dispari;
- $[dp]_R$ per a dispari e b pari.

Sono le stesse classi di equivalenza della relazione R della richiesta precedente.

L'automa minimo per L' è lo stesso identico a quello per L (la prima versione) tranne lo stato finale, che nei due automi è differente.

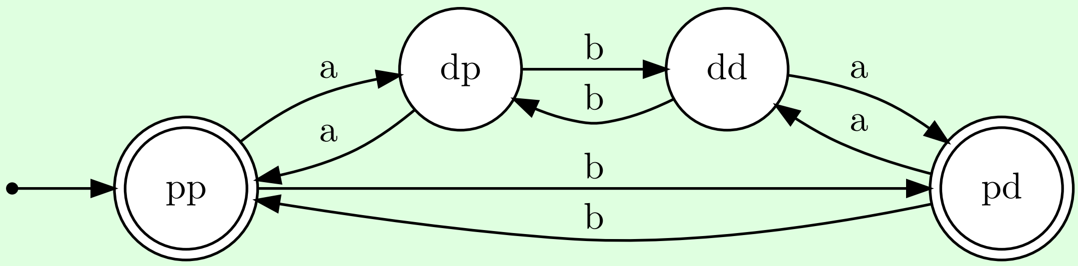


Richiesta 11.1.1.3: Considerate ora il linguaggio L'' formato dalle stringhe in cui il numero di a sia pari e il numero di b sia qualunque.

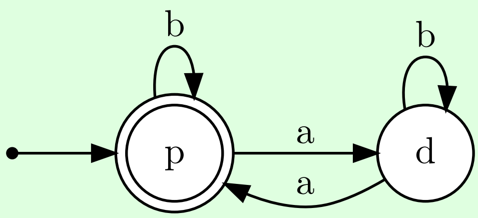
Richieste:

- Verificate che L'' è l'unione di alcune classi di equivalenza della relazione R precedente.
- Costruite un automa che accetti L'' i cui stati corrispondano alle classi di R.
- Determinate le classi di equivalenza della relazione R'' di Myhill-Nerode associata a L''.
- Costruite l'automa deterministico minimo corrispondente.
- Che legame c'è tra R e R'' e tra i due automi ottenuti?

Soluzione 11.1.1.3: Possiamo definire L'' come l'unione delle classi di equivalenza $[pp]_R$ e $[pd]_R$, che vanno contenere tutte le stringhe che hanno un numero di a pari.



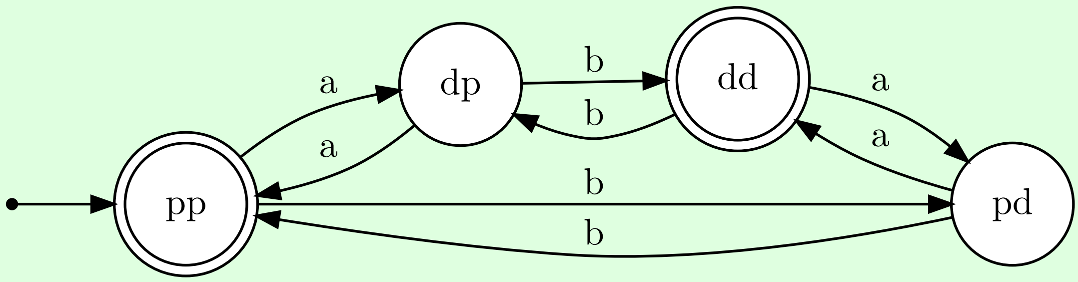
Le classi di equivalenza della relazione R'' sono $[p]_{R''}$ e $[d]_{R''}$, che contengono rispettivamente le stringhe con un numero di a pari e le stringhe con un numero di a dispari.



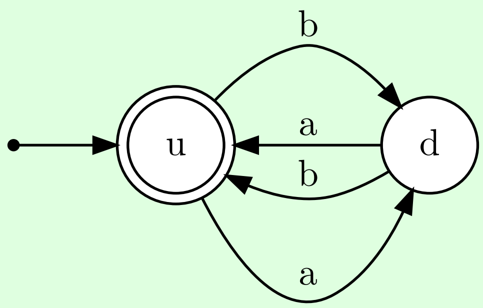
Possiamo dire che R è un raffinamento di R'', e quindi che l'automa di R'' è più compatto e conciso di quello di R, che infatti ha il doppio degli stati.

Richiesta 11.1.1.4: Considerate ora il linguaggio L''' formato dalle stringhe in cui il numero di a e il numero di b siano entrambi pari o entrambi dispari. Rispondete alle domande del punto c precedente, considerando L''' e la rispettiva relazione R''' di Myhill-Nerode al posto di L'' e R''.

Soluzione 11.1.1.4: Possiamo definire L''' come l'unione delle classi di equivalenza $[pp]_R$ e $[dd]_R$, che vanno contenere tutte le stringhe che hanno un numero di a e b entrambi pari o entrambi dispari.



Le classi di equivalenza della relazione R''' sono $[u]_{R''}$ e $[d]_{R''}$, che contengono rispettivamente le stringhe con un numero di a e b entrambi pari o entrambi dispari e le stringhe con un numero di a dispari/pari e di b pari/dispari.



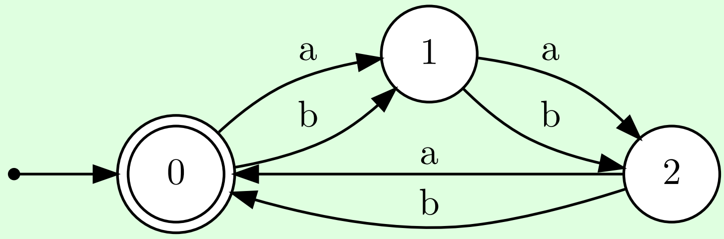
Possiamo dire che R è un raffinamento di R''', e quindi che l'automa di R''' è più compatto e conciso di quello di R, che infatti ha il doppio degli stati.

11.2. Esercizio 02

Esercizio 11.2.1: Calcolate le classi d'equivalenza della relazione di Myhill-Nerode associata a ciascuno dei seguenti linguaggi e, nel caso la relazione sia di indice finito, costruite l'automa deterministico minimo corrispondente.

Richiesta 11.2.1.1: $L = \{w \in \{a,b\}^* \mid \text{la somma del numero di } a \text{ e del numero di } b \text{ è multiplo di } 3\}$.

Soluzione 11.2.1.1: Le classi di equivalenza della relazione R_L sono 3:
- $[0]_{R_L}$ per $a + b$ multiplo di 3;
- $[1]_{R_L}$ per $a + b$ multiplo di $3 + 1$;
- $[2]_{R_L}$ per $a + b$ multiplo di $3 + 2$.



Richiesta 11.2.1.2: $J = \{a^n b^n \mid n \geq 0\}$.

Soluzione 11.2.1.2: Il linguaggio J non è regolare, quindi non siamo sicuri che l'indice della relazione R_J sia finito. E infatti non lo è: le classi di equivalenza sono nella forma

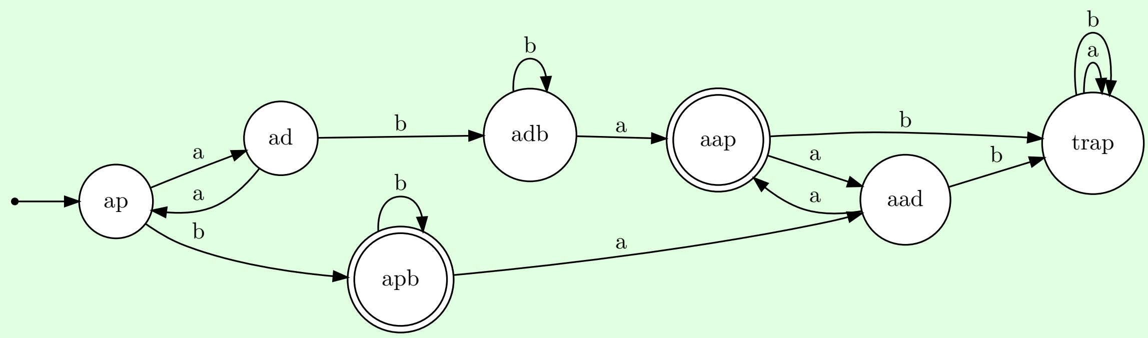
$$[a^n]_{R_J} \mid n \geq 0$$

ma il numero di queste classi è infinito, quindi l'indice di R_J è infinito.

Richiesta 11.2.1.3: $K = \left\{a^i b^n a^j \mid n > 0 \wedge i + j \text{ è pari}\right\}$.

Soluzione 11.2.1.3: Visto che sono furbo, prima ho costruito l'automa minimo e ora scrivo quali sono le classi di equivalenza, urlo del sium.

Le classi di equivalenza della relazione R_L quindi sono 7, che riprendono i nomi degli stati dell'automa successivo, molto facili.



Sono sicuro che questo sia l'automa minimo, ho controllato con l'algoritmo di Hopcroft.

12. Lezione 08 [21/03]

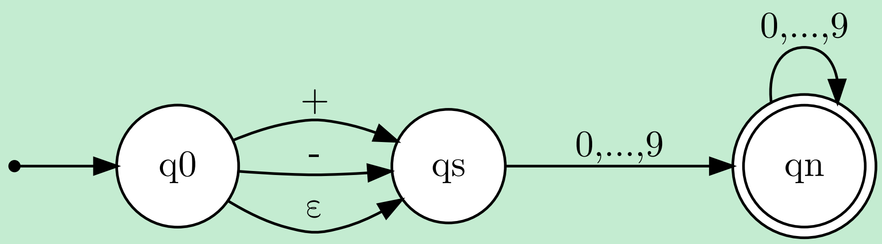
12.1. Altre forme di non determinismo

Nell'ambito degli automi non deterministici avevamo visto delle forme di non determinismo sulle transizioni e sulla scelta dello stato iniziale. Entrambe queste rappresentazioni non rendevano più potenti gli automi: tramite **costruzione per sottoinsiemi** riuscivamo a costruire un DFA analogo con un numero di stati con gap esponenziale.

Un'altra forma di non determinismo che possiamo avere è l'uso delle **ε-produzioni**: 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 12.1.1: Se $\Sigma = \{0, ..., 9, +, -\}$ 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.

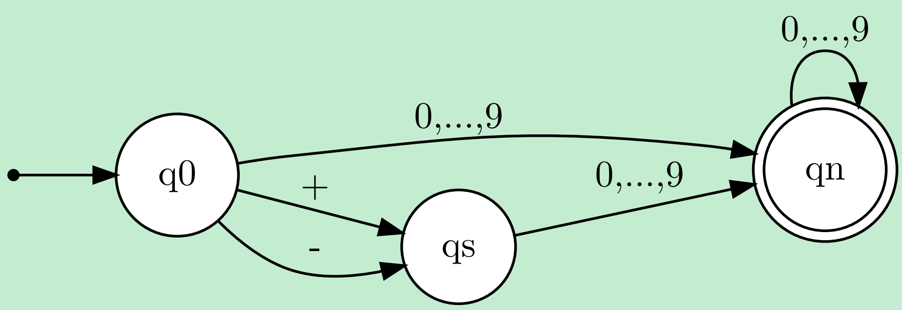
Questa soluzione aumenta la potenza dell'automa? **NO**: ogni sequenza nella forma

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

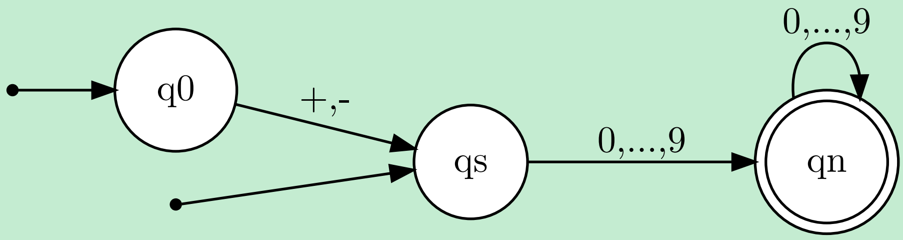
può essere tradotta nella transizione

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

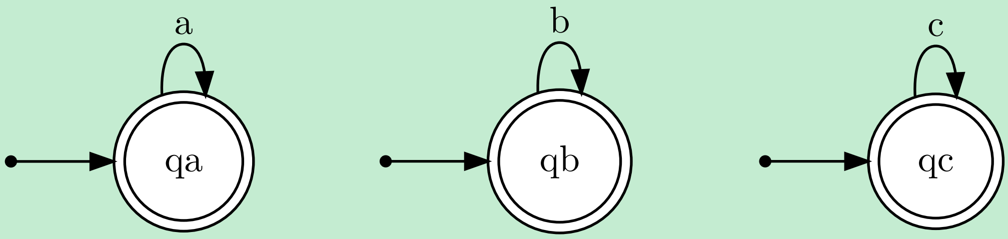
Esempio 12.1.2: Andiamo a rimuovere la ε-transizione usando le sequenze appena descritte.



