

Teoria dei Linguaggi

Indice

1. Lezione 01 [26/02]	5
1.1. Cosa faremo	5
1.2. Storia	5
1.3. Ripasso	5
1.4. Gerarchia di Chomsky	6
2. Lezione 02 [28/02]	7
2.1. Grammatiche	7
2.1.1. Regole di produzione	7
2.1.2. Linguaggio generato da una grammatica	7
2.2. Gerarchia di Chomsky	8
3. Lezione 03 [05/03]	10
3.1. Gerarchia	10
3.2. Decidibilità	10
3.3. Parola vuota	12
3.4. Linguaggi non esprimibili tramite grammatiche finite	12
4. Lezione 04 [07/03]	15
4.1. Linguaggi regolari	15
4.1.1. Macchine a stati finiti deterministiche	15
4.1.2. Macchine a stati finiti non deterministiche	18
4.1.3. Confronto tra DFA e NFA	19
4.1.4. Altre forme di non determinismo	20
5. Lezione 05 [12/03]	21
5.1. Distinguibilità	21
5.2. Linguaggio L_n	22
5.3. Automa di Meyer-Fischer	25
6. Lezione 06 [14/03]	28
6.1. Molti esempi	28
6.2. Fooling set	31
7. Lezione 07 [19/03]	35
7.1. Introduzione matematica	35
7.2. Automa minimo	37
7.2.1. Relazione R_M	37
7.2.2. Relazione R_L	38
7.2.3. E gli NFA?	42
8. Lezione 08 [21/03]	43
8.1. Altre forme di non determinismo	43
8.2. Relazione tra i linguaggi e le grammatiche di tipo 3	45
8.2.1. Dall'automa alla grammatica	45
8.2.2. Dalla grammatica all'automa	46
8.3. Grammatiche lineari	47
8.3.1. Lineari a destra	47
8.3.2. Lineari a sinistra	48
8.3.3. Lineari	48
8.4. Operazioni sui linguaggi	48

8.4.1. Operazioni insiemistiche	48
8.4.2. Operazioni tipiche	49
8.4.3. Teorema di Kleene e espressioni regolari	51
9. Lezione 09 [26/03]	55
9.1. Fine dimostrazione	55
9.2. State complexity	55
9.3. Operazioni	56
9.3.1. Complemento	57
9.3.1.1. DFA	57
9.3.1.2. NFA	57
9.3.1.3. Costruzione per sottoinsiemi	58
9.3.2. Unione	59
9.3.2.1. DFA	60
9.3.2.2. Automa prodotto	60
9.3.2.3. NFA	62
9.3.3. Intersezione	62
9.3.4. Prodotto	62
9.3.4.1. DFA	63
9.3.4.2. Costruzione senza nome	63
9.3.4.3. NFA	64
10. Lezione 10 [28/03]	65
10.1. Prodotto (richiamo)	65
10.2. Chiusura di Kleene	65
10.2.1. DFA	66
10.2.2. NFA	66
10.2.3. Esempi utili per dopo	66
10.2.4. Codici	67
10.2.5. Star height	68
10.3. Espressioni regolari estese	69
10.4. Operazioni esotiche	72
10.4.1. Reversal	72
10.4.1.1. DFA	73
10.4.1.2. NFA	74
10.4.2. Shuffle	74
10.4.2.1. Alfabeti disgiunti	74
10.4.2.2. Stesso alfabeto	75
10.4.2.3. Alfabeto unario	75
11. Lezione 11 [02/04]	76
11.1. Reversal	76
11.2. Algoritmo per automa minimo	77
11.3. Pumping Lemma	81
12. Lezione 12 [04/04]	85
12.1. Problemi di decisione per i linguaggi regolari	85
12.1.1. Linguaggio vuoto e infinito	85
12.1.2. Appartenenza	86
12.1.3. Universalità	86

12.1.4. Inclusione e uguaglianza	86
12.2. Altre operazioni	86
12.2.1. Raddoppio	86
12.2.2. Metà	87
12.3. Automi come matrici	88
12.3.1. Prima applicazione: raddoppio	89
12.3.2. Seconda applicazione: metà	90
12.3.3. Matrici su NFA	90
12.4. Varianti di automi	91
12.4.1. Automi pesati	91
12.4.2. Automi probabilistici	91
13. Lezione 13 [09/04]	93
13.1. Varianti di automi	93
13.1.1. One-way VS two-way	93
13.1.2. Read-only VS read-write	93
13.1.3. Memoria esterna	94
13.2. Automi two-way	95
13.2.1. Esempi vari	95
13.2.2. Definizione formale	97
13.2.3. Potenza computazionale	98
14. Lezione 14 [11/04]	103
14.1. Simulazione	103
14.1.1. Problema di Sakoda & Sipser	103
14.2. Automi a pila	105
14.2.1. Versione non deterministica	105
14.2.2. Accettazione	108
14.2.3. Determinismo VS non determinismo	110
14.2.4. Trasformazioni	111

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 descrittiva 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 $x = a_1 \dots a_n$ 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 dopponi.
- **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;
- **ricognoscitivo:** 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 derivare 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.
- **ricognoscitivo:** 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 $\alpha \rightarrow \beta$, 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 $\exists(\alpha \rightarrow \beta) \in P$ 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 $x \xRightarrow{k} y$, se e solo se $\exists x_0, \dots, x_k \in (V \cup \Sigma)^*$ 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:

- $x \xRightarrow{*} y$ per indicare un numero generico di passi, e questo vale se e solo se $\exists k \geq 0$ tale che $x \xRightarrow{k} y$;
- $x \xRightarrow{+} y$ per indicare che serve almeno un passo, e questo vale se e solo se $\exists k > 0$ tale che $x \xRightarrow{k} y$.

2.1.2. Linguaggio generato da una grammatica

Indichiamo con $L(G)$ il linguaggio generato dalla grammatica G , ed è l'insieme $\{w \in \Sigma^* \mid S \xRightarrow{*} w\}$. 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:

- $S \rightarrow \varepsilon$;
- $S \rightarrow (S)$;
- $S \rightarrow SS$.

Vediamo un esempio più complesso. Siano:

- $\Sigma = \{a, b\}$;
- $V = \{S, A, B\}$;
- $P = \{S \rightarrow aB \mid bA, A \rightarrow a \mid aS \mid bAA, B \rightarrow b \mid bS \mid aBB\}$.

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:

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

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

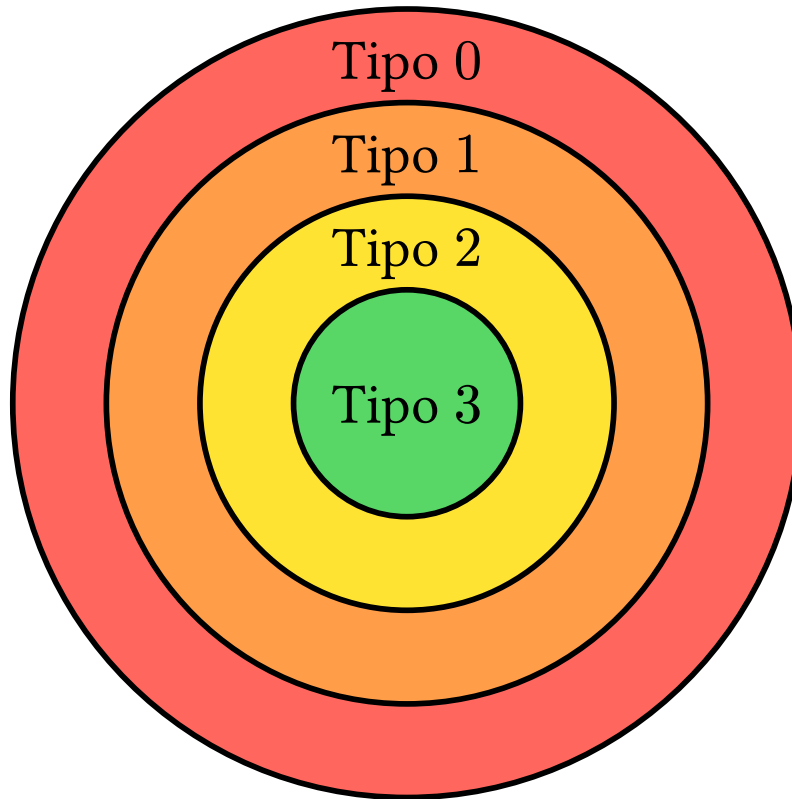
Negli anni '50 Noam Chomsky studia la generazione dei linguaggi formali e crea una **gerarchia di grammatiche formali**. La classificazione delle grammatiche viene fatta in base alle regole di produzione che definiscono la grammatica.

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 $(\alpha \rightarrow \beta) \in P$ allora $ \beta \geq \alpha $, ovvero devo generare parole che non siano più corte di quella di partenza. Sono dette dipendenti dal contesto perché ogni regola $(\alpha \rightarrow \beta) \in P$ può essere riscritta come $\alpha_1 A \alpha_2 \rightarrow \alpha_1 B \alpha_2$, con $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$ che rappresentano il contesto, $A \in V$ e $B \in (V \cup \Sigma)^+$	Automati limitati linearmente
Tipo 2 , dette context-free o libere dal contesto	Le regole in P sono del tipo $\alpha \rightarrow \beta$, con $\alpha \in V$ e $\beta \in (V \cup \Sigma)^+$.	Automati a pila
Tipo 3 , dette grammatiche regolari	Le regole in P sono del tipo $A \rightarrow aB$ oppure $A \rightarrow a$, con $A, B \in V$ e $a \in \Sigma$. Vale anche il simmetrico.	Automati 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 è di tipo 1 allora possiamo costruire una macchina che sia in grado di dire, in tempo finito, se una parola appartiene o meno al linguaggio generato da quella grammatica. Questa macchina è detta **verificatore** e si dice che le grammatiche di tipo 1 sono **decidibili**.

3. Lezione 03 [05/03]

3.1. 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.

3.2. Decidibilità

Teorema 3.2.1 (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 3.2.1.1: 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

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

Calcoliamo induttivamente questi insiemi.

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

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

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

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

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

Per come abbiamo definito gli insiemi, sappiamo che

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

e l'ultima inclusione è vera perché ho fissato la lunghezza massima, non voglio considerare di più perché 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 3.2.2 (Semi-decidibilità dei linguaggi di tipo 0): I linguaggi di tipo 0 sono ricorsivamente enumerabili.

Dimostrazione 3.2.2.1: 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

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

Calcoliamo induttivamente questi insiemi.

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

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

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

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

Noi partendo da U_0 calcoliamo tutti i vari insiemi ottenendo una serie di U_i .

Per come abbiamo definito gli insiemi, sappiamo che

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

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

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

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

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

Ma allora G è semi-decidibile. ■

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

3.3. Parola vuota

Vediamo il problema della **parola vuota**: nelle grammatiche di tipo 2 abbiamo 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:
 - $P_1 = P \cup \{S_1 \rightarrow \alpha \mid (S \rightarrow \alpha) \in P\} \cup \{S_1 \rightarrow \varepsilon\}$;
 - $P_1 = P \cup \{S_1 \rightarrow S\} \cup \{S_1 \rightarrow \varepsilon\}$;
- 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 $S \rightarrow \varepsilon$ purché S non appaia mai sul lato destro delle produzioni;
- tipo 2 permettiamo direttamente $A \rightarrow \beta$ 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**.

3.4. 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 3.4.1: Vale

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

Dimostrazione 3.4.1.1: 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 $\mathbb{N} \approx 2^{\mathbb{N}}$. ■

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 3.4.2: Esistono linguaggi che non sono descrivibili da grammatiche finite.

Dimostrazione 3.4.2.1: 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

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

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

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

4. Lezione 04 [07/03]

4.1. Linguaggi regolari

4.1.1. 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 4.1.1.1: Sia $A = (Q, \Sigma, \delta, q_0, F)$ tale che:

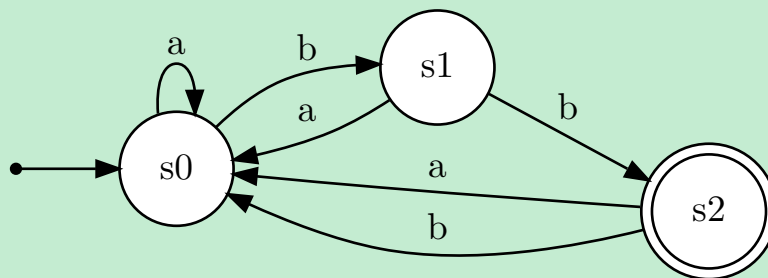
- $Q = \{s_0, s_1, s_2\}$;

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

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

	a	b
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_0

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



Il linguaggio che riconosce questo automa è

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

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

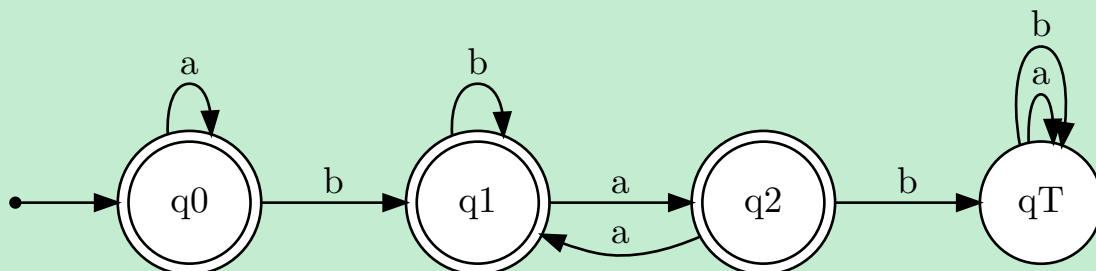
$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}.$$

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 4.1.1.2: 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 .



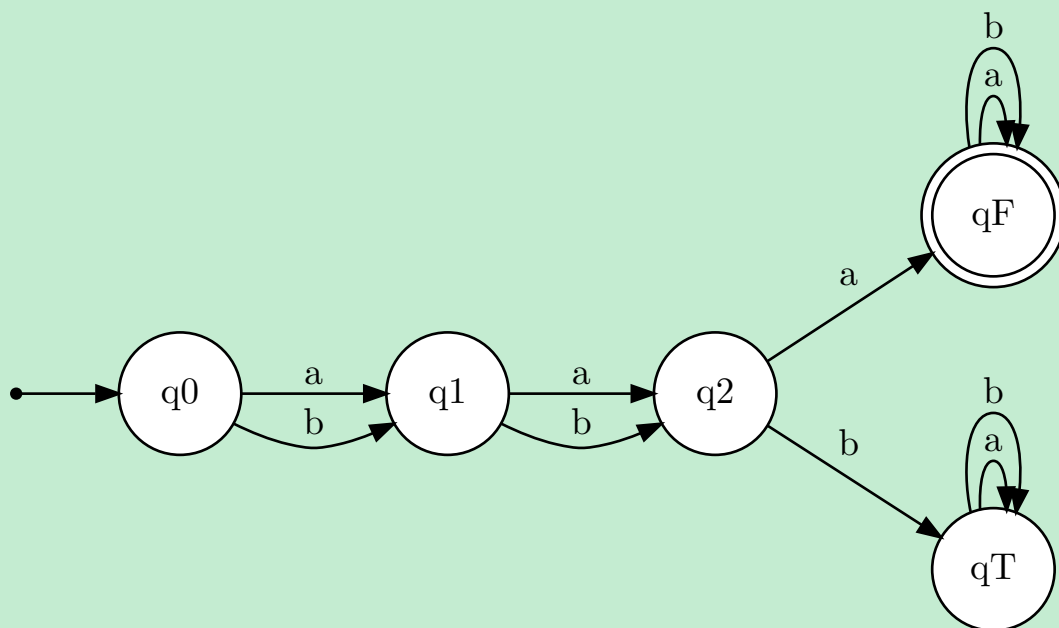
Come vediamo dall'esempio precedente, abbiamo uno stato particolare q_T che è detto **stato trappola**: esso viene utilizzato come «punto di arrivo» per esaurire la lettura dell'input e non accettare la stringa data in input. Finiamo in questo stato se, in uno stato q , leggiamo un carattere che rende 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 4.1.1.3: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

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

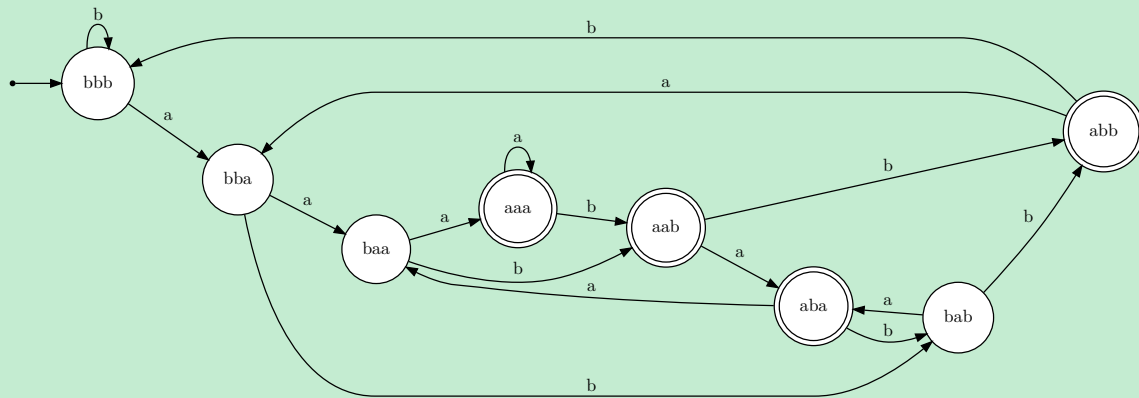
Costruiamo un automa deterministico per L .



Esempio 4.1.1.4: Sia $\Sigma = \{a, b\}$, vogliamo trovare un automa per il linguaggio

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

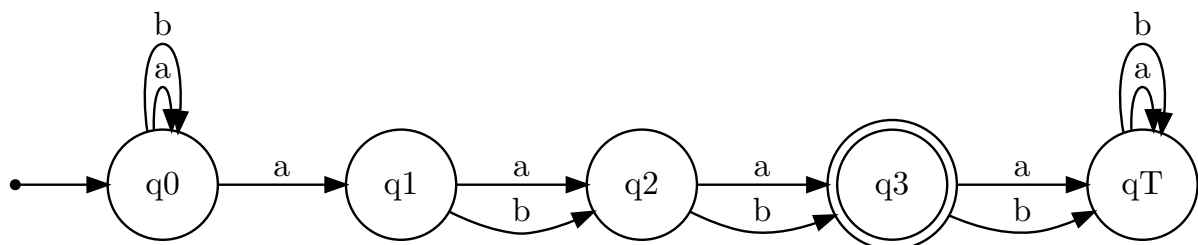
Costruiamo un automa deterministico per L . Qua l'idea è ricordarsi una finestra di 3 simboli e grazie a questa vediamo se il primo carattere che definisce lo stato è una a .



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

4.1.2. 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 .

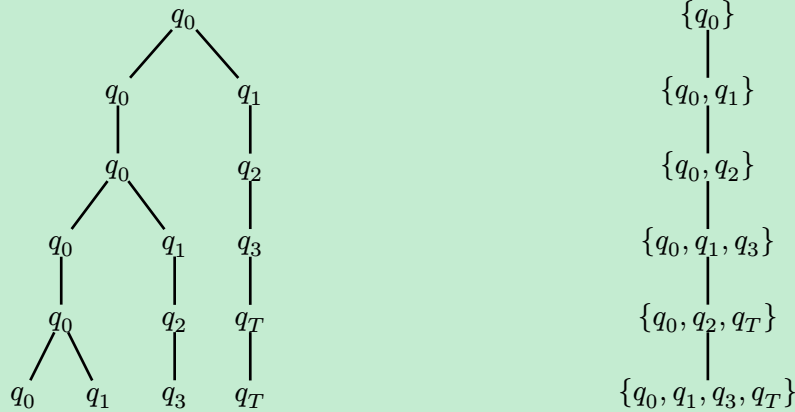
Gli **automi non deterministici**, o **NFA**, sono definiti da una quintupla $A = (Q, \Sigma, \delta, q_0, F)$ definita allo stesso modo dei DFA tranne la funzione di transizione. Essa è la funzione

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

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



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

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

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

definita induttivamente come

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

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

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

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

4.1.3. 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;

- $\delta' : Q' \times \Sigma \rightarrow Q'$ è la nuova funzione di transizione che ci permette di navigare tra i possibili sottoinsiemi, ed è tale che

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

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

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

4.1.4. Altre forme di non determinismo

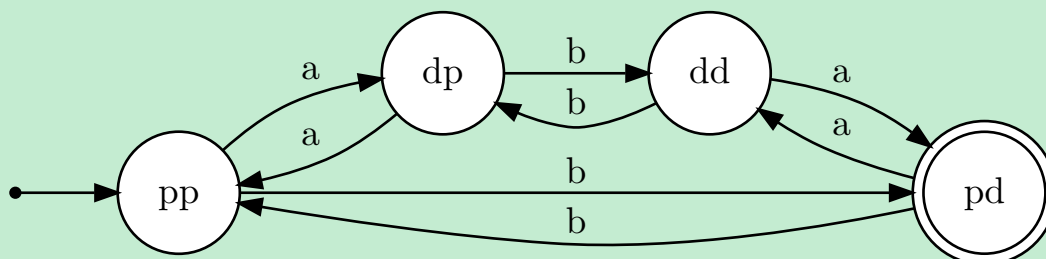
Una ulteriore forma di non determinismo, oltre a quella sulle molteplici transizioni con lo stesso carattere in uno stato, è quella di avere **molteplici stati iniziali**.

5. Lezione 05 [12/03]

5.1. Distinguibilità

Esempio 5.1.1: Sia $\Sigma = \{a, b\}$ e vogliamo un automa che riconosca il linguaggio

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



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

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

Definizione 5.1.1 (Distinguibilità): Sia $L \subseteq \Sigma^*$ un linguaggio. Date $x, y \in \Sigma^*$, allora esse sono **distinguibili** per L se

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

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

Teorema 5.1.1 (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 5.1.1.1: Sia $X = \{x_1, \dots, x_n\}$ e sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA che accetta il linguaggio L . Definiamo gli stati

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

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

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

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

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

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

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

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

Esempio 5.1.2: 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**.

5.2. Linguaggio L_n

Esempio 5.2.1: Riprendiamo il linguaggio della scorsa lezione e diamogli un nome. Dato l'alfabeto $\Sigma = \{a, b\}$, sia

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

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

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

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

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

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

possiamo affermare che:

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

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

Cerchiamo di generalizzare questo concetto.

