Automi, Calcolabilità e Complessità: Compendio

Anthony

Primo semestre 2022

Contents

Part I
Automi

Gli automi a stati finiti sono un modello per computer con una quantità limitata di memoria. Ad esempio il sensore di una porta automatica è un esempio rappresentabile attraverso un automa.

1 Automi finiti

Un automa è rappresentabile attraverso una *tabella delle transizioni*, ovvero una matrice che, per ogni stato e per ogni input, esplicita il comportamento dell'automa.
Un altro modello rappresentativo è quello del diagramma di stato, lo stesso utilizzato nelle *Markov Chain*.

1.1 Esempio di un DFA (Deterministic Finite Automaton) M

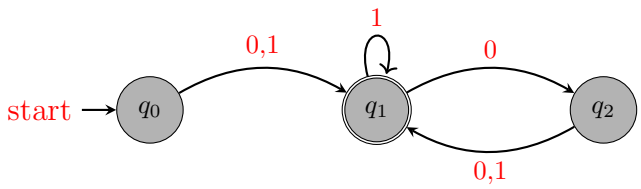


Figure 1: DFA M

Nel diagramma di stato identifichiamo tre stati: q_0, q_1, q_2. Lo stato iniziale q_1 è indicato da un arco entrante e non uscente da uno stato. Lo stato *accettante* q_1 è quello con un doppio cerchio. Gli archi che vanno da uno stato all'altro sono chiamati transizioni.
Quando l'automa riceve una stringa in input la elabora e *accetta* o *rifiuta* l'input. In particolare, se la computazione termina su uno stato accettante, l'automa accetta l'input, altrimenti lo rifiuta.

1.2 Def: DFA (Deterministic Finite Automaton)

Un DFA è una 5-tupla $(Q, \Sigma, \delta, q0, F)$

- Q: Insieme finito degli stati;
- Σ: Alfabeto di input;
- δ: Funzione di transizione.

$$Q \times \Sigma \Rightarrow Q$$

Ovvero prende uno stato $q \in Q$ e un simbolo dell'alfabeto di input $\sigma \in \Sigma$ ed applica σ a q, restituendo uno stato $q' \in Q$.

- q_0: Stato iniziale
- $F \subseteq Q$: Sottoinsieme che comprende tutti e soli gli stati di accettazione.

1.3 Def: Linguaggio di un modello computazionale

Sia M un modello computazionale[1] e sia A l'input che M accetta, A viene detto *linguaggio* di M.

$$L(M) = A$$

ovvero *il linguaggio di M è A*. Banalmente, se M non accetta alcuna stringa, allora $L(M) = \varnothing$

1.3.1 Stringa accettata da un DFA

Una stringa $w \in \Sigma^*$ è accettata da un DFA $M = (Q, \Sigma, \delta, q_0, F)$ se $\delta^*(q_0, w) \in F$ ovvero:

$$L(M) = \{w \in \delta^* : \delta^*(q_0, w) \in F\}$$

1.4 Transizione Estesa

Obiettivo: Vogliamo valutare la funzione di transizione δ su un'intera stringa, non su un singolo carattere. Normalmente δ legge un carattere dell'alfabeto Σ alla volta, ma la *transazione estesa* permette di prendere una stringa analizzandola carattere per carattere.

1.4.1 Notazione di Σ^*

$$Q \times \Sigma^* \Rightarrow Q$$

1.4.2 Proprietà della transizione estesa

1. L'elemento neutro, ovvero la stringa vuota, è in Σ^*

$$\varepsilon \in \Sigma^*$$

$$\delta^*(q, \varepsilon) = \delta(q, \varepsilon)$$

2. Data una stringa ax dove $a \in \Sigma$ e $x \in \Sigma^*$ è possibile decomporre la stringa, applicando prima $\delta(a)$ e poi $\delta^*(x)$.

$$\delta^*(q, ax) = \delta^*(\delta(q, a), x)$$

[1] vale per DFA, NFA, PDA, TM, etc...

1.5 Configurazioni di un modello computazionale

Una configurazione è un insieme di valori che specifica lo stato corrente del modello computazionale, il corrente simbolo di input che sta per essere processato e l'input restante ancora da processare.

1.5.1 Configurazione in un DFA

Una configurazione in un DFA è rappresentata da una coppia $(q_i, x) \in Q \times \Sigma^*$ in cui q_i è lo stato attuale dell'automa, mentre x è la stringa di input ancora da processare.

1.6 Passo di computazione

Un passo computazionale in un modello computazionale è una singola operazione effettuata dal modello generata dal processamento di un singolo simbolo in input applicando a esso la funzione di transizione.

1.6.1 La relazione binaria che ne consegue dai passi computazionali

Siano $p, q \in Q; a \in \Sigma; x \in \Sigma^*$ e sia $\vdash_M$ la relazione binaria definita come segue:

$$(p, ax) \vdash_M (q, x) \Leftrightarrow \delta(q, a) = q$$

Ovvero date due coppie di stati e di stringhe, esse sono in relazione $\vdash_M$ se e solo se restituiscono come output lo stesso stato. Ovvero se e solo se sono parte dello stesso cammino computazionale.
Sia ora $\vdash_M^*$ la chiusura riflessiva e transitiva di $\vdash_M$:

(i) **riflessiva:** $\forall x \in D : R(x, x)$ è vera
$(q, x) \vdash_M^* (q, x)$

(ii) **transitiva:** $\forall x, y, z \in D : R(x, y) \wedge R(y, z) \Rightarrow R(x, z)$
La transitività ci permette di collassare più passi computazionali:

$$\Big[(q, aby) \vdash_M^* (p, by)\Big] \wedge \Big[(p, by) \vdash_M^* (r, y)\Big] \Rightarrow (q, aby) \vdash_M^* (r, y)$$

1.7 Linguaggi regolari

Un linguaggio è chiamato *linguaggio regolare* se un automa finito lo riconosce. Quindi l'insieme dei linguaggi regolari contiene tutti e soli i linguaggi che sono accettati da un qualche DFA.

$$REG = \{L \in \Sigma^* : \exists \text{ DFA } M \text{ t.c. } L(M) = L\}^2$$

[2] Tale definizione sarà poi estesa anche per gli NFA

1.7.1 Eredità delle operazioni e delle proprietà degli insiemi

I linguaggi, essendo un insieme di stringhe, ereditano le operazioni degli insiemi. Ad esempio sia Σ un alfabeto e siano $L_1, L_2 \in \Sigma^*$:

- **Unione**: $L1 \cup L2 = \{x \in \Sigma^* : x \in L1 \vee x \in L2\}$
- **Intersezione**: $L1 \cap L2 = \{x \in \Sigma^* : x \in L1 \wedge x \in L2\}$
- **Complemento**: $\neg L1 = \{x \in \Sigma^* : x \notin L1\}$

Vengono ereditate anche altre proprietà degli insiemi:

- $\neg(L1 \cup L2) = \neg L1 \cap \neg L2$
- $\neg(L1 \cap L2) = \neg L1 \cup \neg L2$
- ...

1.7.2 Operazioni sui linguaggi regolari: Concatenazione

Siano $x, y \in \Sigma^*$ tali che:

$$x = \{a_1, \ldots, a_n : a_i \in \Sigma,\ n \in \mathbb{N}, n > 0\}$$

$$y = \{b_1, \ldots, b_m : b_j \in \Sigma,\ m \in \mathbb{N}, m > 0\}$$

La *concatenazione* di x con y è definita come segue:

$$xy = \{a_1, \ldots, a_n, b_1, \ldots, b_m : a_i, b_j \in \Sigma,\ n, m \in \mathbb{N}, n > 0, m > 0\}$$

Banalmente è possibile effettuare la concatenazione tra una parola in Σ^* con la parola vuota ε.
Sia x la stessa dell'esempio precedente:

$$x\varepsilon = \varepsilon x = x$$

Da ciò possiamo ricavare una definizione ricorsiva di concatenazione.

Concatenazione di linguaggi Più in generale possiamo effettuare una concatenazione tra due linguaggi. Siano L_1 e L_2 due linguaggi:

$$L1 \circ L2 = \{xy : x \in L1 \wedge y \in L2\}$$

Es: Dato $\Sigma = \{a, b\}$:

$$L1 = \{a, ab, ba\}, L2 = \{ab, b\}$$

$$L1 \circ L2 = \{aab, ab, abab, abb, baab, bab\}$$

Da cui ne deduciamo banalmente:

$$L1 \circ \varnothing = \varnothing \circ L1 = L1$$

Potenza: un caso particolare di concatenazione Sia $x \in \Sigma^*, n \in \mathbb{N}$:

$$x^0 = \varepsilon$$

$$x^n = \underbrace{xx \dots x}_{\text{n volte}} \forall n \geq 0$$

$$x^{n+1} = x^n x$$

Lo stesso principio è applicabile ai linguaggi, non solo alle stringhe. Sia $L \in \Sigma^*$:

$$L^0 = \varepsilon$$

$$L^{n+1} = L^n L : \forall n \geq 0$$

1.7.3 Operazioni sui linguaggi regolari: Operazione stella

Operazione unaria applicabile a un singolo linguaggio. L'operazione stella agisce concatenando un numero qualsiasi di stringhe in A per ottenere una stringa nel nuovo linguaggio.

$$L^* = \{x_1 x_2 \dots x_k | k \geq 0, \forall i : x_i \in A\}$$

L'operazione stella è paragonabile a un'unione di potenze dello stesso linguaggio:

$$\begin{aligned} L^* &= \{x_1 x_2 \dots x_k | k \geq 0, \forall i : x_i \in A\} \\ &= \bigcup L^n : n \geq 0 \\ &= \varepsilon \cup L^1 \cup L^2 \cup L^3 \cup \dots \end{aligned}$$

Esempio, sia $L = \{a, b\}$:

$$L^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \dots\}$$

1.8 Def: Classe di linguaggi

Una classe di linguaggi C è un insieme di linguaggi che soddisfano una certa proprietà p. È possibile calcolare la *chiusura* di C rispetto alla proprietà p. In generale, una classe di oggetti è *chiusa* rispetto a un'operazione se l'applicazione di questa operazione a elementi della classe restituisce un oggetto ancora nella classe.

1.8.1 Teorema: REG è chiusa per l'unione

REG è chiusa rispetto all'operazione di unione.

Dim: Siccome $L_1, L_2 \in REG, \exists M_1, M_2$ DFA che li riconoscono rispettivamente. Progettiamo un DFA M che riconosce $L_1 \cup L_2$. Ovvero, dato un certo x, M accetta $x \Leftrightarrow x \in L_1 \cup L_2$.
Per l'assenza di memoria che caratterizza i DFA non possiamo modellare banalmente M in modo che prima simula l'input su M_1 e poi su M_2.
Idea: Possiamo tener traccia degli stati di M_1, M_2 ricordando la coppia di stati in cui si troverebbero. In questo modo M *ricorda* lo stato in cui ciascuna macchina sarebbe se avesse letto l'input fino a quel punto. Le transizioni vanno da coppia in coppia, aggiornando sia la componente di M_1 che quella di M_2. Gli stati accettanti di M sono quelle coppie tali che M_1 o M_2 è in uno stato accettante. Siano M_1, M_2 definiti come segue:

$$M_1 = (Q_1, \Sigma_1, \delta_1, q_1, F_1)$$

$$M_2 = (Q_2, \Sigma_2, \delta_2, q_2, F_2)$$

Costruiamo M:

$$M = (Q, \Sigma, \delta, q0, F)$$

Modellando opportunamente ogni sua componente:

- $Q = \{(r_1, r_2) : r_1 \in Q_1, r_2 \in Q_2\} = Q_1 \times Q_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- δ, la funzione di transizione, è definita come segue:

$$\forall (r_1, r_2) \in Q, \forall a \in \Sigma : \delta((r_1, r_2), a) = (\delta_1(r1, a), \delta_2(r2, a))$$

- $q_0 = (q_1, q_2)$
- $F = \{(r_1, r_2) : r_1 \in F_1 \vee r_2 \in F_2\} = (F_1 \times Q_2) \cup (F_2 \times Q_1)$

■

2 Non determinismo

Quando una macchina *deterministica* in un dato stato legge il simbolo di input successivo sappiamo qual è lo stato sucessivo, perché univocamente determinato dalla sua funzione di transizione. In una macchina *non deterministica* possono esistere diverse scelte per lo stato successivo.

2.1 Esempio di NFA (Non-deterministic Finite Automaton):

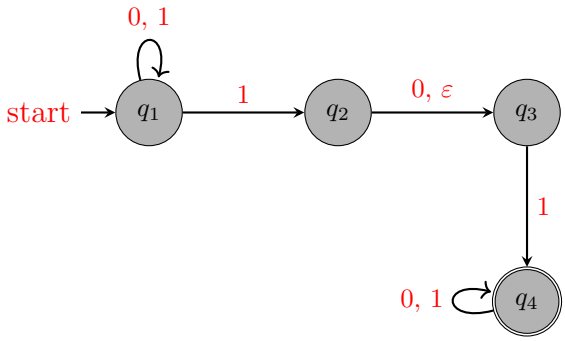


Figure 2: NFA N

In questo esempio N dallo stato q_1 il carattere dell'alfabeto '1' permette di fare due cose: rimanere in q_1 e transitare in q_2. Il discorso è analogo per lo stato q_2: ε divide un ramo computazionale in due.

2.1.1 Un DFA è un caso particolare di NFA

Eseguire un NFA equivale a eseguire più DFA in parallelo, uno per ciascun ramo computazionale. Possiamo notare che l'albero computazionale generato da un NFA non ha una sola foglia come quello del DFA, difatti il non determinismo è una generalizzazione del determinismo, quindi ogni automa finito deterministico è anche un automa finito non deterministico.

2.2 Def: NFA

Dato Σ, sia $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e sia $\mathcal{P}(Q)$ l'insieme delle parti di Q[3], un automa non deterministico a stati finiti (NFA) è una 5-tupla $N = (Q, \Sigma_\varepsilon, \delta, q0, F)$ in cui:

- la definizione per Q, Σ, q_0, F è invariata dalla definizione di DFA;
- $\delta : Q \times \Sigma_\varepsilon \to \mathcal{P}(Q)$ è la funzione di transizione.

2.2.1 Accettazione in un NFA

Diciamo che N accetta w se possiamo scrivere w come $w = y_1 y_2 \dots y_m$ dove $y_i \in \Sigma_\varepsilon$ ed esiste una sequenza di stati $r_0, r_1, \dots, r_m \in Q$ che soddisfa le tre seguenti condizioni:

1. $r_0 = q_0$;
2. $\forall i \in [0, m-1] : r_{i+1} \in \delta(r_i, y_{i+1})$;
3. $r_m \in F$.

[3] collezione di tutti i sottoinsiemi

La condizione 1. fa sì che la macchina inizia dallo stato iniziale. La condizione 2. dice che lo stato r_{i+1} è uno dei possibili stati successivi quando N è nello stato r_i e sta leggendo y_{i+1}. La terza condizione esplicita che la macchina accetta w se l'ultimo stato è uno stato accettante. Alternativamente N accetta w se:

$$\exists q \in F : (q_0, w) \vdash_M^* (q, \varepsilon)$$

2.3 Def: L'insieme dei linguaggi riconosciuti da NFA

$$\mathcal{L}(NFA) = \{L \subseteq \Sigma_\varepsilon : \exists M \text{ NFA t.c. } L(M) = L\}$$

2.4 Teorema: NFA e DFA sono equivalenti

$$\mathcal{L}(NFA) = \mathcal{L}(DFA)$$

Dove $\mathcal{L}(DFA)$ è l'insieme dei linguaggi riconosciuti da DFA.

2.4.1 Dim (due direzioni):

($\Rightarrow$): $L \in \mathcal{L}(DFA) \Rightarrow L \in \mathcal{L}(NFA)$
Banale: se $L \in \mathcal{L}(DFA), \exists$ DFA M che riconosce L, ma M è un caso particolare di NFA, allora $L \in \mathcal{L}(NFA)$

($\Leftarrow$): $L \in \mathcal{L}(NFA) \Rightarrow L \in \mathcal{L}(DFA)$
Assumiamo per ipotesi che $\exists$ NFA $N = (Q_N, \Sigma, \delta_N, q_0^N, F_N)$ che riconosce L. Dobbiamo modellare un DFA $D = (Q_D, \Sigma, \delta_D, q_0^D, F_D)$ che riconosce L.
Idea: creiamo uno stato in Q_D per ogni possibile insieme di stati in Q_N, dopodiché simuliamo deterministicamente ogni livello dell'albero di computazione. Assumiamo inizialmente un caso semplice, senza ε archi:

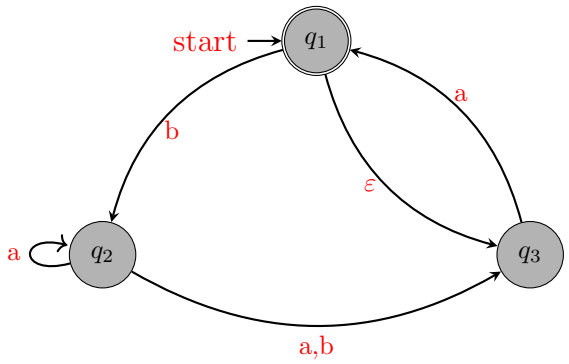


Figure 3: NFA N

Sia quindi $Q_D = \mathcal{P}(Q_N)$. Prendendo come riferimento N abbiamo che:

$$\mathcal{P}(Q_N) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$$

Modelliamo quindi l'automa D con uno stato per ogni oggetto in $\mathcal{P}(Q_N)$.