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.

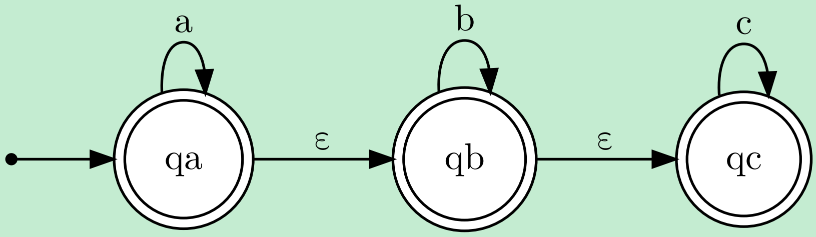


Esempio 12.1.3: Ci vengono dati tre automi, che riconoscono sequenze di a, b e c arbitrarie.

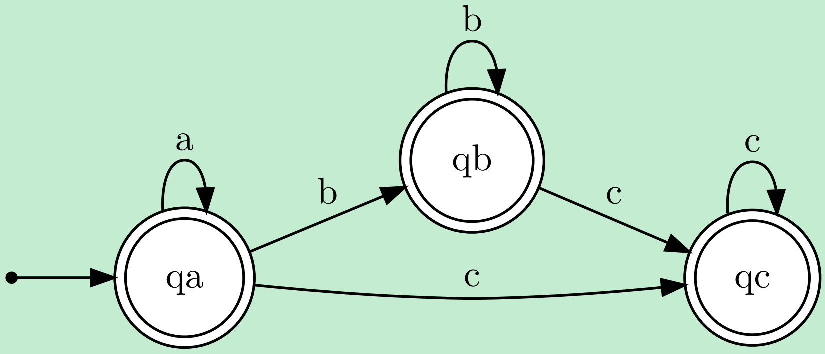


Vogliamo costruire un automa che utilizzi le ε-transizioni usando questi tre moduli per riconoscere il linguaggio

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

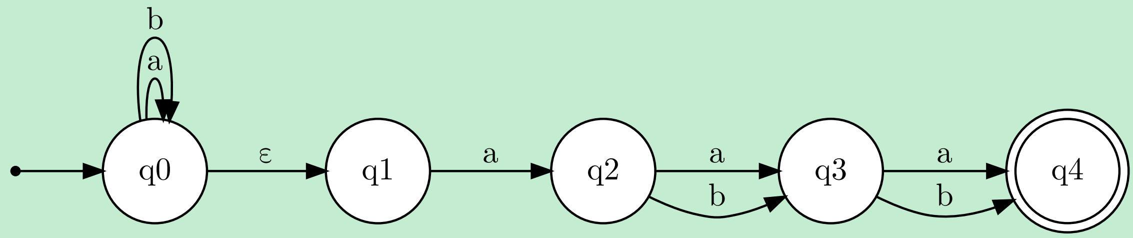


Come lo rendiamo deterministico? Sicuramente non andiamo ad utilizzare gli stati iniziali multipli, che qui ci starebbero molto bene, ma appunto vogliamo un comportamento deterministico.



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

Esempio 12.1.4: Riprendiamo il linguaggio L_n delle stringhe con l'n-esimo carattere da destra uguale ad una a. Avevamo visto un NFA sulle transizioni, vediamone 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 $n + 2$.

12.2. Relazione tra i linguaggi e le grammatiche di tipo 3

Nella lezione precedente abbiamo «dato per buono» il fatto che le grammatiche di tipo 3 sono equivalenti agli automi a stati finiti.

Riprendiamo velocemente la definizione di grammatica di tipo 3: essa è una quadrupla

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

con produzioni nella forma

$$A \longrightarrow aB \mid a \quad \text{tali che} \quad a \in \Sigma \land A, B \in V.$$

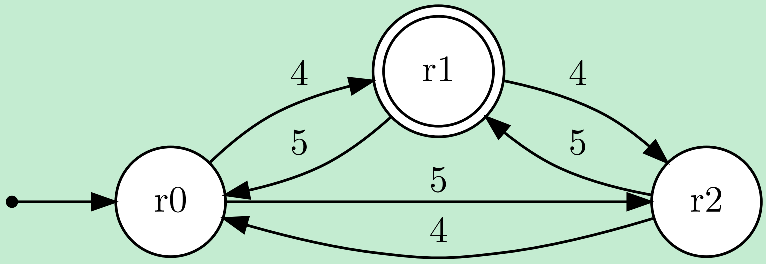
12.2.1. Dall'automa alla grammatica

Dato un automa $A = (Q, \Sigma, \delta, q_0, F)$ per il linguaggio L, costruiamo una grammatica G di tipo 3 che riconosca lo stesso linguaggio L.

Per fare ciò dobbiamo definire le variabili, l'assioma e le produzioni. Definiamo quindi G tale che:
- le **variabili** sono gli stati dell'automa, ovvero $V = Q$;
- l'**assioma** è lo stato iniziale dell'automa, ovvero $S = q_0$;

- le **produzioni** derivano dalle transizioni e sono nella forma:
  - $q \longrightarrow ap$ se la funzione di transizione è tale che $\delta(q,a) = p$;
  - in più alla produzione precedente aggiungiamo la produzione $q \longrightarrow a$ se p è uno stato finale, questo perché posso fermarmi in p.

Esempio 12.2.1.1: Sia $\Sigma = \{4, 5\}$. Ci viene fornito un automa che, date le stringhe sull'alfabeto Σ interpretate come numeri decimali, una volta divise per 3 ci danno 1 come resto.



Costruiamo una grammatica G di tipo 3 analoga a questo automa. Sia quindi G tale che:
- variabili $V = \{r_0, r_1, r_2\}$;
- assioma $S = r_0$;
- produzioni P:
  - $r_0 \longrightarrow 4r_1 \mid 4 \mid 5r_2$;
  - $r_1 \longrightarrow 4r_2 \mid 5r_0$;
  - $r_2 \longrightarrow 4r_0 \mid 5r_1 \mid 5$.

Proviamo a derivare una stringa per vedere se effettivamente funziona:

$$r_0 \longrightarrow 4r_1 \longrightarrow 45r_0 \longrightarrow 455r_2 \longrightarrow 4555.$$

12.2.2. Dalla grammatica all'automa

In maniera analoga, data la grammatica G di tipo 3 creiamo un automa A tale che:
- **stati** $Q = V \cup \{q_F\}$;
- **stato iniziale** $q_0 = S$;
- **stato finali** $F = \{q_F\}$;
- **transizioni** della funzione di transizioni derivano dalle regole di produzione, ovvero:
  - per ogni produzione $(A \longrightarrow aB) \in P$ essa ci dice che dallo stato A leggendo una a andiamo a finire in B, ovvero $\delta(A, a) = B$;
  - per ogni produzione $(A \longrightarrow a) \in P$ essa ci dice che possiamo finire la derivazione, cioè che andiamo da A in uno stato finale tramite a, ovvero $\delta(A, a) = q_F$.

Per essere più precisi, definiamo i passi della funzione di transizione come

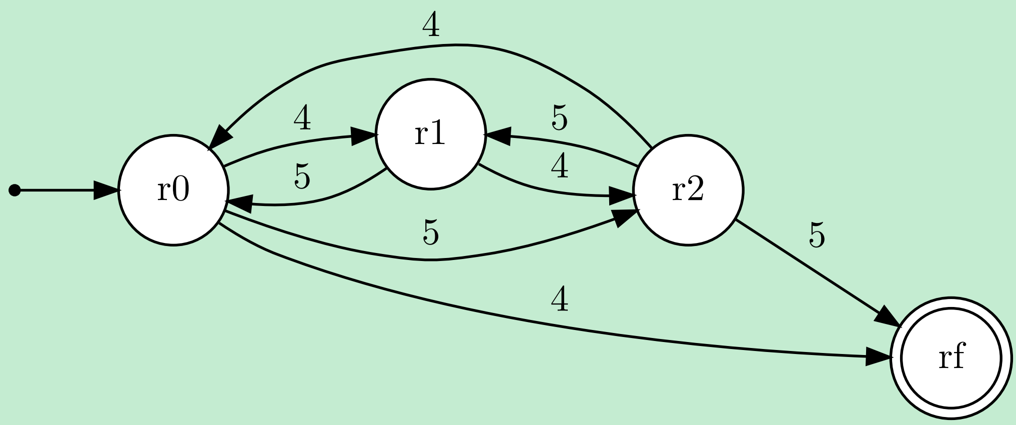
$$\delta(A, a) = \{B \mid (A \longrightarrow aB) \in P\} \cup \{q_F \text{ se } (A \longrightarrow a) \in P\}$$

Esempio 12.2.2.1: Data la grammatica $G = (V, \Sigma, P, S)$ tale che:
- $V = \{r_0, r_1, r_2\}$;
- $S = r_0$;

- produzioni P:
  - $r_0 \longrightarrow 4r_1 \mid 4 \mid 5r_2$;
  - $r_1 \longrightarrow 4r_2 \mid 5r_0$;
  - $r_2 \longrightarrow 4r_0 \mid 5r_1 \mid 5$.

Ricaviamo un automa dalla grammatica G. Per fare ciò definiamo:

- $Q = \{r_0, r_1, r_2, r_f\}$;
- $q_0 = r_0$;
- $F = \{r_f\}$;
- funzione di transizione δ che ha il seguente comportamento:
  - $\delta(r_0, 4) = \{r_1, r_f\}$;
  - $\delta(r_0, 5) = \{r_2\}$;
  - $\delta(r_1, 4) = \{r_2\}$;
  - $\delta(r_1, 5) = \{r_0\}$;
  - $\delta(r_2, 4) = \{r_0\}$;
  - $\delta(r_2, 5) = \{r_1, r_f\}$.



Notiamo come l'automa ottenuto sia non deterministico e, soprattutto, non è l'automa minimo che avevamo invece nell'esempio precedente.

12.3. Grammatiche lineari

12.3.1. Lineari a destra

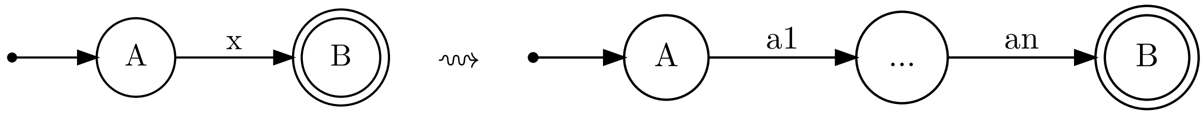
Diamo una mini introduzione alle **grammatiche lineari**, che però vedremo meglio più avanti.

Potrebbero capitarci delle grammatiche che hanno una forma simile a quelle regolari, ma che in realtà non lo sono. Queste grammatiche hanno le produzioni nella forma

$$A \longrightarrow xB \mid x \quad x \in \Sigma^*.$$

Non abbiamo più, come nelle grammatiche regolari, la stringa x formata da un solo terminale, ma possiamo averne un numero arbitrario.

Queste grammatiche sono dette **grammatiche lineari a destra**, ma nonostante questa aggiunta di terminali non aumentiamo la potenza del linguaggio: per generare quella sequenza di terminali x basta aggiungere una serie di regole che rispettano le grammatiche regolari che generino esattamente la stringa x.



Nell'esempio precedente, abbiamo sostituito la stringa $x = a_1...a_n$ con una serie di stati intermedi.

Se $x = \varepsilon$ basta mettere una ε-mossa, tutto molto facile (anche se non ho capito).