Esempio 5.2.2: 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 $\sigma = \sigma_1 \dots \sigma_n$.

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

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

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

Scelgo di attaccare una stringa

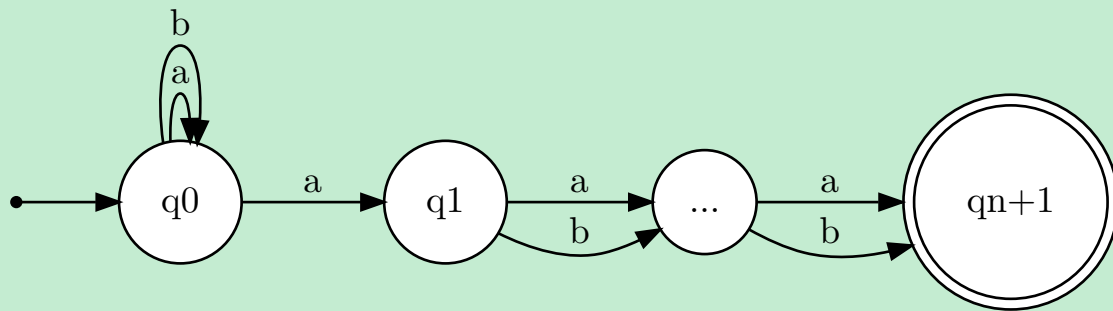
$$z \in \Sigma^{i-1}$$

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

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

Cosa cambia se invece utilizziamo un NFA per L_n ?

Esempio 5.2.3: 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.

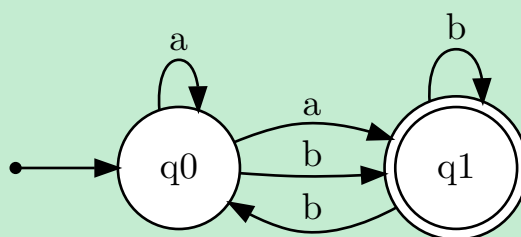


Il numero totale di stati è $n + 1$.

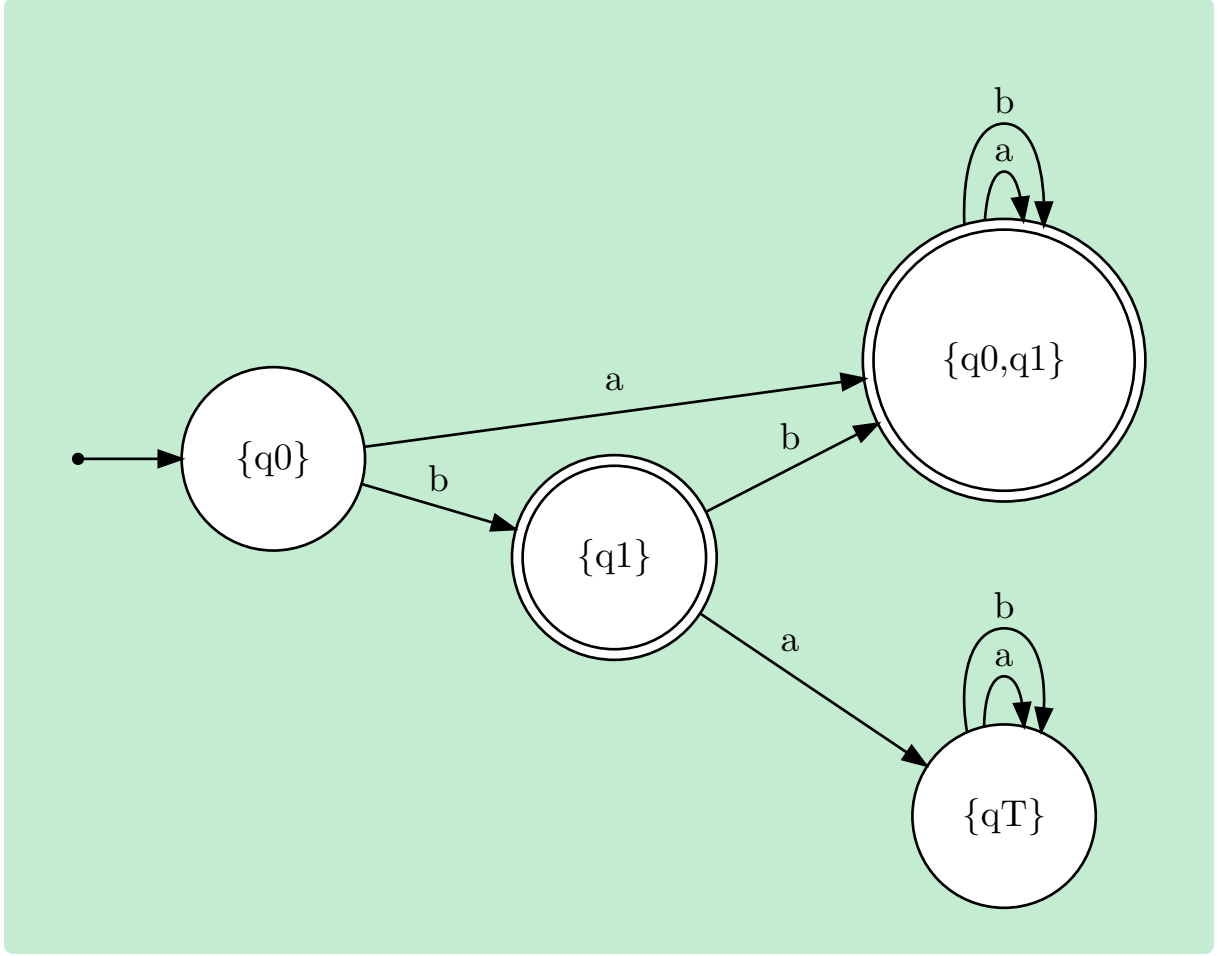
Per L_n abbiamo quindi visto che il numero di stati richiesti per un NFA è $n + 1$, mentre per un DFA è almeno 2^n grazie al teorema sulla distinguibilità. Il salto che abbiamo fatto è quindi **esponenziale**.

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

Esempio 5.2.4: 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 $n \rightarrow 2^n$, quindi vuol dire che forse si riesce a fare meglio. E invece **NO**. Esiste un caso peggiore, un automa che esegue un salto preciso da n a 2^n preciso preciso.

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

5.3. 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.

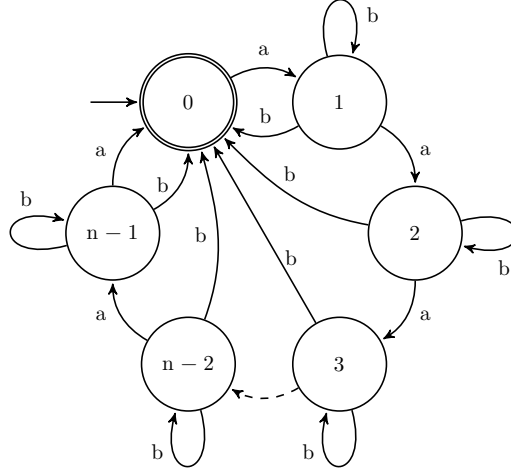
Sia $M_n = (Q, \Sigma, \delta, q_0, F)$ tali che:

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

La funzione di transizione è tale che

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

L'automa M_n lo possiamo disegnare in questo modo.



Teorema 5.3.1: Ogni DFA equivalente a M_n deve avere almeno 2^n stati.

Dimostrazione 5.3.1.1: Sia $S \subseteq \{0, \dots, n-1\}$. Definiamo la stringa

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

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

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

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

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

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

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

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

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

Lemma 5.3.1: Per ogni $S \subseteq \{0, \dots, n-1\}$ vale

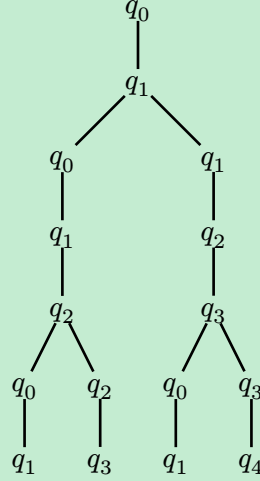
$$\delta(q_0, w_S) = S.$$

Esempio 5.3.1: Sia M_5 una istanza dell'automa di Meyer-Fischer.

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

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

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



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

E ora vediamo la seconda e ultima proprietà.

Lemma 5.3.2: Dati $S, T \subseteq \{0, \dots, n-1\}$, se $S \neq T$ allora w_S e w_T sono distinguibili per il linguaggio $L(M_n)$.

Dimostrazione 5.3.2.1: 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. ■

6. Lezione 06 [14/03]

6.1. 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 6.1.1: 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.

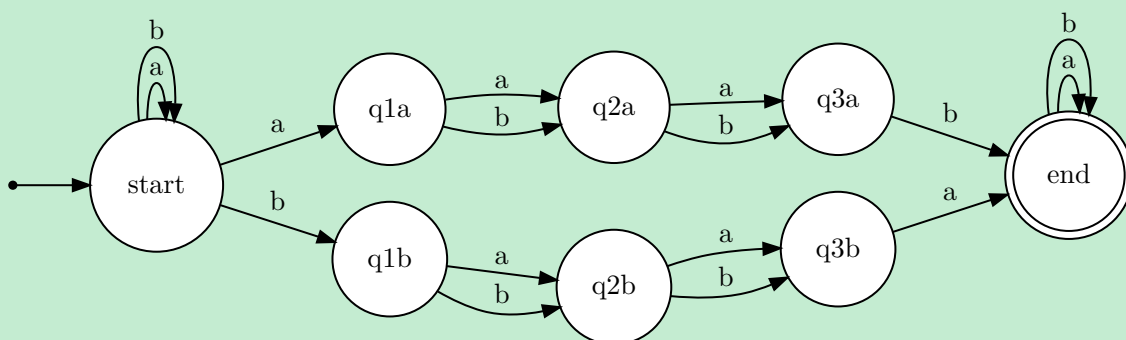
Visto che siamo bravi con le scommesse, andiamo a fare un po' di sano **gambling**.

Esempio 6.1.2: 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 6.1.3: Dato L_n il linguaggio di prima, sia $X = \Sigma^n$.

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

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

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

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

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

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

Esempio 6.1.4: Dato l'alfabeto $\Sigma = \{a, b\}$, definiamo

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

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

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

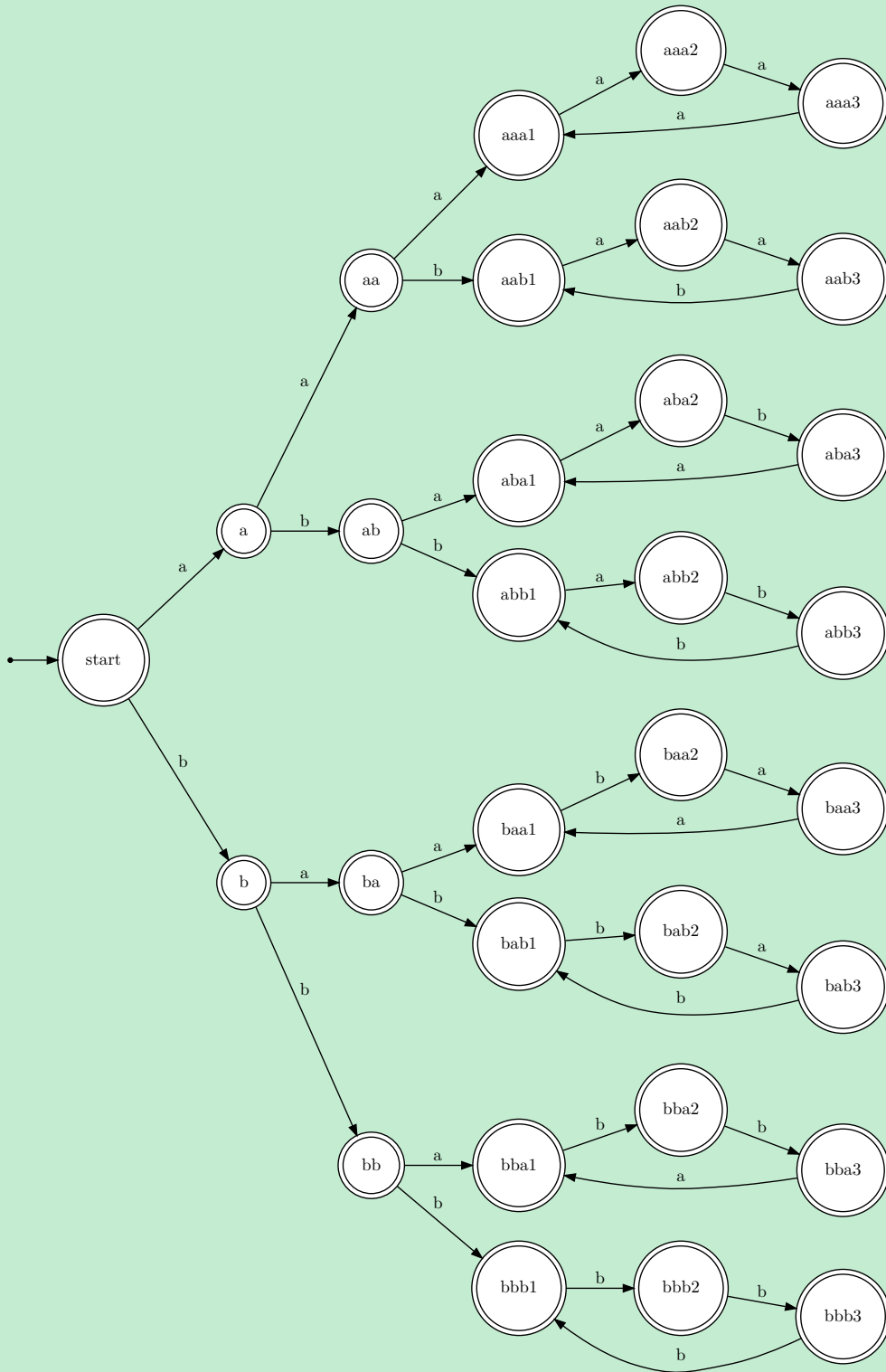
Posso ripetere w quante volte voglio, ma poi la parte finale deve ripetere in parte w .

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

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

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

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

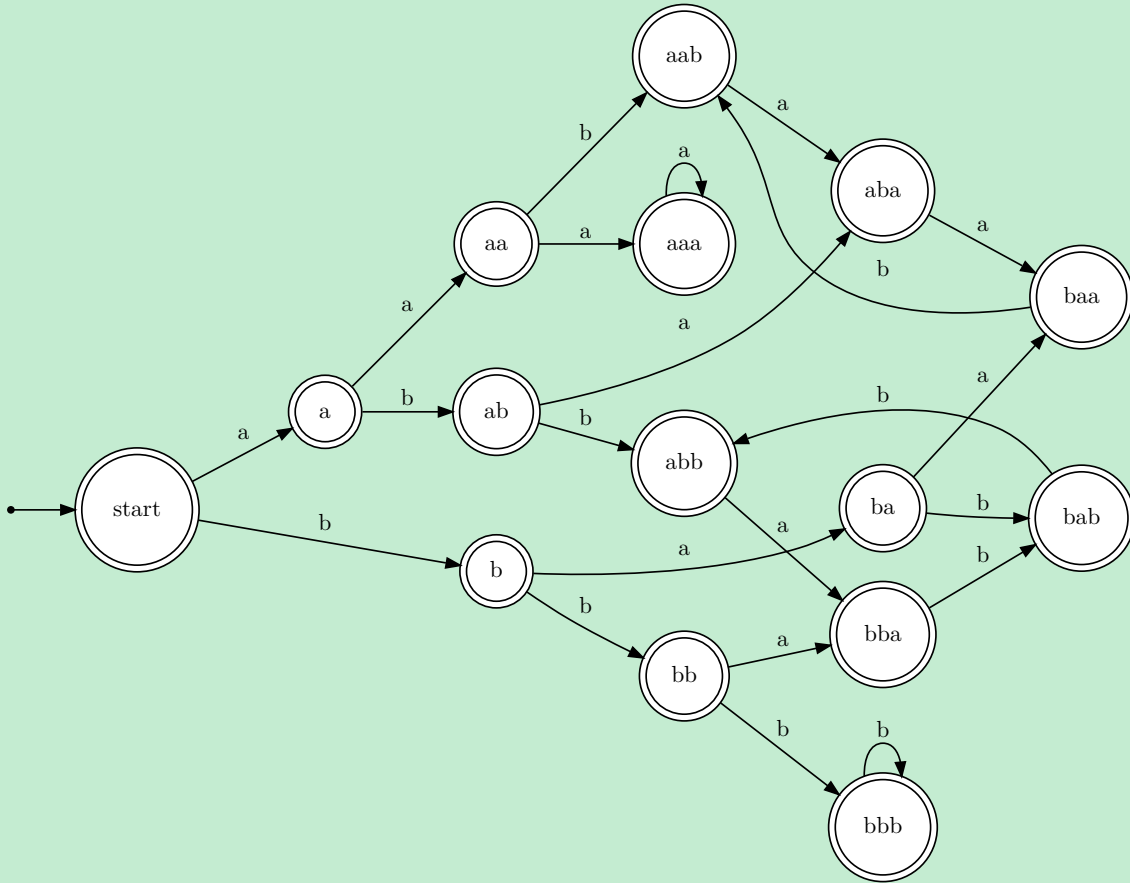


Per il linguaggio generico L'_n , l'albero usa un numero di stati pari a

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

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

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



Questa bellissima versione ha un numero di stati pari a

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

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

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».

6.2. Fooling set

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

Definizione 6.2.1 (Fooling set): Sia $L \subseteq \Sigma^*$. Definiamo

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

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

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

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

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

Noi useremo una versione leggermente diversa del fooling set.

Definizione 6.2.2 (Extended fooling set): Un **extended fooling set** è un fooling set nel quale viene modificata la seconda proprietà, ovvero:

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

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

Teorema 6.2.1 (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 6.2.1.1: Concentriamoci solo sui cammini accettanti che possiamo avere in un NFA per il linguaggio L . Grazie alla prima proprietà di P , sappiamo che le stringhe $z = x_i y_i$ stanno in L . Calcoliamo i cammini per ogni coppia di P , che sono N :

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

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

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

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

Quindi ogni NFA deve avere almeno N stati. ■

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

Esempio 6.2.1: 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.

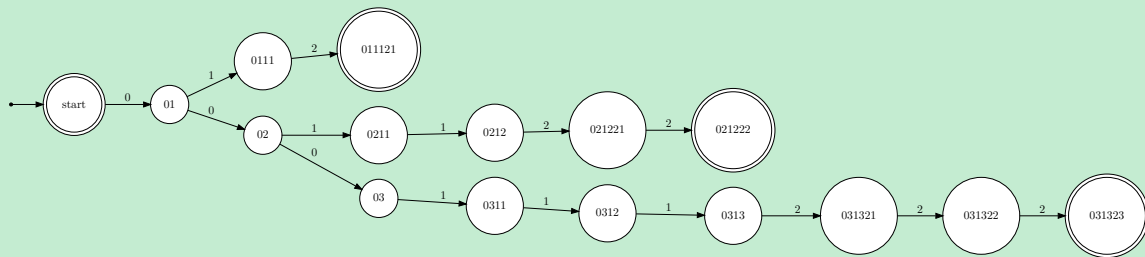
Linguaggio	DFA	NFA
L_n	$\geq 2^n$	$\leq 2n + 2$
L'_n	$\geq 2^n \wedge \leq 2^{n+1}$	$\geq 2^n$

Finiamo con un ultimo esempio.

Esempio 6.2.2: 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

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

Questo è un fooling set per L_n :

- una coppia ci dà la stringa $z = 0^i 1^{j+i-j} 2^i = 0^i 1^i 2^i$ che appartiene al linguaggio;
- prendendo due elementi da due coppie diverse:
 - se sono diverse le i abbiamo un numero di 0 e 2 diversi;
 - se sono uguali le i allora sono diverse le j , ma allora la stringa $0^i 1^{j+i-j'} 2^i$ non appartiene al linguaggio perché $j + i - j' \neq i$.

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

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

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

7. Lezione 07 [19/03]

7.1. 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 7.1.1: 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 $\forall x, y \in S \quad x R y \implies y R x$;
- **transitiva** $\forall x, y, z \in S \quad x R y \wedge y R z \implies x R z$.

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

$$x R y \iff x \bmod 3 = y \bmod 3.$$

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 7.1.2: 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

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

Esempio 7.1.2: Sia R la relazione dell'esempio precedente. Definiamo $\cdot = +$ l'operazione di somma. La relazione R è invariante a destra?

Dobbiamo verificare se

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

ovvero se

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

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 7.1.3: 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

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

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

$$x R' y \iff x \bmod 2 = y \bmod 2.$$

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

$$x R'' y \iff x \bmod 6 = y \bmod 6.$$

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

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

7.2. Automa minimo

7.2.1. Relazione R_M

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

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

tale che

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

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

Lemma 7.2.1.1: La relazione R_M è una relazione di equivalenza.

Dimostrazione 7.2.1.1.1: Facciamo vedere che R_M rispetta RST.

La relazione R_M è riflessiva: banale per la riflessività dell'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 7.2.1.2: La relazione R_M è invariante a destra rispetto all'operazione di concatenazione.

Dimostrazione 7.2.1.2.1: Dobbiamo dimostrare che

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

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. ■

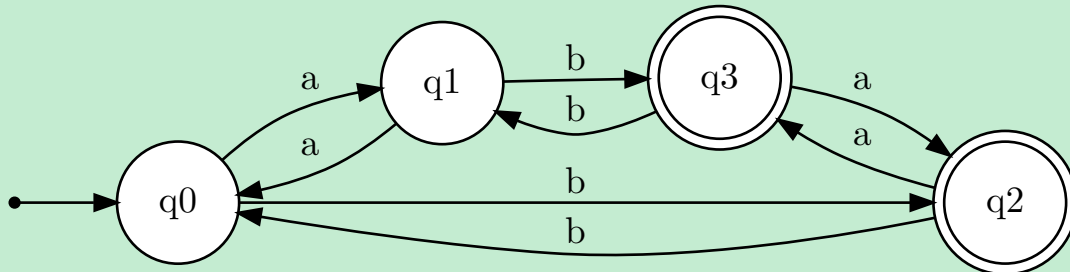
Quante classi di equivalenza abbiamo? Al massimo il numero di stati dell'automa. Come mai diciamo **AL MASSIMO** e non esattamente il numero di stati? Perché in un DFA potremmo avere degli stati che sono irraggiungibili e che quindi non vanno a creare nessuna classe di equivalenza.