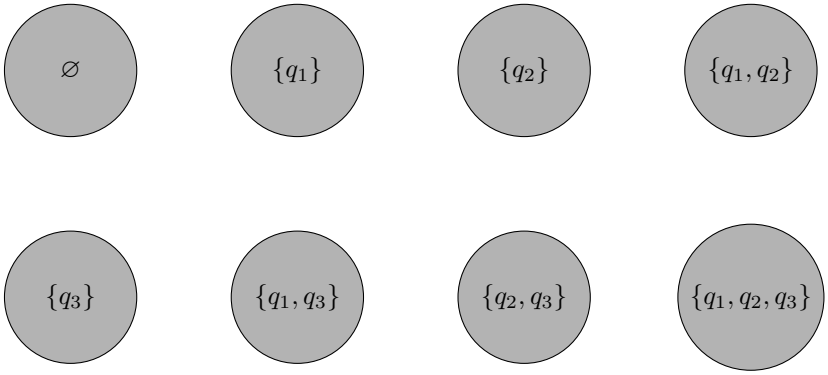


Figure 4: DFA D senza archi

Ora che abbiamo l'insieme degli stati di D, facciamo considerazioni sullo stato iniziale, gli stati finali e δ_D la funzione di transizione prendendo in considerazione gli ε-archi:

- $q_0^D = q_0^N$
  $\Rightarrow$ nel nostro esempio $q_0^D = \{\{1\}\}$
- $F_D = \{R \in Q_D : R \text{ t.c. } R \cap F_N \neq \varnothing\}$
  $\Rightarrow$ nel nostro esempio $F_D = \{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$
  Ovvero: D accetta nello stato R se almeno uno stato $Q_N \in R$ è finale. Vale a dire tutti quelli che contengono $\{1\}$, poiché l'intersezione non è vuota.
- δ_D: sia $R \in Q_D$ e sia a un carattere $\in \Sigma$, allora $\delta_D(R, a) = \bigcup_{r \in R} \delta_N(r, a)$
  Alcuni esempi che prendono come riferimento sempre D e il suo corrispettivo N:
  - da q_1 leggo $a \to \varnothing \Rightarrow \delta_D(\{q_1\}, a) = \varnothing$
  - da q_1 leggo $b \to q_2 \Rightarrow \delta_D(\{1\}, b) = \{q_2\}$
  - da q_2 leggo $a \to \{q_2, q_3\} \Rightarrow \delta_D(\{2\}, a) = \{q_2, q_3\}$

  Dobbiamo analizzare anche gli ε-archi. Per convenzione, consideriamo gli ε-archi solo dopo a lettura di un qualsiasi input. Ad esempio, prendendo come riferimento N, se ci troviamo nello stato q_3 e leggiamo a in input, viene considerato anche l'ε-arco.
  Introduciamo qualche ulteriore notazione. Per ogni stato R di M, definiamo $E(R)$ come la collezione di stati che possono essere raggiunti dagli elementi di R proseguendo solo con ε-archi, includendo gli stessi elementi. Formalmente, per $R \subseteq Q$ sia:

  $E(R) = \{q : q \text{ può essere raggiunto da } R \text{ attraverso 0 o più } \varepsilon\text{-archi }\}$

Modifichiamo quindi la funzione di transizione sostituendo $\delta(r, a)$ con $E(\delta(r, a))$:

$$\delta'(R, a) = \{q \in Q : q \in E(\delta(r, a)) \text{ per qualche } r \in R\}$$

Continuando per ogni stato di D andiamo a modellare un DFA che legge il linguaggio dell'NFA. Di seguito il DFA D dopo le considerazioni effettuate:

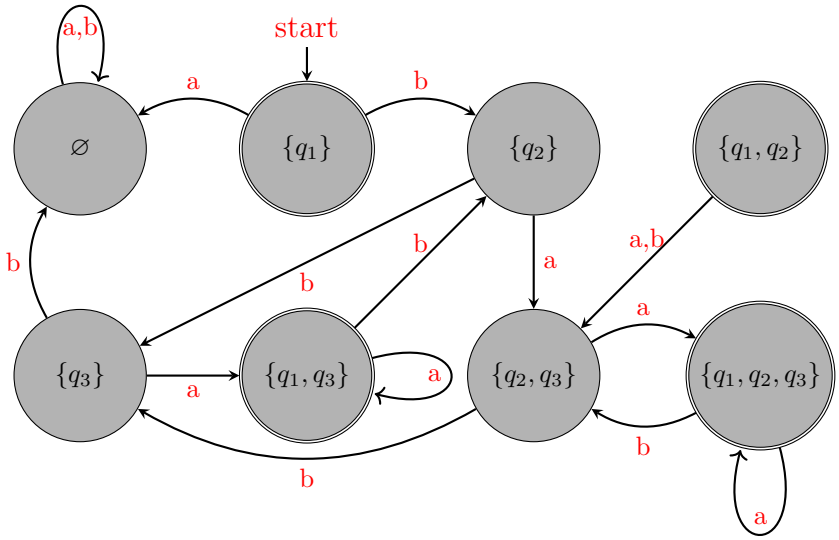


Figure 5: DFA D

A ogni passo di computazione di D su un input, D entra in uno stato che corrisponde al sottoinsieme di stati in cui N si troverebbe con lo stesso input.

■

2.5 Linguaggi regolari (continua dalla sezione 1.7.)

Abbiamo visto che un DFA è un caso particolare di un NFA con un singolo ramo computazionale. Abbiamo anche mostrato che un NFA è simulabile con un DFA, possiamo quindi estendere la nostra definizione di linguaggi regolari.

2.5.1 Def: linguaggi regolari

$$REG = \{L \in \Sigma^* : \exists \text{ NFA } M \text{ t.c. } L(M) = L\}$$

2.6 Chiusure e REG:

2.6.1 REG è chiusa per $\cup$

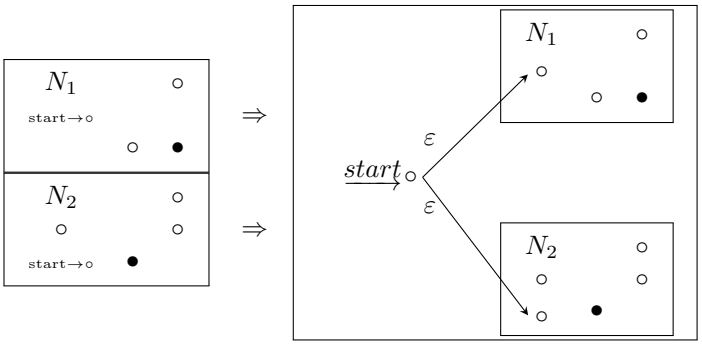
Abbiamo già dimostrato che la classe di linguaggi regolari è chiusa per l'unione. Ora facciamo lo stesso generalizzando per gli NFA.

Dim: chiusura Prendiamo due NFA definiti come segue:

$$N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1), L(N_1) = L_1$$

$$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2), L(N_2) = L_2$$

Costruiamo N t.c. N accetta l'input se e solo se N_1 o N_2 accetterebbero quell'input. Aggiungiamo un nuovo stato q_0 che con ε-archi raggiunge gli stati iniziali di N_1, N_2:



1

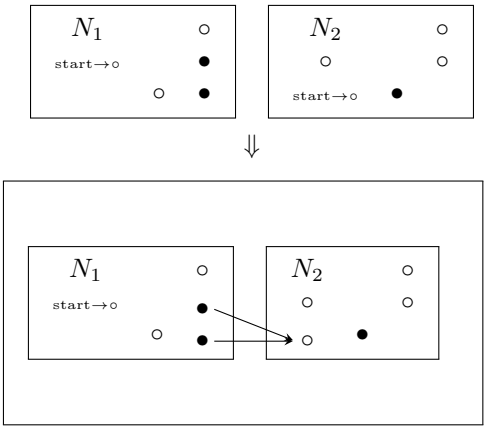
Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ che riconosce $L_1 \cup L_2$:

- $Q = Q_1 \cup Q_2 \cup \{q_0\}$
  Gli stati di N sono tutti gli stati di N_1 e N_2 con l'aggiunta del nuovo stato q_0.
- q_0 è lo stato iniziale di N.
- $F = F_1 \cup F_2$
  Gli stati accettanti di N sono tutti gli stati accettanti di N_1 e N_2.
- Definiamo la δ t.c. $\forall q \in Q, a \in \Sigma_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & q \in Q_1 \\ \delta_2(q,a), & q \in Q_2 \\ \{q_1, q_2\}, & q = q_0, a = \varepsilon \\ \varnothing, & a \neq \varepsilon \end{cases}$$

2.6.2 REG è chiusa per $\circ$

Dati i linguaggi regolari di N_1 e N_2 vogliamo provare che la loro concatenazione è un linguaggio regolare. L'idea è simile alla precedente: prendere due NFA e combinarli.
Poniamo come stato iniziale di N lo stato iniziale di N_1. Gli stati accettanti di N_1 hanno ulteriori ε-archi che permettono di diramarsi in N_2 ogni volta che N_1 è in uno stato accettante, indicando che è stato letto l'input che costituisce una stringa in L_1. Gli stati accettanti di N sono gli stessi di N_2. Quindi N accetta quando l'input può essere diviso in due parti: la prima parte deve essere accettata da N_1, mentre la seconda da N_2.



Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ che riconosce $L_1 \circ L_2$:

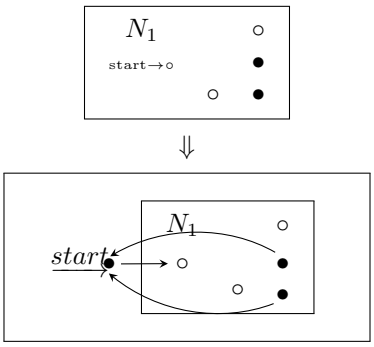
- $Q = Q_1 \cup Q_2$
- $q_0 = q_1$
- $F = F_2$
- Definiamo la δ t.c. $\forall q \in Q, a \in \Sigma_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & q \in Q_1, q \notin F_1 \\ \delta_1(q,a), & q \in F_1, a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\}, & q \in F_1, a = \varepsilon \\ \delta_2(q,a), & q \in Q_2 \end{cases}$$

2.6.3 REG è chiusa per $\star$

Abbiamo un linguaggio regolare L_1 e vogliamo provare che anche L_1^* è regolare. Prendiamo un NFA N_1 per L_1 e lo modifichiamo per riconoscere A_1^*. L'NFA N risultante accetterà il suo input quando esso può essere diviso in varie parti e N_1 accetta ogni parte.
Possiamo costruire N come N_1 con ε-archi supplementari che dagli stati accettanti ritornano allo stato iniziale. In questo modo ogni stringa del

linguaggio sarà composta da sotto-stringhe ben distinte accettate da N_1. Dobbiamo però modificare N per far sì che accetti ε, che è sempre elemento di L_1^*; semplicemente aggiungiamo un altro stato accettante q_0 che con ε-archi giunge agli stati iniziali di N_1.



Costruiamo $N = (Q, \Sigma, \delta, q_0, F)$ che riconosce L_1^*:

- $Q = Q_1 \cup q_0$
  Gli stati di N sono gli stati di N_1 più un nuovo stato iniziale.
- q_0 è il nuovo stato iniziale
- $F = \{q_0\} \cup F_1$
- Definiamo la δ t.c. $\forall q \in Q, a \in \Sigma_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a), & q \in Q_1, q \notin F_1 \\ \delta_1(q,a), & q \in F_1, a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\}, & q \in F_1, a = \varepsilon \\ \{q_1\}, & q = q_0, a = \varepsilon \\ \varnothing, & q = q_0, a \neq \varepsilon \end{cases}$$

2.7 Espressioni Regolari

Le espressioni regolari sono espressioni che descrivono un linguaggio mediante operazioni regolari come $\cup, \circ, \star$.

2.7.1 Esempio di un'espressione regolare

$$(0 \cup 1)0^*$$

Tale espressione regolare rappresenta il linguaggio che consiste di tutte le stringhe che iniziano con uno 0 o un 1 seguito da un qualsiasi numero di 0.

2.7.2 Proprietà e convenzioni

- $(0 \cup 1) \equiv \{0\} \cup \{1\} = 0, 1$
- $0^* \equiv \{0\}^*$
- $(0 \cup 1)0^* \equiv (0 \cup 1) \circ 0^*$
- $1^*\varnothing = \varnothing$
- $\varnothing^* = \varepsilon$

2.7.3 Def: espressioni regolari

Dato un alfabeto Σ, un espressione regolare su Σ o $re(\Sigma)$ è definibile per ricorsione:

- Caso base:
  $\varnothing \in re(\Sigma)$
  $\varepsilon \in re(\Sigma)$
  $a \in re(\Sigma) \quad a \in \Sigma$
- Passo induttivo:
  $R_1 \cup R_2 \quad R_1, R_2 \in re(\Sigma)$
  $R_1 \circ R_2 \quad R_1, R_2 \in re(\Sigma)$
  $R_1^* \quad R_1 \in re(\Sigma)$

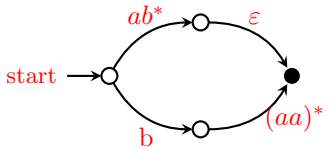
Oss: Sia r un'espressione regolare in Σ, ovvero $r \in re(\Sigma)$, possiamo definire ricorsivamente il linguaggio $L(r)$ associato a r.

- Caso base:
  $r = \varnothing \quad L(r) = \varnothing$
  $r = \varepsilon \quad L(r) = \varepsilon$
  $r = a \quad L(r) = a$
- Caso induttivo:
  $r = R_1 \cup R_2 \quad L(r) = L(R_1) \cup L(R_2)$
  $r = R_1 \circ R_2 \quad L(r) = L(R_1) \circ L(R_2)$
  $r = R^* \quad L(r) = L(R_1)^*$

Poiché, data un'espressione regolare, a essa è associato un linguaggio regolare, allora esiste un DFA che lo riconosce.
Inoltre, se un linguaggio è regolare, allora il linguaggio è generato da un'espressione regolare.

2.8 Automa Generalizzato (GNFA)

Differisce da un NFA perché gli archi, anziché essere etichettati solo dai simboli dell'alfabeto Σ, sono echitettati da espressioni regolari. Esempio GNFA[4]:



Una transizione da uno stato all'altro comporta la lettura di un'intera stringa. Il passaggio avviene quando l'input soddisfa l'espressione regolare.

[4]Non in forma canonica

2.8.1 Osservazioni sugli GNFA:

- È non deterministico, ovvero può avere più rami di computazione
- Accetta se almeno un ramo computazionale termina in uno stato accettato

2.8.2 Forma Canonica GNFA

Un GNFA è in forma canonica se soddisfa le seguenti proprietà:

1. È presente un solo stato iniziale e un solo stato finale ed essi sono distinti.
2. Lo stato iniziale ha solo archi uscenti verso tutti gli altri stati.
3. Lo stato finale ha solo archi entranti provenienti da tutti gli altri stati.
4. Ogni altro stato è collegato con tutti gli altri stati attraverso un singolo arco.

Conversione da DFA a GNFA in forma canonica Dato un qualsiasi NFA o DFA, per soddisfare le proprietà 1., 2. e 3., abbiamo bisogno di due nuovi stati; uno iniziale il quale, attraverso ε-archi, è collegato agli stati iniziali originali, e uno finale tale che i vecchi stati accettanti hanno ε-archi uscenti verso esso.
Per soddisfare la proprietà 4., dati due stati s, s', inseriamo un arco etichettato con $\varnothing$ da s a s' per ogni stato s originariamente non collegato da alcun arco a un altro stato s'. Se invece esistono più archi da s a s', li collassiamo tutti in un singolo arco, unendo le varie etichette con $\cup$.