12.3.2. Lineari a sinistra

Esistono anche le **grammatiche lineari a sinistra**, che hanno le produzioni nella forma

$$A \longrightarrow Bx \mid x \quad x \in \Sigma^*.$$

Si dimostra che anche queste grammatiche non vanno oltre i linguaggi regolari, anche se è un accrocchio passare da queste grammatiche a quelle regolari.

12.3.3. Lineari

E se facciamo un mischione delle due grammatiche precedenti?

Le produzione di queste grammatiche sono nella forma

$$A \longrightarrow xB \mid Bx \mid x \quad \text{tali che} \quad x \in \Sigma^* \land A, B \in V.$$

Queste grammatiche, che generano i cosiddetti **linguaggi lineari**, sono a cavallo tra le grammatiche di tipo 3 e le grammatiche di tipo 2. Quindi siamo un pelo più forti delle grammatiche regolari, ma non quanto le grammatiche CF.

Esempio 12.3.3.1: Definiamo una grammatica che utilizza le seguenti produzioni:

$$S \longrightarrow aA \mid \varepsilon$$
$$A \longrightarrow Sb.$$

Con queste regole di una grammatica lineare stiamo generando il linguaggio

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

che non è un linguaggio di tipo 3.

La cosa che stiamo aggiungendo è una sorta di **ricorsione**, che mi permette di saltare fuori dai linguaggi regolari e catturare di più di prima.

12.4. Operazioni sui linguaggi

Supponiamo di avere in mano una serie di linguaggi. Vediamo una serie di operazioni che possiamo fare su essi per combinarli assieme e ottenere altri linguaggi.

12.4.1. Operazioni insiemistiche

I linguaggi sono insiemi di stringhe, quindi perché non iniziare dalle **operazioni insiemistiche**?

Fissiamo un alfabeto Σ, siano $L', L'' \subseteq \Sigma^*$ due linguaggi definiti sullo stesso alfabeto Σ. Se i due alfabeti sono differenti allora si considera come alfabeto l'unione dei due alfabeti.

Partiamo con l'operazione di **unione**:

$$L' \cup L'' = \{x \in \Sigma^* \mid x \in L' \vee x \in L''\}.$$

Continuiamo con l'operazione di **intersezione**:

$$L' \cap L'' = \{x \in \Sigma^* \mid x \in L' \wedge x \in L''\}.$$

Terminiamo con l'operazione di **complemento**:

$$L'^C = \{x \in \Sigma^* \mid x \notin L'\}.$$

12.4.2. Operazioni tipiche

Passiamo alle **operazioni tipiche** dei linguaggi, definite comunque molto semplicemente.

Partiamo con l'operazione di **prodotto** (o concatenazione):

$$L' \cdot L'' = \{w \in \Sigma^* \mid \exists x \in L' \wedge \exists y \in L'' \mid w = xy\}.$$

In poche parole, concateniamo in tutti i modi possibili le stringhe del primo linguaggio con le stringhe del secondo linguaggio. Questa operazione, in generale, è **non commutativa**, e lo è se Σ è formato da una sola lettera.

Esempio 12.4.2.1: Vediamo due casi particolari e importanti di prodotto

$$L' \cdot \emptyset = \emptyset \cdot L' = \emptyset$$
$$L' \cdot \{\varepsilon\} = \{\varepsilon\} \cdot L' = L'.$$

Andiamo avanti con l'operazione di **potenza**:

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

In poche parole, stiamo prendendo k stringhe da L' e le stiamo concatenando in ogni modo possibile. Possiamo dare anche una definizione induttiva di questa operazione, ovvero

$$L^k = \begin{cases} \{\varepsilon\} & \text{se } k = 0 \\ L^{k-1} \cdot L & \text{se } k > 0 \end{cases}.$$

Infine, terminiamo con l'operazione di **chiusura di Kleene**, detta anche **STAR**. Questa operazione è estremamente simile alla potenza, ma in questo caso il numero k non è fissato e quindi questa operazione di potenza viene ripetuta all'infinito. Vengono quindi concatenate un numero arbitrario di stringhe di L, e teniamo tutte le computazioni intermedie, ovvero

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

Ecco perché scriviamo Σ^*: partendo dall'alfabeto Σ andiamo ad ottenere ogni stringa possibile.

Esiste anche la **chiusura positiva**, definita come

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

Che relazione abbiamo tra le due chiusure? Questo dipende da ε, ovvero:

- se $\varepsilon \in L$ allora $L^* = L^+$ perché $L^1 \subseteq L^+$ e visto che $\varepsilon \in L^1$ abbiamo gli stessi insiemi;
- se $\varepsilon \notin L$ allora $L^+ = L^* / \{\varepsilon\}$ perché l'unico modo di ottenere ε sarebbe con L^0.

Esempio 12.4.2.2: Vediamo una cosa simpatica:

$$\emptyset^* = \{\varepsilon\}.$$

Abbiamo appena generato qualcosa dal nulla, fuori di testa. La generazione si blocca con la chiusura positiva, ovvero

$$\emptyset^+ = \emptyset.$$

Infine, vediamo una cosa abbastanza banale sull'insieme formato dalla sola ε, ovvero

$$\{\varepsilon\}^* = \{\varepsilon\}^+ = \{\varepsilon\}.$$

Con la chiusura di Kleene, partendo da un **linguaggio finito** L, otteniamo una chiusura L^* di cardinalità infinita, perché ogni volta andiamo a creare delle nuove stringhe.

Partendo invece da un **linguaggio infinito** L, otteniamo ancora una chiusura L^* di cardinalità infinita ma ci sono alcune situazioni particolari.

Esempio 12.4.2.3: Vediamo tre linguaggi infiniti che hanno comportamenti diversi.

Consideriamo il linguaggio

$$L_1 = \{a^n \mid n \geq 0\}.$$

Calcolando la chiusura L_1^* otteniamo lo stesso linguaggio L_1 perché stiamo concatenando stringhe che contengono solo a, che erano già presenti in L_1.

Consideriamo ora il linguaggio

$$L_2 = \left\{a^{2k+1} \mid k \geq 0\right\}.$$

Calcolando la chiusura L_2^* otteniamo il linguaggio L_1 perché:
- in L_2^1 ho tutte le stringhe formate da a di lunghezza dispari;
- in L_2^2 ho tutte le stringhe formate da a di lunghezza pari (dispari + dispari).

Già solo con $L_2^0 \cup L_2^1 \cup L_2^2$ generiamo tutto il linguaggio L_1

Consideriamo infine

$$L_3 = \{a^n b \mid n \geq 0\}.$$

Proviamo a calcolare prima la potenza L_3^k di questo linguaggio, ovvero

$$L_3^k = \{a^{n_1} b ... a^{n_k} b \mid n_1, ..., n_k \geq 0\}.$$

La chiusura L_3^* conterrà stringhe in questa forma con k ogni volta variabili.

Abbiamo quindi visto diverse situazioni: nel primo linguaggio non abbiamo dovuto calcolare nessuna chiusura, nel secondo linguaggio abbiamo calcolato un paio di linguaggi, nel terzo linguaggio non ci siamo mai fermati.

12.4.3. Teorema di Kleene e espressioni regolari

Con le operazioni che abbiamo visto noi possiamo creare dei nuovi linguaggi. Tra queste operazioni, possiamo raggruppare **unione**, **prodotto** e **chiusura di Kleene** sotto il cappello delle **operazioni regolari**. Come mai questo nome? Perché esse sono usate per definire i **linguaggi regolari-**

Vediamo tre versioni del seguente teorema, ma ci interesseremo solo della prima e della terza.

Teorema 12.4.3.1 (Teorema di Kleene): La classe dei linguaggi accettati da automi a stati finiti coincide con la più piccola classe contenente i linguaggi

$$\emptyset \quad | \quad \{\varepsilon\} \quad | \quad \{a\}$$

e chiusa rispetto alle operazioni di unione, prodotto e chiusura di Kleene.

Questa prima versione ci dice che possiamo costruire la classe dei linguaggi regolari partendo da tre linguaggi base e applicando in tutti i modi possibili le tre operazioni regolari.

Teorema 12.4.3.2 (Teorema di Kleene): La classe dei linguaggi accettati da automi a stati finiti coincide con la più piccola classe che contiene i linguaggi finiti.

Seconda versione carina, ma che non commentiamo.

Teorema 12.4.3.3 (Teorema di Kleene): La classe dei linguaggi accettati da automi a stati finiti coincide con la classe dei linguaggi espressi con le espressioni regolari.

Tutto bello, ma cosa sono le **espressioni regolari**?

Simbolo/espressione	**Linguaggio associato**
$\emptyset$	$\emptyset$
ε	$\{\varepsilon\}$
a	$\{a\}$
$E_1 + E_2$	$L(E_1) \cup L(E_2)$
$E_1 \cdot E_2$	$L(E_1) \cdot L(E_2)$
E^*	$(L(E))^*$

Le espressioni regolari sono una **forma dichiarativa**, ovvero grazie ad esse dichiariamo come sono fatte le stringhe di un certo linguaggio. Fin'ora avevamo usato gli automi (**forma riconoscitiva**) e le grammatiche (**forma generativa**).

Esempio 12.4.3.1: Vediamo un po' di espressioni regolari per alcuni linguaggi.

Linguaggio	**Espressione regolare**
$L = \{a^n \mid n \geq 0\}$	a^*
$L = \{a^{2k+1} \mid k \geq 0\}$	$(aa)^*a$
$L = \{a^n b \mid n \geq 0\}$	a^*b
$L = \left\{(a^n b)^k \mid n \geq 0 \land k > 0\right\}$	$(a^*b)^k$
L_3 terzultimo simbolo da destra è una a	$(a+b)^*a(a+b)(a+b)$

Il penultimo linguaggio ha una espressione regolare che non abbiamo visto: si tratta di una piccola estensione algebrica che ci permette di unire assieme una serie di fattori identici.

Andiamo ora a dimostrare il teorema di Kleene.

Dimostrazione 12.4.3.3.1: Dobbiamo mostrare una doppia implicazione.

[Automa $\longrightarrow$ RegExp]

Vedi esempio successivo su come fare questa operazione.

[RegExp $\longrightarrow$ Automa]

Non l'ha ancora spiegato. ■

Vediamo un esempio di come passare da un automa ad una espressione regolare.

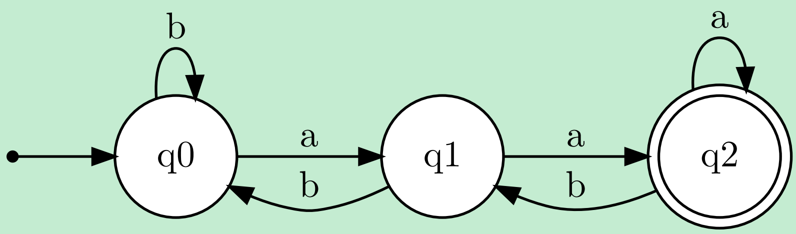
Esempio 12.4.3.2: Per ricavare una espressione regolare da un automa si usa un algoritmo di **programmazione dinamica** molto simile all'algoritmo Floyd-Warshall sui grafi, che cerca i cammini minimi imponendo una serie di vincoli.

Un altro approccio invece cerca di risolvere un **sistema di equazioni** associato all'automa.

Dato un automa, costruiamo un sistema di n equazioni, dove n è il numero di stati dell'automa. Supponendo di numerare gli stati da 1 a n, la i-esima equazione descrive i cambiamenti di stato che possono avvenire partendo dallo stato i.

Ogni **cambiamento di stato** è nella forma aB, dove a è il carattere che causa una transizione e B è lo stato di arrivo. Tutti i cambiamenti di stato a partire da i vanno sommati tra loro. Inoltre, se lo stato i-esimo è uno stato finale si aggiunge anche ε all'equazione.

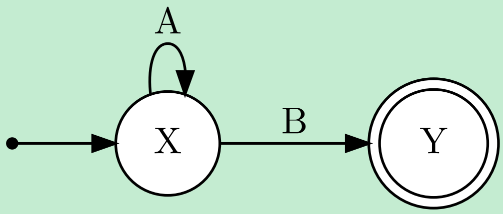
Questa somma di cambiamenti di stati va posta uguale allo stato i-esimo.