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 7.2.1.1: 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.

7.2.2. Relazione R_L

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

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

tale che

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

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 7.2.2.1: La relazione R_L è una relazione di equivalenza.

Dimostrazione 7.2.2.1.1: 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 7.2.2.2: La relazione R_L è invariante a destra rispetto all'operazione di concatenazione.

Dimostrazione 7.2.2.2.1: Dobbiamo dimostrare che

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

Se $(x R_L y)$ allora

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

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 7.2.2.1: 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 7.2.2.1 (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 7.2.2.1.1: Facciamo vedere $1 \implies 2 \implies 3 \implies 1$.

[$1 \implies 2$]

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.

[$2 \implies 3$]

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

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

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 \Rightarrow 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 dalle 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 7.2.2.2 (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.

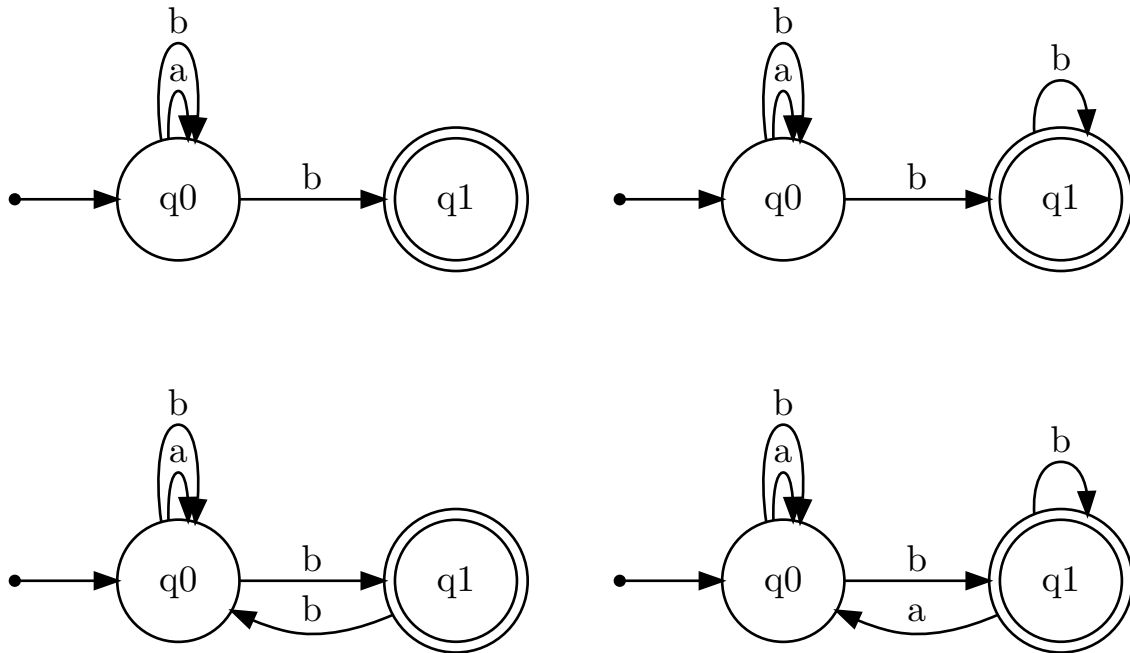
L'automa minimo M' è ottenuto grazie alla relazione R_L .

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.

7.2.3. E gli NFA?

Cosa succede se applichiamo tutti 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.

8. Lezione 08 [21/03]

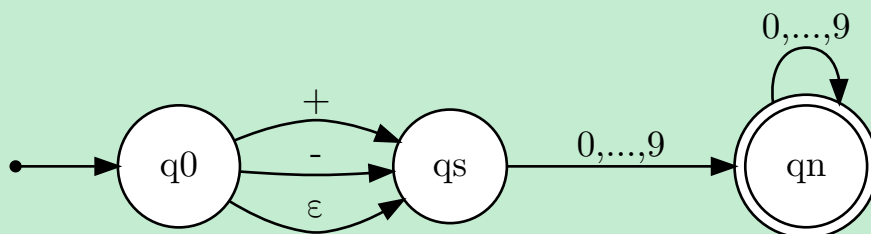
8.1. Altre forme di non determinismo

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



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

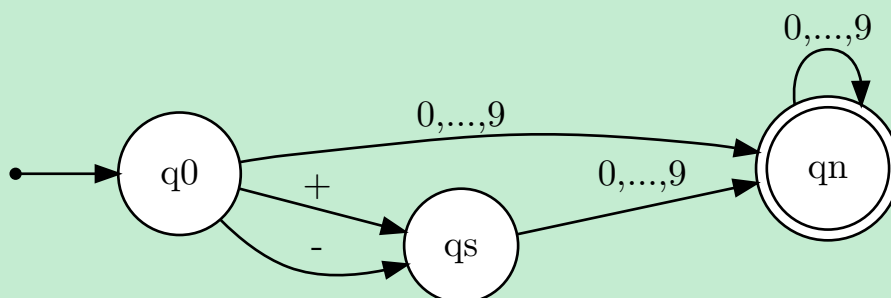
Questa soluzione aumenta la potenza dell'automa? **NO**: ogni sequenza nella forma

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

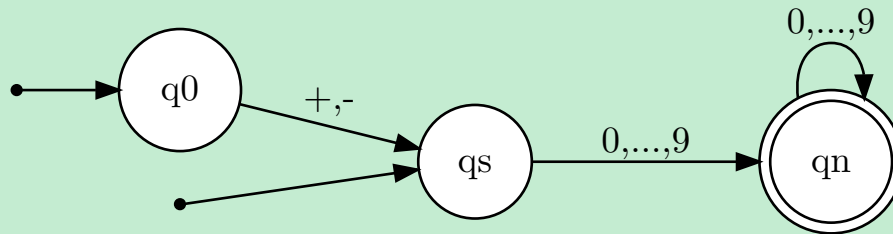
può essere tradotta nella transizione

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

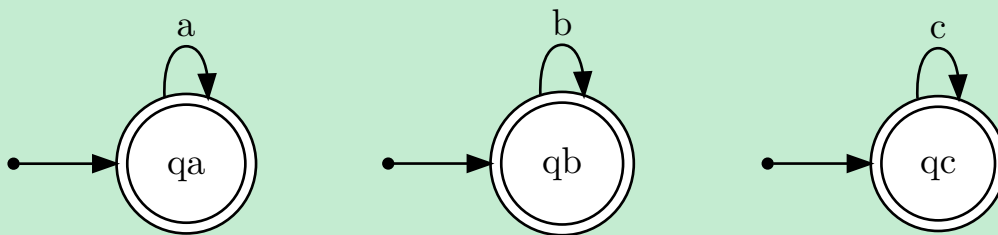
Esempio 8.1.2: Andiamo a rimuovere la ε -transizione usando le sequenze appena descritte.



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.

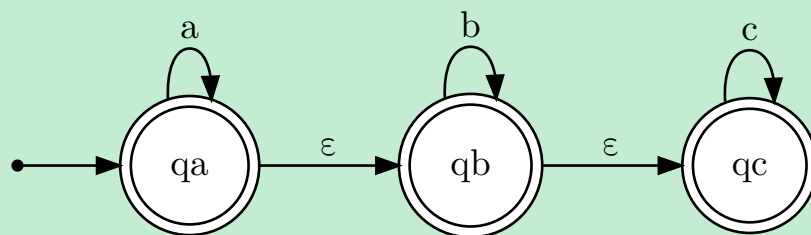


Esempio 8.1.3: Ci vengono dati tre automi, che riconoscono sequenze di a , b e c arbitrarie.

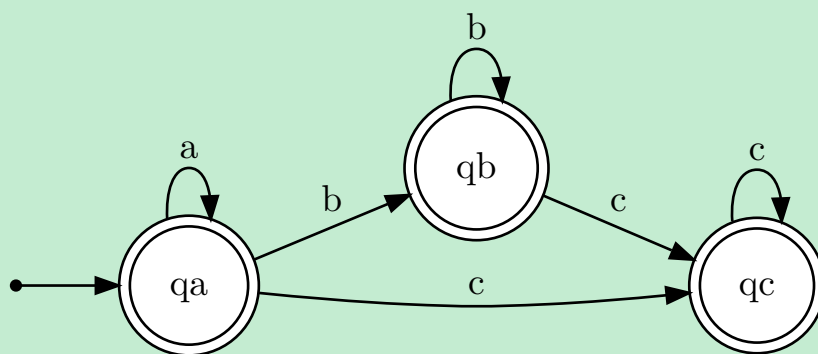


Vogliamo costruire un automa che utilizzi le ε -transizioni usando questi tre moduli per riconoscere il linguaggio

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

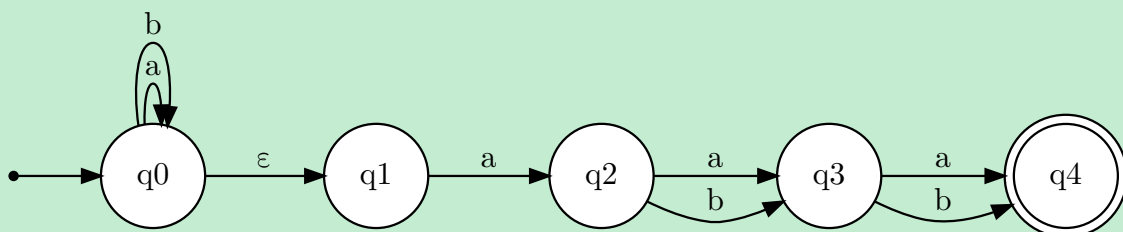


Come lo rendiamo deterministico? Sicuramente non andiamo ad utilizzare gli stati iniziali multipli, che qui ci starebbero molto bene, ma appunto vogliamo un comportamento deterministico.



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

Esempio 8.1.4: Riprendiamo il linguaggio L_n delle stringhe con l' n -esimo carattere da destra uguale ad una a . Avevamo visto un NFA sulle transizioni, vediamo uno non deterministico sulle ε -transizioni fissando il valore a $n = 3$.



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

8.2. 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

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

8.2.1. Dall'automa alla grammatica

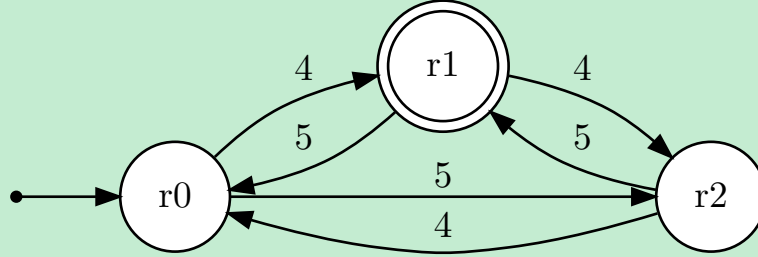
Dato un automa $A = (Q, \Sigma, \delta, q_0, F)$ per il linguaggio L , costruiamo una grammatica G di tipo 3 che riconosca lo stesso linguaggio L .

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:
 - ▶ $q \rightarrow ap$ se la funzione di transizione è tale che $\delta(q, a) = p$;
 - ▶ in più alla produzione precedente aggiungiamo la produzione $q \rightarrow a$ se p è uno stato finale, questo perché posso fermarmi in p .

Esempio 8.2.1.1: 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 :
 - ▶ $r_0 \rightarrow 4r_1 \mid 5r_2$;
 - ▶ $r_1 \rightarrow 4r_2 \mid 5r_0$;
 - ▶ $r_2 \rightarrow 4r_0 \mid 5r_1$.

Proviamo a derivare una stringa per vedere se effettivamente funziona:

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

8.2.2. 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 $(A \rightarrow aB) \in P$ essa ci dice che dallo stato A leggendo una a andiamo a finire in B , ovvero $\delta(A, a) = B$;
 - ▶ per ogni produzione $(A \rightarrow a) \in P$ 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

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

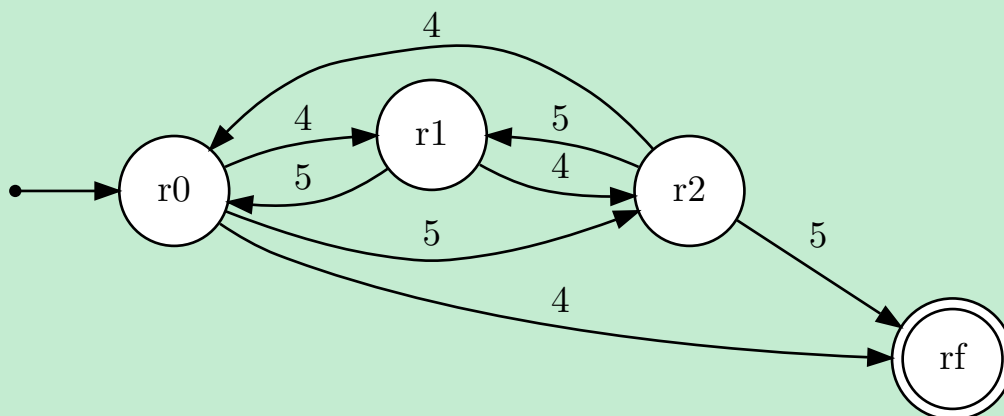
Esempio 8.2.2.1: Data la grammatica $G = (V, \Sigma, P, S)$ tale che:

- $V = \{r_0, r_1, r_2\}$;
- $S = r_0$;

- 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.

8.3. Grammatiche lineari

8.3.1. Lineari a destra

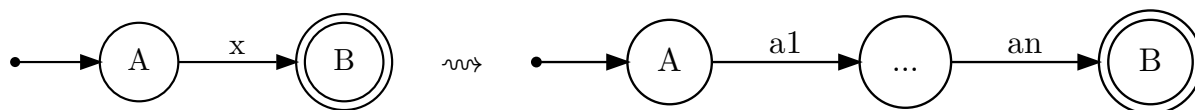
Diamo una mini introduzione alle **grammatiche lineari**, che però vedremo meglio più avanti.

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 $x = a_1 \dots a_n$ con una serie di stati intermedi.

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

8.3.2. Lineari a sinistra

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

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

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

8.3.3. Lineari

E se facciamo un mischione delle due grammatiche precedenti?

Le produzioni di queste grammatiche sono nella forma

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

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 8.3.3.1: Definiamo una grammatica che utilizza le seguenti produzioni:

$$\begin{aligned} S &\rightarrow aA \mid \varepsilon \\ A &\rightarrow Sb. \end{aligned}$$

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.

8.4. 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.

8.4.1. 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**:

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

8.4.2. 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 8.4.2.1: Vediamo due casi particolari e importanti di prodotto

$$\begin{aligned} L' \cdot \emptyset &= \emptyset \cdot L' = \emptyset \\ L' \cdot \{\varepsilon\} &= \{\varepsilon\} \cdot L' = L'. \end{aligned}$$

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

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

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 8.4.2.2: Vediamo una cosa simpatica:

$$\mathbb{Q}^* = \{\varepsilon\}.$$

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

$$\mathbb{Q}^+ = \mathbb{Q}.$$

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 8.4.2.3: 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

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

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

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

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.

8.4.3. 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 8.4.3.1 (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 8.4.3.2 (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 8.4.3.3 (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 8.4.3.1: 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
$L = \{(a^n b)^k \mid n \geq 0 \wedge k > 0\}$	$(a^*b)^k$
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 8.4.3.3.1: 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.

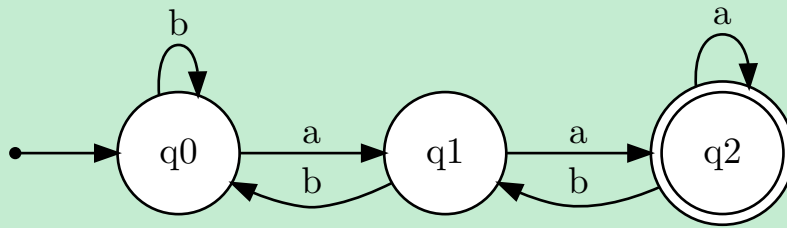
Esempio 8.4.3.2: Per ricavare una espressione regolare da un automa si usa un algoritmo di **programmazione dinamica** molto simile all'algoritmo Floyd-Warshall sui grafi, che cerca i cammini minimi imponendo una serie di vincoli.

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.

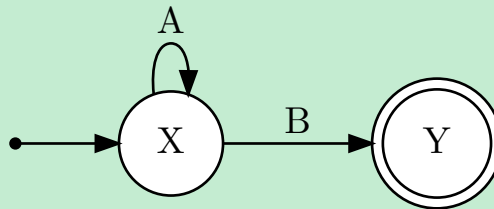
Questa somma di cambiamenti di stati va posta uguale allo stato i -esimo.



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...

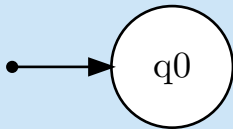
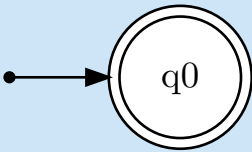
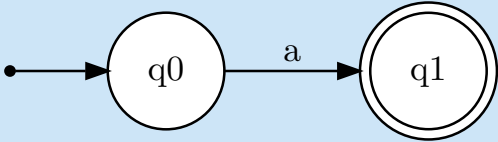
9. Lezione 09 [26/03]

9.1. 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 9.1.1 (Teorema di Kleene): Vedi lezione scorsa.

Dimostrazione 9.1.1.1: 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. ■

9.2. 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 9.2.1 (State complexity deterministica): Sia $L \subseteq \Sigma^*$. Indichiamo con

$$sc(L)$$

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

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

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 9.2.1: Se L non è un linguaggio regolare allora

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

Lemma 9.2.2: Se L è un linguaggio regolare allora

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

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 9.2.1: 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

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

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

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

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

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

9.3. 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.

9.3.1. Complemento

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

9.3.1.1. 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

$$F' = Q/F.$$

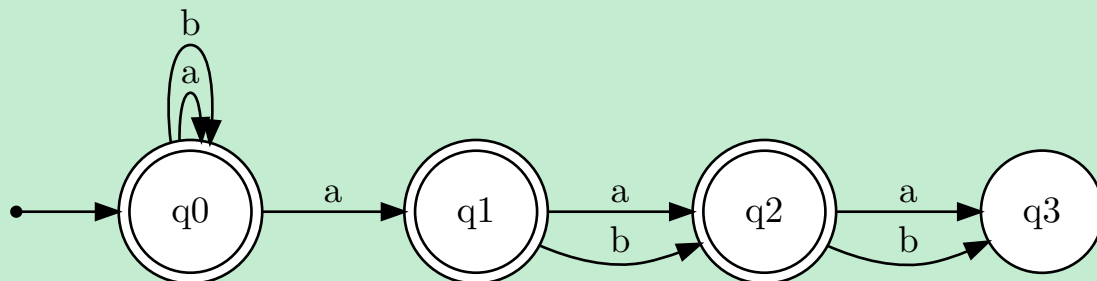
Dobbiamo imporre che A sia **completo** perché ciò che andava nello stato trappola ora deve essere accettato. Ma allora

$$\text{sc}(L^C) = \text{sc}(L).$$

9.3.1.2. NFA

Come ci comportiamo sugli NFA?

Esempio 9.3.1.2.1: 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.

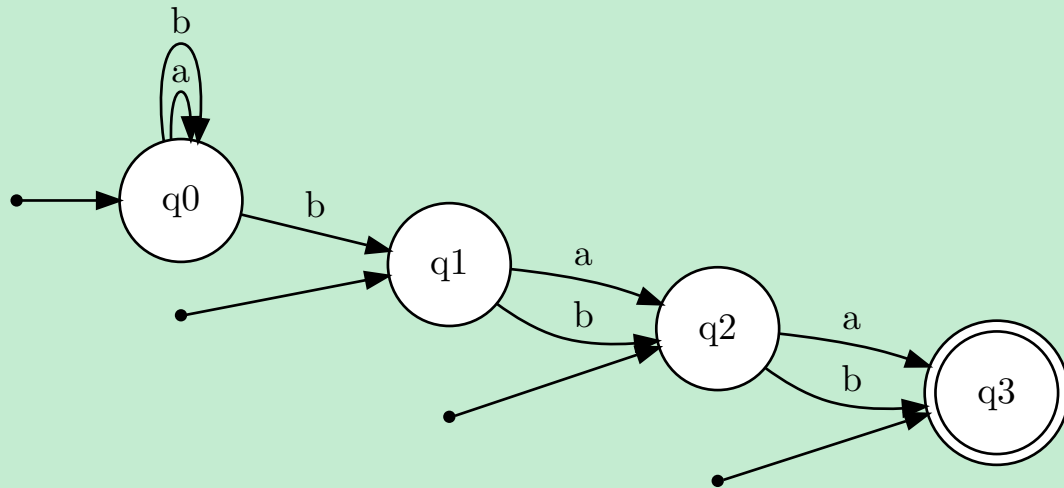


Abbiamo un problema: questo automa **accetta tutto**. Ma perché succede questo? Negli NFA accettiamo se esiste almeno un cammino accettante e rifiutiamo se ogni cammino è rifiutante. Quando accettiamo è molto probabile che ci sia, oltre al cammino accettante, anche qualche cammino rifiutante. Facendo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

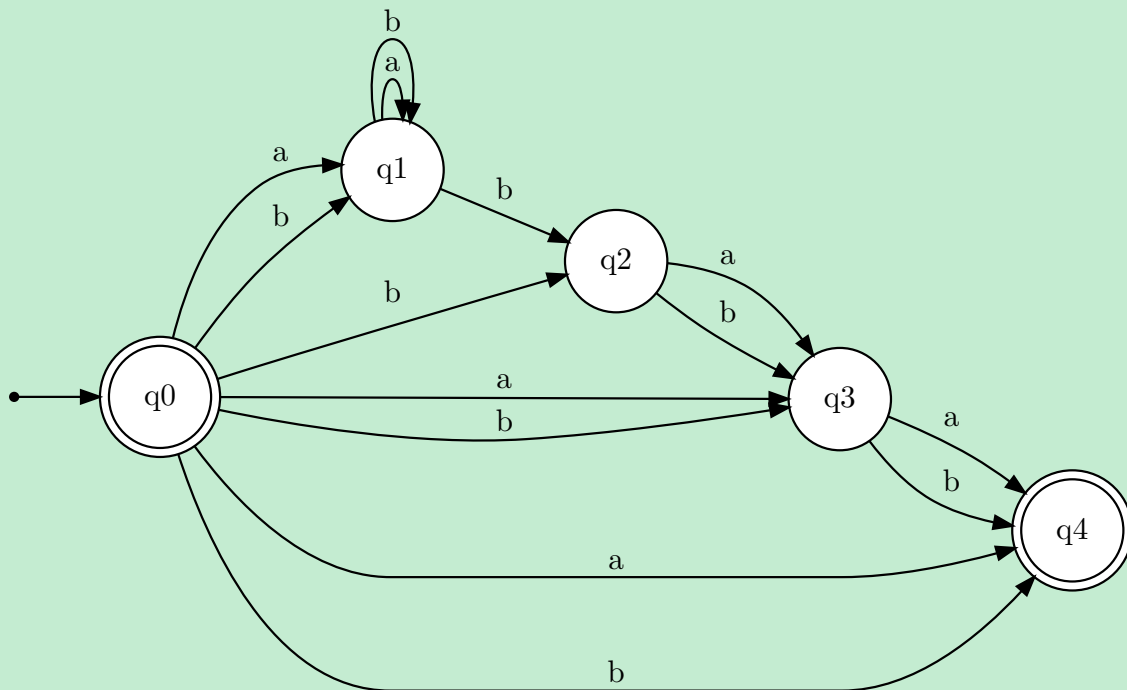
È importantissimo avere il DFA, per via di questa asimmetria tra accettazione e non accettazione.