2.8.3 Def: GNFA

Un GNFA è una 5-tupla $(Q, \Sigma, \delta, q_{start}, q_{acc})$ tale che:

1. Q insieme finito degli stati
2. Σ alfabeto
3. q_{start}, q_{acc} rappresentano lo stato iniziale e finale rispettivamente
4. $\delta : \big(Q \setminus \{q_{acc}\}]\big) \times \big(Q \setminus \{q_{start}\}\big) \to \mathcal{R}^5$

Un GNFA accetta $w \in \Sigma^*$ se e solo se:

$$\begin{aligned} w = &\{w_1, \dots, w_k : w_i \in \Sigma^*, \\ &\exists q_0, q_1, \dots, q_k \text{ t.c. } q_0 = q_{start}, q_k = q_{acc}, \\ &\forall i, w_i \in L(R_i) \text{ dove } R_i = \delta(q_{i-1}, q_i)\} \end{aligned}$$

[5] L'insieme delle espressioni regolari su Σ

2.9 Espressioni regolari (continua)

2.9.1 Equivalenza con gli automi finiti

Le espressioni regolari e gli automi finiti sono equivalenti rispetto allam loro potenza descrittiva. Ogni espressione regolare può essere trasformata in un automa finito che riconosce il linguaggio che essa descrive e viceversa.

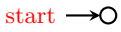
2.9.2 Teorema: L è regolare $\Leftrightarrow$ un'espressione regolare descrive L

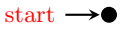
Un linguaggio L è regolare $\Leftrightarrow$ un'espressione regolare descrive L

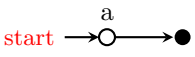
Dim (due direzioni): La dimostrazione prevede due direzioni: se L è descritto da un'espressione regolare, allora L è regolare e se L è regolare, allora è descritto da un'espressione regolare.

Dim: Se un linguaggio è descritto da un'espressione regolare, allora esso è regolare. Data r espressione regolare, costruiamo DFA M_r t.c. $L(M_r) = L(r)$ per induzione usando la definizione:

- Caso base:

  $r = \varnothing$ $\quad L(r) = \varnothing$ $\quad$ 

  $r = \varepsilon$ $\quad L(r) = \varepsilon$ $\quad$ 

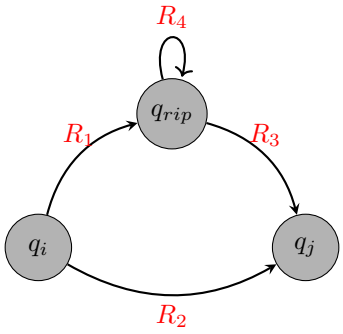
  $r = a \in \Sigma$ $\quad L(r) = a$ $\quad$ 

- Caso induttivo:
  $r = R_1 \cup R_2 \Rightarrow \exists M_1, M_2$ DFA che riconoscono $L(R_1) = L(M_1)$ e $L(R_2) = L(M_2)$
  invoco il **teorema 1.13.1 della chiusura per** $\cup$[6]

Dim: Se un linguaggio è regolare, allora è descritto da un'espressione regolare Se L è regolare, allora è generato da un'espressione regolare. Siccome L è regolare, $\exists$ DFA M t.c. $L(M) = L$. Vogliamo convertire M in un'espressione regolare equivalente al linguaggio L. Per proseguire con la dimostrazione ci serviremo dell'automa generalizzato (GNFA) in forma canonica, mostrato nel paragrafo **1.13.2.**.
Nota: Questa generalizzazione non è necessaria per la dimostrazione, ma la rende più omogenea. Dato M DFA lo converto in un GNFA in forma canonica e da esso, per ricavare l'espressione regolare a lui associata, a ogni passo, elimino uno stato mantenendo la sua forma canonica, aggiornando opportunatamente tutte le etichette di ogni arco di ogni altro stato. Quando non ci saranno più stati da eliminare, ovvero quando gli unici due stati sono quello iniziale e quello finale, l'espressione regolare associata all'unico arco che li unisce è proprio l'espressione regolare equivalente al linguaggio di M.

[6] analogo per $\circ$ e $\star$

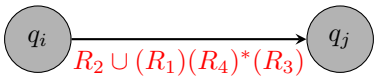
Ad esempio, dato il seguente GNFA in forma canonica:



Supponiamo di voler eliminare lo stato q_{rip}.
Prendiamo una coppia di stati (q_i, q_j) che non contenga quello che abbiamo deciso di eliminare: è sempre possibile, data la forma canonica, raggiungere q_j da q_i in due modi: non attraversando lo stato q_{rip} e attraversando lo stato q_{rip}.
Nell'esempio, è evidente che possiamo raggiungere q_j da q_i attraverso R_2 o attraverso la concatenazione di R_1, R_4^*, R_3.
La nuova etichetta quindi sarà:

$$R_2 \cup (R_1)(R_4)^*(R_3)$$

L'automa, *che ha mantenuto la sua forma canonica*, avrà quindi la forma:



Nota bene: Possiamo soggiornare nello stato R_4 per un tempo non definito, da ciò deriva l'operazione $\star$ associata a (R_4).

Algoritmo di conversione da DFA a espressione regolare:
Dato DFA M ottengo un GNFA in forma canonica G con k stati come mostrato nel paragrafo **1.13.2.1.**.

Mostriamo di seguito l'algoritmo `Convert`(G) che adotta una procedura ricorsiva che trasforma G con k stati in un'espressione regolare:

1: **if** $k = 2$ **then**
2: restituisco l'espressione regolare che collega q_{start} a q_{acc}
3: **end if**
4: **if** $k > 2$ **then**
5: scelgo $q_{rip} \in Q \setminus \{q_{start}, q_{acc}\}$
6: $G' = (Q', \Sigma, \delta', q_{start}, q_{acc})$ ▷ dove $Q' = Q \setminus \{q_{rip}\}$
7: **for all** $(q_i \in Q' \setminus q_{acc}, q_j \in Q' \setminus \{q_{start}\})$ **do**
8: $\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4)$[7]
9: **end for**
10: `Convert(G')`
11: **end if**

È banale osservare che l'algoritmo termina sempre: ogni iterazione ha uno stato in meno della precedente, e il caso base viene incontrato quando abbiamo un numero di stati pari a due.

Dim: Correttezza algoritmo `Convert`(G): Effettuiamo la dimostrazione per induzione sul numero di stati k:

- con $k = 2$ è banale; esistono solo due stati: quello iniziale e quello finale. `Convert`(G)=G, quindi il linguaggio accettato da G è lo stesso linguaggio generato dall'espressione regolare di `Convert`(G).
- Assumendo vero per $k - 1$ stati, basta mostrare che G e G' riconoscono lo stesso linguaggio: Ovvero se G accetta w, allora anche G' accetta w, e viceversa (doppia implicazione):
  - $\Rightarrow$ Osserviamo che c'è un ramo accettante quando G legge w che sarà pari a $(q_{start}, q_1, \ldots, q_{acc})$
    - Se q_{rip} non è presente nella sequenza di stati la dimostrazione è banale.
    - Se q_{rip} viene rimosso da `Convert`(G), l'algoritmo aggiornerà correttamente le espressioni regolari di tutte le etichette, quindi G' accetta w.
  - $\Leftarrow$ Se G' accetta w, gli archi in G' tengono conto di tutte le possibili transizioni di stato o dirette o che passano per q_{rip}, quindi G accetta w. Poiché G' ha $k - 1$ stati e la dimostrazione segue dall'ipotesi induttiva.

Questo termina la dimostrazione del paragrafo **1.14.2.1.2.**

■

[7] dove $R_1 = \delta(q_i, q_{rip}), R_2 = \delta(q_{rip}, q_{rip}), R_3 = \delta(q_{rip}, q_j), R_4 = \delta(q_i, q_j)$

2.10 Tutti i linguaggi sono regolari?

No, non tutti i linguaggi sono linguaggi regolari. Ricordiamo che ogni linguaggio finito ammette un DFA il cui numero di stati è finito. Cosa accade quando l'input, ad esempio, non è di dimensione finita?

2.10.1 Esempio di un linguaggio non regolare:

$$L = \{0^n 1^n : n \geq 0\} \quad \text{stringhe potenzialmente infinite di 1 e 0.}$$

Se cerchiamo di trovare un DFA che riconosca L, scopriamo che la macchina sembra aver bisogno di ricordare quanti simboli uguali a 0 sono stati analizzati quando legge l'input. Poiché il numero di simboli uguali a 0 non è limitato, la macchina dovrà tener traccia di un numero infinito di possibilità, e ciò è impossibile usando un numero di stati finiti.

2.10.2 Il pumping lemma

La tecnica per trovare la non regolarità deriva dal teorema del *pumping lemma*. Il teorema afferma che tutti i linguaggi regolari hanno una proprietà particolare; se non possiamo mostrare che tale proprietà è presente in un linguaggio, siamo sicuri che esso non è regolare.
La proprietà afferma che tutte le stringhe nel linguaggio possono essere *replicate* se la loro lunghezza raggiunge almeno uno specifico valore, chiamato **lunghezza del pumping**. Questo significa che ogni tale stringa contiene una parte che può essere ripetuta un numero qualsiasi di volte, ottenendo una stringa che appartiene ancora al linguaggio.

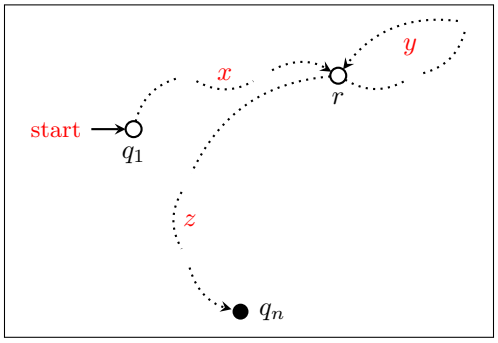
Teorema del pumping lemma: se L è un linguaggio regolare, allora esiste un numero p chiamato *lunghezza del pumping* tale che se s è una qualsiasi stringa in L di lunghezza almeno p, allora s può essere divisa in tre parti $s = xyz$ soddisfacenti le seguenti condizioni:

1. $\forall i \geq 0, xy^i z \in L$
2. $|y| > 0$
3. $|xy| \leq p$

Dim: Siano:

1. $M = (Q, \Sigma, \delta, q_1, F)$ il DFA che riconosce il linguaggio L
2. $p = \#$ stati
3. $w = w_1 w_2 \dots w_n \in L : n \geq p$
4. $r_1, r_2, \dots, r_{n+1} : n + 1 > p$ la sequenza di stati attraversati da M con input w, ovvero $\delta(r_i, w_i) = r_{i+1}$

Per il *pigeonhole principle*, tra i primi $p+1$ stati ce ne è almeno uno che si ripete, sia esso r. È possibile dividere w in tre componenti x, y, z. La componente x è la parte di w che compare prima di r. La componente y è la parte tra le due occorrenze di r, mentre la componente z è la parte restante di w.
Esempio: automa M



Osserviamo che la divisione di w soddisfa le tre condizioni. Supponiamo di eseguire M con l'input $xyyyz$. Sappiamo che x porta M da q_1 a r, e poi gli y portano a tre cicli su r, infine z porta l'automa in q_n. Essendo q_n uno stato accettante, M accetta l'input $xyyyz$.
Più in generale, M accetterà le stringhe del tipo $xy^iz : i \geq 0$. Questo prova la prima del pumping lemma.
La verifica della seconda condizione è banale: $|y| > 0$, poiché si tratta della parte che cicla su r.
Per ottenere la terza condizione, ci assicuriamo che r sia la prima ripetizione nella sequenza. Per il principio della piccionaia, i primi $p+1$ stati nella sequenza devono contenere almeno una ripetizione, quindi $|xy| \leq p$

■

Es 1: dimostrare che un linguaggio non è regolare Mostrare che $L = \{0^n1^n : n \geq 0\}$ non è regolare.
Per assurdo, sia L regolare, ovvero $\exists p$ t.c. $\forall w \in L$ è scomponibile come nell'enunciato. Costruisco w t.c. per ogni scomposizione legale $w = xyz$:

1. $|y| > 0$
2. $|xy| <= p$
3. $\exists i$ t.c. $xy^iz \in L$

Contraddizione: prendo $w = 0^p1^p$, dove p è pari alla *lunghezza del pumping*, supponendo esista, con $|w| = n, 2p > p$.
Siccome $|y| \leq p$, e y non è vuoto, y contiene almeno uno 0:

$$w = \underbrace{0\ldots0}_{x}\underbrace{0\ldots0}_{y}\underbrace{1\ldots1}_{z}$$

Questo conclude la dimostrazione poiché w ha un numero di simboli uguali a 0 maggiore rispetto a quelli uguali a 1.
Più in generale abbiamo tre casi specifici che riguardano y:

1. La stringa y consiste solo di simboli uguali a 0, come nell'esempio appena mostrato. In questo caso, la stringa $xyyz$ ha più simboli uguali a 0 che a 1, quindi non è un elemento in L, violando la condizione 1 del pumping lemma.

2. La stringa y consiste solo di simboli uguali a 1, analogo all'esempio precedente.

3. La stringa y consiste sia di simboli uguali a 0 che uguali a 1. In questo caso, la stringa $xyyz$ può avere lo stesso numero di simboli uguali per 0 e per 1, ma essi non saranno nell'ordine corretto: ci sarà qualche 1 prima di qualche 0, e quindi non è elemento di L.

Pertanto è inevitabile la contraddizione se assumiamo che L sia regolare, quindi L non è regolare.

Es 2: dimostrare che un linguaggio non è regolare Mostrare che $L = \{w \in \{0,1\}^* : |w|_0 = |w|_1\}$ non è regolare, dove $|w|_0$ rappresenta il numero di simboli 0, e $|w|_0$ il numero di 1.
Per assurdo, sia L regolare allora $\exists p$ t.c. vale il pumping lemma.
Sia s la stringa 0^p1^p. Poiché $s \in L$, e ha lunghezza maggiore di p, il pumping lemma assicura che s è divisibile in 3 parti: $s = xyz$, tali che per ogni $i \geq 0$ la stringa xy^iz è in L.
Sia $x = \varepsilon$, $z = \varepsilon$ e $y = 0^p1^p$, allora xy^iz ha sempre lo stesso numero di 0 uguali al numero di 1, e quindi è in L, e *sembra* che possa essere iterata.
Tuttavia la terza condizione del pumping lemma ci torna utile: essa stabilisce che s deve essere divisa in modo che $|xy| \leq p$, ma nel nostro caso $|x| = 0, |y| = 2p \rightarrow |xy| = 2p$, e quindi s non può essere iterata.
Applicando come dovere la terza condizione, ovvero $|xy| \leq p$, allora y può consistere solo di simboli uguali a 0, perciò $xyyz \notin L$. Quindi non possiamo iterare su s, e questo ci fornisce la contraddizione desiderata.

3 Linguaggi context-free

Abbiamo osservato che non tutti i linguaggi sono regolari, come $L = \{0^n1^n : n \geq 0\}$. Ora presentiamo le *grammatiche acontestuali*, o *context-free grammars*, un metodo più potente per descrivere linguaggi.

3.1 Grammatiche acontestuali (context-free grammar)

Una grammatica consiste di un insieme di *regole di sostituzione*, chiamate **produzioni**. Ogni regola è costituita da una *variabile non terminale* e da una stringa separati da una freccia. La stringa consiste di variabili non terminali e

altri simboli detti *variabili terminali* o semplicemente *terminali*. Una della variabili non terminali è chiamata *variabile iniziale* e le variabili terminali non compaiono mai a sinistra di alcuna regola. L'insieme di regole specificano, data una stringa s espressa nella grammatica, quali simboli potranno andare a sostituire le variabili non terminali di s. Una sequenza di sostituzioni per ottenere una stringa è chiamata una *derivazione*.
Una grammatica acontestuale in forma normale di Chomsky genera un linguaggio acontestuale.

3.1.1 Esempio di grammatica CF: G_1

Sia G_1 la seguente grammatica e sia A la sua variabile iniziale:

$$G_1 = \begin{cases} A \to 0A1 \\ A \to B \\ B \to \# \end{cases}$$

In G_1, $0, 1, \#$ sono variabili terminali, mentre A, B non terminali. Con la grammatica vogliamo generare stringhe, utilizzando la seguente procedura:

1. Viene scritta la variabile iniziale, poiché ogni grammatica ha associata sempre una variabile iniziale.
2. Viene sostituita la variabile con una qualsiasi regola associata a essa.
3. Si ripete il procedimento finché la stringa ottenuta non è composta esclusivamente da variabili terminali.

Esempio di stringa generata dalla grammatica Data la grammatica acontestuale G_1, è possibile generare la stringa 000#111 effettuando i seguenti passi:

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

3.1.2 Curiosità: grammatiche acontestuali e parser

Le grammatiche acontestuali sono utili, ad esempio, al parser di un linguaggio di programmazione per constatare se un pezzo di codice contenga un errore sintattico. Le espressioni di linguaggi di programmazione, difatti, possono essere espresse attraverso grammatiche acontestuali.

3.1.3 Def: CFG

Una CFG, o grammatica acontestuale, è una 4-tupla (V, Σ, R, S) dove:

- V è l'insieme finito di variabili;
- Σ è l'insieme finito di simboli terminali e $V \cap \Sigma = \varnothing$;
- R è l'insieme finito di regole;

– S rappresenta la variabile iniziale.