Associamo all'automa precedente un sistema di **3** equazioni, nel quale indichiamo gli stati con le variabili X_i e i caratteri sono quelli dell'alfabeto $\{a, b\}$. Il sistema è il seguente:

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

Ora dobbiamo risolvere questo sistema di equazioni. Per fare ciò, dobbiamo introdurre una **regola fondamentale** che ci permetterà di risolvere tutti i sistemi che vedremo.



Il sistema di equazioni per questo automa è

$$\begin{cases} X = AX + BY \\ Y = \varepsilon \end{cases}.$$

Sostituendo $Y = \varepsilon$ nella prima equazione otteniamo

$$X = AX + B.$$

L'espressione regolare per questo automa è

$$A^*B.$$

Visto che le due cose che abbiamo scritto devono essere identiche, ogni volta che abbiamo una equazione nella forma

$$X = AX + B$$

la possiamo sostituire con l'equazione

$$X = A^*B.$$

Riprendiamo il sistema dell'automa dell'esempio e andiamo a risolvere le nostre equazioni:

$$\begin{cases} X_0 = a(aX_2 + bX_0) + bX_0 \\ X_2 = aX_2 + b(aX_2 + bX_0) + \varepsilon \end{cases}$$

$$\begin{cases} X_0 = aaX_2 + abX_0 + bX_0 \\ X_2 = aX_2 + baX_2 + bbX_0 + \varepsilon \end{cases}$$

$$\begin{cases} X_0 = (ab + b)X_0 + aaX_2 \\ X_2 = (a + ba)X_2 + bbX_0 + \varepsilon \end{cases}$$

$$\begin{cases} X_0 = (ab + b)X_0 + aaX_2 \\ X_2 = (a + ba)^*(bbX_0 + \varepsilon) \end{cases}$$

$$X_0 = (ab + b)X_0 + aa((a + ba)^*(bbX_0 + \varepsilon))$$

$$X_0 = (ab + b)X_0 + aa(a + ba)^*bbX_0 + aa(a + ba)^*$$

$$X_0 = (ab + b + aa(a + ba)^*bb)X_0 + aa(a + ba)^*.$$

Applicando un'ultima volta la regola fondamentale otteniamo l'espressione regolare

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

E pensare che l'algoritmo basato su Floyd-Warshall è anche più difficile...

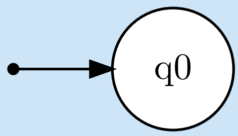
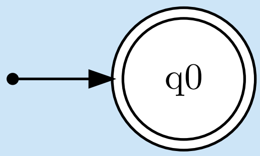
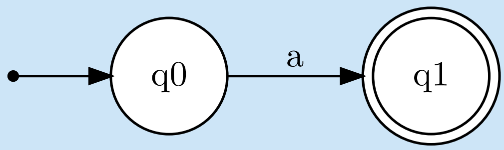
13. Lezione 09 [26/03]

13.1. Fine dimostrazione

Per finire la dimostrazione del **teorema di Kleene** dobbiamo essere in grado di passare dalle espressioni regolari ai linguaggi di tipo 3.

Teorema 13.1.1 (Teorema di Kleene): Vedi lezione scorsa.

Dimostrazione 13.1.1.1: Costruiamo degli automi per le espressioni regolari di base e poi costruiamo gli automi per le operazioni che usiamo per chiudere questa classe di linguaggi.

Espressione regolare	**Automa**
$\emptyset$	
ε	
a	

Per essere precisi, dovremmo utilizzare dei DFA che sono completi, quindi dobbiamo considerare anche lo stato trappola. In realtà, se vogliamo fare un conto asintotico non ci interessa molto, ma se vogliamo il numero preciso di stati allora quello stato è necessario.

Per vedere la composizione di questi stati usando le operazioni lineari, penso vada bene il prossimo capitolo sulle operazioni. ■

13.2. State complexity

Vogliamo studiare il numero di stati che sono necessari per definire un automa. Vediamo due quantità che sono chiave in questo studio.

Definizione 13.2.1 (State complexity deterministica): Sia $L \subseteq \Sigma^*$. Indichiamo con

$$\mathrm{sc}(L)$$

il minimo numero di stati di un DFA completo per L.

Abbiamo poi visto che l'automa con questo numero di stati è anche **unico**.

Definizione 13.2.2 (State complexity non deterministica): Sia $L \subseteq \Sigma^*$. Indichiamo con

$$\mathrm{nsc}(L)$$

il minimo numero di stati di un NFA per L.

In questo caso abbiamo visto che l'NFA minimo **non è unico**. Inoltre, non abbiamo la nozione di **completo** perché la funzione di transizione associa ad ogni passo di computazione una serie di scelte, che può essere anche la scelta vuota.

Lemma 13.2.1: Se L non è un linguaggio regolare allora

$$\mathrm{sc}(L) = \mathrm{nsc}(L) = \infty.$$

Lemma 13.2.2: Se L è un linguaggio regolare allora

$$\mathrm{sc}(L) < \infty \wedge \mathrm{nsc}(L) < \infty.$$

Avevamo inoltre il bound per passare da NFA a DFA, che nel caso peggiore trasformava n stati di un NFA in 2^n stati di un DFA con l'automa di **Meyer-Fischer**.

Esempio 13.2.1: Sia L_n il solito linguaggio dell'n-esimo simbolo da destra uguale ad a.

Avevamo visto un NFA che utilizzava $n+1$ stati, quindi

$$\mathrm{nsc}(L_n) \leq n + 1.$$

Si dimostra poi l'uguaglianza dei due valori utilizzando un fooling set.

Avevamo poi visto un DFA che utilizzava 2^n stati, quindi

$$\mathrm{sc}(L_n) = 2^n.$$

Con un insieme di stringhe distinguibili avevamo mostrato che servivano almeno 2^n stati, ma con la realizzazione effettiva abbiamo uguagliato il bound.

13.3. Operazioni

Estendiamo la nozione di state complexity alle operazioni sui linguaggi. Data un'operazione che preservi la **regolarità** su n linguaggi, ognuno con la propria state complexity, ci chiediamo quale sia la state complexity dell'operazione considerata sui linguaggi dati.

13.3.1. Complemento

Dato il linguaggio L con $\mathrm{sc}(L) = n$, vogliamo valutare la quantità $\mathrm{sc}\left(L^C\right)$ del **complemento** di L.

13.3.1.1. DFA

Se abbiamo un DFA per L, passare a L^C è molto facile: tutte le stringhe che prima accettavo ora le devo rifiutare e viceversa. Parlando in termini di dell'automa, invertiamo ogni stato finale in non finale e viceversa, mantenendo intatte le transizioni.

Dato $A = (Q, \Sigma, \delta, q_0, F)$ un DFA per L, costruisco l'automa $A' = (Q, \Sigma, \delta, q_0, F')$ un DFA per L^C tale che

$$F' = Q/F.$$

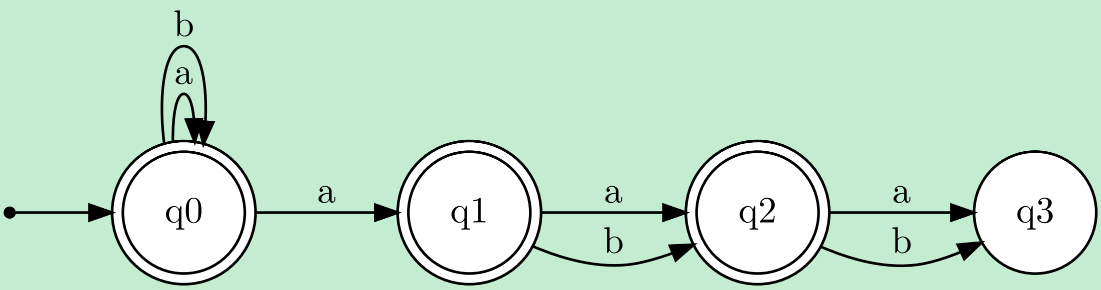
Dobbiamo imporre che A sia **completo** perché ciò che andava nello stato trappola ora deve essere accettato. Ma allora

$$\mathrm{sc}\left(L^C\right) = \mathrm{sc}(L).$$

13.3.1.2. NFA

Come ci comportiamo sugli NFA?

Esempio 13.3.1.2.1: Sia L_3 l'istanza del linguaggio L_n classico con $n = 3$. Andiamo a vedere un automa che cerca di calcolare L_3^C con la tecnica che abbiamo appena visto nei DFA.



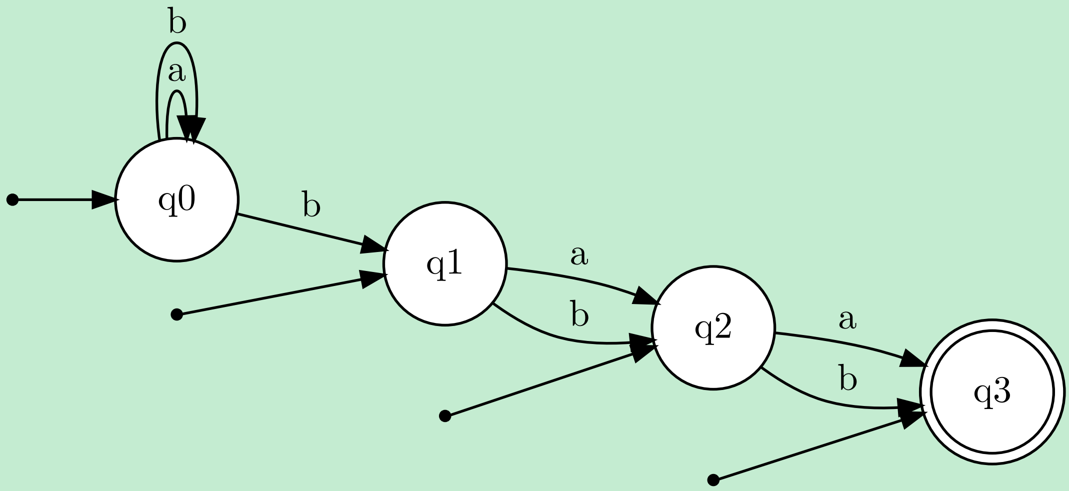
Abbiamo un problema: questo automa **accetta tutto**. Ma perché succede questo? Negli NFA accettiamo se esiste almeno un cammino accettante e rifiutiamo se ogni cammino è rifiutante. Quando accettiamo è molto probabile che ci sia, oltre al cammino accettante, anche qualche cammino rifiutante. Facendo il complemento, accettiamo ancora quando abbiamo almeno un cammino accettante, ma questo deriva da uno dei cammini rifiutanti precedenti.

È importantissimo avere il DFA, per via di questa asimmetria tra accettazione e non accettazione.

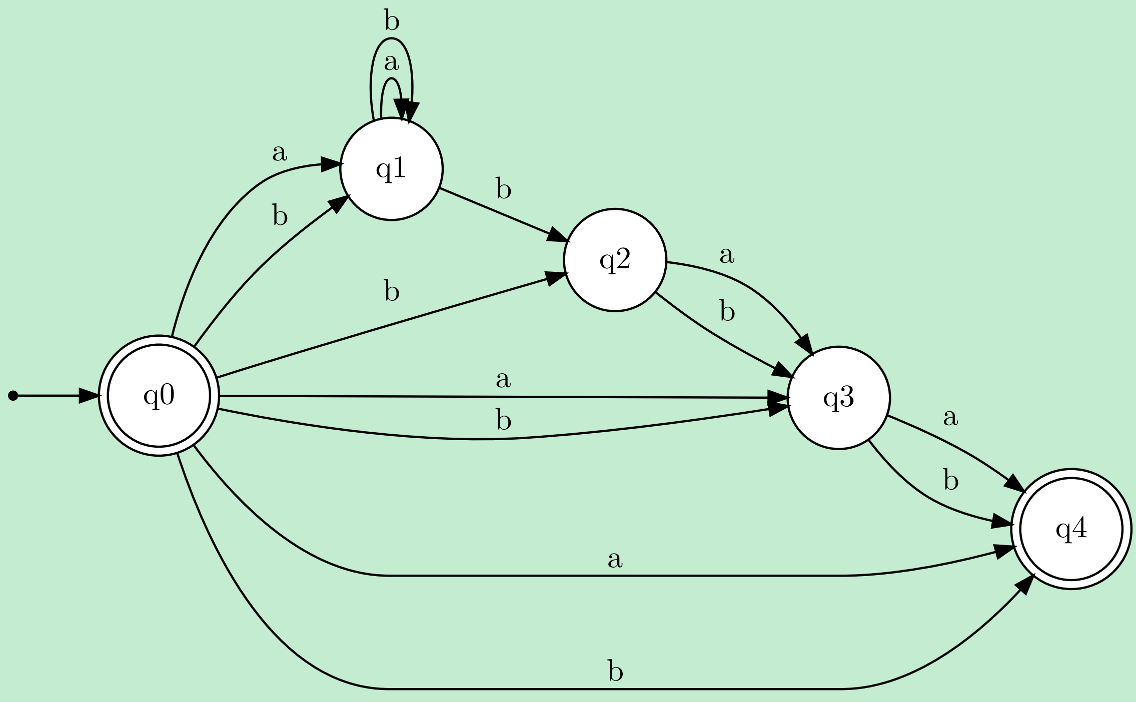
Ma se volessimo per forza un NFA per il complemento? Questo va molto a caso, dipende da linguaggio a linguaggio, potrebbe essere molto facile da trovare come molto difficile.