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 9.3.1.2.2: 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.

9.3.1.3. 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

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

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

$$2n + 2 \rightarrow 2^n,$$

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

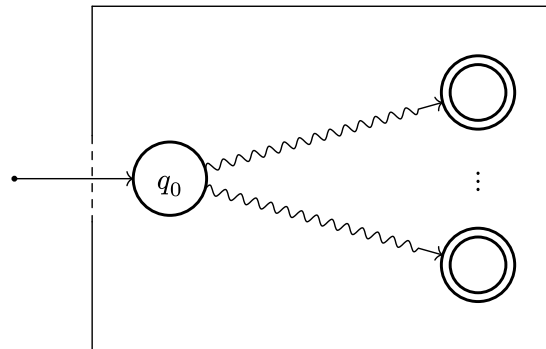
Abbiamo quindi visto che del complemento negli NFA non ce ne facciamo niente, questo proprio per la natura del non determinismo.

9.3.2. Unione

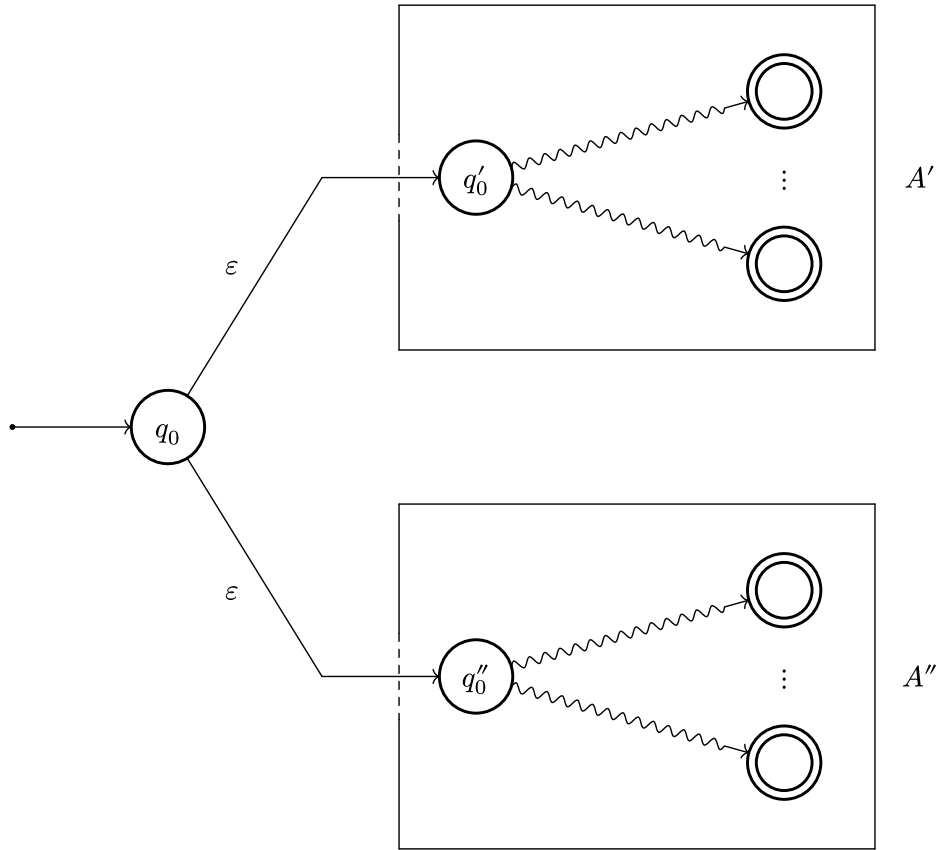
Dati due linguaggi $L', L'' \subseteq \Sigma^*$ rispettivamente riconosciuti dagli automi $A' = (Q', \Sigma, \delta', q'_0, F')$ e $A'' = (Q'', \Sigma, \delta'', q''_0, F'')$, vogliamo costruire un automa per l'**unione**

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

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?

9.3.2.1. 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.

9.3.2.2. 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

$$\text{sc}(L' \cup L'') \leq \text{sc}(L') \cdot \text{sc}(L''),$$

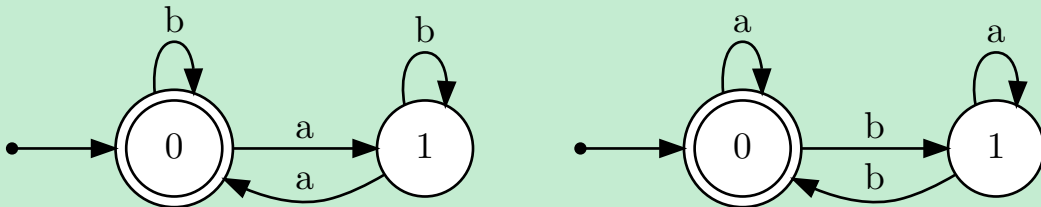
quindi abbiamo una soluzione notevolmente migliore. Inoltre, non si può fare meglio di così.

Esempio 9.3.2.2.1: Fissati due valori m, n positivi, definiamo i linguaggi

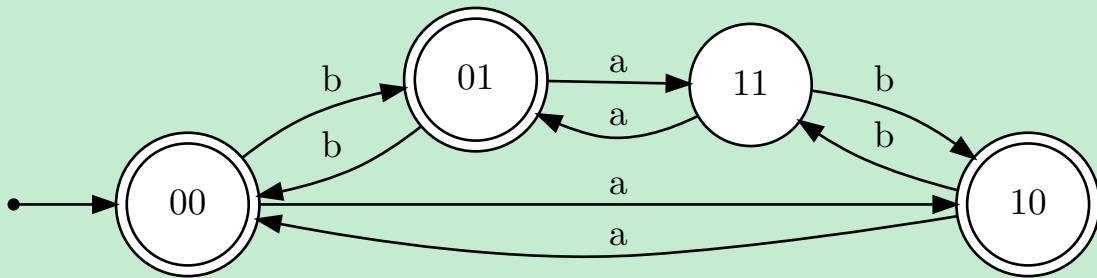
$$L' = \{x \in \{a, b\}^* \mid \#_a(x) \text{ è multiplo di } m\}$$

$$L'' = \{x \in \{a, b\}^* \mid \#_b(x) \text{ è multiplo di } n\}.$$

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



9.3.2.3. 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

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

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

9.3.3. Intersezione

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

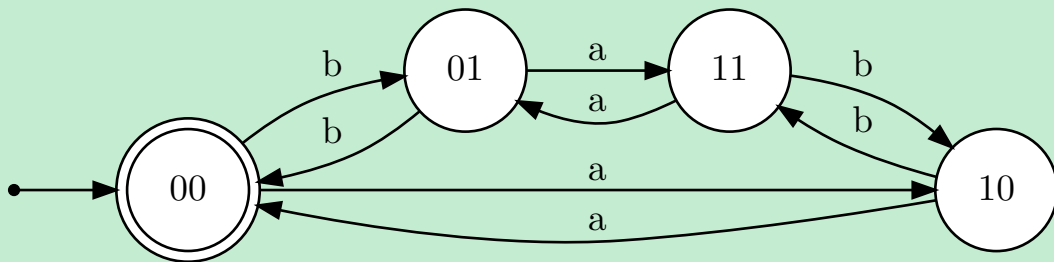
Per i DFA, possiamo utilizzare la costruzione dell'automa prodotto appena definita modificando l'insieme degli stati finali F rendendolo l'insieme

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

Ma allora la state complexity vale

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

Esempio 9.3.3.1: 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 possibili 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

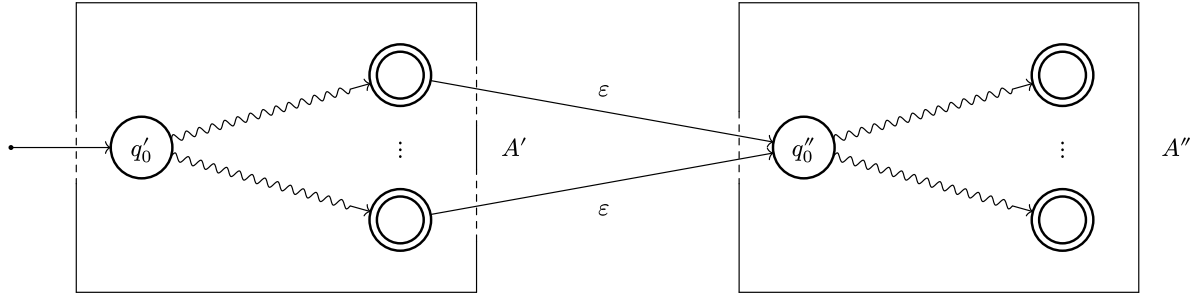
$$\text{nsc}(L' \cap L'') \leq \text{nsc}(L') \cdot \text{nsc}(L'').$$

9.3.4. Prodotto

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

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

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?

9.3.4.1. 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

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

possiamo dire che

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

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

9.3.4.2. 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

$$q_0 = \begin{cases} (q'_0, \emptyset) & \text{se } q'_0 \notin F' \\ (q'_0, \{q''_0\}) & \text{se } q'_0 \in F' \end{cases}$$

- 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.

9.3.4.3. 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

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

10. Lezione 10 [28/03]

10.1. 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'automata 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 $n+1$.

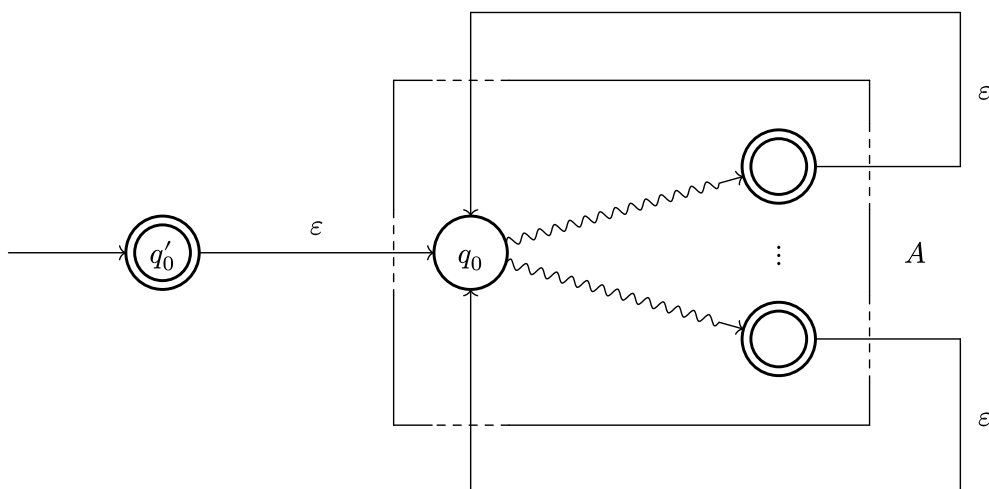
10.2. Chiusura di Kleene

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

Questa operazione è definita come

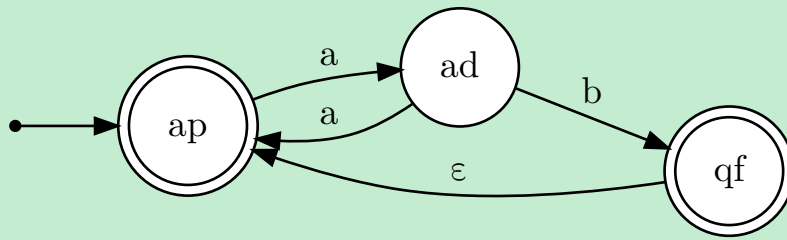
$$L^* = \bigcup_{k \geq 0} L^k = \{w \in \Sigma^* \mid \exists x_1, \dots, x_k \in L \mid k \geq 0 \mid w = x_1 \dots x_k\}.$$

Un automa per la star deve cercare di scomporre la stringa in ingresso in più stringhe di L . Possiamo prendere spunto dall'automata per la concatenazione: facciamo partire l'automata 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.



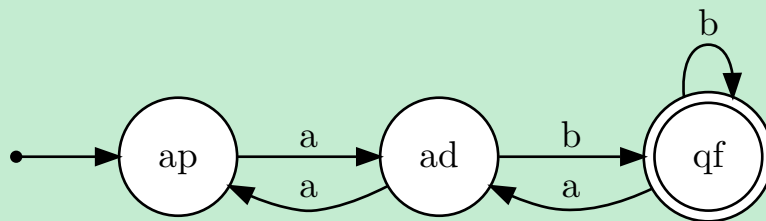
Abbiamo bisogno dello stato iniziale q'_0 perché dentro l'automata ci possono essere delle transizioni che mi fanno ritornare indietro e mi fanno accettare di più di quello che dovrei.

Esempio 10.2.1: 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'automato di partenza.



10.2.1. DFA

Ci piace questa soluzione, perché stiamo aumentando solo di uno il numero di stati dell'automato, 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

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

10.2.2. 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

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

10.2.3. 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 10.2.3.1: 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

$$aaabaaaaaaabaabaaaab$$

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 10.2.3.2: 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.

10.2.4. Codici

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

Definizione 10.2.4.1 (Codice): Dato $X \subseteq \Sigma^*$, diciamo che X è un **codice** se e solo se

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

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.

10.2.5. 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 10.2.5.1: Date le espressioni regolari

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

$$(a + b)^*$$

$$(ab^*)^*$$

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 10.2.5.1 (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.

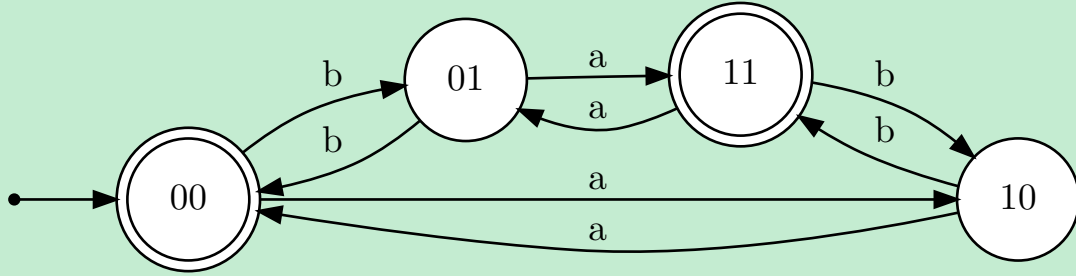
Teorema 10.2.5.1 (Un bro nel 1966): Vale

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

Il linguaggio W_q è definito come

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

Esempio 10.2.5.2: Proviamo a disegnare W_q per $q = 1$.



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

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

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

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

10.3. 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 10.3.1: 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.

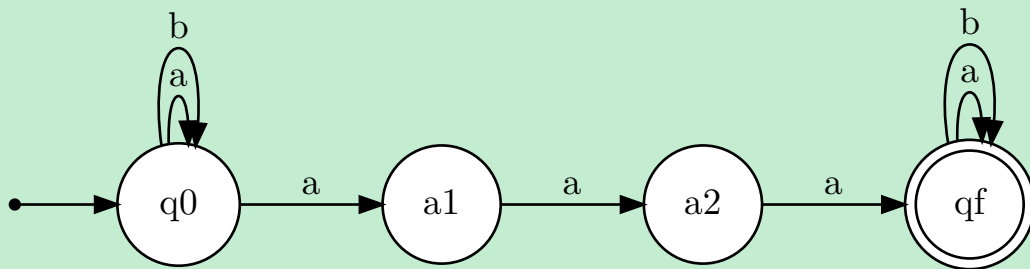
Per il linguaggio $L_a = \{w \in \{a, b\}^* \mid \#_a(w) \text{ pari}\}$ possiamo scrivere l'espressione regolare $(b + ab^*a)^*$, mentre per il linguaggio $L_b = \{w \in \{a, b\}^* \mid \#_b(w) \text{ pari}\}$ possiamo scrivere l'espressione regolare $(a + ba^*b)^*$.

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

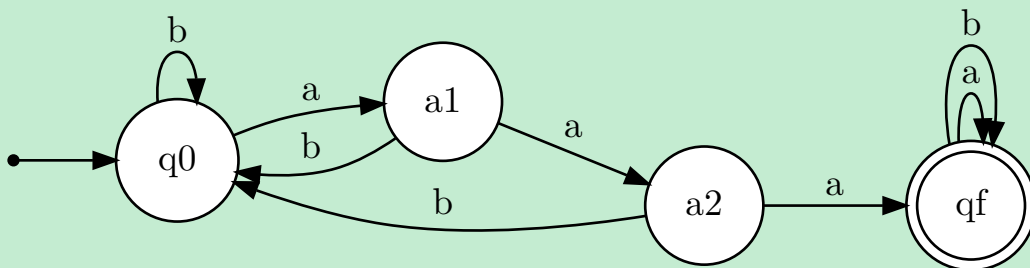
$$(b + ab^*a)^* \cap (a + ba^*b)^*.$$

Esempio 10.3.2: 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.

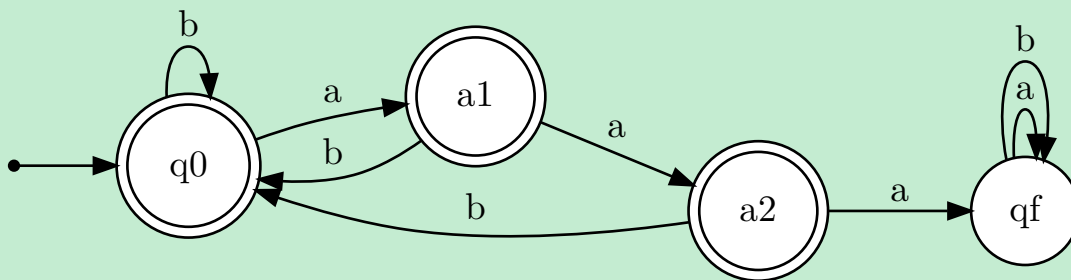
Partiamo da automi che accettano stringhe con tre a consecutive e poi complementiamo. In ordine, vediamo un NFA e un DFA per questo linguaggio di transizione.



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 10.3.1 (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

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

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

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

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.

10.4. Operazioni esotiche

10.4.1. 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

$$w = a_1 \dots a_n \implies w^R = a_n \dots a_1.$$

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

Lemma 10.4.1.1: L'operazione di reversal preserva la regolarità.

Dimostrazione 10.4.1.1.1 (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'automato 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.

10.4.1.1. 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

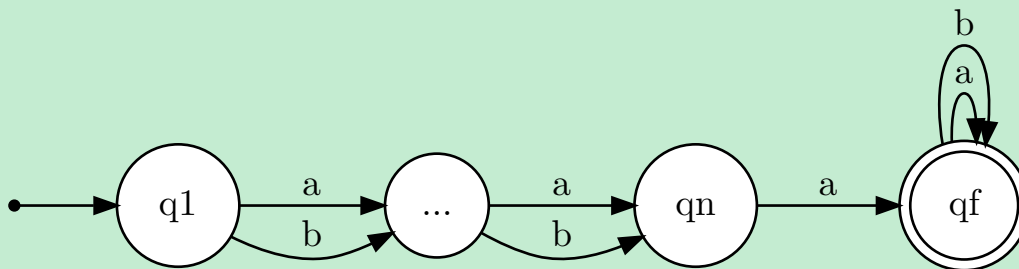
$$\text{sc}(L^R) \leq 2^{\text{nsc}(L^R)}.$$

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

Esempio 10.4.1.1.1: 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 10.4.1.1.2: 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).

10.4.1.2. 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

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

10.4.2. 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 10.4.2.1: Date le stringhe $aabb$ e b , possiamo ottenere le stringhe

$baabb$

$ababb$

$aabbb$

$aabbb$

$aabbb$

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

Esempio 10.4.2.2: Date le stringhe $aabb$ e ab , possiamo ottenere molte stringhe con lo shuffle. Ne vediamo un paio per vedere la non atomicità dell'operazione.

$aababb$

$aaabbb$

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.

10.4.2.1. 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.

10.4.2.2. 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

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

10.4.2.3. 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.

11. Lezione 11 [02/04]

11.1. Reversal

Abbiamo già visto l'operazione di reversal applicato ad un linguaggio, ovvero dato L regolare definiamo

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

anch'esso regolare formato da tutte le stringhe di L lette in senso opposto. In poche parole, se $w = a_1 \dots a_n$ allora $w^R = a_n \dots a_1$. Ovviamente, il reversal è **idempotente**, ovvero

$$(L^R)^R = L.$$

Cosa succede se applichiamo il reversal ad una **grammatica**?

Data una grammatica G di tipo 2 per L , come otteniamo una grammatica per L^R ? Abbiamo scelto una grammatica di tipo 2 perché le regole sono «standard» nella forma $A \rightarrow \alpha$ con $\alpha \in (V \cup \Sigma)^+$. Questa costruzione è facile: invertiamo ogni parte destra delle regole di derivazione, ovvero definiamo

$$\forall (A \rightarrow \alpha) \quad \text{definiamo } A \rightarrow \alpha^R.$$

Esempio 11.1.1: Data una grammatica per L con regola di produzione

$$A \rightarrow aaAbBb,$$

una grammatica per L^R ha come regola di produzione

$$A \rightarrow bBbAaa.$$

Prendiamo ora una grammatica di tipo 3, che sappiamo essere capace di generare i linguaggi regolari, a meno della parola vuota. Le regole di produzione sono nella forma

$$A \rightarrow aB \mid a.$$

Avevamo visto due estensioni delle grammatiche regolari: una di queste è rappresentata dalle **grammatiche lineari a destra**, ovvero quelle grammatiche con regole di produzione

$$A \rightarrow wB \mid w \quad \text{tale che } w \in \Sigma^*.$$

Avevamo anche dimostrato che queste grammatiche sono equivalenti alle grammatiche di tipo 3 trasformando la stringa w in una serie di derivazioni che rispettino le grammatiche regolari.