Siano u, v e $w \in \Sigma \cup V$ e $A \to w \in R$, diciamo che uAv *produce* uwv, e lo denotiamo con $uAv \Rightarrow uwv$.
Diciamo inoltre che u *deriva* v, e lo denotiamo con $u \overset{*}{\Rightarrow}$, se $u = v$ o se esiste una sequenza $u_1, u_2, \ldots, u_k$ con $k \geq 0$ tale che:

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

3.1.4 Esempio di grammatica CF: G_2, la grammatica delle parentesi correttamente annidate

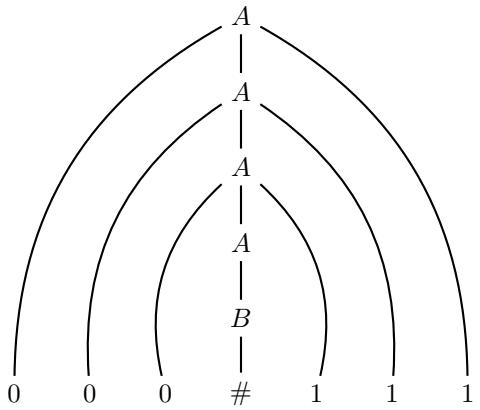
Consideriamo la grammatica $G_2 = (\{S\}, \{a, b\}, R, S)$ con R definito come segue:

$$S \to aSb|SS|\varepsilon$$

Questa grammatica genera stringhe come *abab*, *aaabbb*, etc. È possibile vedere questo linguaggio come la corretta chiusura di parentesi, ovvero vedere la a come parentesi aperta '(', e la b come la chiusa ')'. Visto in questo modo, $L(G_2)$ è il linguaggio di tutte le stringhe di parentesi correttamente annidate.

3.2 Albero sintattico

Possiamo quindi introdurre il concetto di albero sintattico. Esso rappresenta graficamente la stessa informazione della sequenza di sostituzioni che ci porta da una variabile non terminale iniziale a una stringa composta da variabili terminali. Possiamo osservare qui sotto l'albero sintattico dell'esempio 1.16.1.1:



3.2.1 Esempio di grammatica CF: G_3 espressioni aritmetiche

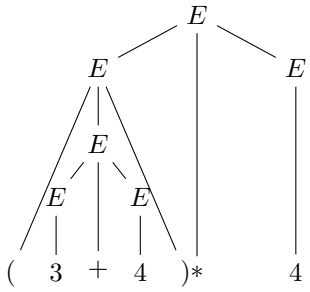
Definiamo la grammatica context-free G_3 delle espressioni aritmetiche che ha come variabile iniziale E

$$G_3 = \begin{cases} E \to E + E \\ E \to E * E \\ E \to (E) \\ E \to 0|1|2|\dots|9 \end{cases}$$

Facciamo qualche osservazione:

1. Le variabili terminali sono quindi tutti i simboli: parentesi, *, +, numeri. Una variabile terminale non compare mai a sinistra di nessuna regola R_i.
2. La grammatica può generare, ad esempio, la stringa $(3 + 4) * 4$, ma non può generare una stringa come $3+) * +$
3. Esistono più modi per generare una stringa composta da sole variabili terminali.

Albero sintattico di una stringa w generata da G_3: Sia $w = (3 + 4) * 4$ una stringa generata da G_3, possiamo modellare il suo albero sintattico:



3.2.2 Esempio di grammatica CF: G_4 espressioni aritmetiche alternative

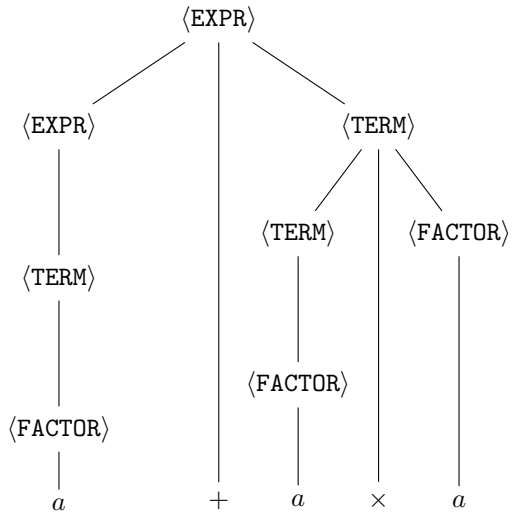
Si consideri la grammatica $G_4 = (V, \Sigma, R, \langle \texttt{EXPR} \rangle)$, definita come segue:

$V = \{\langle \texttt{EXPR} \rangle, \langle \texttt{TERM} \rangle, \langle \texttt{FACTOR} \rangle\}$

$\Sigma = \{a, +, \times, (,)\}$

$$R = \begin{cases} \langle \texttt{EXPR} \rangle \to \langle \texttt{EXPR} \rangle + \langle \texttt{TERM} \rangle | \langle \texttt{TERM} \rangle, \\ \langle \texttt{TERM} \rangle \to \langle \texttt{TERM} \rangle \times \langle \texttt{FACTOR} \rangle | \langle \texttt{FACTOR} \rangle, \\ \langle \texttt{FACTOR} \rangle \to (\langle \texttt{EXPR} \rangle) | a \end{cases}$$

Albero sintattico di una stringa w generata da G_4: Sia $w = a + a \times a$ una stringa generata da G_4, possiamo modellare il suo albero sintattico:



Ambiguità della grammatica G_4 C'è da fare un'osservazione importante: la grammatica G_4 non definisce regole di precedenza sulle operazioni, quindi la rappresentazione può avere ambiguità.

3.3 Problema: dato un linguaggio, progettare la sua grammatica

Presentiamo alcune linee guida che ci torneranno utili per la progettazione delle grammatiche:

3.3.1 Progettazione di grammatiche: unione di grammatiche

Supponiamo di avere $G_i = (V_i, \Sigma_i, R_i, S_i)$ con associato il linguaggio $L(G_i)$. Possiamo definire $G = (V, \Sigma, R, S)$ t.c. $L(G) = \bigcup_i L(G_i)$ tale che:

- $V = \bigcup_i V_i \cup S$
- $\Sigma = \bigcup_i \Sigma_i$
- $R = \bigcup_i R_i \cup S \to S_1 | S_2 | \dots | S_k$[8]

Dim: $L(G) = \bigcup_i L(G_i)$ Per effettuare la dimostrazione dobbiamo mostrare che $L(G) \subseteq \bigcup_i L(G_i)$ e che $L(G) \supseteq \bigcup_i L(G_i)$.

[8] Ovvero esiste una regola che mappa S, la nuova variabile iniziale, alla variabile iniziale S_i della grammatica $G_i \forall i$

$L(G) \subseteq \bigcup_i L(G_i)$ Sia $w \in \bigcup_i L(G_i)$, allora:

1. $\exists j$ t.c. $w \in L(G_j)$
2. $S_j \overset{*}{\Rightarrow} w \in G_j$

Ma in G esiste $S \to S_j$ per definizione, quindi:

$$G : S \to S_j \overset{*}{\Rightarrow} w$$

Questo ci mostra che $w \in L(G)$

$L(G) \supseteq \bigcup_i L(G_i)$ La dimostrazione è analoga alla precedente. ■

Esempio: unione di due grammatiche Sia $S_1 = (V_1, \Sigma_1, R_1, S_1)$ una CFG definita come segue:

- $V_1 = \{S_1\}$
- $\Sigma_1 = \{0, 1\}$
- $R_1 = \{S_1 \to 0S_1 1 | \varepsilon\}$

e sia $S_2 = (V_2, \Sigma_2, R_2, S_2)$ un'altra CFG tale che:

- $V_2 = \{S_2\}$
- $\Sigma_2 = \{0, 1\}$
- $R_2 = \{S_2 \to 1S_2 0 | \varepsilon\}$

Vogliamo progettare $L = S_1 \cup S_2$, ovvero:

$$L = \{S_1 = \{0^n 1^n : n \geq 0\} \cup S_2 = \{1^m 0^m : m \geq 0\}\}$$

Definiamo $L = \{V, \Sigma, R, S\}$:

- $V = V_1 \cup V_2 \cup S = \{S, S_1, S_2\}$
  dove S è la nuova variabile iniziale
- $\Sigma = \Sigma_1 \cup \Sigma_2 = \{0, 1\}$
- $R = R_1 \cup R_2 \cup \{S \to S_1 | S_2\}$
  Ovvero $R = \begin{cases} S \to S_1 | S_2 \\ S_1 \to 0S_1 1 | \varepsilon \\ S_2 \to 1S_2 0 | \varepsilon \end{cases}$

3.3.2 Progettazione di grammatiche: passaggio da DFA a CFG

Vogliamo, dato un DFA M, definire una grammatica acontestuale $G_M = (V, \Sigma, R)$:

(i) Introduco V_i variabile per ogni stato q_i di M

(ii) $V_i \to aV_j \in R$ se in M vale $\delta(q_i, a) = q_j$

(iii) Se q_i è stato di accettazione, allora $V_i \to \varepsilon \in R$

(iv) q_0 è la variabile iniziale

Oss: i DFA sono casi speciali di grammatiche A ogni DFA è associabile una grammatica che lo rappresenta.

3.3.3 Progettazione di grammatiche: linguaggi non regolari

Una CFG può definire stringhe di lunghezza infinita, cosa che gli NFA non possono fare. Le grammatiche acontestuali possono quindi definire un linguaggio non regolare.
Dato $L = \{0^n 1^n : n \geq 0\}$, linguaggio non regolare, possiamo definire la seguente grammatica $G_{NR} = (V, \Sigma, R)$ che lo rappresenta:

- $V = \{R\}$
- $\Sigma = \{0, 1\}$
- $R = \{0R1\}$

3.3.4 Progettazione di grammatiche: curiosità

Esistono regole che modellano grammatiche più complesse, come cicli while, for, etc. Tali grammatiche sfruttano la ricorsione per definire tale complessità.

3.4 L'ambiguità delle grammatiche acontestuali

Abbiamo visto che per alcuni linguaggi acontestuali è possibile generare la stessa espressione in due modi diversi; questo può risultare in ambiguità. Un parser di un linguaggio di programmazione, però, non è mai ambiguo. Questa ambiguità può essere evitata con delle convenzioni: data un'espressione, dobbiamo sempre sostituire prima la variabile più a sinsitra.

3.4.1 Derivazione a sinistra

Una derivazione di una stringa in una CFG è una derivazione a sinistra se a ogni passo la variabile sostituita è quella più a sinistra.

Esempio di derivazione a sinistra: data la stringa $aAbCc$, dove A, C sono variabili, la prossima regola che uso è quella in cui a sinistra è presente una A.

3.5 CFG: La forma normale di Chomsky

Una CFG è in forma normale di Chomsky, o forma canonica, se ha solo regole del tipo:

- $A \to BC$ dove B, C sono due variabili diverse dalla variabile iniziale;
- $A \to a$ dove a è un solo simbolo terminale;
- $S \to \varepsilon$ dove S è la variabile iniziale.

3.6 Teorema: ogni linguaggio acontestuale è generato da una CFG canonica

Ogni linguaggio context-free è generato da una grammatica context-free in forma normale di Chomsky, o forma canonica.

3.6.1 Dim

Mostriamo un esempio che andrà di pari passo con la dimostrazione. Sia G la seguente CFG: trasformeremo G in forma normale di Chomsky; in grassetto saranno visualizzati i cambiamenti.

$$G = \begin{cases} S \to ASA|aB \\ A \to B|S \\ B \to b|\varepsilon \end{cases}$$

Trasformazione in forma normale di Chomsky: passo 1 La CFG iniziale, mostrata sulla sinistra, necessita di una variabile iniziale S_0 non presente alla destra di alcuna regola: aggiungiamola.

(G)		(1)
		$\boldsymbol{S_0 \to S}$
$S \to ASA\|aB$	$\Rightarrow$	$S \to ASA\|aB$
$A \to B\|S$	$\Rightarrow$	$A \to B\|S$
$B \to b\|\varepsilon$	$\Rightarrow$	$B \to b\|\varepsilon$

Trasformazione in forma normale di Chomsky: passo 2a Eliminiamo le ε-regole: eliminiamo una ε-regola $V \to \varepsilon$, dove V non è la varibiale iniziale poi, per ogni occorrenza di V sul lato destro di una regola, aggiungiamo una nuova regola con quell'occorrenza cancellata. In altre parole, se $R \to uVvVw$ è una regola in cui u, v, w sono stringhe di variabili e terminali, aggiungiamo la regola $R \to uvw$. Dobbiamo fare in modo che tutte le occorrenze siano presenti, quindi aggiungiamo le regole $R \to uVvw$ e $R \to uvVw$. In

particolare, se abbiamo una regola della forma $R \to V$, aggiungiamo la sola regola $R \to \varepsilon$.
In questo esempio iniziamo eliminando la regola $B \to \varepsilon$:

(1)		(2a)
$S_0 \to S$	$\Rightarrow$	$S_0 \to S$
$S \to ASA \vert aB$	$\Rightarrow$	$S \to ASA \vert aB \vert \boldsymbol{a}$
$A \to B \vert S$	$\Rightarrow$	$A \to B \vert S \vert \boldsymbol{\varepsilon}$
$B \to b \vert \varepsilon$	$\Rightarrow$	$B \to \boldsymbol{b}$

Trasformazione in forma normale di Chomsky: passo 2b Ora è presente una nuova ε-regola: $A \to \varepsilon$; eliminiamola ricordandoci di mantenere tutte le occorrenze possibili di A sulla destra.

(2a)		(2b)
$S_0 \to S$	$\Rightarrow$	$S_0 \to S$
$S \to ASA \vert aB \vert a$	$\Rightarrow$	$S \to ASA \vert aB \vert a \vert \boldsymbol{SA} \vert \boldsymbol{AS} \vert \boldsymbol{S}$
$A \to B \vert S \vert \varepsilon$	$\Rightarrow$	$A \to \boldsymbol{B} \vert \boldsymbol{S}$
$B \to b$	$\Rightarrow$	$B \to b$

Trasformazione in forma normale di Chomsky: passo 3a Ora eliminiamo le regole unitarie: eliminiamo una regola unitaria generica $A \to B$ poi, per ogni regola $B \to u$, aggiungiamo la regola $A \to u$, a meno che essa non sia una regola unitaria precedentemente cancellata.
In questo passo eliminiamo le regole unitarie $S \to S$, che non porta a cambiamenti, e $S_0 \to S$:

(2b)		(3a)
$S_0 \to S$	$\Rightarrow$	$S_0 \to S \vert \boldsymbol{ASA} \vert \boldsymbol{aB} \vert \boldsymbol{a} \vert \boldsymbol{SA} \vert \boldsymbol{AS}$
$S \to ASA \vert aB \vert a \vert SA \vert AS \vert S$	$\Rightarrow$	$S \to \boldsymbol{ASA} \vert \boldsymbol{aB} \vert \boldsymbol{a} \vert \boldsymbol{SA} \vert \boldsymbol{AS}$
$A \to B \vert S$	$\Rightarrow$	$A \to B \vert S$
$B \to b$	$\Rightarrow$	$B \to b$

Trasformazione in forma normale di Chomsky: passo 3b Eliminiamo la regola unitaria $A \to B$:

$(3a)$		$(3b)$
$S_0 \to S\|ASA\|aB\|a\|SA\|AS$	$\Rightarrow$	$S_0 \to S\|ASA\|aB\|a\|SA\|AS$
$S \to ASA\|aB\|a\|SA\|AS$	$\Rightarrow$	$S \to ASA\|aB\|a\|SA\|AS$
$A \to B\|S$	$\Rightarrow$	$A \to \boldsymbol{S\|b}$
$B \to b$	$\Rightarrow$	$B \to b$

Trasformazione in forma normale di Chomsky: passo 3c Eliminiamo la regola unitaria $A \to S$:

$(3b)$		$(3c)$
$S_0 \to S\|ASA\|aB\|a\|SA\|AS$	$\Rightarrow$	$S_0 \to S\|ASA\|aB\|a\|SA\|AS$
$S \to ASA\|aB\|a\|SA\|AS$	$\Rightarrow$	$S \to ASA\|aB\|a\|SA\|AS$
$A \to S\|b$	$\Rightarrow$	$A \to S\|b\|\boldsymbol{ASA\|aB\|a\|SA\|AS}$
$B \to b$	$\Rightarrow$	$B \to b$

Trasformazione in forma normale di Chomsky: passo 4
Trasformiamo le regole rimanenti, cioè quelle che presentano più di una variabile non terminale sulla destra. Per far ciò, creiamo delle variabili di transizione. Ad esempio, sia una regola da rimpiazzare del tipo $A \to u_1 u_2 \dots u_k$ con $k \geq 3$ e u_i variabile o terminale, la rimpiazziamo con un insieme di regole del tipo $A \to u_1 A_1$, $A_1 \to u_2 A_2 \dots A_{k-2} \to u_{k-1} u_k$[9]

(4)

$S_0 \to AA_1|UB|a|SA|AS$

$S \to AA_1|UB|a|SA|AS$

$A \to b|AA_1|UB|a|SA|AS$

$A_1 \to SA$

$U \to a$

$B \to b$

Ora la grammatica è in forma normale di Chomsky: ogni variabile a destra delle regole è composta o da esattamente due variabili non terminali, oppure da un solo simbolo terminale e l'unica ε-regola, se presente, è della variabile iniziale. In questo esempio, tuttavia, essa non è presente.