Esempio 13.3.1.2.2: Sempre per il linguaggio L_3, diamo due NFA per riconoscere L_3^C.

Una prima soluzione utilizza una serie di stati iniziali multipli.



Una seconda soluzione utilizza invece il non determinismo puro.



Questo approccio di cercare a tutti i costi un NFA può essere difficoltoso. Vediamo un algoritmo che ci permette di avere un automa per L^C, per ci darà un automa deterministico.

13.3.1.3. Costruzione per sottoinsiemi

Sia $A = (Q, \Sigma, \delta, q_0, F)$ un NFA per L, voglio un automa per il linguaggio L^C. Un modo sistematico e ottimo per avere un automa sotto mano è passare al DFA di A e poi eseguire la costruzione del complemento che abbiamo visto prima.

Quanti stati abbiamo? Sappiamo che abbiamo un salto esponenziale passando dall'NFA al DFA, e poi uno stesso numero di stati, quindi

$$\mathrm{nsc}\left(L^C\right) \leq 2^{\mathrm{nsc}(L)}.$$

Possiamo fare di meglio? Sicuramente esistono esempi di salti che non sono esattamente esponenziali, come i linguaggi delle coppie di elementi uguali/diversi a distanza n, che avevano un salto del tipo

$$2n + 2 \longrightarrow 2^n,$$

ma si può costruire un esempio che faccia un salto esponenziale perfetto.

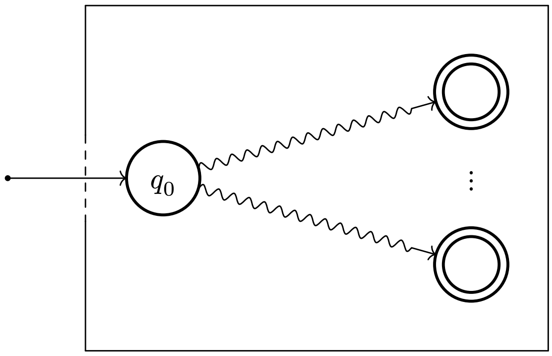
Abbiamo quindi visto che del complemento negli NFA non ce ne facciamo niente, questo proprio per la natura del non determinismo.

13.3.2. Unione

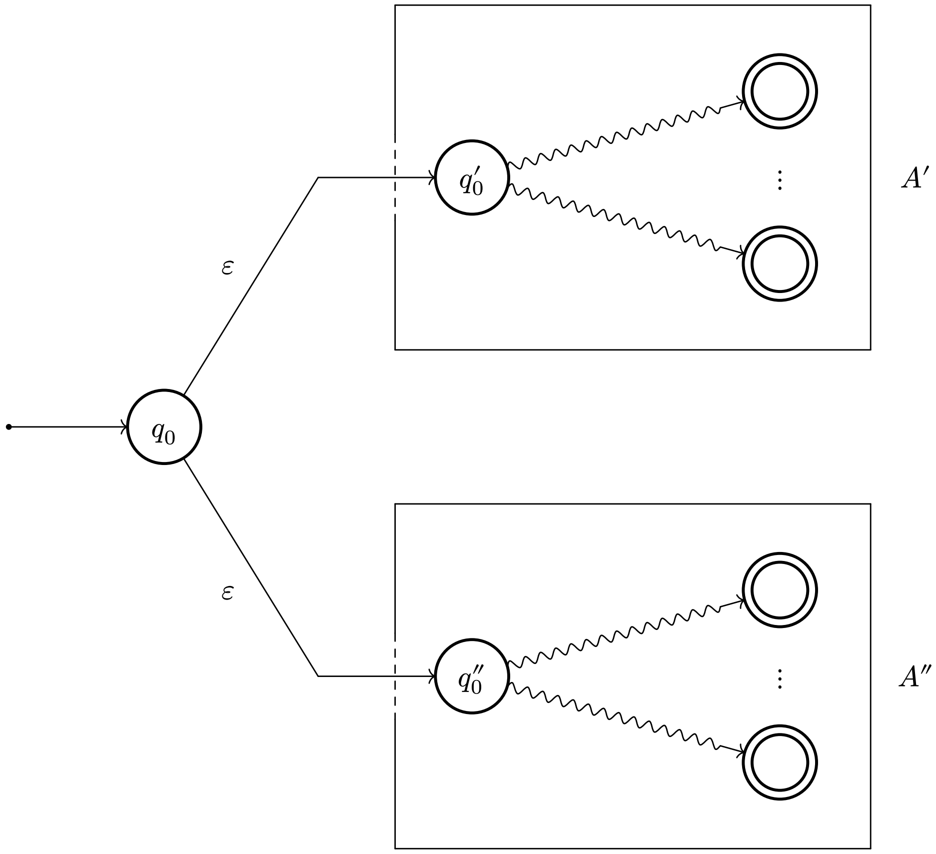
Dati due linguaggi $L', L'' \subseteq \Sigma^*$ rispettivamente riconosciuti dagli automi $A' = (Q', \Sigma, \delta', q_0', F')$ e $A'' = (Q'', \Sigma, \delta'', q_0'', F'')$, vogliamo costruire un automa per l'**unione**

$$L' \cup L''.$$

Per risolvere questo problema pensiamo agli automi come se fossero delle scatole, che prendono l'input nello stato iniziale e poi arrivano alla fine nell'insieme degli stati finali.



L'idea per costruire l'automa l'unione è combinare i due automi A' e A'' usando il non determinismo per scegliere in quale automa finire con una ε-mossa.



Visto che il linguaggio dell'unione deve stare in almeno uno dei due, metto una scommessa all'inizio per vedere se andare nel primo o nel secondo automa. Bella soluzione, funziona, ma non ci piace tanto, come mai?

13.3.2.1. DFA

Non ci piace tanto questa soluzione perché se partiamo da due DFA andiamo a finire in un NFA. Infatti, la componente, non deterministica viene inserita con le due ε-mosse iniziali. La stessa componente non deterministica l'avremmo inserita con gli stati iniziali multipli, che sarebbero stati in corrispondenza dei due stati iniziali q'_0 e q''_0 senza lo stato q_0.

Se vogliamo rimanere nel mondo DFA dobbiamo unire i due automi con questa costruzione e poi passare al DFA con la costruzione per sottoinsiemi. Il numero di stati dell'NFA è

$$\text{nsc}(L' \cup L'') \leq 1 + \text{nsc}(L') + \text{nsc}(L''),$$

quindi con la costruzione per sottoinsiemi arriveremmo ad avere un numero di stati pari a

$$\text{sc}(L' \cup L'') \leq 2^{\text{nsc}(L' \cup L'')}.$$

Questa costruzione è altamente **inefficiente**. Si può fare molto meglio.

13.3.2.2. Automa prodotto

Utilizzando una costruzione particolare, la **costruzione dell'automa prodotto**, siamo in grado di abbassare di brutto la complessità in stati dei DFA per l'unione di linguaggi.

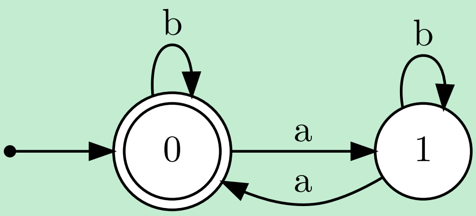
L'**automa prodotto** fa partire in parallelo i due automi, e alla fine controlla che almeno uno dei due abbia dato un cammino accettante. Definiamo quindi $A = (Q, \Sigma, \delta, q_0, F)$ tale che:

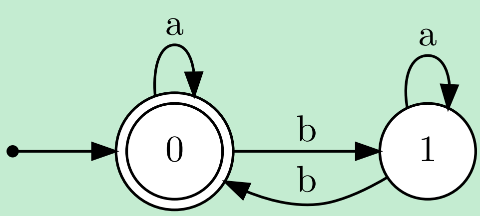
- gli **stati** rappresentano i due automi che viaggiano in parallelo, come se avessi due pc davanti, ognuno che lavora da solo. Gli stati sono quindi l'insieme
$$Q = Q' \times Q'';$$
- lo **stato iniziale** è la coppia di stati iniziali, ovvero
$$q_0 = (q_0', q_0'');$$
- la **funzione di transizione** lavora ora sulle coppie di stati, che deve portare avanti in parallelo, quindi
$$\delta((q, p), a) = (\delta'(q, a), \delta''(p, a));$$
- gli **stati finali** sono tutte le coppie dove riesco a finire in almeno uno stato finale, ovvero
$$F = \{(q, p) \mid q \in F' \vee p \in F''\}.$$

Come cambia la complessità dell'automa rispetto alla costruzione per sottoinsiemi? Qua il numero di stati è
$$\mathrm{sc}(L' \cup L'') \leq \mathrm{sc}(L') \cdot \mathrm{sc}(L''),$$
quindi abbiamo una soluzione notevolmente migliore. Inoltre, non si può fare meglio di così.

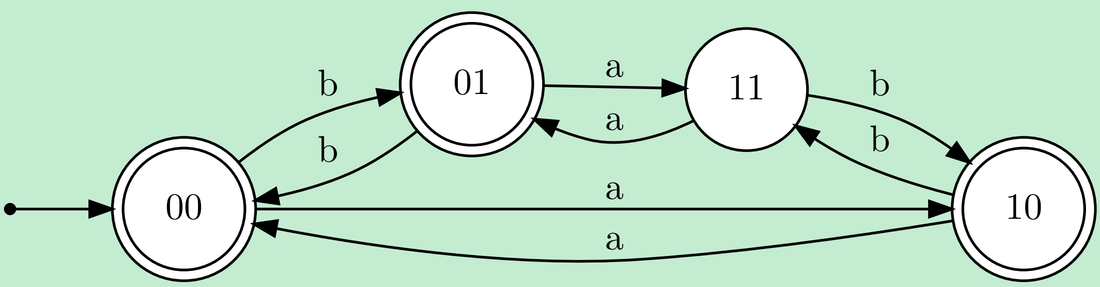
Esempio 13.3.2.2.1: Fissati due valori m, n positivi, definiamo i linguaggi
$$L' = \{x \in \{a, b\}^* \mid \#_a(x) \text{ è multiplo di } m\}$$
$$L'' = \{x \in \{a, b\}^* \mid \#_b(x) \text{ è multiplo di } n\}.$$

I due automi A' e A'' per L' e L'' sono molto semplici, devono solo contare il numero di a e b. Vediamo un esempio con $m = n = 2$.





Costruiamo l'automa prodotto per il linguaggio $L = L' \cup L''$.



13.3.2.3. NFA

Negli NFA non abbiamo nessun problema: partiamo da NFA e vogliamo restare in NFA, quindi non servono ulteriori costruzioni per avere un automa di questa classe. Il numero di stati è

$$\mathrm{nsc}(L' \cup L'') \leq 1 + \mathrm{nsc}(L') + \mathrm{nsc}(L'').$$

Perdiamo il termine noto di questa quantità se non usiamo ε-mosse ma stati iniziali multipli.

13.3.3. Intersezione

Per l'**intersezione** di linguaggi non dobbiamo definire molto di nuovo.