L'altra estensione sono le grammatiche **lineari a sinistra**, con regole di produzione

$$A \rightarrow Bw \mid w \quad \text{tale che } w \in \Sigma^*.$$

Cosa possiamo dire di queste?

Lemma 11.1.1: Le grammatiche lineari a sinistra generano i linguaggi regolari.

Dimostrazione 11.1.1.1: Partiamo da una G lineare a sinistra e applichiamo il **reversal**: otteniamo una grammatica G' lineare a destra, visto che applichiamo la trasformazione

$$A \rightarrow Bw \mid w \implies A \rightarrow wB \mid w.$$

Possiamo quindi dire che

$$L(G') = (L(G))^R.$$

Sappiamo che i linguaggi regolari sono chiusi rispetto all'operazione di reversal, quindi essendo $L(G')$ regolare allora anche $L(G)$ lo deve essere. Quindi anche le grammatiche lineari a sinistra generano i linguaggi regolari. ■

Se prendiamo un linguaggio che è sia lineare a destra e a sinistra otteniamo le **grammatiche lineari**, che però generano di più rispetto alle grammatiche regolari.

11.2. Algoritmo per automa minimo

Dato $M = (Q, \Sigma, \delta, q_0, F)$ un DFA senza stati irraggiungibili, costruiamo l'automa per il reversal del linguaggio accettato da M . Definiamo quindi l'automa $M^R = (Q, \Sigma, \delta^R, F, \{q_0\})$ definito dalla **funzione di transizione**

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

Questo automa, ovviamente, è NFA per via del non determinismo sulle transizioni e del non determinismo sugli stati iniziali multipli. A noi non piace, vogliamo ancora un DFA, quindi applichiamo la **costruzione per sottoinsiemi**.

Definiamo allora l'automa $N = \text{sub}(M^R)$ DFA ottenuto dalla **subset construction**, definito dalla tupla $(Q'', \Sigma, \delta'', q_0'', F'')$ tale che:

- $Q'' \leq 2^Q$ **insieme degli stati raggiungibili**, ovvero andiamo a rimuovere dall'automa tutti gli stati che sono irraggiungibili; con stati, ovviamente, ci stiamo riferendo ai vari sottoinsiemi;
- δ'' **funzione di transizione** tale che

$$\delta''(\alpha, a) = \bigcup_{p \in \alpha} \delta^R(p, a);$$

- $q_0'' = F$ **stato iniziale** preso da M^R , che aveva già un insieme come stato iniziale;
- F'' **insieme degli stati finali** tale che

$$F'' = \{\alpha \in Q'' \mid q_0 \in \alpha\}$$

perché per il reversal lo stato iniziale diventava finale.

Vediamo adesso un risultato abbastanza strano di questa costruzione.

Lemma 11.2.1: N è il DFA minimo per il reversal del linguaggio riconosciuto da M .

Dimostrazione 11.2.1.1: Per prima cosa, dimostriamo che N **riconosce** il reversal del linguaggio riconosciuto da M . Ma questo è banale: l'abbiamo fatto per costruzione, usando prima il reversal e poi la costruzione per sottoinsiemi.

Dimostriamo quindi che N è **minimo**. In un automa minimo, tutti gli stati sono distinguibili tra loro. Analogamente, al posto di dimostrare questo, possiamo fare vedere che

$$\forall A, B \in Q'' \quad A, B \text{ non distinguibili} \implies A = B.$$

Assumiamo quindi che $A, B \in Q''$ siano due stati non distinguibili e che allora vale $A = B$. Questi due stati sono sottoinsiemi derivanti dalla costruzione per sottoinsiemi, quindi per dimostrare l'uguaglianza di insiemi devo dimostrare che

$$A \subseteq B \wedge B \subseteq A.$$

Partiamo con $A \subseteq B$. Sia $p \in A$, allora, visto che tutti gli stati sono raggiungibili, esiste una stringa $w \in \Sigma^*$ tale che, nell'automa M , vale

$$\delta(q_0, w) = p.$$

Per come abbiamo definito l'NFA per il reversal, allora

$$q_0 \in \delta^R(p, w^R).$$

Usando invece il DFA, abbiamo che

$$q_0 \in \delta''(A, w^R)$$

perché abbiamo assunto che $p \in A$. Ma allora w^R è accettata da N partendo da A .

Ora, visto che A e B non sono distinguibili per ipotesi, la stringa w^R è accettata da N partendo anche da B , cioè

$$q_0 \in \delta''(B, w^R)$$

quindi esiste un elemento $p' \in B$ tale che

$$q_0 \in \delta^R(p', w^R)$$

e quindi che

$$\delta(q_0, w) = p'.$$

L'automa è deterministico, quindi $p = p'$, e quindi che $p \in B$, e quindi

$$A \subseteq B.$$

La dimostrazione è analoga per la seconda inclusione. ■

Questa costruzione è un po' strana e contro quello che abbiamo sempre fatto: la costruzione per sottoinsiemi di solito fa esplodere il numero degli stati, mentre stavolta ci dà l'automa minimo per il reversal. Cosa possiamo fare con questo risultato?

Algoritmo di Brzozowski

Sia K un DFA per il linguaggio L senza stati irraggiungibili

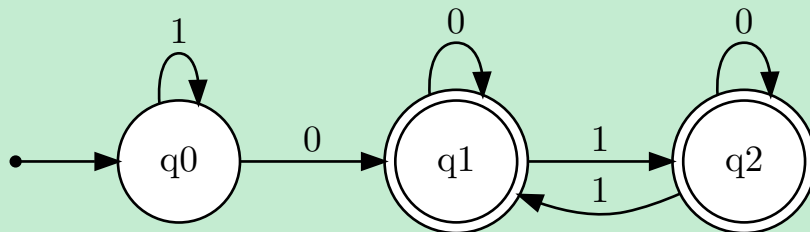
- 1: Costruiamo K^R NFA per L^R
 - 2: Costruiamo $M = \text{sub}(K^R)$ DFA per L^R
 - 3: Costruiamo M^R NFA per $(L^R)^R = L$
 - 4: Costruiamo $N = \text{sub}(M^R)$ DFA per L
-

Teorema 11.2.1: N è l'automa minimo per L .

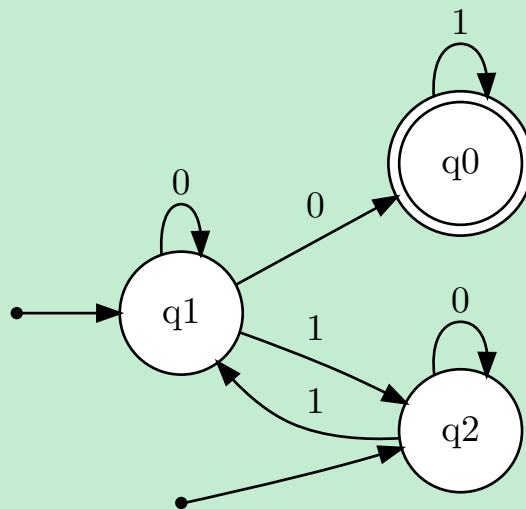
Dimostrazione 11.2.1.2: La dimostrazione è una conseguenza del lemma precedente: la prima coppia reversal+subset costruisce l'automa minimo per L^R , mentre la seconda coppia reversal+subset costruisce l'automa minimo per $(L^R)^R = L$. ■

Grazie all'**algoritmo di Brzozowski** noi abbiamo a disposizione un algoritmo di minimizzazione un po' strano dal punto di vista pratico che però ottiene l'automa minimo, anche se non è il più efficiente, ce ne sono di migliori.

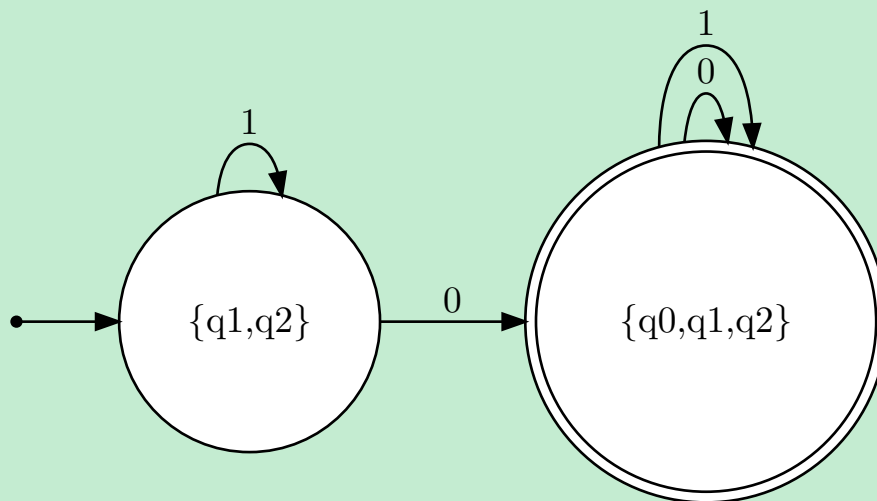
Esempio 11.2.1: Ci viene dato il DFA K della seguente figura.



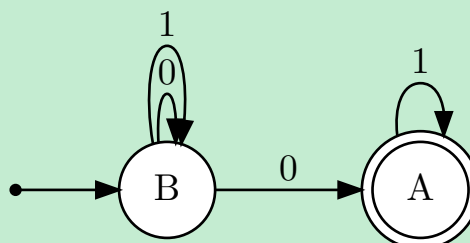
Notiamo subito che lo stato q_2 è ridondante. Andiamo a costruire K^R .



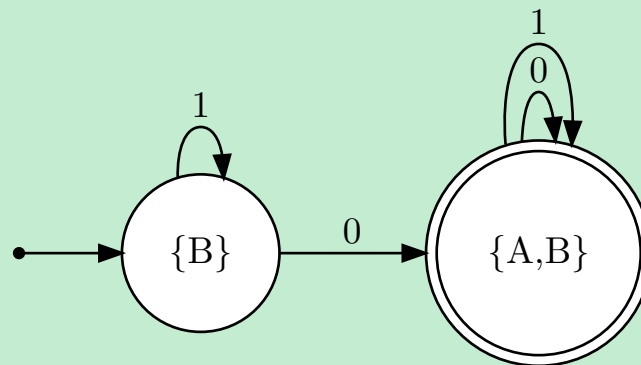
Rendiamo DFA questo automa, e chiamiamolo M .



Facciamo il renaming, chiamando A l'insieme iniziale e B l'insieme finale. Rifacciamo ora la costruzione del reversal, ottenendo M^R .



Infine, facciamo la costruzione per sottoinsiemi su M^R ottenendo N .



11.3. Pumping Lemma

Come facciamo a dimostrare che un linguaggio non è regolare? Che tecniche abbiamo?

Prima di tutto abbiamo il **criterio di distinguibilità**: se troviamo un insieme X di parole distinguibili tra loro per un linguaggio L , allora ogni DFA per L ha almeno $|X|$ stati. Come lo utilizziamo? Se $|X| = \infty$ allora servono un numero infinito di stati, cosa che negli automi a **stati finiti** non è possibile.

Esempio 11.3.1: Definiamo il linguaggio

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

Gli automi a stati finiti **non sanno contare**, quindi non posso contare quante a ci sono nella stringa e poi verificare lo stesso numero di b .

Definiamo l'insieme $X = \{a^n \mid n \geq 0\}$. Esso è formato da stringhe distinguibili tra loro: infatti, dati a^i e a^j per distinguere utilizzo $z = b^i$.

Un altro modo per dimostrare la non regolarità è far vedere che il linguaggio dato fa saltare qualche proprietà di chiusura.

Esempio 11.3.2: Definiamo il linguaggio

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

che palesemente non è regolare perché non posso contare, ma come lo dimostro?

Con la **distinguibilità** posso usare lo stesso insieme X di prima, ma facciamo finta di non saperlo fare.

Sappiamo che i linguaggi regolari sono chiusi rispetto all'operazione di intersezione. Prendiamo quindi un linguaggio regolare e facciamo l'intersezione con L , ad esempio facciamo

$$L \cap a^* b^*.$$

Visto che a^*b^* è un linguaggio regolare e l'intersezione chiude i linguaggi regolari, anche il linguaggio risultante deve essere regolare, ma questa intersezione genera il linguaggio dell'esempio precedente, perché date tutte le stringhe con a e b uguali filtriamo tenendo solo quelle che hanno tutte le a all'inizio e poi tutte le b .

Visto che il linguaggio risultante non è regolare, non lo è nemmeno L .

L'ultimo metodo che abbiamo a disposizione è il **pumping lemma per i linguaggi regolari**.

Lemma 11.3.1 (Pumping lemma per i linguaggi regolari): Sia L un linguaggio regolare. Allora esiste una costante N tale che $\forall z \in L$, con $|z| \geq N$, possiamo scrivere z come

$$z = uvw$$

con:

1. $|uv| \leq N$;
2. $v \neq \varepsilon$;
3. $\forall k \geq 0 \quad uv^k w \in L$.

Un po' strano: il succo di questo lemma è che se prendiamo delle stringhe lunghe prima o poi qualcosa si deve ripetere. Infatti, i tre punti ci dicono questo:

1. il primo punto ci dice che la parte che contiene la parte ripetuta è all'inizio e non è troppo lontana;
2. il secondo punto ci dice che effettivamente viene ripetuto qualcosa;
3. il terzo punto ci dice che possiamo ripetere all'infinito la parte centrale senza uscire dal linguaggio, appunto pumping, pompare.

In poche parole, la scomposizione di z avviene nei punti di ripetizione: u è la parte prima della ripetizione, v è la parte che viene ripetuta e w è la parte dopo la ripetizione.

Questa è una **condizione necessaria**: se faccio vedere se un linguaggio viola questo lemma allora non è regolare, ma potrebbe non bastare questo per far vedere che un linguaggio non è regolare.

Dimostrazione 11.3.1.1: Sia $A = (Q, \Sigma, \delta, q_0, F)$ un DFA per L . Sia $N = |Q|$.

Prendiamo una stringa $z = a_1 \dots a_m \in L$ con $|z| \geq N$. Un qualsiasi cammino accettante per z è nella forma

$$q_0 = p_0 \xrightarrow{a_1} p_1 \xrightarrow{a_2} \dots \xrightarrow{a_m} p_m \in F$$

che attraversa $N + 1$ stati, ma noi avendone N vuol dire che almeno uno stato lo stiamo visitando $2+$ volte.

Ma allora esistono $i, j \mid i < j$ tali che $p_i = p_j$. Andiamo a scomporre la nostra stringa z come $z = uvw$ tali che

$$\begin{aligned}
u &= a_1 \dots a_i \\
v &= a_{i+1} \dots a_j \\
w &= a_{j+1} \dots a_m.
\end{aligned}$$

Visto che $i < j$ allora la parte centrale ha almeno un elemento, quindi $v \neq \varepsilon$.

Inoltre, visto che $p_i = p_j$ vuol dire che partendo da p_i , leggendo la parte di stringa v , finiamo in p_j . Ma allora è possibile ripetere un numero questo cammino un numero arbitrario di volte.

Infine, per assunzione la lunghezza della stringa è $|z| = m \geq N$. Quando arriviamo all' N -esimo carattere abbiamo visto $N + 1$ stati, ovvero sono già passato in uno stato ripetuto, quindi $|uv| \leq N$ perché la ripetizione deve avvenire prima dell'inizio dell'ultima parte della stringa. ■

Vediamo come utilizzare il pumping lemma per dimostrare la non regolarità. Generalmente, faremo delle dimostrazioni per assurdo: assumendo la regolarità faremo vedere che esiste una stringa tale che ogni sua scomposizione possibile fa cadere almeno uno dei punti del pumping lemma.

Esempio 11.3.3: Definiamo il linguaggio $L = \{a^n b^n \mid n \geq 0\}$.

Per assurdo sia L un linguaggio regolare. Sia N la costante del pumping lemma. Definiamo la stringa $z = a^N b^N$ che rispetta la minima lunghezza delle stringhe. Infatti, $|z| = 2N \geq N$.

Scriviamo ora z come $z = uvw$. Deve valere il punto 1, ovvero $|uv| \leq N$, ma questo implica che u e v sono formate da solo a , quindi

$$u = a^i \wedge v = a^j \wedge w = a^{N-i-j} b^N \quad | \quad j \neq 0$$

perché deve valere $v \neq \varepsilon$.

Sappiamo inoltre che la ripetizione arbitraria di v mi mantiene nel linguaggio, ovvero che $\forall k \geq 0$ allora $uv^k w \in L$ ma questo non è vero: se scegliamo $k = 0$ la stringa uw non è più accettata perché essa è nella forma

$$uw = a^i a^{N-i-j} b^N = a^{N-j} b^N$$

e ovviamente $N - j \neq N$, quindi L non è regolare.

Facciamo un ultimo esempio dell'applicazione del pumping lemma.

Esempio 11.3.4: Definiamo il linguaggio

$$L = \{a^{2^n} \mid n \geq 0\} = \{a^{2^0}, a^{2^1}, \dots\} = \{a, aa, aaaa, \dots\}$$

insieme delle potenze di due scritte in unario.

Questo ovviamente non è regolare. Come lo dimostriamo con il pumping lemma?

Sia N la costante del PL per L . Prendiamo una stringa di L lunga almeno N , ovvero la stringa $z = a^{2^N}$ la cui lunghezza è $|z| \geq N$. Scomponiamo ora z come $z = uvw$, cosa possiamo dire?

Sappiamo che $v \neq \varepsilon$, quindi $|v| \geq 1$.

Inoltre, sappiamo della ripetizione arbitraria di v , quindi le stringhe $uv^k w$ devono stare in L . Cosa possiamo dire della lunghezza di questa stringa? Sappiamo che

$$|uv^k w| = 2^N + |v|(k-1) = 2^N + j(k-1) \stackrel{k=2}{=} 2^N + j < 2^{N+1}.$$

Che valore assume j ? La potenza successiva è 2^{N+1} ma j essendo un pezzo di z è al massimo 2^N quindi $j < 2^N$ e quindi $2^N + j < 2^{N+1}$.

Ma allora L non è regolare.

12. Lezione 12 [04/04]

12.1. Problemi di decisione per i linguaggi regolari

I **problemi di decisione** sono dei problemi che hanno come unica risposta **SI** oppure **NO**. Sui linguaggi regolari abbiamo una serie di problemi di decisione interessanti che possono essere risolti in maniera automatica. Questa lista di problemi diventerà ancora più briosa quando andremo nei linguaggi context-free.

12.1.1. Linguaggio vuoto e infinito

Dato un linguaggio L possiamo chiederci se $L \neq \emptyset$, ovvero se L **non è vuoto**, o se L è **infinito**.

Lemma 12.1.1.1: Sia L un linguaggio regolare e sia N la costante del pumping lemma per L . Allora:

1. $L \neq \emptyset \iff L$ contiene almeno una stringa di lunghezza $< N$;
2. L è infinito $\iff L$ contiene almeno una stringa z con $N \leq |z| < 2N$.

Dimostrazione 12.1.1.1.1: Partiamo con la dimostrazione del punto 1.

[\Leftarrow] Se L contiene una stringa di lunghezza $< N$ allora banalmente $L \neq \emptyset$.

[\Rightarrow] Se $L \neq \emptyset$ sia $z \in L$ la stringa di lunghezza minima in L . Per assurdo sia $|z| \geq N$, ma allora per il pumping lemma possiamo dividere z come $z = uvw$. Sappiamo dal terzo punto che possiamo ripetere un numero arbitrario di volte la stringa v e stare comunque in L . Ripetiamo v un numero di volte pari a 0: otteniamo la stringa $z' = uw$ che appartiene a L per il pumping lemma. Avevamo detto che z era la stringa più corta di L , ma abbiamo appena mostrato che $|z'| < |z|$: questo è assurdo e quindi $|z| < N$.

Dimostriamo ora il punto 2.

[\Leftarrow] Se L contiene una stringa z di lunghezza $N \leq |z| \leq 2N$, visto che $|z| \geq N$ applichiamo il pumping lemma per scomporre z in $z = uvw$. Per il terzo punto possiamo ripetere un numero arbitrario di volte il fattore v e rimanere comunque in L . Ma allora L è infinito.

[\Rightarrow] Se L è infinito, sicuramente esiste una stringa z che è lunga almeno N . Tra tutte queste stringhe, scegliamo la più corta di tutte. Per assurdo sia $|z| \geq 2N$, ma allora possiamo scomporre z come $z = uvw$ con $|uv| \leq N$ e $v \neq \epsilon$. Adesso andiamo a ripetere un numero di volte pari a 0 il fattore v , ottenendo $z' = uw \in L$. Quale è la sua lunghezza? Possiamo dire che $|z'| = |z| - |v|$ ma $|v|$ è al massimo N per il primo punto analizzato, quindi $|z'|$ è almeno N e al massimo $2N$, ma questo è assurdo perché z era la più corta tra tutte le stringhe e, per assurdo, l'avevamo posta di lunghezza $\geq 2N$. ■

Questo lemma ci dice che se vogliamo sapere se un linguaggio non è vuoto basta generare tutte le stringhe di lunghezza fino a N escluso e vedere se ne abbiamo una nel linguaggio, mentre se vogliamo sapere se un linguaggio è infinito basta generare tutte le stringhe di lunghezza compresa tra N incluso e $2N$ escluso e vedere se ne abbiamo una nel linguaggio.

Quale è il problema? Sicuramente è un approccio **inefficiente**, visto che dobbiamo generare un numero esponenziale di casi da analizzare. Possiamo fare di meglio? **SI**: per vedere se un

linguaggio non è vuoto devo cercare un **cammino** dallo stato iniziale ad uno stato finale, mentre per vedere se un linguaggio è infinito potrei cercare i **cicli** sui cammini del punto precedente. Ci siamo quindi ricondotti a dei **problemi su grafi**, che sappiamo risolvere efficientemente.