[9] l'ultima regola $A_{k-2} \to u_{k-1} u_k$ ha tali indici poiché vengono collassate due regole insieme: avendo $A_{k-1} \to u_{k-1} Ak$ e $A_k \to u_k$ possiamo eliminare Ak

3.7 Automi a Pila (pushdown automata)

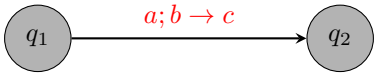
Introduciamo un nuovo tipo di modello computazionale, chiamato *automa a pila*. Questi automi sono come gli automi finiti non deterministici NFA ma hanno una specie di *memoria* chiamata *pila* o *stack*. Tale pila consente a questa tipologia di automi di riconoscere alcuni linguaggi non regolari.
Gli automi sono computazionalmente equivalenti alle grammatiche context-free: data una grammatica acontestuale, possiamo trovare un automa a pila che la riconosce, e dato un automa a pila, esiste una grammatica acontestuale associata a quell'automa..

3.7.1 Operazioni automa a pila

Due operazioni:

- pop: rimozione dalla pila
- push: inserimento in cima alla pila

3.7.2 Notazione grafica di un PDA



L'arco permette all'automa, quando si trova nello stato q_1, di passare allo stato q_2 leggendo a e facendo pop di b in cima alla pila, effettuando poi il push di c.
Oss: sia b che c possono essere ε, quindi è possibile fare o solo push o solo pop, oppure niente:

- $b \neq \varepsilon, c \neq \varepsilon \to$ pop di b e push di c;
- $b = \varepsilon, c \neq \varepsilon \to$ push di c;
- $b \neq \varepsilon, c = \varepsilon \to$ pop di b.

3.7.3 Alfabeto della pila

Per implementare un PDA abbiamo bisogno di descrivere sintatticamente l'insieme di simboli che possono andare nella pila. Dobbiamo quindi definire un alfabeto Γ che verrà chiamato *alfabeto della pila*.

3.7.4 Definizione PDA

Un PDA è una 6-tupla $(Q, \Sigma, \Gamma, \delta, q_o, F)$ dove Q, Σ, q_0, F sono come in NFA, inoltre:

- Γ è l'alfabeto finito della pila
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \to \mathcal{P}(Q \times \Gamma_\varepsilon)$

Un automa a pila $M = (Q, \Sigma, \Gamma, \delta, q_o, F)$ accetta un input w se $w = w_1 w_2 \dots w_m$ dove $w_i \in \Sigma_\varepsilon$ per ogni i ed esistono sequenze di stati $r_0, r_1, \dots, r_n \in Q$ e di stringhe $s_0, s_1, \dots, s_m \in \Gamma^*$ che soddisfano le tre condizioni seguenti:

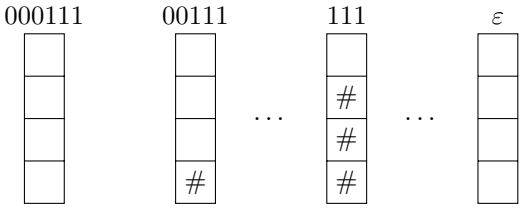
1. $r_0 = q_0$ e $s_0 = \varepsilon$, ovvero M inizia correttamente nello stato iniziale con la pila vuota;
2. $\forall i = 0, \dots, m-1$ abbiamo che $(r_{i+1}, a) \in \delta(r_i, w_{i+1}, b)$ dove $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$. Questa condizione afferma che M si muove correttamente in funzione allo stato, al simbolo di pila e al prossimo simbolo in input.
3. $r_m \in F$. Questa condizione afferma che alla fine dell'input M si trova in uno stato accettante.

Relazione di transizione Dal punto 2. appena mostrato, ne ricaviamo la seguente *relazione di transizione*: dati $p, q \in Q$, $b, c \in \Gamma_\varepsilon$, $y \in \Gamma_\varepsilon^*$, $a \in \Sigma_\varepsilon$, $x \in \Sigma_\varepsilon^*$:

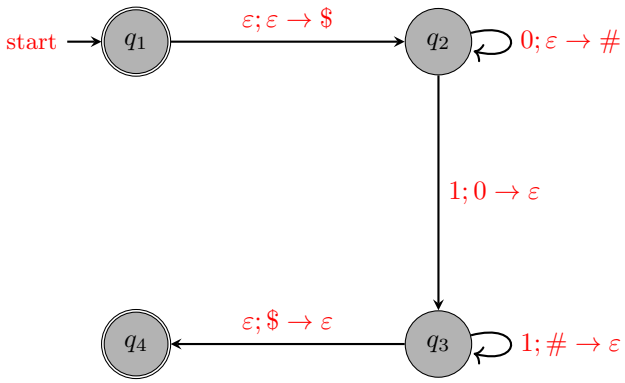
$$(p, ax, by) \vdash_M \Big((q, x, cy) \to (q, c) \in \delta(p, a, b) \Big)$$

3.7.5 Esempi di PDA

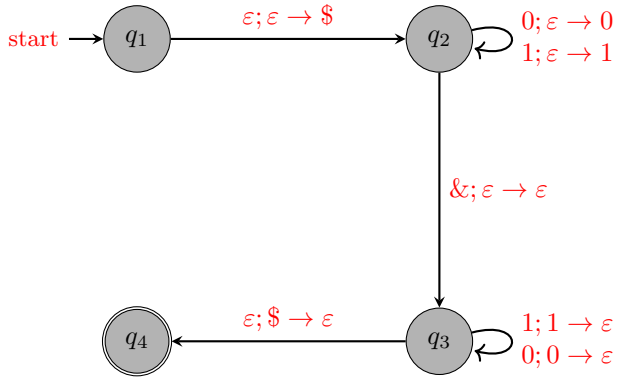
Esempio 1. Realizzare un PDA per il linguaggio non regolare $L = \{0^n 1^n : n \geq 0\}$.
Idea: fino a che leggo 0 faccio push di un generico simbolo # di Γ nella pila. Quando leggo 1 faccio l'operazione pop dello stesso simbolo. Quando termina l'input, se la pila è vuota, accetto.



Di seguito il PDA di L: utilizziamo il carattere \$ come primo carattere per contrassegnare l'inizio della pila.



Esempio 2. Realizzare un PDA per il linguaggio non regolare $L = \{w\&w^T : w \in 0, 1^*\}$.
Idea: leggo w fino al punto appena precedente al carattere & e la salvo in uno stack. Nella pila ora saranno estratti caratteri in ordine inverso a w, ovvero proprio w^T.



3.7.6 Teorema: un linguaggio è acontestuale ⇔ esiste un PDA che lo riconosce

Un linguaggio L è acontestuale ⇔ esiste PDA che lo riconosce.

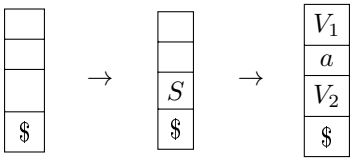
Dim: due direzioni

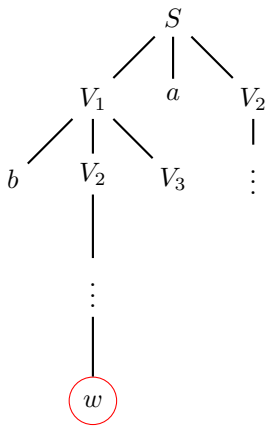
Dim: L acontestuale ⇒ esiste PDA che lo riconosce Dato L, sappiamo che c'è una grammatica acontestuale G che genera L. Possiamo trasformare G in un PDA equivalente.
Idea: dato $w \in L$, controllo con il non determinismo che esiste una serie di produzioni in G che conduce a w. In particolare, dato un PDA P che riconosce G, P inizia la sua computazione scrivendo la variabile iniziale nella pila. Esso

poi passa attraverso una serie di stringhe intermedie effettuando opportune sostituzioni, infine può giungere a una stringa che contiene solo simboli terminali; ha usato la grammatica per derivarne una stringa. P accetta se la stringa è identica alla stringa che ha ricevuto in input. Sia quindi G la seguente grammatica:

$$G = \begin{cases} S \to V_1 a V_2 \\ V_1 \to b V_2 V_3 \\ V_1 \to c V_2 V_4 \\ \vdots \end{cases}$$

Usiamo lo stack per scriverci tutte le regole della grammatica; se considero tutti i pattern possibili, ovvero tutte le stringhe che posso generare, controllo se almeno una di queste è w.

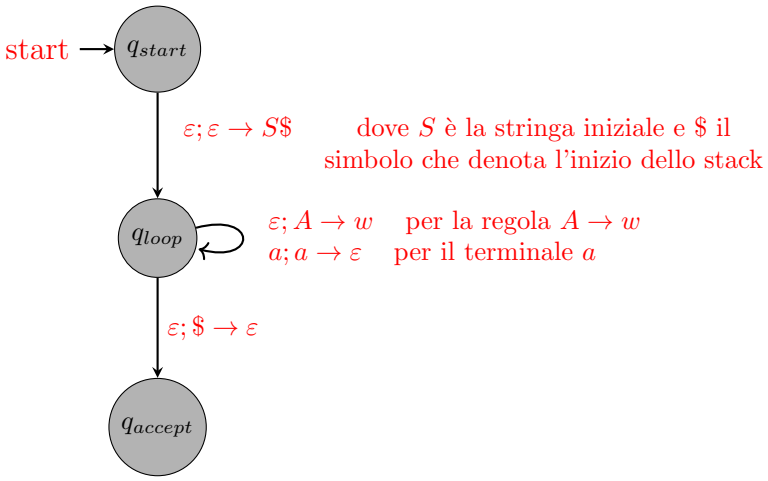




Se una foglia dell'albero computazionale di P è proprio pari a w, accetto.

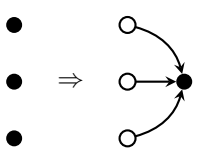
Pseudocodice:

1: Inserisce \$ e la variabile iniziale nella pila;
2: **while** Esistono variabili non terminali **do**
3: **if** Sulla cima della pila c'è una variabile A **then**
4: Uso il non-determinismo per sostituire con pop/push A usando le regole di G;
5: **else if** Sulla cima della pila c'è il terminale a **then**
6: Faccio pop controllando che a sia il carattere seguente di input;
7: **else if** Leggo \$ **then**
8: Accetto;
9: **end if**
10: **end while**

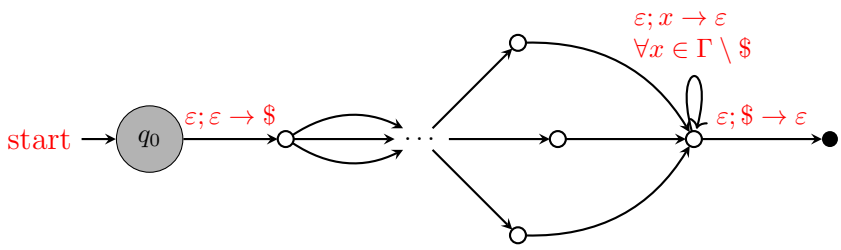


Dim: Se L è riconosciuto da un PDA $\Rightarrow$ L è acontestuale. Idea: dato un generico PDA P, definisco la sua grammatica CFG G che genera tutte le stringhe che P accetta. Per raggiungere il risultato, progettiamo una grammatica che fa un po' di più: per ciascuna coppia di stati p e q in P, la grammatica avrà una variabile A_{pq} in grado di generare tutte le stringhe che possono portare P da p con pila vuota a q con pila vuota. Tali stringhe sono quindi in grado di condurre da p a q indipendentemente dal contenuto della pila, lasciando la pila in q nella stessa condizione in cui era in p.
Definiamo P con le seguenti caratteristiche:

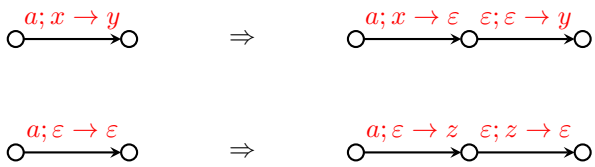
1. Ha un unico stato accettante q_{accept}



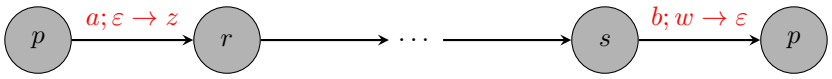
2. Viene aggiunto il simbolo \$ e e svuota semprela pila prima di accettare



3. Ciascuna transizione effettua un push o un pop non contemporaneamente.

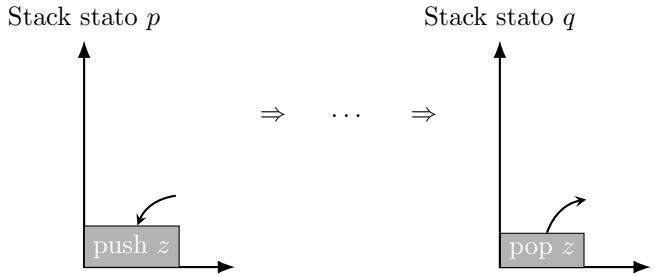


Oss: per consentire a P di soddisfare la terza caratteristica, sostituiamo ciascuna transizione che contemporaneamente elimina e inserisce simboli con una sequenza di due transizioni che attraversa un nuovo stato. Inoltre sostituiamo ogni transizione che non effettua né push né pop con una sequenza di due transizioni che prima inserisce un simbolo generico e poi lo elimina. In questo modo garantiamo l'invariante che il PDA inizi con la pila vuota e termina la propria computazione con la pila vuota.
Per ogni coppia (p, q) di stati definisco anche le variabili non terminali A_{pq} e A_{qp}. $S = A_{q_0 q_{acc}}$.
Ovvero per ogni coppia (p, q) di stati, aggiungo le regole che portano da p a q con la pila vuota. La variabile non terminale A_{pq} genererà tutte queste stringhe.

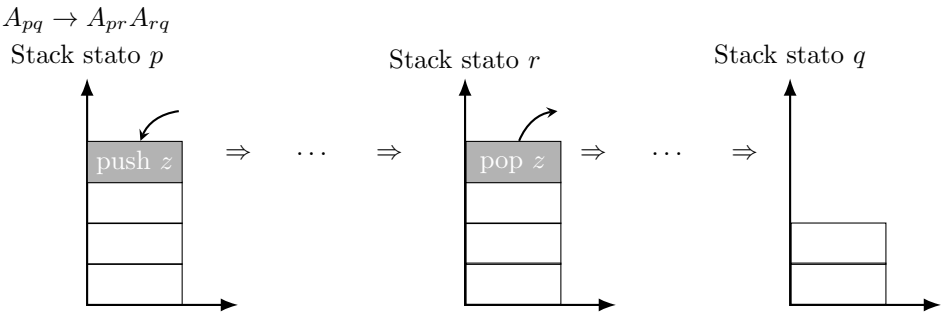


Abbiamo quindi due casi:

Caso 1: $w = z$ ovvero trovandomi allo stato p effetto push di z e tale z viene rimossa quando entro nello stato q.



Caso 2: $w \neq z$ ovvero trovandomi nello stato p effettuo la push di z e tale z viene rimossa quando entro in un certo stato r. La computazione di r continuerà, portando eventualmente allo stato q.



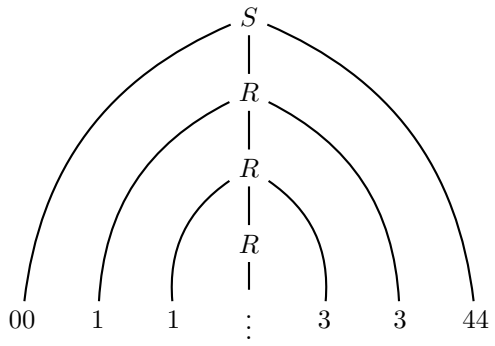
3.8 Tutti i linguaggi sono context-free?

La risposta alla domanda è no; difatti esiste una versione generalizzata del pumping lemma applicabile a linguaggi acontestuali. Questo vale a dire che esistono linguaggi non riconoscibili nemmeno da PDA.
L'idea è simile a quella del pumping lemma per linguaggi regolari; siccome un DFA, seppur con un numero di stati finito, può accettare stringhe di lunghezza infinita, abbiamo già osservato per il pigeonhole principle che c'è sempre almeno uno stato che si ripete.
Anche con i PDA vale lo stesso: una grammatica con regole finite può generare stringhe infinite. Questo significa che, a un certo punto, esiste almeno una regola che si ripete. Questo rende la stringa ricevuta in input *scomponibile* in un certo modo. Il pumping lemma ci dirà che il linguaggio non è acontestuale se tale proprietà non è soddisfatta.