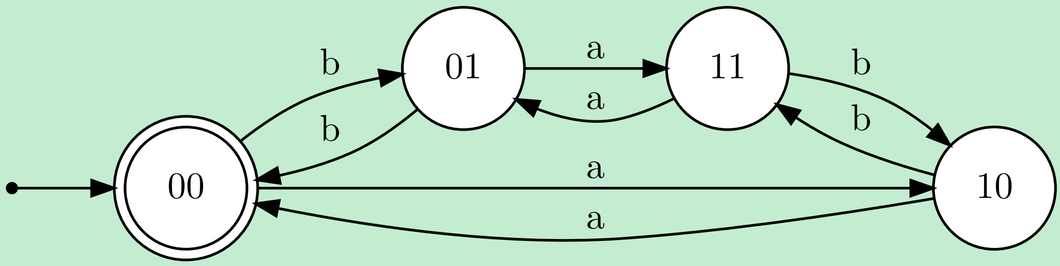
Per i DFA, possiamo utilizzare la costruzione dell'automa prodotto appena definita modificando l'insieme degli stati finali F rendendolo l'insieme

$$F = \{(q, p) \mid q \in F' \land p \in F''\}.$$

Ma allora la state complexity vale

$$\mathrm{sc}(L' \cap L'') \leq \mathrm{sc}(L') \cdot \mathrm{sc}(L'').$$

Esempio 13.3.3.1: Riprendendo i due linguaggi di prima, l'automa prodotto viene costruito nello stesso modo, ma cambia l'insieme degli stati finali, che si riduce al singleton $\{00\}$.



Per gli NFA, possiamo riutilizzare la costruzione dell'automa prodotto per permetterci di navigare tutte le possibile coppie di cammini, e scommettendo bene su entrambi i cammini possiamo accettare la stringa. Va sistemata un pelo la definizione della funzione di transizione, ma la costruzione rimane uguale. Vale quindi

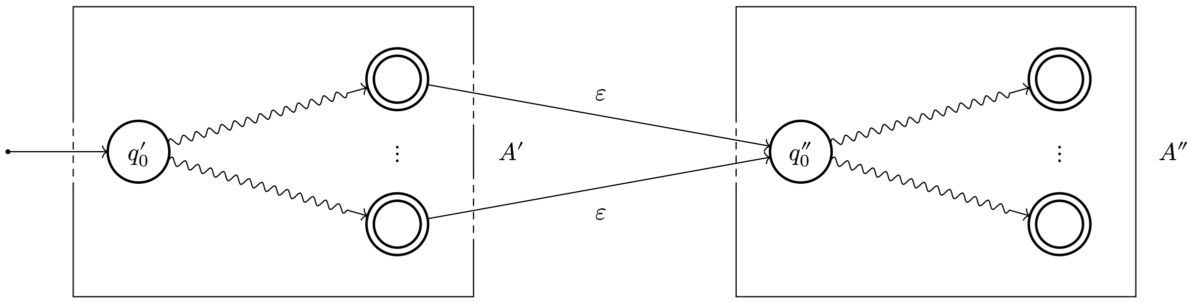
$$\mathrm{nsc}(L' \cap L'') \leq \mathrm{nsc}(L') \cdot \mathrm{nsc}(L'').$$

13.3.4. Prodotto

Riprendiamo velocemente la definizione di **prodotto** di linguaggi. Dati due linguaggi L' e L'', allora

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

Sfruttiamo la rappresentazione black box degli automi: mettendoli in serie utilizzando le ε-mosse.



In poche parole, ogni volta che arriviamo in uno stato finale di A' facciamo partire la computazione su A'', ma in A' andiamo avanti a scandire la stringa. Stiamo scommettendo di essere arrivati alla fine della stringa x e di dover iniziare a leggere la stringa y.

Bella costruzione, ma ci va veramente bene una roba del genere?

13.3.4.1. DFA

La risposta, come prima, è **NO**: se partiamo da due DFA andiamo a finire in un NFA, che non ci va bene perché per poi tornare in un DFA ci costa un salto esponenziale. Visto che

$$\mathrm{nsc}(L' \cdot L'') = \mathrm{nsc}(L') + \mathrm{nsc}(L''),$$

possiamo dire che

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

Come prima, possiamo ottimizzare questa costruzione, anche se non di molto stavolta.

13.3.4.2. Costruzione senza nome

Il problema dell'esplosione del doppio esponenziale deriva dal fatto che, quando arrivo in uno stato finale del primo automa, devo far partire il secondo automa, ma il primo continua ancora a scandagliare la stringa perché deve scommettere.

La soluzione inefficiente di prima prendeva i due automi A' e A'', li univa in un NFA ed effettuava la costruzione per sottoinsiemi. La soluzione che facciamo adesso **incorpora** i sottoinsiemi nei passi del DFA, così da evitare l'esecuzione non deterministica.

Costruisco l'automa $A = (Q, \Sigma, \delta, q_0, F)$ che, ogni volta che A' finisce in uno stato finale, avvia anche A'' dal punto nel quale si trova. Esso è definito da:

- gli **stati** sono tutte le coppie di stati di A' con i sottoinsiemi di A'', così da incorporare i sottoinsiemi nel DFA direttamente, ovvero
$$Q = Q' \times 2^{Q''};$$
- lo **stato iniziale** dipende se siamo già in una configurazione che permette lo start di A'', ovvero
$$q_0 = \begin{cases} (q_0', \varnothing) & \text{se } q_0' \notin F' \\ (q_0', \{q_0''\}) & \text{se } q_0' \in F' \end{cases};$$
- la **funzione di transizione** deve lavorare sulla prima componente ma anche su tutte quelle presenti nella seconda componente, quindi essa è definita come
$$\delta((q, \alpha), a) = \begin{cases} (\delta'(q, a), \{\delta''(p, a) \mid p \in \alpha\}) & \text{se } \delta'(q, a) \notin F' \\ (\delta'(q, a), \{\delta''(p, a) \mid p \in \alpha\} \cup \{q_0''\}) & \text{se } \delta'(q, a) \in F' \end{cases};$$
- gli **stati finali** sono quelli nei quali riusciamo ad arrivare con il secondo automa, ovvero

$$F = \{(q, \alpha) \mid \alpha \cap F'' \neq \emptyset\}.$$

La prima componente la mandiamo avanti deterministicamente, ma la manteniamo sempre accesa per far partire la seconda computazione. Quest'ultima è anch'essa deterministica, ma simula un po' il comportamento non deterministico.

Il numero di stati massimo che abbiamo è

$$\text{sc}(L' \cdot L'') = \text{sc}(L')2^{\text{sc}(L'')},$$

che rappresenta comunque un gap esponenziale ma abbiamo abbassato di un po' la complessità.

13.3.4.3. NFA

Come per l'unione, qua siamo molto tranquilli: partiamo da NFA e arriviamo in NFA, quindi a noi va tutto bene. La state complexity, come visto prima, è

$$\text{nsc}(L' \cdot L'') \leq \text{nsc}(L') + \text{nsc}(L'').$$

14. Lezione 10 [28/03]

14.1. Prodotto (richiamo)

Piccole osservazioni da aggiungere alla lezione precedente:

- abbiamo un suffisso e un prefisso, ma non sapendo quando finisce il primo e inizia il secondo devo scommettere quando arrivo in uno stato finale del primo automa; ci possono essere tante suddivisioni, devo indovinare quella giusta;
- l'automa A che abbiamo costruito ha una prima parte deterministica, mentre la seconda è sì deterministica ma emula la costruzione per sottoinsiemi;
- non possiamo evitare il salto esponenziale: se concateniamo $(a+b)^*$ con $a(a+b)^{n-1}$ otteniamo il solito L_n, che ha bisogno di 2^n stati partendo da $n+1$.

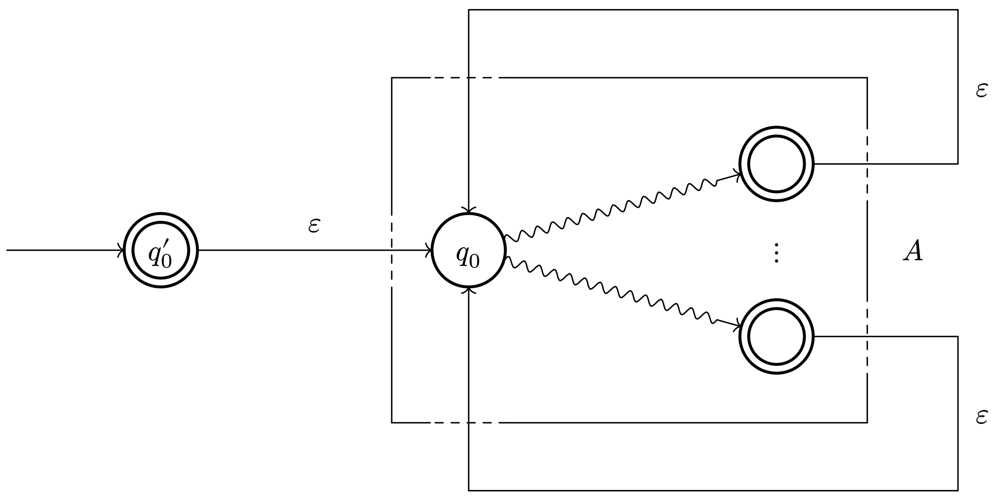
14.2. Chiusura di Kleene

Con questa ultima operazione chiudiamo la dimostrazione del teorema di Kleene.

Questa operazione è definita come

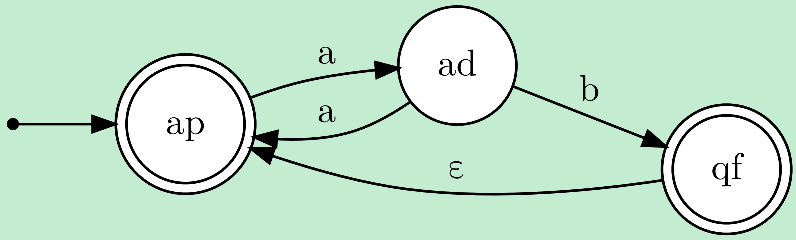
$$L^* = \bigcup_{k \geq 0} L^k = \{w \in \Sigma^* \mid \exists x_1, ..., x_k \in L \mid k \geq 0 \mid w = x_1, ..., x_k\}.$$

Un automa per la star deve cercare di scomporre la stringa in ingresso in più stringhe di L. Possiamo prendere spunto dall'automa per la concatenazione: facciamo partire l'automa per L, ogni volta che arriviamo in uno stato finale lo facciamo ripartire dallo stato iniziale. Devo accettare anche la parola vuota, quindi aggiungiamo uno stato iniziale finale.



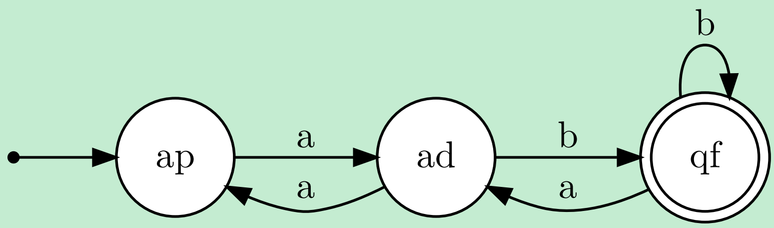
Abbiamo bisogno dello stato iniziale q_0' perché dentro l'automa ci possono essere delle transizioni che mi fanno ritornare indietro e mi fanno accettare di più di quello che dovrei.

Esempio 14.2.1: Consideriamo il seguente automa **sbagliato** per la chiusura di Kleene del linguaggio delle sequenze dispari di a seguite da una b. Se vogliamo l'espressione regolare per questo linguaggio, essa è $(aa)^*ab$.



Non abbiamo inserito uno stato iniziale aggiuntivo. Che succede? La stringa aa adesso viene accettata, anche se palesemente non appartiene al linguaggio, così come la stringa $abaa$.

Un automa per la star di questo linguaggio, inoltre, è molto facile da trovare perché la lettera b fa da marcatore, e il numero di stati rimane quello dell'automa di partenza.



14.2.1. DFA

Ci piace questa soluzione, perché stiamo aumentando solo di uno il numero di stati dell'automa, ma siamo caduti nel non determinismo, e partire da DFA e finire in NFA non ci piace molto.