12.1.2. Appartenenza

Dato L un linguaggio regolare e $x \in \Sigma^*$ una stringa, il problema di **appartenenza** si chiede se $x \in L$. Questo lo sappiamo fare tranquillamente in tempo lineare: basta eseguire il DFA (se ce l'abbiamo) e vedere se finiamo in uno stato finale.

12.1.3. Universalità

Dato L un linguaggio regolare, il problema di **universalità** si chiede se $L = \Sigma^*$, ovvero L contiene tutte le stringhe della chiusura di Kleene dell'alfabeto Σ . Questo sembra difficile, ma possiamo sfruttare le operazioni che rendono chiusi i linguaggi regolari: infatti, possiamo chiederci invece se

$$L = \Sigma^* \iff L^C = \emptyset$$

e questo lo sappiamo fare grazie al lemma precedente.

12.1.4. Inclusione e uguaglianza

Infine, l'ultimo problema che vediamo prende due linguaggi L_1 e L_2 regolari e si chiede se $L_1 \subseteq L_2$. Questo problema si chiama problema dell'**inclusione** e lo possiamo risolvere manipolando quello che ci viene chiesto: infatti, al posto dell'inclusione, possiamo chiederci se

$$L_1 \subseteq L_2 \iff L_1 \cap L_2^C = \emptyset,$$

che sappiamo fare tranquillamente grazie al lemma.

Se non volessimo utilizzare le proprietà di chiusura, possiamo costruire un **automa prodotto** che ha come stati finali tutte le coppie di stati dove il primo accetta L_1 e il secondo rifiuta L_2 .

E se invece volessimo risolvere il problema di **uguaglianza**, ovvero quello che si chiede se $L_1 = L_2$? Basta dimostrare la doppia inclusione $L_1 \subseteq L_2 \wedge L_2 \subseteq L_1$ e siamo a cavallo. Un algoritmo diverso utilizza le classi di equivalenza, ma non lo vedremo.

12.2. Altre operazioni

Vediamo qualche **operazione** un pelo più complicata.

12.2.1. Raddoppio

Sia L regolare, definiamo l'operazione α tale che

$$\alpha(L) = \{x \in \Sigma^* \mid xx \in L\}.$$

Esempio 12.2.1.1: Dato il linguaggio $L = \{a^n b^n \mid n \geq 0\}$ allora

$$\alpha(L) = \{\varepsilon\}.$$

Dato invece il linguaggio $L = a^*$ allora

$$\alpha(L) = a^*.$$

Infine, dato il linguaggio $L = \{a^n \mid n \text{ pari}\}$ allora

$$\alpha(L) = a^*.$$

Se L è regolare, riesco a dimostrare che anche $\alpha(L)$ è regolare? Abbiamo a disposizione un automa A per L , vogliamo sapere se il mio input, raddoppiato, viene accettato da L .

Come possiamo ragionare? Vogliamo cercare un cammino che fa da q_0 a $q_f \in F$ leggendo la stringa xx . Nell'automa A , leggendo x , finiamo in uno stato p : dobbiamo cercare di indovinare questo p per far partire la computazione una seconda volta e arrivare in $q_f \in F$.

L'idea è quindi di scommettere lo stato che raggiungiamo con A leggendo x , e poi mandare in parallelo due copie di A , uno dall'inizio e uno dallo stato indovinato.

Formalizziamo questo automa. Dato $A = (Q, \Sigma, \delta, q_0, F)$ DFA per L , costruiamo l'automa

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

tale che:

- l'insieme degli stati $Q = Q^3$ è formato da triple

$$[p, q', q'']$$

dove:

- ▶ p è lo stato che abbiamo scommesso di raggiungere con x in A ;
- ▶ q' è lo stato che portiamo avanti in A a partire da q_0 ;
- ▶ q'' è lo stato che portiamo avanti in A a partire da p ;
- l'insieme degli stati iniziali (multipli)

$$I' = \{[p, q_0, p] \mid p \in Q\}$$

dove scommettiamo un qualunque stato p come stato intermedio;

- l'insieme degli stati finali

$$F' = \{[p, p, q] \mid q \in F\}$$

formato da tutti gli stati dove l'automa A finisce in p nella computazione iniziale e finisce in uno stato finale nella computazione da p

- la funzione di transizione δ' è tale che

$$\delta'([p, q', q''], a) = [p, \delta(q', a), \delta(q'', a)]$$

che manda avanti i due automi in parallelo.

Purtroppo, quello che otteniamo è un **NFA** per via di tutti gli stati iniziali multipli.

12.2.2. Metà

Un'altra operazione strana che vediamo prende un linguaggio regolare L e calcola

$$\frac{1}{2}L = \{x \in \Sigma^* \mid \exists y \mid |y| = |x| \wedge xy \in L\}.$$

In poche parole, prendo le stringhe di L di lunghezza pari e prendo la prima metà di queste.

Esempio 12.2.2.1: Dato il linguaggio $L = \{a^n b^n \mid n \geq 0\}$ allora

$$\frac{1}{2}L = a^*.$$

Si può dimostrare che questa operazione **mantiene la regolarità**, ma come facciamo? Possiamo ricondurre questo problema a quello precedente, variando un po' la seconda computazione.

In questo caso facciamo molte più scommesse: al posto di mandare avanti in parallelo i due automi, con la scommessa sullo stato intermedio p , qua mandiamo avanti il primo automa normalmente e il secondo lo mandiamo avanti non deterministicamente prendendo ogni simbolo possibile di Σ . Infatti, la stringa y è randomica, la dobbiamo inventare noi.

Cambia quindi la **funzione di transizione** δ' , che prende ancora la tripla dello stato ma ora:

- mantiene invariato lo stato scommessa;
- manda avanti deterministicamente il primo automa;
- per ogni carattere di Σ fa partire una computazione con quel carattere.

Quello che otteniamo è un **turbo NFA**, se non volessimo utilizzarlo? Abbiamo rappresentazioni alternative che ci bypassano il non determinismo?

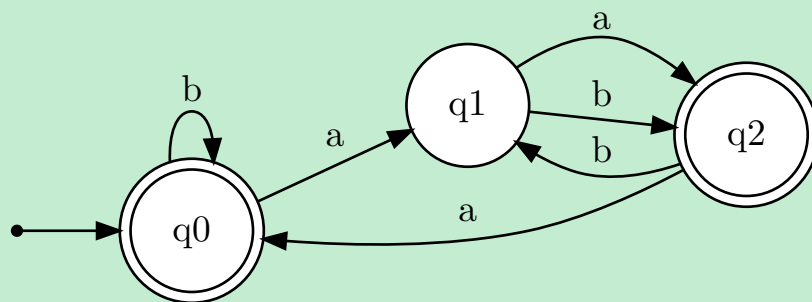
12.3. Automi come matrici

Possiamo rappresentare gli automi come **matrici di adiacenza**: esse sono matrici indicizzate, su righe e colonne, dagli stati dell'automa, e ogni cella è un valore booleano che viene posto a 1 se e solo se abbiamo una transizione dallo stato riga allo stato colonna.

Queste matrici sono dette **matrici di transizione**, e rappresentano le transizioni che possiamo fare all'interno dell'automa. Queste matrici possono essere anche **associate ad una lettera** di Σ , e queste rappresentano le transizioni che possono essere fatte nell'automa con quella lettera. Se invece le matrici sono **associate ad una stringa** rappresentano le transizioni che possono essere fatte nell'automa leggendo quella stringa, ma queste le vedremo meglio dopo.

Il **numero di possibili matrici** che possiamo costruire è finito: esso è $2^{n \times n}$, con $n = |Q|$. Questa informazione ci servirà dopo per costruire degli automi.

Esempio 12.3.1: Costruiamo le matrici di transizione del seguente automa.



Andiamo a calcolare le due matrici di transizione delle lettere a e b .

$$M_a = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \qquad M_b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Ovviamente, se l'automa è un **DFA** abbiamo un solo 1 per ogni riga.

Se calcoliamo $M_a M_b$ otteniamo la matrice che ci dice in che stato finiamo leggendo ab in base allo stato di partenza che scegliamo. Infatti:

$$M_{ab} = M_a M_b = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

che indica esattamente gli stati che raggiungiamo leggendo la stringa ab .

Possiamo banalmente estendere questa moltiplicazione ad una stringa generica $w = a_1 \dots a_n$, calcolando la matrice M_w come

$$M_w = M_{a_1} \dots M_{a_n}.$$

Queste matrici ci danno informazioni molto interessanti: ogni riga ci dice in che stato possiamo arrivare partendo dallo stato della riga leggendo una certa sequenza di caratteri. Come possiamo utilizzare questa matrice a nostro vantaggio?

12.3.1. Prima applicazione: raddoppio

Riprendiamo l'operazione α : possiamo utilizzare le matrici per risolvere questo problema evitando il non determinismo. Se noi avessimo M_x sarebbe molto facile rispondere a $xx \in L$:

- prendiamo la riga dello stato iniziale q_0 e vediamo la colonna p che contiene l'1 della riga;
- prendiamo la riga p e vediamo la colonna q_f che contiene l'1 della riga;
- verifichiamo se $q_f \in F$.

Con queste tabelle è molto facile risolvere α : ce le teniamo nello stato e mano a mano costruiamo l'automa con le tabelle nuove, e poi alla fine verifichiamo quello scritto sopra.

Definiamo quindi l'automa

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

tale che:

- l'**insieme degli stati** tiene tutte le possibili matrici booleane con indici in Q , ovvero

$$Q' = \{0, 1\}^{|Q| \times |Q|};$$

- lo **stato iniziale** è la matrice identità $I_{|Q|}$ perché all'inizio non viene letto niente (viene letta ε) e quindi non ci spostiamo dallo stato nel quale siamo;
- la **funzione di transizione** δ' è tale che

$$\delta'(M, a) = MM_a;$$

- l'**insieme degli stati finali** contiene tutti gli stati che hanno delle matrici con le proprietà descritte all'inizio della sezione, ovvero

$$F' = \{M \mid \exists p \in Q \mid M[q_0, p] = 1 \wedge \exists q \in F \mid M[p, q] = 1\}.$$

Sicuramente questo è un automa a stati finiti: il numero di matrici, anche se esponenziale, è comunque un numero finito. Inoltre, questo automa che otteniamo è DFA, a differenza del precedente.

12.3.2. Seconda applicazione: metà

Torniamo ora sull'operazione $\frac{1}{2}L$: come possiamo fare questo con le matrici definite poco fa? Se prima le matrici per entrambi gli automi erano le stesse, qua la seconda parte, visto che viene inventata, si riduce ad un problema di **raggiungibilità del grafo**.

Quello che faremo è mandare avanti il primo automa deterministicamente e il secondo invece verrà rappresentato dalle potenze della **matrice dell'automata** per vedere la raggiungibilità. Qua con matrice dell'automata intendiamo la prima versione che abbiamo definito della matrice.

La matrice dell'automata la otteniamo come somma booleana di tutte le matrici M_x associate al carattere $x \in \Sigma$. Facendo poi la potenza k -esima di questa matrice riusciamo a vedere la raggiungibilità dopo aver letto k simboli.

Definiamo quindi l'automata

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

tale che:

- l'**insieme degli stati** è formato da tutte le coppie

$$Q' = Q \times \{0, 1\}^{|Q| \times |Q|}$$

dove la prima componente rappresenta lo stato dell'automata che viene mandato avanti deterministicamente e il secondo rappresenta tutte le potenze della matrice dell'automata;

- lo **stato iniziale** parte dallo stato iniziale e dalla matrice identità, ovvero

$$q'_0 = (q_0, I_{|Q|});$$

- la **funzione di transizione** δ' è tale che

$$\delta'([q, K], a) = (\delta(q, a), KM);$$

- l'**insieme degli stati finali** contiene tutte le coppie dove la matrice, osservata nella riga definita dalla prima componente, contiene un 1 in una colonna di uno stato finale, ovvero

$$F' = \{[p, K] \mid \exists q \in F \mid K[p, q] = 1\}.$$

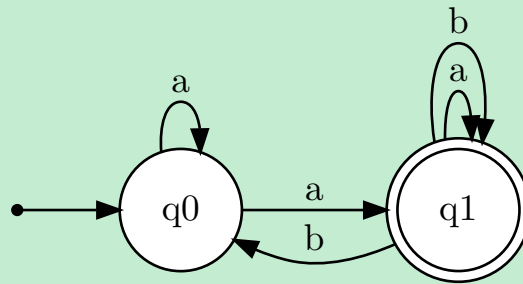
Il numero di stati è considerevole, considerando anche il prodotto cartesiano, ma abbiamo ottenuto un DFA, che invece prima non avevamo.

12.3.3. Matrici su NFA

Per ora abbiamo calcolato la matrice delle transizioni dei DFA, che succede se abbiamo un NFA? Ovviamente, in un NFA, avendo la possibilità di fare delle computazioni parallele, nella riga di una matrice associata ad una lettera possiamo avere più valori a 1.

Calcolando però le potenze della matrice dell'automata cosa otteniamo?

Esempio 12.3.3.1: Dato il seguente automa divertiamoci con qualche matrice.



Le matrici di transizione associate alle lettere a e b sono:

$$M_a = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$$

$$M_b = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

Proviamo a calcolare la matrice M_{aa} calcolandola con la somma intera:

$$M_{aa} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}.$$

Questa matrice non rappresenta più la raggiungibilità con una matrice booleana, ma **conta il numero di cammini** che abbiamo nell'automato per raggiungere quello stato. Usando la somma booleana invece avremmo ancora la matrice che definisce la raggiungibilità.

I numeri che vediamo scritti nella tabella sono i **gradi di ambiguità** delle varie stringhe (se ci limitiamo a quelle accettate): questo rappresenta appunto il numero di modi in cui possiamo arrivare a quella stringa partendo dallo stato che indicizza la riga. Il **grado di ambiguità del grafo** è il massimo grado di ambiguità delle stringhe accettate.

12.4. Varianti di automi

Per finire questa lezione infinita, vediamo qualche **variante** di automi.

12.4.1. Automi pesati

La prima variante che vediamo sono gli **automi pesati**. Essi associano ad ogni transizione un peso. Il **peso di una stringa** viene calcolato come la somma dei pesi delle transizioni che la stringa attraversa per essere accettata. Questo peso poi può essere usato in problemi di ottimizzazione, come trovare il cammino di peso minimo, ma questo ha senso solo su NFA.

12.4.2. Automi probabilistici

Un tipo particolare di automi pesati sono gli **automi probabilistici**, che come pesi sulle transizioni hanno la probabilità di effettuare quella transizione. Visto che parliamo di **probabilità**, i pesi sono nel range $[0, 1]$ e, dato uno stato, tutte le transizioni uscenti sommano a 1. In realtà, potremmo sommare a meno di 1 se nascondiamo lo stato trappola. Con questi automi possiamo chiederci con che probabilità accettiamo una stringa.

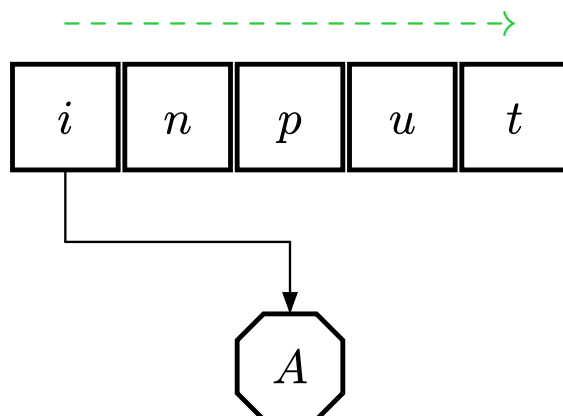
Questi automi li possiamo usare come **riconoscitori a soglia**: tutte le parole oltre una certa soglia le accettiamo, altrimenti le rifiutiamo.

Questi automi comunque non sono più potenti dei DFA: si può dimostrare che se la soglia λ è **isolata**, ovvero nel suo intorno non cade nessuna parola, allora possiamo trasformare questi automi probabilistici in DFA. Se la soglia non è isolata riusciamo a riconoscere una strana classe di linguaggi, che però ora non ci interessa.

13. Lezione 13 [09/04]

13.1. Varianti di automi

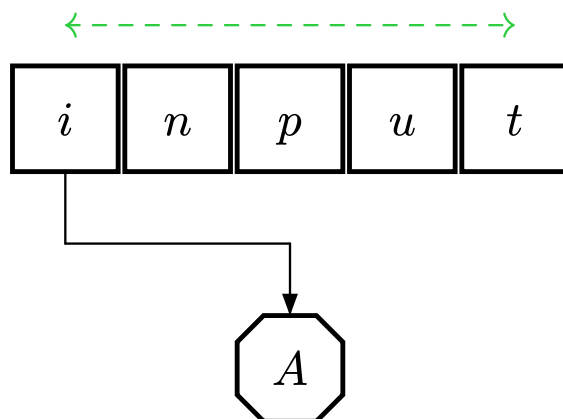
Come possiamo **rappresentare** un automa a stati finiti? Questa macchina è molto semplice: abbiamo un **nastro** che contiene l'input, esaminato da una **testina in sola lettura** che, spostandosi **one-way** da sinistra verso destra, permette ad un **controllo a stati finiti** di capire se la stringa in input deve essere accettata o meno.



La classe di linguaggi che riconosce un automa a stati finiti è la classe dei **linguaggi regolari**. Ma possiamo fare delle modifiche a questo modello? Se sì, che cosa possiamo cambiare?

13.1.1. One-way VS two-way

Se permettiamo all'automa di spostarsi da sinistra verso destra ma anche viceversa, andiamo ad ottenere gli **automi two-way**, che in base alla possibilità di leggere e basta o leggere e scrivere e in base alla lunghezza del nastro saranno in grado di riconoscere diverse classi di linguaggi.



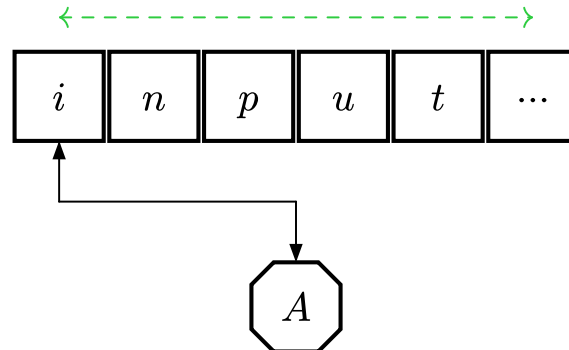
13.1.2. Read-only VS read-write

Se manteniamo l'automa one-way, rendere il nastro anche in lettura non modifica per niente il comportamento dell'automa: infatti, anche se scriviamo, visto che siamo one-way non riusciremo mai a leggere quello che abbiamo scritto.

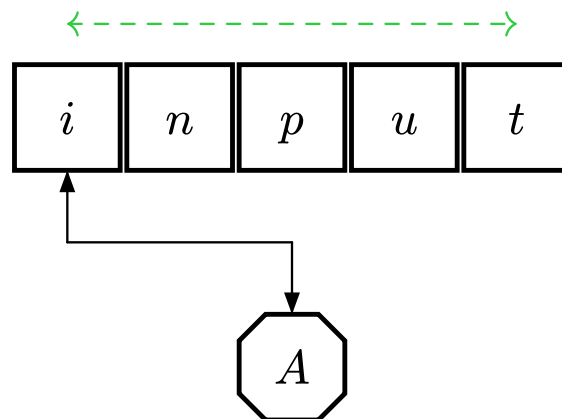
Consideriamo quindi un automa two-way che però mantiene la read-only del nastro: la classe che otteniamo è ancora una volta quella dei **linguaggi regolari**, e questo lo vedremo oggi.

Rendiamo ora la testina capace di poter scrivere sul nastro che abbiamo a disposizione. Ora, in base a come è fatto il nastro abbiamo due situazioni:

- se rendiamo il nastro illimitato oltre la porzione occupata dall'input, andiamo ad riconoscere i linguaggi di tipo 0, ovvero otteniamo una **macchina di Turing**:



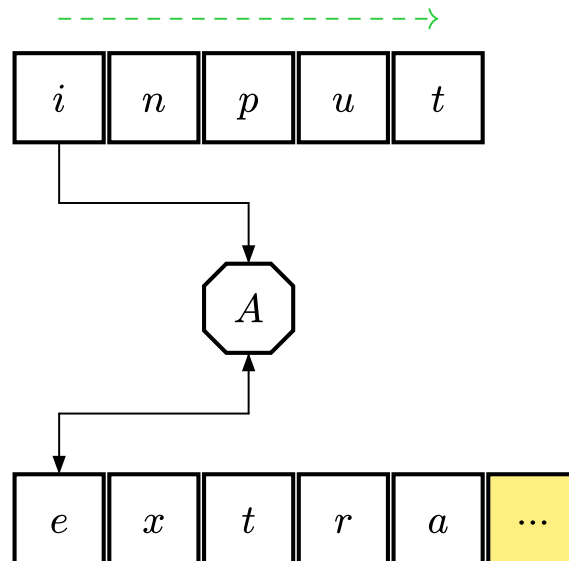
- se invece lasciamo il nastro grande quando l'input andiamo a riconoscere i linguaggi di tipo 1, ovvero otteniamo un **automa limitato linearmente**. Quest'ultima cosa vale perché nelle grammatiche di tipo 1 le regole di produzione non decrescono mai, e un automa limitato linearmente per capire se deve accettare cerca di costruire una derivazione al contrario, accorciando mano a mano la stringa arrivando all'assioma S :



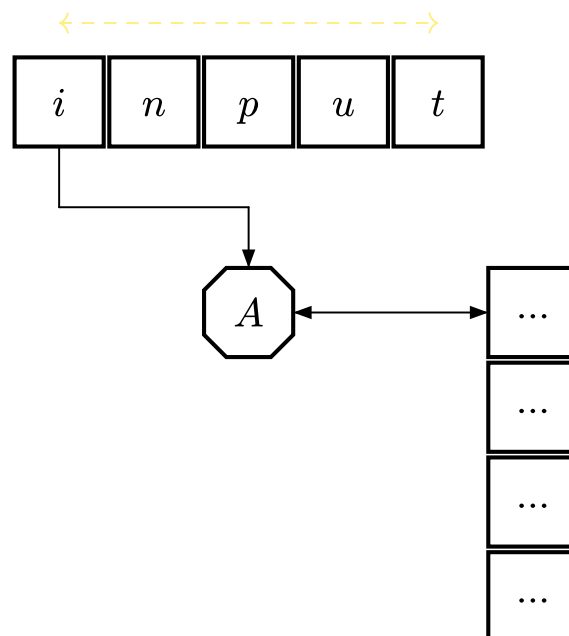
13.1.3. Memoria esterna

L'ultima modifica che possiamo pensare per queste macchine è l'aggiunta di una **memoria esterna**.