Es: una grammatica con il seguente insieme di regole finite:

- $S \to 00R44$;
- $R \to 1R3|2$;

può generare stringhe infinite.



Nell'esempio appena mostrato, la regola R può esser ripetuta tante volte. Il pumping lemma ci dice che se un linguaggio è acontestuale, allora esiste una lunghezza p detta *lunghezza del pumping*, per tutte le stringhe lunghe almeno p essa è scomponibile in $uvxyz$.

3.8.1 Il pumping lemma per i linguaggi context-free

Se A è un CFL, allora:

$$\exists p : \forall S \in A \text{ t.c. } |S| \geq p \text{ allora } s = uvxyz$$

per cui valgono:

(i) $\forall i \geq 0, uv^i xy^i z \in A$

(ii) $|vy| > 0$

(iii) $|vxy| \leq p$

La dimostrazione di questa versione del pumping lemma non viene effettuata ed è consultabile sul libro.

Mostrare che $L = \{a^n b^n c^n : n \geq 0\}$ non è CF Proseguiamo con la dimostrazione per assurdo: supponiamo che L sia context-free, ovvero esiste un certo p che soddisfa il pumping lemma, o PL; mostreremo che esiste una stringa s di lunghezza almeno p che viola almeno una delle condizioni del pumping lemma.
Sia $s = a^p b^p c^p$, $|s| = 3p \geq p$.

Considero i modi di scomporre $s = uvxyz$; abbiamo due casi:

Caso 1: v e y contengono al più un solo simbolo tra a, b, c.

$$aaaabbbbcccc \rightarrow \begin{cases} \text{Es1: } a\underbrace{aaa}_{v}bbbbbcc\underbrace{c}_{y}cc \\ \text{Es2: } a\underbrace{a}_{v}bbbbbcccc, y = \varepsilon \end{cases}$$

Prendendo come riferimento il primo esempio, con $i = 2$ abbiamo che uv^2xy^2z dove $v^2 = aaaa, y^2 = cc$, ottenendo quindi la stringa $aaaaaabbbbccccc$, che non è nel linguaggio perché ogni simbolo ha un numero diverso di occorrenze.
Contenendo un solo simbolo, quando si effettua l'iterazione rimarrà sempre almeno un simbolo che rimane scoperto e quindi la stringa non è nel linguaggio.

Caso 2: almeno uno tra v e y contiene almeno due simboli diversi.
$aaabbbccc \rightarrow aa\underbrace{aab}_{v}bb\underbrace{b}_{y}cccc$

Prendendo $i = 2$ otteniamo la stringa $uv^2xy^2z = aaabaabbbbccc$ che non è nel linguaggio, poiché alcuni simboli si alternano.

Mostrare che $L = \{ww : w \in \{0,1\}^*\}$ non è CF Supponiamo sia CF, quindi deve esistere p del PL. Sia $s = 0^p1^p0^p1^p$, $|s| = 4p \geq p$. Visualizziamo una stringa d'esempio nel seguente modo: vediamo il centro della stringa, ovvero la parte che separa una w dall'altra, come il *centro* della stringa. In ogni w, inoltre, vediamo come la separazione tra due simboli $0, 1$ come un *confine*.

Es: $00000111110000011111 \rightarrow 00000 \underset{\downarrow conf}{|} 11111 \underset{\downarrow centro}{|} 00000 \underset{\downarrow conf}{|} 11111$ Ora supponiamo $s = uvxyz$: possiamo avere tre casi distinti:

Caso 1: vxy non oltrepassa i confini.
Es: $00000\underbrace{11111}_{vxy}0000011111$
All'iterare di questa stringa, solo il numero di 1 crescerà, e quindi evidentemente non è nel linguaggio.

Caso 2: vxy oltrepassa il confine.
Es: $00\underbrace{000111}_{vxy}110000011111$
Siccome $|vxy| \leq p$ non supero il centro. Ponendo $i = 0$, ovvero effettuando *pumping down*, il centro si sposta a destra, quindi la stringa non è della forma ww. $001100 \underset{\substack{\downarrow \\ nuovo \\ centro}}{|} 00011111$

Caso 3: vxy sorpassa il centro.

Es: $0000011\underbrace{11100}_{vxy}00011111$

Se effettuo il pumping down, ovvero pongo $i = 0$ otteniamo la stringa: $00000\underbrace{11}_{<p}\underbrace{000}_{<p}11111$ che non è una stringa del linguaggio poiché la sottostringa tra i due confini non contiene lo stesso numero di 0 e 1.

3.9 CFG e chiusure

3.9.1 Unione

Vero banalmente per $\cup$.

3.9.2 Intersezione

Sappiamo che $L = \{a^n b^n c^n : n \geq 0\}$ non è acontestuale, ma possiamo *spezzare* L in due sotto-linguaggi:

- $L_1 = \{a^n b^n c^i : n \geq 0, i \geq 0\}$
- $L_2 = \{a^i b^n c^n : n \geq 0, i \geq 0\}$

Di seguito le regole di L_1:

- $S \to TU$
- $T \to aTb|\varepsilon$
- $U \to cU|\varepsilon$

Analogamente per L_2:

- $S \to UT$
- $T \to bTc|\varepsilon$
- $U \to aU|\varepsilon$

L_1 e L_2 sono entrambi linguaggi acontestuali, ma la loro intersezione, ovvero L, non lo è.

3.9.3 Complemento

Per quanto riguarda il complemento, un modo di dimostrarlo è prendendo di riferimento l'intersezione; possiamo vedere l'intersezione come un'unione di complementi complementata.

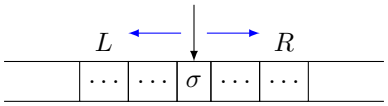
$$\overline{\overline{A} \cap \overline{B}} = \overline{\overline{A} \cup \overline{B}}$$

Supponendo che il complemento sia context-free, allora se A fosse context-free, $\overline{A}$ sarebbe context free. Possiamo dire lo stesso per $\overline{B}$. Quindi anche il complemento della loro unione è context-free, ma esso è analogo all'intersezione applicando DeMorgan, che abbiamo già dimostrato che non è CF.

4 La macchina di Turing

Introduciamo un nuovo modello computazionale: la *macchina di Turing*.
Prima di definirla formalmente, elenchiamo di seguito le principali differenze rispetto ai modelli già visti:

- La TM può sia leggere che scrivere su un *nastro*;
- La *testina del nastro* può spostarsi o a destra o a sinistra;
- Il *nastro* è infinito.



La testina, denotata dalla freccia verticale, può spostarsi a sinistra e a destra. Una volta letto un input sul nastro può decidere se sovrascriverlo con un altro simbolo, per poi spostarsi sul nastro. Possiamo evidenziare tre tipologie di stati sulla macchina di Turing: iniziale, di accettazione e di rifiuto. Sono definiti inoltre due alfabeti: Σ, ovvero l'alfabeto di input, e Γ, ovvero l'alfabeto di nastro. È bene osservare che esiste un carattere speciale nell'alfabeto Γ: $\sqcup$, denominato "*blank*" che non è presente nell'alfabeto di input.
La macchina di Turing di per sé è deterministica, ma è possibile definire macchine di Turing non deterministiche.

4.1 Def: Macchina di Turing

Una Macchina di Turing (o MdT o TM) è $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{acc}, q_{rej})$ tale che:

- Q è l'insieme di stati finito;
- Σ è l'alfabeto finito di input;
- Γ è l'alfabeto finito di nastro;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times L, R$, dove L e R sono le direzioni (sinistra e destra rispettivamente) che effettua la testina;
- $q_0 \in Q$ stato iniziale;
- $q_{acc} \in Q$ stato accettante;
- $q_{rej} \in Q$ stato di rifiuto;
- $q_{rej} \neq q_{acc}$.

4.1.1 Tesi di Church-Turing

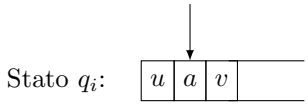
La **tesi di Church-Turing** afferma che la macchina di Turing può effettuare qualsiasi computazione effettuabile su un normale computer.
Oss: ci sono linguaggi che una macchina di Turing, e quindi un computer, non può riconoscere.

4.1.2 Convenzioni informali delle TM

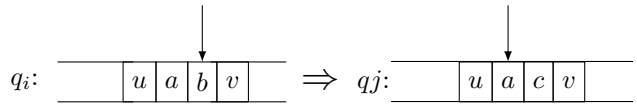
Di seguito elenchiamo qualche convenzione che utilizzeremo all'interno del corso.

- all'inizio in q_0 l'input $w = w_1 w_2 \dots w_n \in \Sigma^*$ è sul nastro. Il resto del nastro è $\sqcup$;
- la computazione segue la δ e, per convenzione, la testina non si muove a sinistra se si trova all'estremità, ovvero all'inizio, del nastro;
- la computazione termina se viene raggiunto q_{acc} o q_{rej}, ma è possibile definire una δ che, dato un input w, vada in loop, ovvero non raggiunga mai alcun stato q_{acc} o q_{rej};
- chiamiamo *snapshot* la configurazione (nastro, stato, posizione della testina). Siano C_1 e C_2 due configurazioni di una macchina di Turing M, si dice che C_1 produce C_2 se M può passare da C_1 a C_2 in un unico passo.
  È possibile definire graficamente una specifica configurazione. Ad esempio, sia uq_iav una generica configurazione di M dove:
  - $u, v \in \Gamma^*$;
  - $a \in \Gamma$ è il simbolo in lettura;
  - $q_i \in Q$ lo stato corrente.

  La configurazione può essere rappresentata nel seguente modo:



Di seguito un esempio di un passo computazionale, dove supponiamo di avere $a, b \in \Gamma$ e $u, v \in \Gamma^*$ e gli stati $q_i, q_j \in Q$:



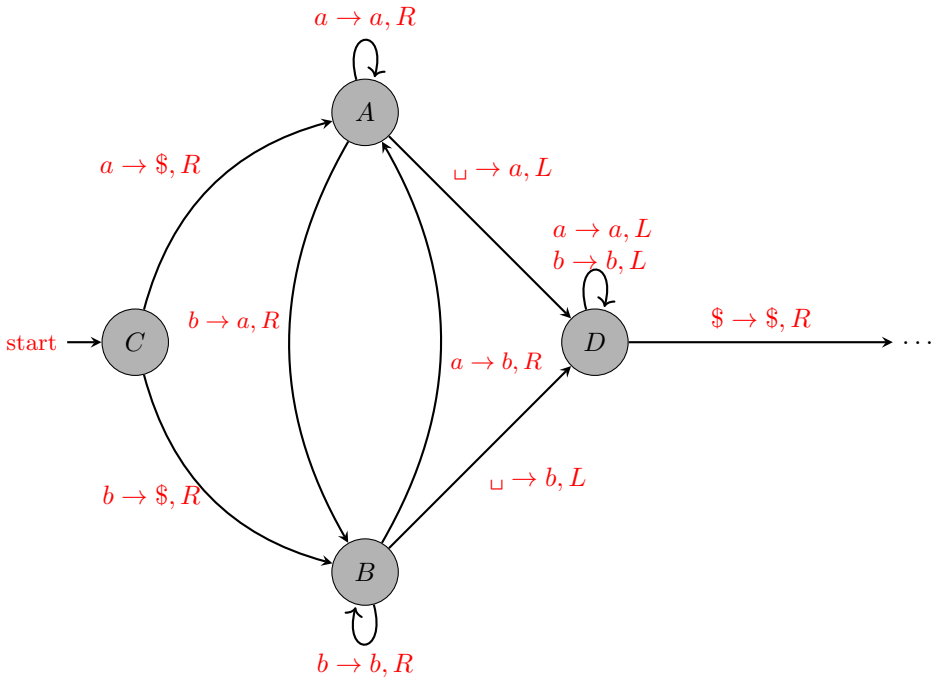
La transizione da q_i a q_j è definita da $\delta(q_i, b) = (q_j, c, L)$; informalmente: la testina era in b, scrive c sovrascrivendo b e si muove a sinistra. Quindi q_i produce q_j.

In generale, una TM M accetta $w \in \Sigma^*$ se esistono $C_1, \ldots, C_k$ t.c.:

- C_1 è la configurazione iniziale di M su input w;
- $C_i \Rightarrow C_{i+1}$, ovvero ogni C_i produce C_{i+1};
- C_k è di accettazione.

Una macchina di Turing invece rifiuta quando legge un input per cui la δ non è definita.

Riconoscere la fine del nastro Una convenzione per riconoscere la fine del nastro è quella di inserire un simbolo speciale come \$. Ovvero, se un nastro contiene la stringa $w_1|w_2|\ldots$, lo stesso nastro con la convenzione sarà del tipo $\$|w_1|w_2|\ldots$
È bene osservare che possiamo trasformare qualsiasi TM in una TM che inserisce il simbolo \$ all'estremo del nastro, ad esempio la seguente TM applica la convenzione a un nastro generico:



4.2 Def: Linguaggio Turing-riconoscibile

Un linguaggio è Turing-riconoscibile se esiste una TM che lo riconosce.

4.3 Def: TM decisore

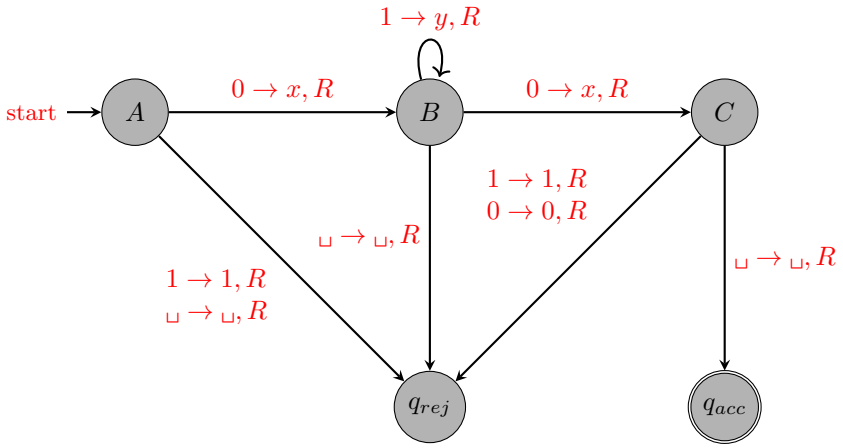
Una TM è un decisore se, per ogni linguaggio L riconoscibile dalla TM, essa non va mai in loop per ogni input che può ricevere. Ovvero ogni computazione finisce sempre o nello stato di accettazione oppure di rifiuto $\Rightarrow$ una TM *decide* L se è un decisore e se riconosce L.

4.4 Def: Linguaggio Turing-decidibile

Un linguaggio è Turing-decidibile se esiste una TM che lo decide.

4.5 Esempi di TM

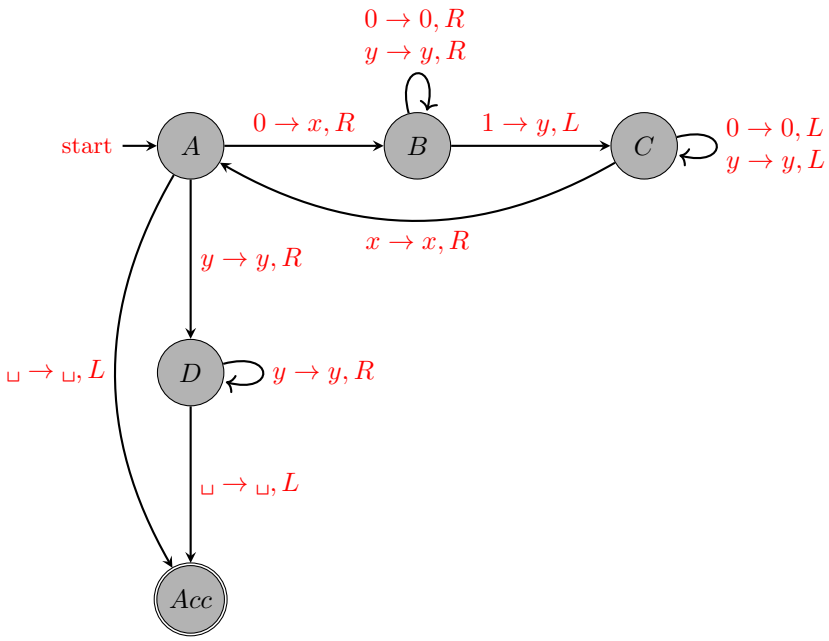
1. Una TM che riconosce il linguaggio regolare $L = \{01^*0\}$:



   Ad esempio, da A a B, leggendo 0, scrivo x e vado a destra. La sovrascrittura di x, nell'esempio, non è davvero necessaria ma solo dimostrativa. Rimaniamo quindi in B finché leggiamo simboli uguali a 1 e li rimpiazziamo con y, andando a destra.

$$01110 \Rightarrow x1110 \Rightarrow xy110 \Rightarrow \cdots \Rightarrow xyyyx$$

2. Una TM che riconosce il linguaggio acontestuale $L = \{0^n1^n : n \geq 0\}$:



Leggendo uno 0 lo sovrascrivo con x, poi vado a destra nel nastro finché trovo un 1 e lo sovrascrivo con y. Torno a sinistra e ripeto il procedimento per ogni prossimo 0. Se a fine input il nastro contiene solo simboli x e y, o se il nastro è *blank*, accetto.