Purtroppo, come spesso ci succede, per tornare nei DFA dobbiamo, nel caso peggiore, applicare la costruzione per sottoinsiemi e fare un salto esponenziale nel numero degli stati. In poche parole

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

14.2.2. NFA

L'abbiamo formulato nella scorsa sezione: se partiamo da NFA otteniamo ancora degli NFA, quindi ci piace, e lo facciamo in modo semplice aggiungendo solo uno stato, ovvero

$$\mathrm{nsc}(L^*) \leq \mathrm{nsc}(L) + 1.$$

14.2.3. Esempi utili per dopo

Per ora abbiamo visto l'esempio del linguaggio $L = (aa)^* b$, che era estremamente comodo da scomporre per via della presenza di una sola b nella stringa in input. Vediamo ora qualche altro esempio notevole per la prossima sezione.

Esempio 14.2.3.1: Definito il linguaggio $L = aaaba^*$, dobbiamo calcolare L^*.

Questo linguaggio è «facilmente» scomponibile: ogni volta che troviamo una b torniamo indietro di 3 caratteri e dividiamo la stringa in quel punto.

Ad esempio, la stringa

$$aaabaaaaaabaaabaaaab$$

viene suddivisa nel seguente modo:

$$aaabaaaa \mid aaab \mid aaaba \mid aaab.$$

L'automa comunque esegue un sacco di test non deterministici ogni volta che legge delle a dopo una b, perché rimaniamo sempre in uno stato finale.

Esempio 14.2.3.2: Definito invece il linguaggio $L = a(b + baab)a^*$, dobbiamo calcolare L^*.

Questo linguaggio invece è più difficile da scomporre. Ad esempio, data la stringa

$$abaabaaaaabaaba$$

essa la possiamo dividere in

$$aba \mid abaaaa \mid abaaba$$

oppure la possiamo dividere in

$$abaabaaaa \mid abaaba.$$

Per questa stringa abbiamo già due modi di scomposizione possibili.

Cambiamo la stringa: data la stringa

$$abaabaabaabaaaba$$

essa la possiamo dividere in

$$aba \mid aba \mid aba \mid abaa \mid aba$$

oppure la possiamo dividere in

$$abaaba \mid abaabaa \mid aba.$$

In generale, si possono creare delle stringhe che hanno un numero di suddivisioni enorme.

14.2.4. Codici

A cosa servono i tre esempi che abbiamo introdotto nella sezione precedente?

Definizione 14.2.4.1 (Codice): Dato $X \subseteq \Sigma^*$, diciamo che X è un **codice** se e solo se

$$\forall w \in X^* \quad \exists! \text{ decomposizione di } w \text{ come } x_1 ... x_k \mid x_i \in L \land k \geq 0.$$

Dei tre esempi che abbiamo visto, solo i primi due sono dei codici: è facile dimostrare che lo sono, soprattutto il primo per via della b che fa da delimitatore. L'ultimo esempio, invece, abbiamo visto che ha delle stringhe scomponibili in più modi, quindi non è un codice.

Tra tutti i codici a noi interessano quelli che possono essere **decomposti in tempo reale**: essi sono chiamati **codici prefissi**, e sono dei codici tali che

$$\forall i \neq j \quad x_i \text{ non è prefisso di } x_j.$$

In poche parole, il codice contiene parole che **non** sono prefisse di altre. Essi sono i più **efficienti**.

Dei due codici che abbiamo a disposizione, il primo è un codice prefisso: ogni volta che troviamo una b sappiamo che dobbiamo dividere. Il secondo, invece, deve aspettare una b e poi tornare indietro di 3 posizioni per avere la decomposizione.

14.2.5. Star height

Per definire le espressioni regolari abbiamo a disposizione le tre operazioni

$$+ \quad | \quad \cdot \quad | \quad ()^*$$

Concentriamoci un secondo sulla chiusura di Kleene e vediamo un esempio.

Esempio 14.2.5.1: Date le espressioni regolari

$$(a^* b^*)^* \qquad (a+b)^* \qquad (ab^*)^*$$

ci chiediamo che linguaggio stanno denotando. Questo è facile: $\{a, b\}^*$. Ognuna lo fa a modo proprio, in base a come ha risolto il sistema delle equazioni dell'automa associato.

Nell'esempio abbiamo visto diverse espressioni per lo stesso linguaggio, ma quante star mi servono per definire completamente un linguaggio?

Definizione 14.2.5.1 (Star height): La **star height** è il massimo numero di star innestate in una espressione regolare. Possiamo definire la quantità induttivamente, ovvero

$$\begin{aligned} h(\emptyset) = h(\varepsilon) = h(a) &= 0 \\ h(E' + E'') = h(E' \cdot E'') &= \max\{h(E'), h(E'')\} \\ h(E^*) &= 1 + h(E). \end{aligned}$$

Nell'esempio precedente, le espressioni regolari hanno star height rispettivamente 2, 1 e 2. A noi piacerebbe scrivere un'espressione regolare con la minima star height.

Sia L un linguaggio regolare. Definiamo con $h(L)$ la **minima altezza** delle espressioni regolari che definiscono L, ovvero

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

Questa quantità, nei linguaggi infiniti, è almeno 1, non posso usare meno star.

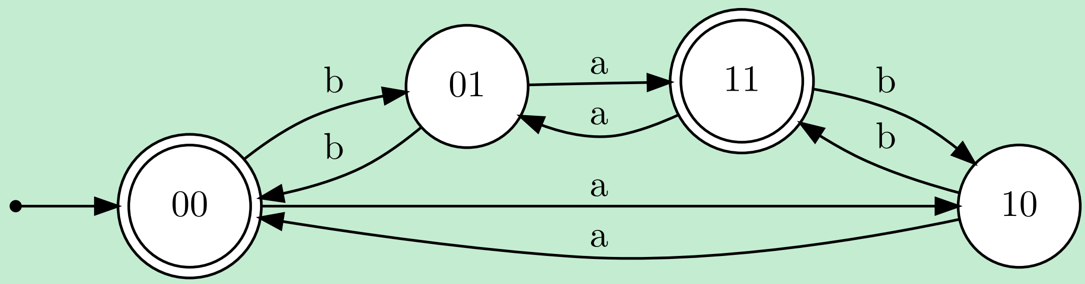
Teorema 14.2.5.1 (Un bro nel 1966): Vale

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

Il linguaggio W_q è definito come

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

Esempio 14.2.5.2: Proviamo a disegnare W_q per $q = 1$.



La sua espressione regolare, dopo un po' di conti, è la seguente:

$$L = \left(a(bb)^*a + a(bb)^*ba(aa + ab(bb)^*ba)^*ab(bb)^*a\right)^*\left(a(bb)^*ba(aa + ab(bb)^*ba)^*ab(bb)^*b\right).$$

A quanto pare ho sbagliato a risolvere il sistema, sono scarso scusate.

Abbiamo quindi fatto vedere che se fissiamo il numero $q > 0$ di star che vogliamo usare in una espressione regolare, riusciamo a trovare un linguaggio W_q che usa quel numero di star.

Teorema 14.2.5.2: Se $|\Sigma| = 1$ è sufficiente una star innestata per definire completamente L, ovvero vale che

$$\forall L \subseteq \{a\}^* \text{ regolare} \quad h(L) \leq 1.$$

14.3. Espressioni regolari estese

Cosa succede se nelle espressioni regolari, oltre alle operazioni regolari di unione, concatenazione e chiusura, utilizziamo anche le operazioni di **intersezione** e **negazione**?

Esse sono definite **espressioni regolari estese**, e sono molto potenti ma devono essere usate con cautela. Vediamo il perché con qualche esempio.

Esempio 14.3.1: Sia

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

Se mi chiedono l'espressione regolare di questo linguaggio mi mandano a quel paese, ma usando le espressioni regolari estese questo è molto facile.

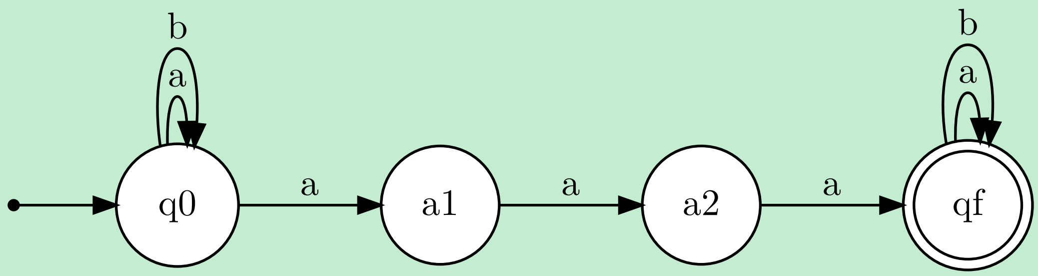
Per il linguaggio $L_a = \{w \in \{a,b\}^* \mid \#_a(w) \text{ pari}\}$ possiamo scrivere l'espressione regolare $(b + ab^*a)^*$, mentre per il linguaggio $L_b = \{w \in \{a,b\}^* \mid \#_b(w) \text{ pari}\}$ possiamo scrivere l'espressione regolare $(a + ba^*b)^*$.

Utilizzando l'intersezione possiamo banalmente concatenare le due espressioni, ovvero

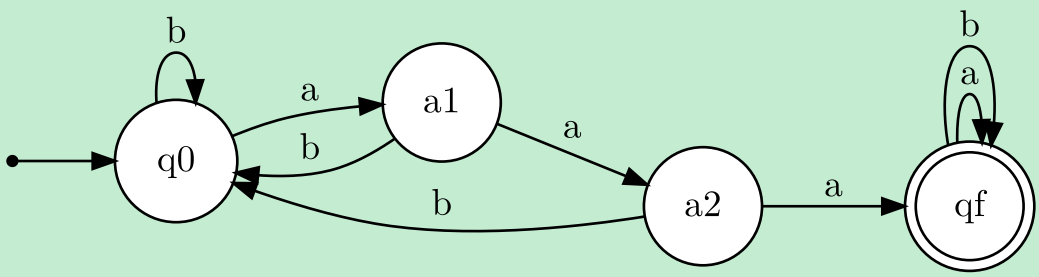
$$(b + ab^*a)^* \cap (a + ba^*b)^*.$$

Esempio 14.3.2: Vogliamo un'espressione regolare per le stringhe che **non** contengono tre a consecutive. Per fare ciò, costruiamo un automa e poi ricaviamo l'espressione regolare.

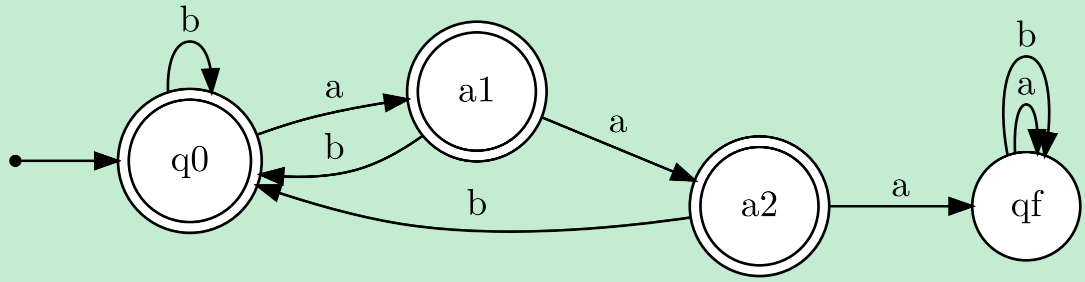
Partiamo da automi che accettano stringhe con tre a consecutive e poi complementiamo. In ordine, vediamo un NFA e un DFA per questo linguaggio di transizione.



Avevamo visto che il complemento non si comportava bene con gli NFA, quindi diamo un DFA, che invece funzionava bene con il complemento.



Siamo rimasti in un numero di stati accettabile. Andiamo a complementare questo DFA, che manterrà lo stesso numero di stati, fortunatamente.



Ora di questo dovrei farci l'espressione regolare. Sicuramente esce una bestiata enorme, con un po' di star qua e là visto che il linguaggio è infinito.