Dato un automa one-way con nastro read-only, se aggiungiamo un secondo nastro in read-write che funga da memoria esterna, otteniamo le due situazioni che abbiamo visto per gli automi two-way con possibilità di scrivere sul nastro di input.



Un caso particolare è se la memoria esterna è codificata come una **pila** illimitata, ovvero riesco a leggere solo quello che c'è in cima, allora andiamo a riconoscere i linguaggi di tipo 2, ottenendo quindi un **automa a pila**. Se passiamo infine ad un two-way con una pila diventiamo più potenti ma non sappiamo di quanto.



13.2. Automi two-way

Tra tutte queste varianti, fissiamoci sugli **automi two-way**, ovvero quelli che hanno il nastro in sola lettura e hanno la possibilità di andare avanti e indietro nell'input. Vediamo prima di tutto qualche linguaggio per il quale possiamo usare un automa two-way.

13.2.1. Esempi vari

Esempio 13.2.1.1: Riprendiamo l'operazione α . Dato L regolare, α era tale che

$$\alpha(L) = \{x \in \Sigma^* \mid xx \in L\}.$$

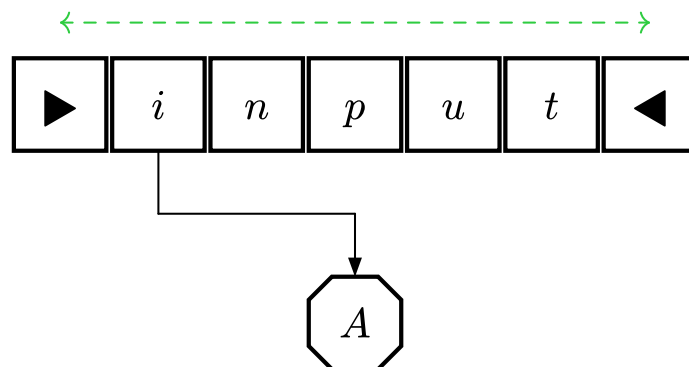
Abbiamo A un DFA che accetta L , come posso costruire un two-way per $\alpha(L)$? Potremmo leggere x la prima volta, ricordarci in che stato siamo arrivati, tornare indietro e poi ripartire a leggere x dallo stato nel quale eravamo arrivati e vedere se finiamo in uno stato finale.

Per fare ciò, ci serve sapere dove finisce il nastro: vedremo come fare tra poco.

Il numero di stati nel two-way è $3n$:

- n stati di A che usiamo per leggere x ;
- n stati che tengono traccia dello stato nel quale siamo arrivati con x e che ci permettono di ritornare all'inizio della stringa;
- n stati che fanno ripartire la computazione dallo stato nel quale siamo arrivati con x e controllano se finiamo in uno stato finale.

Abbiamo sollevato poco fa il problema: come facciamo a capire dove finisce il nastro? Andiamo a inserire dei **marcatori**, uno a sinistra e uno a destra, che delimitano la stringa. Se per caso arriviamo su un marcatore non possiamo andare oltre: possiamo solo rientrare sul nastro. In realtà, vedremo che in un particolare caso usciremo dai bordi.



Vediamo ancora un po' di esempi.

Esempio 13.2.1.2: Definiamo

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

il solito linguaggio dell' n -esimo simbolo da destra uguale ad una a .

Avevamo visto che con un NFA avevamo $n + 1$ stati, mentre con un DFA avevamo 2^n stati perché ci ricordavamo una finestra di n simboli. Ora diventa tutto più facile: ci spostiamo, ignorando completamente la stringa, sul marcatore di destra, poi contiamo n simboli e vediamo se accettare o rifiutare.

Come numero di stati siamo circa sui livelli dell'NFA, visto che dobbiamo solo scorrere la stringa per intero e poi tornare indietro di n .

Esempio 13.2.1.3: Definiamo infine

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

il linguaggio delle parole che hanno due a distanti n . Come lo scriviamo un two-way per K_n ?

Potremmo partire dall'inizio e scandire la stringa x . Ogni volta che troviamo una a andiamo a controllare n simboli dopo e vediamo se troviamo una seconda a :

- se sì, accettiamo;
- se no, torniamo indietro di $n - 1$ simboli per andare avanti con la ricerca.

Il numero di stati è:

- 1 che ricerca le a ;
- n stati per andare in avanti;
- 1 stato di accettazione;
- $n - 1$ stati per tornare indietro.

Ma allora il numero di stati è $2n + 1$.

Abbiamo trovato una buonissima soluzione per l'esempio precedente, ma se volessimo una soluzione alternativa che utilizza un automa sweeping? Ma cosa sono ste cose?

Un **automa sweeping** è un automa che non cambia direzione mentre si trova nel nastro, ma è un automa che rimbalza avanti e indietro sugli end marker. La soluzione che abbiamo trovato non usa automi sweeping perché se il simbolo a distanza n è una b noi invertiamo la direzione e torniamo indietro.

Esempio 13.2.1.4: Cerchiamo una soluzione che utilizzi un automa sweeping per K_n .

Supponiamo di numerare le celle del nastro da 1 a k . Partendo nello stato 1, andiamo a guardare tutte le celle a distanza n : se troviamo una a e poi subito dopo ancora una a accettiamo, altrimenti andiamo avanti fino a quando rimbalziamo sul marker, tornando indietro e andando sulla cella 2. Da qui facciamo ripartire la computazione, andando ogni volta avanti di una cella.

In generale, dalla cella $p \in \{1, \dots, n\}$ noi visitiamo tutte le celle $tn + p$.

Con questo approccio, il numero di stati è $O(n^2)$ perché dobbiamo muoverci di n simboli un numero n di passate. Possiamo farlo con un numero lineare di stati se facciamo la ricerca modulo n anche al ritorno, ma è questo è negli esercizi.

13.2.2. Definizione formale

Abbiamo visto come è costruito un automa two-way, ora vediamo la definizione formale. Definiamo

$$M = (Q, \Sigma, \delta, q_0, q_f)$$

un **2NFA** tale che:

- Q rappresenta l'insieme degli stati;

- Σ rappresenta l'**alfabeto** di input;
- q_0 rappresenta lo **stato iniziale**;
- δ rappresenta la **funzione di transizione**, ed è tale che

$$\delta : Q \times (\Sigma \cup \{\blacktriangleright, \blacktriangleleft\}) \longrightarrow 2^{Q \times \{+1, -1\}},$$

ovvero prende uno stato e un simbolo dell'alfabeto compresi gli end marker e ci restituisce i nuovi stati e che movimento dobbiamo fare con la testina. Ho dei **divieti**: se sono sull'end marker sinistro non ho mosse che mi portano a sinistra, idem ma specchiato su quello di destra con una piccola eccezione, che vediamo tra poco;

- q_f è lo **stato finale** e si raggiunge «passando» oltre l'end marker destro, unico caso in cui si può superare un end marker.

Con questo modello possiamo incappare in **loop infiniti**, che:

- nei DFA non ci fanno accettare;
- negli NFA magari indicano che abbiamo fatto una scelta sbagliata e c'era una via migliore.

Ci sono poi diverse modifiche che possiamo fare a questo modello, ad esempio:

- possiamo estendere le mosse con la **mossa stazionaria**, ovvero quella codificata con 0 che ci mantiene nella posizione nella quale siamo, ma possono essere eliminate con una coppia di mosse sinistra+destra o viceversa;
- possiamo utilizzare un **insieme di stati finali**;
- possiamo **non** usare gli end marker, rendendo molto difficile la scrittura di automi perché non sappiamo dove finisce la stringa.

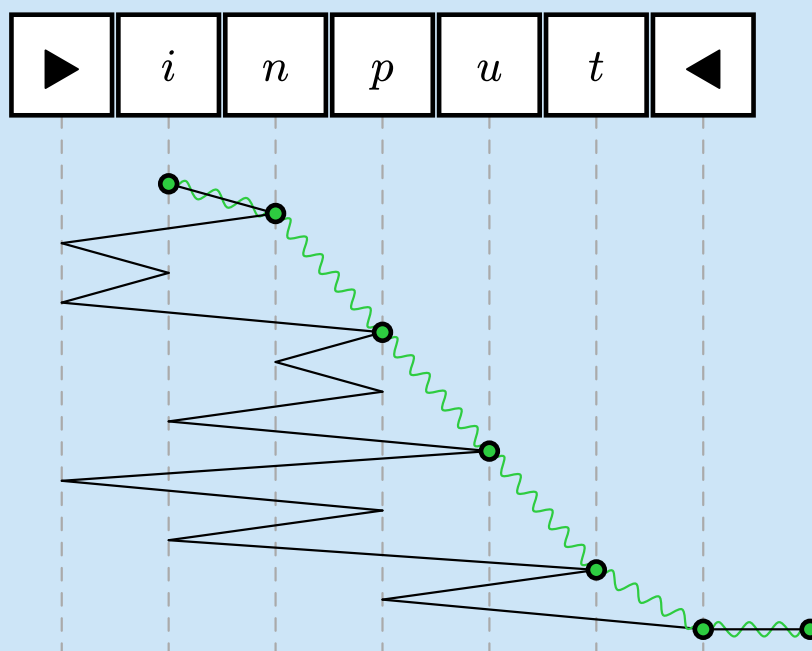
13.2.3. Potenza computazionale

Avere a disposizione un two-way sembra darci molta potenza, ma in realtà non è così: infatti, questi modelli sono equivalenti agli automi a stati finiti one-way, detti anche **1DFA**.

Teorema 13.2.3.1: Vale

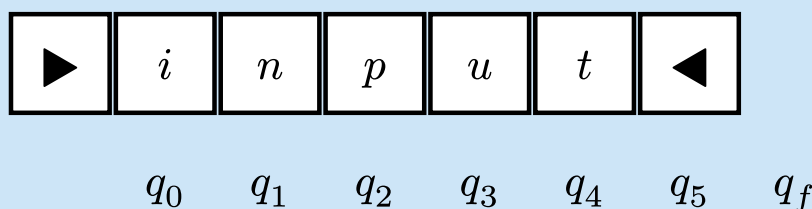
$$L(2DFA) = L(1DFA).$$

Dimostrazione 13.2.3.1.1: Abbiamo a disposizione un 2DFA nel quale abbiamo inserito un input che viene accettato. Vogliamo cambiare la computazione del 2DFA in una computazione di un 1DFA. Vediamo che stati vengono visitati nel tempo.



Prima di tutto, dobbiamo ricordarci che nei DFA non abbiamo end marker, quindi abbiamo solo l'input. Nell'automa two-way ci sono momenti dove entro nelle celle per la prima volta: nel grafico sopra sono segnati in verde. Chiamiamo questi stati $q_{i \geq 0}$.

Usiamo delle **scorciatoie**: visto che nel 1DFA non possiamo andare avanti a indietro, dobbiamo tagliare via le computazioni che tornano indietro e vedere solo in che stato esco.



Come vediamo, a me interessa sapere in che stato devo spostarmi a partire dalla mia posizione, evitando quello che viene fatto tornando all'indietro. Per tagliare le parti che tornano indietro usiamo delle **matrici**, molto simili a quelle della lezione precedente. Quelle matrici erano nella forma $M_w[p, q]$ che conteneva un 1 se e solo se partendo da q finivo in p leggendo w .

Le matrici che costruiamo ora sono nella forma

$$\tau_w : Q \times Q \longrightarrow [0, 1]$$

che mi vanno a definire il primo stato che incontriamo quando leggiamo un nuovo carattere della stringa.

Nella matrice abbiamo $\tau_w[p, q] = 1$ se e solo se esiste una sequenza di mosse che:

- inizia sul simbolo più a destra della porzione di nastro che contiene $(\blacktriangleright w)$ nello stato p ;

- termina quando la testina esce a destra dalla porzione di nastro considerata nello stato q .

Ad esempio, considerando l'esempio sopra, vale

$$\tau_{\text{inp}}[q_2, q_3] = 1.$$

Vediamo come ottenere induttivamente queste tabelle. Partiamo con $w = \varepsilon$: la porzione di nastro che stiamo considerando è formata solo da \blacktriangleright , ma non potendo andare a sinistra l'unica mossa che possiamo fare è andare a destra, quindi andare in un nuovo stato, ovvero

$$\tau_\varepsilon[p, q] = 1 \iff \delta(p, \blacktriangleright) = (q, +1).$$

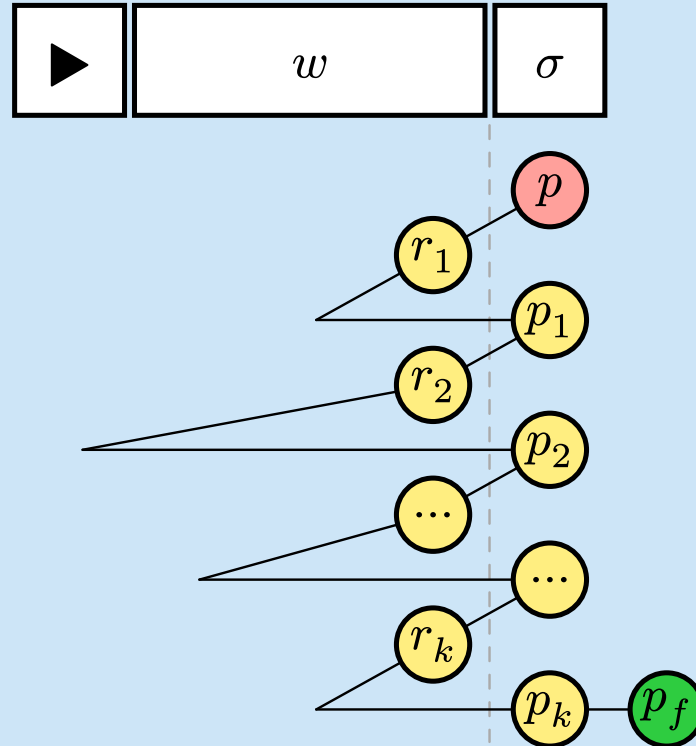
Supponiamo di aver calcolato la tabella di w , vediamo come costruire induttivamente la tabella di $w\sigma$, con $w \in \Sigma^*$ e $\sigma \in \Sigma$. Se vale

$$\delta(p, \sigma) = (q, +1)$$

la tabella è molto facile, perché sto subito uscendo dallo stato p , ovvero

$$\tau_{w\sigma}[p, q] = 1.$$

Se invece andiamo indietro dobbiamo capire cosa fare.



Ogni volta che da p_i torniamo indietro finiamo in uno stato r_{i+1} , che poi dopo un po' di giri finisce per forza in p_{i+1} . Andiamo avanti così, fino ad un certo p_k , dal quale usciamo e andiamo in q . In poche parole

$$\tau_{w\sigma}[p, q] = 1$$

se e solo se esiste una sequenza di stati

$$p_0, p_1, \dots, p_k, r_1, \dots, r_k \mid k \geq 0$$

tale che:

- $p_0 = p$, ovvero parto dallo stato p , per definizione;
- $\delta(p_{i-1}, \sigma) = (r_i, -1) \quad \forall i \in \{1, \dots, k\}$, ovvero in tutti i p tranne l'ultimo io torno indietro;
- $\tau_w[r_i, p_i] = 1$, ovvero da r_i giro in w e poi torno in p_i ;
- $\delta(p_k, \sigma) = (q, +1)$, ovvero esco fuori dal $w\sigma$.

Notiamo che se prendiamo $k = 0$ abbiamo la situazione precedente in cui uscivo direttamente. Inoltre, k è il numero massimo di stati del DFA perché se faccio ancora un giro in w dopo p_k vado in uno stato già visto ed entro in un loop infinito.

Notiamo una cosa **importantissima**: se due stringhe hanno la stessa tabella, ovvero $\tau_w = \tau_{w'}$, allora l'aggiunta di un qualsiasi carattere σ genera tabelle risultanti uguali, ovvero $\tau_{w\sigma} = \tau_{w'\sigma}$. Ma allora esiste una funzione

$$f_\sigma : [Q \times Q \rightarrow [0, 1]] \rightarrow [Q \times Q \rightarrow [0, 1]]$$

che genera una tabella a partire da una data, ed è tale che

$$\forall w \in \Sigma^* \quad \tau_{w\sigma} = f_\sigma(\tau_w).$$

In poche parole, la nuova tabella dipende solo da σ e non da w , e questa tabella è esattamente quella calcolata con i 4 punti messi sopra. Le tabelle, inoltre, sono tantissime ma sono un numero finito.

Siamo pronti per costruire il 1DFA che tanto stiamo bramando. Noi avevamo $M = (Q, \Sigma, \delta, q_0, F)$ che è un 2DFA, vogliamo costruire

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

1DFA che sia equivalente a M . Esso è tale che:

- Q è l'**insieme degli stati** e lo usiamo tenere traccia dello stato nel quale siamo e della tabella che usiamo per calcolare lo stato successivo, ovvero

$$Q' = Q \times [Q \times Q \rightarrow [0, 1]];$$

- q'_0 è lo **stato iniziale** ed è la coppia

$$q'_0 = (q_0, \tau_\epsilon);$$

- δ' è la **funzione di transizione** che manda avanti l'automa, ovvero

$$\delta'((p, T), \sigma) = (q, T')$$

con:

- T' che mi dà indicazioni sullo stato nel quale arrivo con σ , che ho però appena letto, quindi $T' = f_\sigma(T)$;
- vale $T'[p, q] = 1$ perché io devo uscire in q partendo da p ;

- F' è l'**insieme degli stati finali**, ostico perché nel two-way abbiamo gli end marker, nel one-way non li abbiamo. Per accettare dovevo sfiorare l'end marker di destra e finire in q_f , ma questa informazione la ricavo dalla tabella del right marker, ovvero

$$F' = \{(q, T) \mid (f_{\blacktriangleleft}(T))[q, q_f] = 1\}.$$

Ma allora stiamo simulando un 2DFA con un 1DFA, ma gli 1DFA riconoscono la classe dei linguaggi regolari, quindi anche la classe degli automi a stati finiti two-way riconosce la classe dei linguaggi regolari. ■

Che considerazioni possiamo fare sul numero di stati? Sappiamo che:

- il numero di stati è $|Q| = n$;
- il numero di tabelle è $|[Q \times Q] \rightarrow [0, 1]| = 2^{n^2}$.

Ma allora il numero di stati è

$$|Q'| \leq n2^{n^2}.$$

Come vediamo, la simulazione è **poli-esponenziale**.

14. Lezione 14 [11/04]

14.1. Simulazione

La scorsa lezione abbiamo visto gli automi two-way e abbiamo dimostrato che hanno la stessa potenza computazionale degli automi a stati finiti. Avevamo visto la trasformazione da 2DFA a 1DFA, ma la stessa trasformazione può essere fatta per il passaggio da 2NFA a 1NFA.

Ma quanto costano queste trasformazioni?

Nel caso partissimo da un 2DFA e volessimo arrivare in un 1DFA, il costo in termini di stati è

$$\leq \dots,$$

mentre cambiando il punto di partenza con un 2NFA il salto diventa ancora peggiore:

$$\leq 2^n 2^{n^2} = 2^{n^2+n}.$$

Ma questo ce lo potevamo aspettare: abbiamo già un salto esponenziale da NFA a DFA, quindi ciao.

Ci sono due simulazioni che sono però molto particolari e importanti.

14.1.1. Problema di Sakoda & Sipser

La prima trasformazione che vediamo è quella da 2NFA a 2DFA: qua non possiamo usare la costruzione per sottoinsiemi perché ad un certo punto potresti avere il non determinismo su una mossa che però ti sposta la testina su due caratteri diversi della stringa, e questo non è possibile. Ci serve quindi una trasformazione alternativa, ma ci arriviamo dopo.

La seconda trasformazione è quella da 1NFA a 2DFA: questa trasformazione cerca di capire se, dando il two-way ad un automa deterministico, esso è capace di simulare il non determinismo.

Vediamo un paio di esempi.

Esempio 14.1.1.1: Definiamo

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

il classicissimo linguaggio dell' n -esimo carattere da destra pari ad una a .

Sappiamo che:

- esiste un 1NFA di $n + 1$ stati;
- esiste un 1DFA di 2^n stati.

Abbiamo visto un automa two-way per questo linguaggio, che usa poco più di n stati, quindi in questo caso riusciamo a togliere il non determinismo a basso costo.

Esempio 14.1.1.2: Definiamo

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

il solito linguaggio con due a a distanza n .

Avevamo visto che un 1NFA per questo linguaggio usava $n + 2$ stati, quindi una quantità lineare in n . Per un 2DFA abbiamo visto che esiste anche qui una soluzione lineare in n , quindi anche qui eliminiamo il non determinismo a basso costo.

Abbiamo visto due esempi che sembrano dare buone notizie, ma riusciamo a dimostrare che si riesce sempre a fare un 2DFA di n stati partendo da un 1NFA di n stati? Purtroppo, nessuno ci è mai riuscito.

Questi problemi sono i **problemi di Sakoda & Sipser**, ideati nel 1978 e che riguardano il costo della simulazioni di automi non deterministici one-way e two-way per mezzo di automi two-way deterministici, ovvero si chiedono se il movimento two-way aiuta nell'eliminazione del non determinismo.

Cosa sappiamo su questi problemi? Diamo qualche **upper** e **lower bound**.

Per il problema da 1NFA a 2DFA, si sfrutta la **costruzione per sottoinsiemi** per ottenere un 1DFA, che è anche un 2DFA che non torna mai indietro, ottenendo quindi un numero di stati

$$\leq 2^n.$$

Un lower bound per questo problema invece è

$$\geq n^2.$$

Per il problema da 2NFA a 2DFA, si fa un passaggio intermedio all'1NFA e poi al 1DFA, che come prima è anche 2DFA, quindi gli stati sono

$$\leq 2^{n^2+n}.$$

Il lower bound, invece, è lo stesso del problema precedente.