4.6 Tipologie di TM

Di seguito mostriamo diverse tipologie di Turing Machine:

4.6.1 TM come subroutine

Siccome una Turing-machine simula un algoritmo, possiamo utilizzare una TM come subroutine di un'altra TM.

4.6.2 TM stay put

Turing machine che simula il comportamento di rimanere ferma a termine di una computazione.

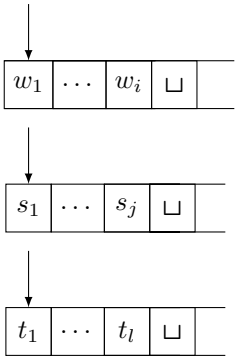
$$S' : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$$

In particolare, $\delta(q, a) = (q_1, b, S)$ si può simulare con due movimenti; uno a destra e uno a sinistra. Per rendere possibile ciò, abbiamo bisogno di un nuovo stato di transizione r e un simbolo speciale $*$ nell'alfabeto di nastro atto solo alla simulazione:

$$\delta(q, a) = (r, b, R); \delta(r, *) = (q_1, *, L)$$

4.6.3 TM multinastro

Supponiamo di volere una TM con $k \geq 1$ nastri; ad esempio sia $k = 3$:



$$\delta : Q \times \Gamma^k \to Q \times \Gamma^* \times \{L, R, S\}^k$$

Tale TM legge tutti i nastri insieme. Ad esempio un passo computazionale è della forma:

$$\delta(q_i, a_1, a_2, \ldots, a_k) = (q_i, b_1, \ldots, b_k, L_1, R_2, \ldots, S_k)$$

Teorema: Una TM multinastro è equivalente a una TM singolo nastro Per ogni TM multi-nastro ne esiste una con singolo nastro equivalente.

Dim: Idea: Memorizzare tutte le informazioni necessarie ed eseguire M con un singolo nastro simulando il comportamento multi-nastro. Per distinguere i k nastri, li separiamo con il simbolo speciale #. Oltre a conoscere dove termina un dato nastro, M deve tenere traccia delle posizioni delle testine: possiamo *marcare* il contenuto del nastro per contrassegnare la posizione in cui si trova la testina di un dato nastro:

#	$\dot{w}_1$	$\cdots$	w_i	#	$\dot{s}_1$	$\cdots$	s_j	#	$\dot{t}_1$	$\cdots$	t_l	

Vengono marcati quindi con un puntino, o un asterisco, i caratteri puntati dalla testina di ogni nastro. A ogni passo di computazione viene applicata la δ mostrata sopra; viene quindi *smarcato* il carattere dove si trova la testina, si sposta la testina in accordo alla δ e viene marcato il nuovo carattere puntato. Dobbiamo mantenere l'invariante che il proprio nastro, per ogni testina, è infinito; è bene osservare che nell'attuale implementazione è possibile che una testina *scavalchi* il carattere #, invadendo quello di altre testine. Per risolvere tale problematica, andando a destra, quando ogni testina legge il simbolo #, viene shiftato tutto il contenuto del nastro da destra in poi, ricavando quindi

una nuova posizione per la testina. Se, invece, una testina prova a spostarsi a sinistra invadendo eventualmente il carattere #, come per le normali TM quando incontrano la loro estremità del nastro, la testina rimane ferma. Quindi, in generale, la TM M:

1. Mette il nastro nel formato $\dot{w}_1 \dots w_n \# \dot{s}_1 \dots s_m \# \dots$
2. Per simulare una singola mossa, la M scansiona il suo nastro dal primo simbolo #, che segna l'estremità sinistra del primo nastro, fino al $(k+1)$-esimo #, che segna l'estremità destra dell'ultimo nastro per determinare i simboli puntati dalle testine virtuali. Successivamente la M effettua un secondo passaggio per aggiornare i nastri in accordo alla funzione di transizione δ.
3. Se in un qualsiasi momento M sposta una delle testine virtuali a destra su un simbolo #, significa che la TM corrispondente ha spostato la testina in una parte di nastro vuota, non letta in precedenza, quindi la M scrive un simbolo $\sqcup$ shiftando a destra di una cella il contenuto del nastro.

Corollario L è turing-riconoscibile $\Leftrightarrow \exists$ TM multinastro che lo riconosce.

4.6.4 TM non deterministica (o NTM)

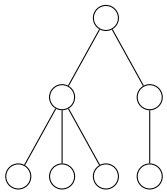
Una TM non deterministica (o NTM) è tale che:

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$[10]

Ovvero è tale che, dato uno stato e un input, la TM non deterministica transita in uno o più stati:

$$\delta(q, a) = (q', b, L); \delta(q, a) = (q'', c, R)$$

La computazione è quindi descritta da un albero:



Ogni nodo dell'albero rappresenta una configurazione. In particolare la radice è la configurazione iniziale e si procede lungo l'albero tra le varie altre configurazioni.

[10] insieme delle parti

In generale, una configurazione in una macchina di Turing è una stringa $uqav$ tale che:

1. u è l'input già analizzato;
2. q è lo stato corrente;
3. a è il prossimo carattere in input;
4. v è la stringa ancora da leggere dopo a.

Accettazione in una TM non deterministica Una macchina di Turing non deterministica accetta se e solo se un ramo computazionale accetta.

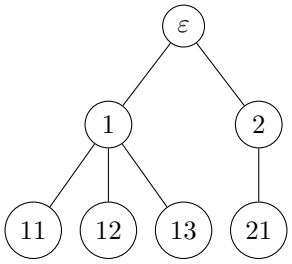
Per ogni NTM N esiste una TM D deterministica equivalente Data una Turing machine non deterministica N possiamo sempre modellare una Turing machine deterministica D equivalente.

Dim Idea: D simula la computazione di N provando tutte le scelte che può fare N durante la sua computazione non deterministica. Se D trova lo stato di accettazione su uno qualsiasi dei rami, allora D accetta. Questo è possibile perché ogni ramo di computazione dell'albero rappresenta una singola scelta non deterministica.
È bene notare che la visita dell'albero da parte di D deve essere svolta accuratamente perché D potrebbe non poter visitare tutto l'albero; una visita in profondità, difatti, è una scelta errata poiché D potrebbe incappare in un loop. La visita va quindi effettuata in ampiezza: questa strategia esplora tutti i cammini che terminano alla stessa profondità prima di esplorare ogni cammino che termina alla profondità successiva. Questo metodo di visita garantisce che D visiti ogni nodo dell'albero finché non incontra una configurazione di accettazione. Modelliamo la TM D con tre nastri:

- N_1 contiene l'input;
- N_2 è il nastro di lavoro o simulazione;
- N_3 contiene l'indirizzo, ovvero tiene traccia della posizione di D nell'albero delle computazioni di N.

Consideriamo la rappresentazione dei dati su N_3: ogni nodo dell'albero può avere al massimo b figli, dove b è la dimensione del più grande insieme di scelte possibili date dalla funzione di transizione di N. A ogni nodo della struttura assegniamo un indirizzo che è una stringa sull'alfabeto $\Gamma_b = \{1, 2, \ldots, b\}$. Ad esempio, assegniamo l'indirizzo 231 al nodo a cui si arriva partendo dalla radice, spostandosi al suo secondo figlio, spostandosi ancora da tale nodo al suo terzo figlio ed infine spostandosi al primo figlio di quest'ultimo nodo.

Es: b=3



Oss: se $b \geq 10$ è possibile aggiungere un nuovo carattere che separa le stringhe della configurazione disambiguando. Ad esempio se $b = 12$ la configurazione 113 potrebbe rappresentare la configurazione $11, 3$ o $1, 1, 3$.
Possiamo definire un semplice algoritmo per una TM deterministica D che effettua la simulazione di una generica TM non deterministica N:
`Simula TM non deterministica(N):`

1: N_1 contiene w_1 e N_2, N_3 sono vuoti;
2: Copia N_1 su N_2 e inizializzo N_3 a ε;
3: **while** Non si trova in uno stato di accettazione **do**
4: i:= indirizzo su N_3;
5: Esegue N su nastro N_2 usando il cammino dalla radice al nodo i;
6: **if** N ha accettato **then**
7: Accetta;
8: **else**
9: Passa al cammino successivo per ampiezza;
10: **end if**
11: Calcola il prossimo indirizzo di visita e scrivilo su N_3;
12: **end while**

Oss: l'algoritmo non tiene conto dell'efficienza: ogni cammino ricomincia da capo; vengono percorsi più volte gli stessi sotto-cammini

Corollario 1 L è Turing-decidibile $\Leftrightarrow$ esiste una NTM che lo decide.

Corollario 2 L è Turing-riconoscibile $\Leftrightarrow$ esiste una NTM che lo riconosce.

4.6.5 Enumeratore

I linguaggi Turing-riconoscibili sono anche detti *ricorsivamente enumerabili*, da ciò deriva il nome di questa variante.
Possiamo vedere un enumeratore come una TM collegata a una stampante: stampa solamente stringhe in output, generando così un linguaggio, ovvero un insieme di stringhe enumerate. All'inizio il nastro è vuoto, inoltre l'ordine delle stringhe generate è arbitrario e alcune di esse possono essere ripetute.

Teorema: Un linguaggio è Turing-riconoscibile $\Leftrightarrow$ esiste un enumeratore che lo enumera Un linguaggio è Turing-riconoscibile se e solo se esiste un enumeratore che lo enumera.

Dim ($\Leftarrow$) supponiamo esista E che enumera il linguaggio L.
La TM su input w simula E; ogni volta che E produce un output controlla se questo è uguale a w. Se sì, accetta.

Dim ($\Rightarrow$) supponiamo esista M che riconosce L: dobbiamo definire l'enumeratore corrispondente, ovvero che enumeri L.
Sia $s_1, s_2, \ldots$ una lista di tutte le possibili stringhe dell'alfabeto di L: lanciamo M su tutte le s_i.
`E(M):`

1: **for** $i = 1, 2, \ldots$ **do** ▷ Loop infinito
2: **for** $j = 1$ to i **do**
3: Simula M usando s_j come input per i passi
4: **if** M accetta s_j entro i passi **then**
5: stampa s_j
6: **end if**
7: **end for**
8: **end for**

Se M accetta una particolare stringa s, alla fine s apparirà nella lista stampata da E. In realtà, essa apparirà un numero infinito di volte perché M ritorna all'inizio su ogni stringa per la ripetizione del ciclo for al passo 1.

Part II
Calcolabilità

In questa parte saranno illustrati i poteri e i limiti degli algoritmi.

5 Notazione: codifica di un oggetto O

Dato un oggetto O, come un grafo, TM, DFA, NFA, etc, indichiamo con $\langle O \rangle$ la sua codifica binaria.

6 Decidibilità

Il matematico Hilbert cercava un *processo in base al quale un problema è risolvibile in un numero finito di passi*: quello che oggi chiamiamo algoritmo. In particolare, il *problema di Hilbert* era trovare le radici di un generico polinomio.

$$D = \{\langle p \rangle : p \text{ polinomio con radici intere e } \langle p \rangle \text{ è la codifica binaria di } p\}$$

Oss: D è un linguaggio Turing-riconoscibile, ma non decidibile.
Questo ci fa osservare che esistono linguaggi decidibili, e che alcuni di essi sono collegati a linguaggi regolari e alle grammatiche acontestuali, e linguaggi non decidibili, come D. Esistono anche linguaggi che non sono Turing-riconoscibili.

6.1 Esempi di linguaggi decidibili

Mostriamo alcuni esempi di linguaggi decidibili.

6.1.1 Teorema: A_{DFA} è decidibile

$$A_{DFA} = \{\langle B, w \rangle : B \text{ DFA accetta } w \subseteq \{0,1\}^*\}$$

A_{DFA} è decidibile.

Dim Definiamo una TM M che decide A_{DFA}:
$M =$ su input $\langle B, w \rangle$ dove B è un DFA e w una stringa:

1: Simula B su input w;
2: Se B accetta *accetta*, altrimenti *rifiuta*.

La TM M appena mostrata è un decisore: riconosce il linguaggio, accetta l'input nei casi accettanti e lo rifiuta nei casi rifiutanti; A_{DFA} è un linguaggio decidibile.

6.1.2 Teorema: A_{NFA} è decidibile

$$A_{NFA} = \{\langle B, w\rangle : B \text{ NFA accetta } w \subseteq \{0,1\}^*\}$$

A_{NFA} è decidibile.

Dim Dato B definisco B' DFA equivalente, poi eseguiamo M della dimostrazione del teorema precedente[11] con input $\langle B', w\rangle$

6.1.3 Teorema: A_{REX} è decidibile

$$A_{REX} = \{\langle R, w\rangle : R \text{ espressione regole che genera } w \subseteq 0, 1^*\}$$

A_{REX} è decidibile.

Dim Dato R definisco B'' NFA equivalente, poi segue dalla dimostrazione del teorema precedente[12].

6.1.4 Teorema: E_{DFA} è decidibile

Vediamo un altro esempio di linguaggio decidibile, il *test del vuoto*: supponiamo di avere un DFA A e vogliamo sapere se A accetta almeno una stringa.

$$E_{DFA} = \{\langle A\rangle : A \text{ è DFA e } L(A) \neq \varnothing\}$$

Dim Idea: si utilizza un algoritmo simile a quello già visto per vedere se un grafo ha un cammino tra due nodi[13]. Il DFA accetta se e solo se dallo stato inziale, percorrendo le transizioni di stati, esiste un cammino che lo porta a uno stato accettante.
Tutto quello che dobbiamo fare, dato un DFA con il suo diagramma di stato, immaginabile come un grafo, osservare se c'è un cammino dallo stato iniziale a uno dei nodi che nel grafo rappresenta uno stato di accettazione.
`Decisore per` E_{DFA}`:`
$T =$ Su input $\langle A\rangle$, dove A è un DFA:

1: Marca lo stato iniziale di A;
2: **while** Esistono nodi da marcare **do**
3: Marca qualsiasi stato che ha una transizione proveniente da uno stato già marcato;
4: **end while**
5: Se nessuno stato di accettazione risulta marcato *accetta*, altrimenti *rifiuta*.

[11] Sezione 6.1.1.
[12] Sezione 6.1.2.
[13] Visto nella dimostrazione di equivalenza tra TM e NTM. Sezione 4.6.4.2.1.

6.1.5 Teorema: EQ_{DFA} è decidibile

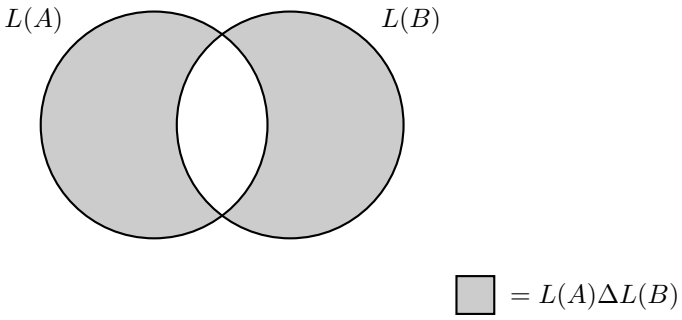
Vogliamo capire se due DFA riconoscono esattamente lo stesso linguaggio.

$$EQ_{DFA} = \{\langle A, B\rangle : A, B \text{ sono DFA e } L(A) = L(B)\}$$

EQ_{DFA} è decidibile.

Dim Possiamo dimostrarlo usando la differenza simmetrica:

$$L(A)\Delta L(B) = (L(A) \cap \overline{L(B)}) \cup (L(B) \cap \overline{L(A)})$$



L'osservazione è che, siccome la differenza simmetrica tra A e B viene definita solo attraverso complemento, unione e intersezione, e siccome i linguaggi regolari sono chiusi rispetto a tutte e tre le operazioni, è definibile un DFA C tale che il linguaggio che riconosce C è proprio la differenza simmetrica dei due linguaggi, ovvero $L(C) = L(A)\Delta L(B)$
Oss: $L(C) = \varnothing \Leftrightarrow L(A) = L(B)$
Definiamo quindi la TM M' progettata con il seguente algoritmo:
$F =$ "Su input $\langle A, B\rangle$, dove A e B sono DFA:

1: Costruisce il DFA C che accetta $L(A)\Delta L(B)$;
2: Esegue la TM T della dimostrazione di E_{DFA}[14];
3: Se M accetta *accetta*, altrimenti *rifiuta*.