Con le espressioni regolari estese possiamo evitare l'uso delle star. Prima prendiamo tutte le stringhe che hanno tre a consecutive: esse sono nella forma

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

Dobbiamo applicare il complemento a questa espressione per ottenere L, quindi

$$\overline{(a+b)^* aaa (a+b)^*}.$$

L'insieme di tutte le stringhe su un alfabeto lo possiamo vedere come il complemento dell'insieme vuoto rispetto all'insieme delle stringhe su quell'alfabeto, ovvero

$$(a+b)^* = \overline{\emptyset}.$$

Ma allora l'espressione regolare diventa

$$\overline{\overline{\emptyset} aaa \overline{\emptyset}},$$

che come vediamo è un'espressione che non utilizza alcuna star.

Siamo stati in grado di non usare le star per un linguaggio infinito, cosa che nelle espressioni regolari classiche non è possibile. Bisogna stare attenti però: il complemento è molto comodo ma è anche molto insidioso perché fa saltare il numero di stati esponenzialmente se usiamo degli NFA.

Come nelle espressioni regolari, possiamo chiederci il **minimo numero di star** che sono necessarie per descrivere completamente un linguaggio.

Definizione 14.3.1 (Star height generalizzata): L'**altezza generalizzata**, o star height generalizzata, è il massimo numero di star innestate di una espressione regolare estesa. Possiamo definire la quantità induttivamente, ovvero

$$\begin{aligned} \mathrm{gh}(\emptyset) = \mathrm{gh}(\varepsilon) = \mathrm{gh}(a) &= 0 \\ \mathrm{gh}(E' + E'') = \mathrm{gh}(E' \cdot E'') = \mathrm{gh}(E' \cap E'') &= \max\{\mathrm{gh}(E'), \mathrm{gh}(E'')\} \\ \mathrm{gh}\left(E^C\right) &= \mathrm{gh}(E) \\ \mathrm{gh}(E^*) &= 1 + \mathrm{gh}(E). \end{aligned}$$

Come prima, dato un linguaggio L, possiamo definire la quantità

$$\mathrm{gh}(L) = \min\{g(E) \mid L = L(E)\}$$

come la minima star height generalizzata di tutte le espressioni regolari estese che generano L.

Le espressioni regolari estese sono molto comode, ma di queste non si sa quasi niente:
- si sa che esistono linguaggi di altezza 0 (pefforza);
- si sa che esistono linguaggi di altezza 1;
- non si sa niente sui linguaggi di altezza almeno 2.

Quale è il cambio di marcia tra le espressioni regolari estese e quelle classiche? L'operazione di **not**: questa ci permette di dichiarare cosa non ci interessa, mentre nelle espressioni regolari classiche noi possiamo solo dire cosa vogliamo, avendo un modello dichiarativo.

14.4. Operazioni esotiche

14.4.1. Reversal

Sia $L \subseteq \Sigma^*$ un linguaggio. Chiamiamo

$$L^R = \{w^R \mid w \in L\}$$

il linguaggio delle stringhe ottenute ribaltando tutte le stringhe di L, ovvero

$$w = a_1 ... a_n \Longrightarrow w^R = a_n ... a_1.$$

Questa operazione sul linguaggio L viene detta **reversal**.

Lemma 14.4.1.1: L'operazione di reversal preserva la regolarità.

Dimostrazione 14.4.1.1.1 (Espressioni regolari): Le espressioni regolari di base possono essere definite nel seguente modo:

$$(\emptyset)^R = \emptyset$$
$$(\varepsilon)^R = \varepsilon$$
$$(a)^R = a.$$

Ora vediamo come definiamo le espressioni regolari induttive:

$$(E_1 + E_2)^R = E_1^R + E_2^R$$
$$(E_1 \cdot E_2)^R = E_2^R \cdot E_1^R$$
$$(E^*)^R = (E^R)^*.$$

Queste espressioni sono ancora regolari, quindi reversal preserva la regolarità. ■

Possiamo dimostrare questo lemma anche usando gli **automi**. Supponiamo di avere $A = (Q, \Sigma, \delta, q_0, F)$ DFA che riconosce L. Vogliamo trovare un automa A' per L^R: questo è facile, basta mantenere la struttura dell'automa invertendo il senso delle transizioni, rendendo finale lo stato iniziale e rendendo iniziali tutti gli stati finali.

In poche parole, definiamo l'automa

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

definito dalla funzione di transizione δ' tale che

$$\delta'(q, a) = \{p \mid \delta(p, a) = q\}.$$

In poche parole, visto che ho reversato le transizioni, devo vedere tutti gli stati p che finivano in q con a per poter fare il cammino inverso.

14.4.1.1. DFA

Ci piace questa soluzione? **NO**: siamo partiti da un DFA e abbiamo ottenuto un NFA per via degli stati iniziali multipli F e per il fatto che la funzione di transizione può mappare in più stati con la stessa lettera letta.

Se voglio ottenere un DFA devo fare il classicissimo salto esponenziale con la costruzione per sottoinsiemi. Con il reversal dell'automa non abbiamo cambiato il numero degli stati, quindi

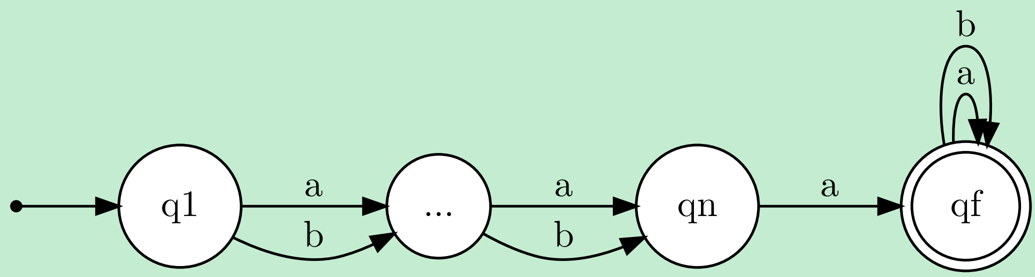
$$\text{sc}(L^R) \leq 2^{\text{nsc}(L^R)}.$$

Possiamo fare di meglio o nel caso peggiore abbiamo questo salto esponenziale?

Esempio 14.4.1.1.1: Prendiamo

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

il linguaggio che ha l'n-esimo simbolo da sinistra uguale ad una a, che riconosco con il seguente automa deterministico.



In questo caso, il numero di stati è esattamente n, visto che ne abbiamo $n - 1$ all'inizio per eliminare gli $n - 1$ caratteri iniziali e poi uno stato per verificare di avere una a.

Il suo reversal è $L^R = L_n$, il solito linguaggio dell'n-esimo simbolo da destra uguale ad una a. Abbiamo visto che un NFA per L_n ha $n + 1$ stati mentre il DFA ha 2^n stati, visto che deve osservare finestre di n caratteri consecutivi.

Il gap esponenziale non riusciamo purtroppo ad evitarlo.

Esempio 14.4.1.1.2: Molto curiosa la situazione inversa: se partiamo da L_n riconosciuto da un DFA di 2^n stati noi otteniamo un reversal $L_n^R = L$ che, minimizzato, ha n stati (escluso lo stato trappola).

14.4.1.2. NFA

Se partiamo invece da un NFA, con la costruzione precedente manteniamo lo status di NFA, mantenendo inoltre il numero degli stati, quindi siamo contenti, ottenendo

$$\mathrm{nsc}(L^R) = \mathrm{nsc}(L).$$

14.4.2. Shuffle

L'operazione di **shuffle**, applicata a due stringhe, le prende e le mescola, mantenendo l'ordine dei caratteri mentre le mischiamo. In poche parole, possiamo pensare di avere due **mazzieri**, ognuno dei quali tiene una stringa come lista ordinata (non lessicograficamente, ma proprio come è stata scritta) dei suoi caratteri. Ad ogni iterazione scegliamo a quale mazziere chiedere un carattere, e lo aggiungiamo alla stringa finale.

Come vediamo, l'inserimento che facciamo **non è atomico**: posso chiedere al mazziere che voglio ad ogni iterazione, l'unica cosa che mi viene chiesta è di mantenere l'ordine.

Esempio 14.4.2.1: Date le stringhe $aabb$ e b, possiamo ottenere le stringhe

$baabb$ $\qquad$ $ababb$ $\qquad$ $aabbb$ $\qquad$ $aabbb$ $\qquad$ $aabbb$

Ovviamente, le ultime tre stringhe sono tutte uguali e vanno considerate come stringa unica.

Esempio 14.4.2.2: Date le stringhe $aabb$ e ab, possiamo ottenere molte stringhe con lo shuffle. Ne vediamo un paio per vedere la non atomicità dell'operazione.

$aababb$ $\qquad$ $aaabbb$

L'operazione di shuffle, se la applichiamo ai **linguaggi** L' e L'', è il linguaggio di tutte le stringhe ottenute tramite shuffle di una stringa di L' con una stringa di L''. Ovviamente prendiamo tutte le possibili coppie di stringhe per ottenere il linguaggio completo.

14.4.2.1. Alfabeti disgiunti

Consideriamo due DFA A' e A'' per i due linguaggi appena definiti. Il caso più semplice di automa per lo shuffle parte con gli alfabeti per i due linguaggi **disgiunti**, ovvero L' definito sull'alfabeto $\Sigma' = \{a, b\}$ e L'' definito sull'alfabeto $\Sigma'' = \{c, d\}$.

Prendiamo spunto dall'**automa prodotto**: uniamo i due automi e ne mandiamo avanti uno alla volta in base al carattere che leggiamo. Definiamo quindi

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

tale che:

- gli **stati** sono tutte le possibili coppie di stati dei due automi, ovvero
$$Q = Q' \times Q'';$$
- lo **stato iniziale** è formato dai due stati iniziali base, ovvero
$$q_0 = (q_0', q_0'');$$
- l'**alfabeto** è l'unione dei due alfabeti di base, ovvero
$$\Sigma = \Sigma' \cup \Sigma'';$$
- la **funzione di transizione**, in base al carattere che legge, deve mandare avanti uno dei due automi e mantenere l'altro nello stesso stato, ovvero
$$\delta((q,p),x) = \begin{cases} (\delta'(q,x),p) & \text{se } x \in \Sigma' \\ (q,\delta''(p,x)) & \text{se } x \in \Sigma'' \end{cases};$$
- gli **stati finali** sono tutte le coppie formate da stati finali, perché devo riconoscere sia la prima che la seconda stringa, ovvero
$$F = \{(q,p) \mid q \in F' \wedge p \in F''\}.$$

Il numero di stati di questo automa è il prodotto del numero di stati dei due automi, ovvero

$$\text{sc}(\text{shuffle}(L', L'')) = \text{sc}(L') \cdot \text{sc}(L'').$$

Abbiamo considerato solo il caso in cui A' e A'' sono DFA. Per il caso non deterministico, basta modificare leggermente la funzione di transizione ma il numero di stati rimane invariato.

14.4.2.2. Stesso alfabeto

Se invece i due linguaggi sono definiti sullo stesso alfabeto Σ come ci comportiamo? Dobbiamo fare affidamento sul **non determinismo**: dobbiamo scommettere se il carattere che abbiamo letto deve mandare avanti il primo automa o il secondo automa. Tra tutte le possibili computazioni ce ne deve essere una che termina in una coppia di stati entrambi finali. Se nessuna computazione termina in uno stato accettabile allora rifiutiamo la stringa data.

Se partiamo da due DFA otteniamo un NFA con un numero di stati uguale al prodotto degli stati dei due automi iniziali. Questa situazione non ci piace, quindi torniamo in un DFA facendo un salto esponenziale con la costruzione per sottoinsiemi, quindi

$$\text{sc}(\text{shuffle}(L', L'')) \leq 2^{\text{nsc}(\text{shuffle}(L', L''))}.$$

Invece, se partiamo da due NFA otteniamo ancora un NFA, quindi la situazione ci piace. Il numero di stati l'abbiamo già definito ed è uguale a

$$\text{nsc}(\text{shuffle}(L', L'')) = \text{nsc}(L') \cdot \text{nsc}(L'').$$

14.4.2.3. Alfabeto unario

Infine, se i due linguaggi sono definiti sull'**alfabeto unario**, ovvero

$$L', L'' \subseteq \{a\}^*$$

l'operazione di shuffle collassa banalmente l'operazione di **prodotto**, perché alla fine stiamo facendo shuffle su stringhe che sono formate sempre da una e una sola lettera, quindi non conta come le mischiamo ma conta la lunghezza finale della stringa che ci esce.