Ci sono casi particolari che hanno delle dimostrazioni precise:

- se utilizziamo dei 2DFA sweeping il costo per la trasformazione è **esponenziale**, ma questo non risolve il problema perché (???) ci sono automi non sweeping che per diventarlo hanno un salto esponenziale (???)
- se $\Sigma = \{a\}$:
 - ▶ se facciamo la trasformazione da 2NFA a 2DFA l'upper bound è

$$e^{O(\log^2(n))},$$

ovvero una funzione super polinomiale ma meno di una esponenziale. Inoltre, se si dimostra che esiste un lower bound super polinomiale, allora abbiamo dimostrato che

$$L = NL \text{ (???)};$$

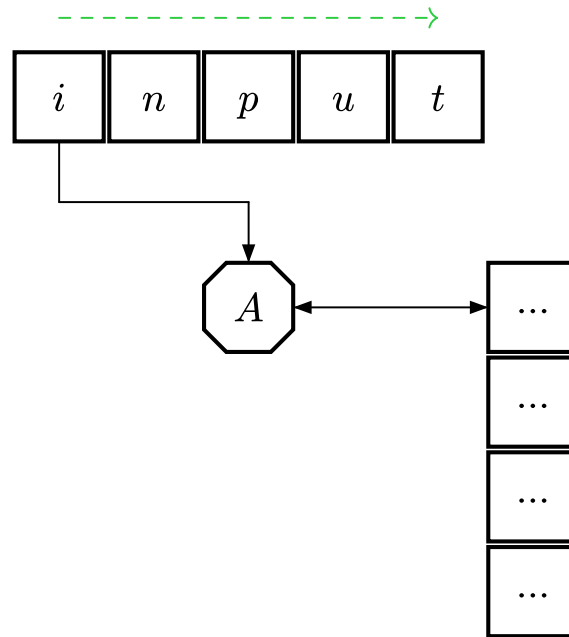
- ▶ se facciamo la trasformazione da 1NFA a 2DFA l'upper bound diventa esattamente n^2 , quindi la trasformazione fatta è ottimale.

Dei ricercatori hanno trovato degli **automi completi** per questi problemi, ovvero degli automi che permettono lo studio dei problemi solo su questi pochi automi scelti per poi far «arrivare» tutte le conseguenze a tutti gli altri automi. Scritto malissimo, sono tipo gli NP-completi.

Per ora, la **congettura** che circola tra la gente è che i costi siano **esponenziali nel caso peggiore**.

14.2. Automi a pila

Lasciamo finalmente stare gli automi a stati finiti per passare ad una nuova classe di riconoscitori: gli **automi a pila**. Essi sono praticamente degli automi a stati finiti con testina di lettura one-way ai quali viene aggiunta una **memoria infinita con restrizioni di accesso**, ovvero l'accesso avviene solo sulla cima della memoria, con politica LIFO.



Come vediamo, la parte degli automi a stati finiti ce l'abbiamo ancora, ma ora abbiamo una **memoria esterna**, che nell'immagine è sulla destra, che possiamo utilizzare con una politica di accesso LIFO. Per via di questa politica, questi automi sono anche detti **automi pushdown**, o **PDA**, perché quando inserisci qualcosa lo fai a spingere giù.

14.2.1. Versione non deterministica

Vediamo subito la definizione formale **non deterministica** dei PDA.

Sia M un PDA definito dalla tupla

$$(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

tale che:

- Q è un **insieme finito non vuoto di stati**, che rappresenta il controllo a stati finiti;
- Σ è un **alfabeto finito non vuoto di input**;
- Γ è un **alfabeto finito non vuoto di simboli della pila**;
- δ è la **funzione di transizione**;
- $q_0 \in Q$ è lo **stato iniziale**;
- $Z_0 \in \Gamma$ è il **simbolo iniziale sulla pila**;
- $F \subseteq Q$ è un **insieme di stati finali**.

La **funzione di transizione** è definita come segue:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow 2^{Q \times \Gamma^*}.$$

In poche parole, consideriamo lo **stato corrente**, il **simbolo sulla testina** o una ε -mossa e il **simbolo sulla cima della pila** per capire in che stato dobbiamo muoverci e che stringa andare ad inserire sulla pila. La lettura del carattere in cima alla pila lo va a **distruggere**.

Questa versione però non ci piace molto perché Γ^* è potenzialmente un **insieme infinito**, e non ci piace avere un insieme infinito di possibilità, quindi sostituiamo la definizione della funzione di transizione con questa analoga, ma molto migliore per noi:

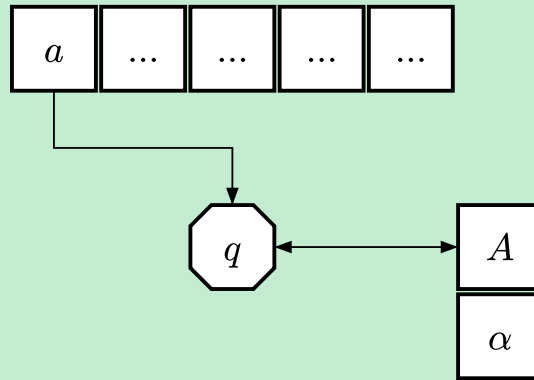
$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \longrightarrow \text{PF}(Q \times \Gamma^*).$$

Con PF intendiamo l'**insieme delle parti finite**, ovvero un insieme finito di possibilità prese dall'insieme delle parti. Ora sì che la definizione ci piace.

Facciamo qualche esempio. Come convenzione useremo le **maiuscole** per i simboli della pila.

Esempio 14.2.1.1: Facciamo che la funzione di transizione sia definita in questo modo:

$$\delta(q, a, A) = \{(q_1, \varepsilon), (q_2, BCC)\}.$$

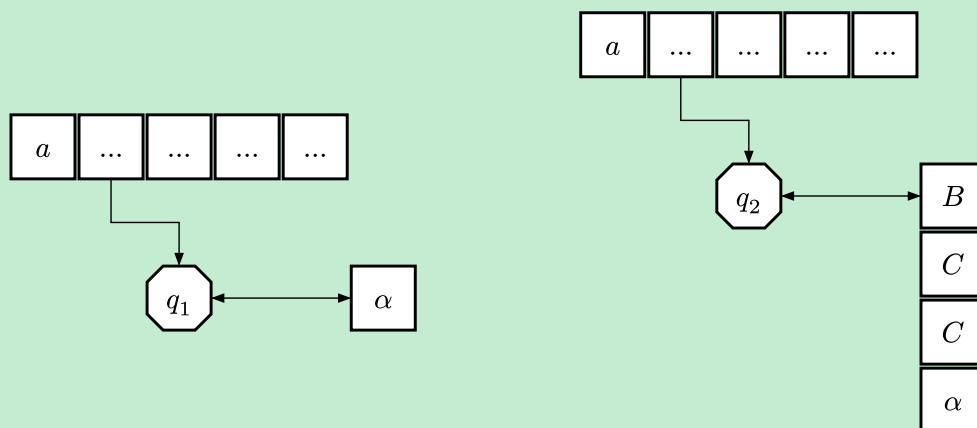


Con α nel disegno si intende una stringa in Γ^* perché oltre ad A potremmo avere altro.

Cosa vuol dire quella regola della funzione di transizione? Ci sta dicendo che se ci troviamo nello stato q , leggiamo a sul nastro leggiamo A sulla cima della pila, possiamo:

- andare in q_1 e non mettere altro sulla pila, praticamente consumando un simbolo in input;
- andare in q_2 e mettere sulla pila la stringa BCC .

Vediamo la rappresentazione dei due casi nei quali possiamo finire.

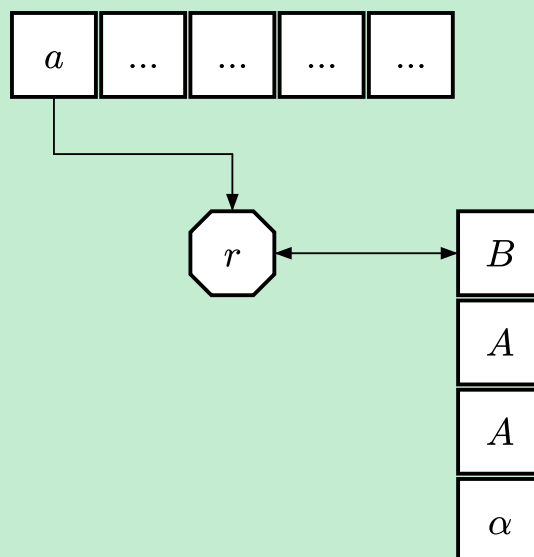


Per convenzione, quando inseriamo una stringa sulla pila, l'inserimento avviene da destra verso sinistra. In poche parole, se inseriamo la stringa $X \in \Gamma^*$ nella pila, se la togliessimo noi leggeremmo, in ordine, esattamente X . In altre parole ancora, quando leggiamo una stringa da inserire è come se la stessimo leggendo dall'alto verso il basso.

Abbiamo la possibilità anche di fare delle ε -mosse: supponiamo di aggiungere la regola

$$\delta(q, \varepsilon, A) = \{(r, BAA)\}.$$

Ora abbiamo tre scelte a disposizione. Le ε -mosse possiamo vederle come delle **mosse interne**, che avvengono senza leggere l'input, e che ci permettono di spostarci negli stati modificando eventualmente la pila.



Una **configurazione** è una fotografia dell'automa in un dato istante di tempo, e ci dice quali sono le informazioni rilevanti per il futuro per definire al meglio la macchina, ovvero:

- lo **stato corrente**;
- il **contenuto del nastro** che ci manca da leggere;
- il **contenuto della pila**.

Una configurazione è quindi una **tripla**

$$(q, ay, A\alpha)$$

che contiene lo stato corrente, il contenuto del nastro ancora da leggere indicato dal carattere corrente a unito al resto della stringa y e il contenuto della pila indicato dal carattere in testa A e dal resto della pila α .

Una **mossa** è l'applicazione della funzione di transizione, ovvero un passaggio

$$(q, ay, A\alpha) \longrightarrow (p, y, \gamma\alpha) \iff (p, \gamma) \in \delta(q, a, A).$$

Analogamente, un passaggio che usa le ε -mosse è un passaggio

$$(q, ay, A\alpha) \longrightarrow (p, ay, \gamma\alpha) \longrightarrow (p, \gamma) \in \delta(q, \varepsilon, A).$$

Una **computazione** è una serie di mosse che partono da una configurazione iniziale e mi portano in una configurazione finale. Di queste ultime parleremo tra poco. Torniamo sulle computazioni.

Come con i passi di derivazione, una computazione che usa una sola mossa si indica con

$$C' \vdash C''.$$

Se invece una computazione impiega k passi, si indica con

$$C' \vdash^k C''.$$

Infine, per indicare una computazione con un numero generico di passi, maggiori o uguali a zero, si usa

$$C' \vdash^* C''.$$

14.2.2. Accettazione

Abbiamo parlato di arrivare in una configurazione accettante, ma quando **accettiamo**? Dobbiamo capire da dove partire e dove arrivare.

Quando partiamo abbiamo la stringa w sul nastro, ci troviamo nello stato iniziale q_0 e abbiamo Z_0 sulla pila: questa è detta **configurazione iniziale** ed è la tripla

$$(q_0, w, Z_0).$$

Le **configurazioni finali** dipendono dal tipo di nozione di accettazione che vogliamo utilizzare.

L'**accettazione per stati finali** ci obbliga a leggere tutto l'input e a finire in uno stato finale, con la pila che contiene quello che vuole, ovvero dobbiamo arrivare in una configurazione

$$(q, \varepsilon, \gamma)$$

dove lo stato q è finale. Il **linguaggio accettato per stati finali** è l'insieme

$$L(M) = \left\{ w \in \Sigma^* \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \gamma) \mid q \in F \wedge \gamma \in \Gamma^* \right\}.$$

Questa nozione è comoda perché vede i PDA come una **estensione** degli automi a stati finiti.

L'**accettazione per pila vuota** invece è una nozione più naturale: tutto ciò che metto nella pila lo devo anche buttare via. Possiamo arrivare in un qualsiasi stato, basta aver svuotato la pila. Il **linguaggio accettato per pila vuota** è l'insieme

$$N(M) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash (q, \varepsilon, \varepsilon) \mid q \in Q\}.$$

Se svuotiamo la pila prima di finire l'input allora quella computazione si blocca perché noi dobbiamo sempre leggere qualcosa dalla pila.

Le due accettazioni sono **equivalenti**, o meglio, possiamo passare da una accettazione all'altra ma i linguaggi che accettano sono differenti. A parità di automa M , gli insiemi $L(M)$ e $N(M)$ in generale sono diversi, ma possiamo passare da un modello all'altro mantenendo il linguaggio accettato con facilità. Una ulteriore nozione di accettazione unisce stati finali e pila vuota, ma rimane comunque equivalente. Avere due nozioni è comodo: se una versione ci esce estremamente comoda allora la andiamo ad utilizzare, altrimenti andremo ad utilizzare l'altra.

Vediamo ora qualche esempio.

Esempio 14.2.2.1: Prendiamo il nostro migliore amico, il linguaggio

$$L = \{a^n b^n \mid n \geq 1\}.$$

Lo possiamo riconoscere con un PDA, visto che abbiamo visto che non è regolare? Bhe sì: con i DFA non riusciamo a ricordare il numero di a e poi confrontare questo numero con le b , mentre ora riusciamo a farlo, le pile sanno contare.

Possiamo pensare ad un automa che ogni volta che legge una a butta una A dentro la pila, e quando legge una b toglie una A dalla pila. Accettiamo se abbiamo messo n caratteri A dentro la pila e poi ne abbiamo tolti n , quindi qua viene comoda l'**accettazione per pila vuota**.

Andiamo a definire la funzione di transizione.

Iniziamo a togliere Z_0 dalla pila e inseriamo la prima A in segno di aver letto la prima a della stringa, che abbiamo per forza per definizione, quindi

$$\delta(q_0, a, Z_0) = \{(q_0, A)\}.$$

Utilizziamo lo stato q_0 per leggere tutte le a della stringa, ovvero

$$\delta(q_0, a, A) = \{(q_0, AA)\}.$$

Appena troviamo una b iniziamo a cancellare e cambiamo stato, visto che non ci aspettiamo più delle a nella stringa, quindi

$$\delta(q_0, b, A) = \{(q_1, \varepsilon)\}.$$

Inseriamo ε sulla stringa perché la A da cancellare per la lettura di b è già stata cancellata dalla lettura.

Andiamo a terminare la lettura delle b , quindi

$$\delta(q_1, b, A) = \{(q_1, \varepsilon)\}.$$

Abbiamo detto che accettiamo per pila vuota, quindi $L = N(M)$.

Esempio 14.2.2.2: Se invece volessimo accettare il linguaggio precedente per **stati finali**?

Non dobbiamo cancellare Z_0 dalla pila perché se una stringa viene accettata cancella tutta la pila, quindi ci serve un carattere fittizio dentro per poterlo leggere e spostarci in uno stato finale. Modifichiamo quindi la mossa iniziale con la mossa

$$\delta(q_0, a, Z_0) = \delta(q_0, AZ_0).$$

Se abbiamo una stringa del linguaggio alla fine delle b dobbiamo spostarci in uno stato finale, quindi aggiungiamo la regola

$$\delta(q_1, \varepsilon, Z_0) = \{(q_f, Z_0)\} \mid q_f \in F.$$

Con queste modifiche abbiamo $L = L(M)$.

Esempio 14.2.2.3: Se invece volessimo accettare anche ε ? Il linguaggio diventa

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

Con l'**accettazione per pila vuota**, nello stato iniziale possiamo aggiungere una regola che svuota subito la pila, ovvero aggiungiamo la regola

$$\delta(q_0, \varepsilon, Z_0) = \{(q_0, \varepsilon)\}.$$

Stiamo scommettendo che l'input è già finito, ovvero abbiamo solo ε sul nastro, ma questo ha appena aggiunto il **non determinismo** al nostro automa a pila.

Con l'**accettazione per stati finali** invece ci spostiamo direttamente nello stato q_f a partire da q_0 , ovvero aggiungiamo la regola

$$\delta(q_0, \varepsilon, Z_0) = \{(q_f, \varepsilon)\}.$$

Come prima, abbiamo aggiunto del **non determinismo** all'automa a pila, ma questo lo possiamo togliere: come facciamo a fare ciò?

Introduciamo uno stato q_I finale che diventa anche iniziale al posto di q_0 , quindi ora

$$F = \{q_I, q_f\}.$$

Se inseriamo sul nastro la stringa vuota allora noi accettiamo, perché siamo in uno stato finale e non abbiamo altri simboli da leggere. Per passare poi al vecchio automa mettiamo una regola

$$\delta(q_I, a, Z_0) = \{(q_0, AZ_0)\}.$$

14.2.3. Determinismo VS non determinismo

Con il termine **non determinismo** non intendiamo le ε -mosse da sole, quelle le possiamo avere, ma intendiamo un mix tra mosse che leggono e mosse che non leggono.

Definizione 14.2.3.1 (Determinismo): Sia M un PDA. Allora M è **deterministico** se:

1. ogni volta che ho una ε -mossa da un certo stato e con un certo simbolo sulla pila, non ho mosse che leggono simboli dal nastro a partire dallo stesso stato e con lo stesso simbolo sulla pila, ovvero

$$\forall q \in Q \quad \forall A \in \Gamma \quad \delta(q, \varepsilon, A) \neq \emptyset \implies \forall a \in \Sigma \quad \delta(q, a, A) = \emptyset;$$

2. come nel caso classico, considero un carattere, o anche ε , allora a parità di stato corrente e simbolo sulla pila, ho al massimo una transizione possibile, ovvero

$$\forall q \in Q \quad \forall A \in \Gamma \quad \forall \sigma \in \Sigma \cup \{\varepsilon\} \quad |\delta(q, \sigma, A)| \leq 1.$$

A differenza del caso classico, il determinismo e il non determinismo non sono ugualmente potenti: un automa a pila non deterministico è **più potente** di un automa a pila deterministico, che riconosce una sottoclasse di linguaggi diversa dai linguaggi di tipo 2, che sono riconosciuti dai PDA non deterministici.

14.2.4. Trasformazioni

Avevamo parlato dell'**equivalenza** dell'accettazione per stati finali e per pila vuota: infatti, esistono due trasformazioni che permettono di passare da un automa all'altro, mantenendo il linguaggio di partenza riconosciuto inalterato. L'equivalenza infatti ci diceva che, partendo da un automa M che riconosce per stati finali, abbiamo una trasformazione che ci dà M' che riconosce per pila vuota che riconosce lo stesso linguaggio di M , e viceversa.

Stati finali \rightarrow pila vuota Dobbiamo trasformare un automa che accetta per stati finali in un automa che accetta per pila vuota. Con quest'ultimo simuliamo il primo, e ogni volta che vado in uno stato finale mi sposto in uno **stato di svuotamento**, che se raggiunto in mezzo blocca la pila, ma se raggiunto alla fine mi fa accettare.

Abbiamo quindi $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ un PDA con $L = L(M)$. Definiamo ora

$$M' = (Q \cup \{q_e, q'_0\}, \Sigma, \Gamma \cup \{X\}, \delta', q'_0, X, \emptyset)$$

un PDA tale che:

- per metterci in una situazione piacevole per la fine usiamo un **truccaccio** definito dalla regola

$$\delta(q'_0, \varepsilon, X) = \{(q_0, Z_0 X)\},$$

ovvero prima di far partire la computazione dell'automa M andiamo ad inserire un carattere X in fondo alla pila, vedremo dopo perché;

- l'automa deve eseguire **tutte le mosse** di M , ovvero

$$\forall q \in Q \quad \forall \sigma \in \Sigma \cup \{\varepsilon\} \quad \forall Z \in \Gamma \quad \delta(q, \sigma, Z) \subseteq \delta'(q, \sigma, Z),$$

che scritto così significa che tutte le mosse che trovavamo nell'applicazione di delta ad una certa tripla le abbiamo anche nella nuova funzione di transizione, che però conterrà anche altro, che vedremo tra poco;

- aggiungiamo uno **stato di svuotamento** per pulire la pila, definito dalle regole

$$\begin{aligned}\forall q \in F \quad \forall Z \in \Gamma \cup \{X\} \quad (q_e, \varepsilon) &\in \delta'(q, \varepsilon, Z) \\ \forall Z \in \Gamma \cup \{X\} \quad \delta'(q_e, \varepsilon, Z) &= \{(q_e, \varepsilon)\},\end{aligned}$$

ovvero con la prima regola, ogni volta che mi trovo in uno stato finale **non deterministicamente** mi posso spostare nello stato di svuotamento, mentre con la seconda regola effettivamente svuoto.

A cosa ci serve il **carattere** X ? Facciamo finta di non mettere il carattere X . Se M accetta una stringa x arrivando con la pila vuota nessun problema, non ci spostiamo nello stato di svuotamento ma abbiamo la pila vuota quindi ottimo. Se invece M non accetta una stringa x ma arriva alla fine con la pila vuota, il simbolo X messo all'inizio ci copre da una eventuale accettazione errata, perché non riusciremo ad andare nello stato di svuotamento per avere la pila vuota, anche se M ci finisce in quel modo. Diciamo che abbiamo messo X come se fosse una guardia, che ci copre questo preciso caso.

Purtroppo, con questa costruzione abbiamo buttato dentro del **non determinismo** quando facciamo i passaggi in q_e da uno stato finale.

Pila vuota \rightarrow stati finali Il percorso opposto invece parte da un PDA

$$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

tale che $L = N(M)$. Definiamo il PDA

$$M' = (Q \cup \{q'_0, q_f\}, \Sigma, \Gamma \cup \{X\}, \delta', q'_0, X, \{q_f\})$$

che come idea ha quella di simulare M e, ogni volta che arriva con pila vuota, ci spostiamo nello stato finale. Vediamo i vari passi:

- come prima, usiamo un **truccaccio** per infilare X sotto la pila, quindi abbiamo la regola

$$\delta'(q'_0, \varepsilon, Z_0) = \{(q_0, Z_0 X)\}$$

che usiamo per inserire X come trigger per andare in uno stato finale;

- simuliamo l'automa M senza aggiungere niente, quindi

$$\forall q \in Q \quad \forall \sigma \in \Sigma \cup \{\varepsilon\} \quad \forall Z \in \Gamma \quad \delta'(q, \sigma, Z) = \delta(q, \sigma, Z);$$

- ogni volta che leggiamo X sulla cima della pila vuol dire che M ha svuotato la pila, quindi devo andare nello stato finale, ovvero

$$\forall q \in Q \quad \delta'(q, \varepsilon, X) = \{(q_f, \varepsilon)\};$$

ovviamente, se andiamo in questo stato a metà stringa ci blocchiamo, altrimenti se ci andiamo alla fine è tutto ok.

A differenza di prima, se partiamo da un automa **deterministico**, quello che otteniamo è ancora un automa **deterministico**.