6.1.6 Teorema: EQ_{REX} è decidibile

EQ_{REX}, ovvero il linguaggio che comprende le coppie di due espressioni regolari che generano lo stesso linguaggio, è decidibile.

$$EQ_{REX} = \{\langle R_1, R_2\rangle : R_1, R_2 \text{ sono espressioni regolari e } L(R_1) = L(R_2)\}$$

Questo è dimostrabile da ciò che abbiamo già visto perché possiamo sempre convertire R_1, R_2 in GNFA equivalenti e tali GNFA possono essere convertiti in DFA. Da qui segue dal teorema precedente.

[14] Mostrata nella sezione 6.1.4.

6.1.7 Teorema: A_{CFG} è decidibile

Poniamoci ora il problema di capire se una data grammatSica G genera una particolare parola fissata w.

$$A_{CFG} = \{\langle G, w\rangle : G \text{ è una grammatica che genera la stringa } w\}$$

A_{CFG} è decidibile.

Dim Prima idea: utilizzare G per passare tutte le sue possibili derivazioni e vedere se ne esiste una che genera w. Il problema è che potremmo dover provare infinite derivazioni, e quindi la TM non sarebbe un decisore ma solo un riconoscitore.
Idea corretta: Ci torna utile la proprietà che abbiamo definito per la forma normale di Chomsky: se G è una CFG in forma normale e $w \in L(G)$ con $|w| = n$, ogni derivazione di w in G richiede al più $2n - 1$ passi. Questo ci definisce un *upper-bound* per generare una qualsiasi stringa di un dato linguaggio. Proviamo quindi tutte le derivazioni lunghe $2n - 1$ passi e controlliamo se w è nelle parole generate. Questo ci porta la macchina di Turing con il seguente algoritmo:
$S =$ su input $\langle G, w\rangle$, dove G è una CFG e w una stringa:

1: Converte G in una grammatica equivalente in forma normale di Chomsky;
2: Lista tutte le derivazioni di $2n - 1$ passi dove $n = |w|$, oppure di 1 passo se $n = 0$;
3: Se una di tali derivazioni genera w *accetta*, altrimenti *rifiuta*.

6.1.8 Teorema: E_{CFG} è decidibile

Ora consideriamo il test del vuoto per le grammatiche acontestuali.

$$E_{CFG} = \{\langle G\rangle : G \text{ è una grammatica CFG e } L(G) = \varnothing\}$$

E_{CFG} è decidibile.

Dim Prima idea: usiamo la TM vista in precedenza per tutte le possibili stringhe; verificare che esse non siano accettate. Tuttavia un linguaggio finito può generare un numero infinito di stringhe, quindi la TM potrebbe non terminare mai.
Idea corretta: una grammatica genera almeno una stringa se esiste una sequenza di sostituzioni che, partendo da una variabile iniziale, genera una stringa con tutti simboli terminali. Se ciò succede, la grammatica genera almeno una stringa.
Potremmo usare un algoritmo simile a quello per i grafi: determina per ogni variabile se essa è in grado di generare una stringa con soli terminali, e tiene traccia di questa informazione marcando tale variabile. Dapprima, l'algoritmo marca tutti i simboli terminali della grammatica, poi scandisce tutte le regole della grammatica. Se trova una regola che consente a qualche variabile di

essere sostituita con una stringa di soli terminali, che sono già tutti marcati, l'algoritmo marca tale variabile. Proseguiamo finché non ci sono più variabili da marcare e osserviamo se la variabile iniziale è stata marcata. Questo ci permette di definire la TM R.
R = su input $\langle G \rangle$, dove G è una CFG:

1: Marca tutti i simboli terminali di G;
2: **while** Esistono variabili da marcare: **do**
3: Marca tutte le variabili A tale che in G c'è una regola $A \to U_1, \ldots, U_k$ e U_i è marcato $\forall i$;
4: **end while**
5: Se la variabile iniziale è marcata *accetta*, altrimenti *rifiuta*.

6.1.9 Teorema: ogni linguaggio context-free è decidibile

Sia A un CFL. Il nostro obiettivo è mostrare che A è decidibile. Possiamo utilizzare la TM S mostrata nel teorema di decidibilità di A_{CFG} come subroutine:
M_G = su input w:

1: Esegue la TM S su input $\langle G, w \rangle$;
2: Se S accetta, *accetta*, altrimenti *rifiuta*.

6.1.10 Chiusura di linguaggi decidibili

Claim: I linguaggi decidibili sono chiusi rispetto alle operazioni di complemento, di unione e di intersezione.

7 Indecidibilità

Vogliamo comprendere e dimostrare che alcuni problemi concreti sono irrisolvibili; la tecnica per capirlo è la *diagonalizzazione*.
Il problema di fondo è estendere i linguaggi studiati per DFA e CFG al caso di Turing-machine. Consideriamo il seguente linguaggio:

$$A_{TM} = \{\langle M, w \rangle : M \text{ è una TM e } M \text{ accetta } w\}$$

Abbiamo dimostrato che se M è un DFA o una CFG, il linguaggio è decidibile, ma se M è una TM il linguaggio è indecidibile. Prima di poter dimostrare il teorema abbiamo necessità di introdurre il concetto di diagonalizzazione e di definire qualche altro concetto.

7.1 Diagonalizzazione

Prima di introdurre il metodo della diagonalizzazione, cerchiamo di capire perché fu introdotto.

7.1.1 Diagonalizzazione: introduzione

Vogliamo dimostrare che due insiemi hanno la stessa cardinalità. Se gli insiemi sono finiti, possiamo semplicemente contare gli elementi e dare una risposta. Ma cosa succede se gli insiemi sono infiniti? Come possiamo sapere se $\mathbb{N}$ ha la stessa cardinalità dei numeri pari? Esistono *diversi tipi* di infinito? Per risolvere problemi del genere utilizziamo la *diagonalizzazione*, introdotta da Cantor. Prima di proseguire, ricordiamo il concetto di biiezione e di numerabilità.

Biiezione Siano A e B due insiemi e $f : A \to B$ una funzione da A in B:

1. f è **iniettiva** se $a \neq b \to f(a) \neq f(b)$;
2. f è **sueriettiva** se $\forall b \in B \exists a \in A | f(a) = b$
3. f è **biiettiva** se è suriettiva che iniettiva.

A e B hanno la stessa cardinalità se esiste una funzione biiettiva da A a B che è totale su A, ovvero il cui dominio è tutto A. Da ciò ne concludiamo che:

1. ogni elemento $a \in A$ viene mappato in uno e uno solo elemento $b \in B$;
2. per ogni elemento $b \in B$ esiste un unico elemento $a \in A$ mappato in b.

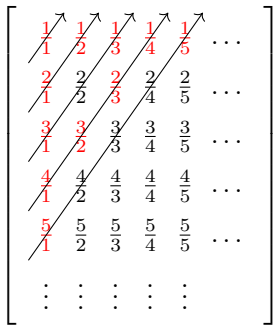
Esempio di funzione biiettiva Siano $\mathbb{N}$ l'insieme dei numeri naturali e $\mathbb{E}$ l'insieme dei numeri naturali pari, dimostriamo che $|\mathbb{N}| = |\mathbb{E}|$: $n \in \mathbb{N} \to 2n \in \mathbb{E}$

n	$f(n)$
0	0
1	2
2	4
3	6
$\vdots$	$\vdots$

Numerabilità Un insieme è *numerabile* se è finito oppure ha la stessa cardinalità di $\mathbb{N}$.

Esempio: $\mathbb{Q}$ è numerabile $\mathbb{Q}$, l'insieme dei numeri razionali positivi, è numerabile:

$$\mathbb{Q} = \{\frac{m}{n} : m, n \in \mathbb{N}\}$$



Mettiamo quindi in relazione gli elementi di $\mathbb{Q}$ con gli elementi di $\mathbb{N}$ lungo le diagonali, iniziando dall'angolo in ato a sinistra, evitando di accoppiare gli elementi già inseriti, per esempio $\frac{2}{2}$ sarebbe ripetuto perché abbiamo già accoppiato $\frac{1}{1}$.
In questo modo riusciamo a definire una biiezione tra i due insiemi infiniti.

Esempio: $\mathbb{R}$ non è numerabile $\mathbb{R}$, l'insieme dei numeri razionali positivi, non è numerabile. Limitiamoci solo all'intervallo $[0,1]$ di $\mathbb{R}$ e dimostriamo che non è numerabile. La dimostrazione è per contraddizione: supponiamo che per assurdo esista una biiezione f tra $\mathbb{N}$ e $[0,1]$ e mostreremo che esiste un elemento $d \in [0,1]$ non accoppiato da f con alcun elemento di $\mathbb{N}$, ma ciò è impossibile, perché abbiamo supposto che esista una biiezione f, quindi f non può esistere. Sia d l'elemento diagonale, proviamo a costruirlo come segue:
La i-esima cifra decimale di d è diversa dalla i-esima cifra decimale di $f(i)$.

$$
\begin{aligned}
f(1) &= 0.\mathbf{5}105110\ldots \\
f(2) &= 0.4\mathbf{1}32043\ldots \\
f(3) &= 0.82\mathbf{4}5026\ldots \\
f(4) &= 0.233\mathbf{0}126\ldots \\
f(5) &= 0.4107\mathbf{2}46\ldots \\
f(6) &= 0.993783\mathbf{8}\ldots \\
f(7) &= 0.0105135\ldots \\
&\vdots
\end{aligned}
$$

Quindi, osservando l'esempio:

1^a cifra di d può essere $4 \neq 5 \rightarrow 1^a$ cifra di $f(1)$;

2^a cifra di d può essere $3 \neq 1 \rightarrow 2^a$ cifra di $f(2)$;

3^a cifra di d può essere $7 \neq 4 \rightarrow 3^a$ cifra di $f(3)$;

...

Quindi $d = 0.437\cdots \in [0,1]$ ma siccome f è una biiezione $f : \mathbb{N} \to [0,1]$ e $d \in [0,1]$ *deve* esistere un elemento $n \in \mathbb{N}$ tale che $f(n) = d$, ma per costruzione di d la n-esima cifra decimale di d deve essere *diversa* dalla n-esima cifra decimale di $f(n)$, ma $f(n) = d$.
Ciò è impossibile, quindi non esiste una biiezione tra $\mathbb{N}$ e $[0,1]$, e quindi l'insieme non è numerabile.

7.2 Teorema: esistono linguaggi non Turing-riconoscibili

Esistono linguaggi non Turing-riconoscibili, e quindi non decidibili.

7.2.1 Dim

Mostreremo che l'insieme di tutte le TM è numerabile, mentre l'insieme dei linguaggi non è numerabile. Quindi, siccome ci sono più linguaggi che TM, deve necessariamente esistere un linguaggio non risconosciuto da nessuna TM per il pidgeonhole principle. Definiamo quindi due insiemi:

1. $\mathcal{M} = \{\langle M \rangle : M \text{ è una macchina di Turing } \}$
2. $\mathcal{L} = \{L : L \subseteq \Sigma^*\}$

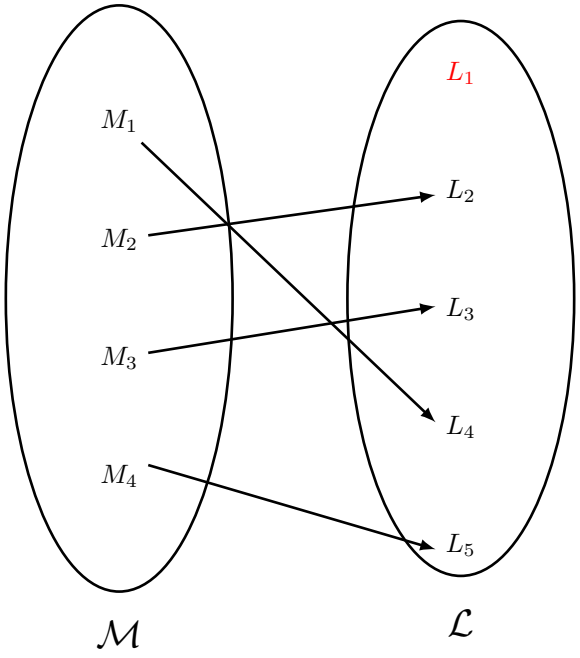


Figure 6: Pidgeonhole principle tra $\mathcal{M}$ e $\mathcal{L}$

$\mathcal{M}$ è numerabile Per dimostrare il teorema osserviamo che $\mathcal{M}$ è un insieme composto da stringhe, quindi ordinabile in modo canonico o quasi-lessicografico. Denotiamo il seguente ordinamento con il simbolo $\preceq$: Siano $x, y \in \Sigma^*$ e sia $<_l$ l'ordine lessicografico, ovvero l'ordine alfabetico, tra i caratteri di un alfabeto Σ ed esteso a parole di Σ^*:

$$x \prec y \Leftrightarrow \begin{cases} |x| < |y| \\ |x| = |y| \text{ e } x <_l y \end{cases}$$

Possiamo facilmente osservare che $\preceq$ è un ordine totale, ovvero che tutti gli elementi sono ordinati, e lineare, cioè possiamo disporre le stringhe ordinate su una riga immaginaria. Ciò implica che esiste una biiezione in $\mathbb{N}$:

$$f : \mathbb{N} \to \Sigma^* | f(i) = i\text{-esimo elemento in } \Sigma^* \text{ rispetto a } \preceq$$

In particolare è possibile definire una biiezione $f' : \mathbb{N} \to \Sigma^*$, quindi Σ^* è numerabile, e possiamo osservare che $\mathcal{M} \subseteq \Sigma^*$, pertanto $\mathcal{M}$ è numerabile.

$\mathcal{L}$ non è numerabile Per continuare la dimostrazione possiamo verificare che $\mathcal{L}$, l'insieme di tutti i linguaggi, non è numerabile, ovvero ha la stessa cardinalità di $\mathbb{R}$. Possiamo utilizzare la tecnica della diagonalizzazione: a ogni linguaggio può essere associato un certo numero reale espresso in binario e a ogni numero reale è associato un unico linguaggio. In particolare, dato Σ^* l'insieme di tutte le stringhe possibii, se un certo $L \in \mathcal{L}$ contiene l'i-esima stringa di Σ^*, allora alla i-esima posizione della funzione che modella la biiezione verso i reali sarà associato un 1, altrimenti uno 0. Più formalmente, sia $\Sigma^* = \{s_1, s_2, ...\}$ dove $s_i \prec s_{i+1}$. Poiché $L \subseteq \Sigma^*$, possiamo definire χ_L come segue:

$$\chi_L = \begin{cases} 1 & s_i \in L \\ 0 & s_i \notin L \end{cases}$$

Prendendo come riferimento il solo intervallo non numerabile di $\mathcal{R}$ $[0, 1]$:

Σ^*	$=\{$	$\epsilon,$	$0,$	$1,$	$00,$	$01,$	$11,$	$000,$	$001,$	$\ldots\}$
$L \in \mathcal{L}$	$=\{$		$0,$		$00,$	$01,$		$000,$	$001,$	$\ldots\}$
χ_L	$=$	$.0$	1	0	1	1	0	1	1	$\ldots$

Ogni linguaggio $L \in \mathcal{L}$ è quindi associato a un certo numero reale e, viceversa, a ogni numero reale è associato un unico linguaggio. Quindi $\mathcal{L}$ non è numerabile.

■

7.3 Teorema: A_{TM} è indecidibile

$$A_{TM} = \{\langle M, w\rangle : M \text{ è una TM e } M \text{ accetta la stringa di input } w\}$$

Con le capacità accumulate possiamo finalmente introdurre formalmente il teorema dapprima rimandato e dimostrarlo. In altre parole, è impossibile stabilire se una TM accetta una determinata stringa senza andare in loop.

7.3.1 Dim

Per assurdo, sia A_{TM} decidibile e sia H il suo decisore:

$$H(\langle M, w\rangle) = \begin{cases} accetta & \text{se } M \text{ accetta } w \\ rifiuta & \text{se } M \text{ rifiuta } w \end{cases}$$

Se H esiste possiamo costruire il seguente D che usa H come subroutine e produce la sua risposta contraria:
$D =$ su input $\langle M\rangle$:

1: Esegue H su input $\langle M, \langle M\rangle\rangle$;
2: Se H accetta *rifiuta*. Se H rifiuta *accetta*.

Abbiamo quindi definito D come segue:

$$D(\langle M\rangle) = \begin{cases} accetta & \Leftrightarrow & H \text{ rifiuta } \langle M, \langle M\rangle\rangle & \Leftrightarrow & M \text{ non accetta } \langle M\rangle \\ rifiuta & \Leftrightarrow & H \text{ accetta } \langle M, \langle M\rangle\rangle & \Leftrightarrow & M \text{ accetta } \langle M\rangle \end{cases}$$

Possiamo osservare che D è una TM decisore, quindi avrà la sua codifica $\langle D\rangle$ e in quanto decisore appartiene a A_{TM}. Supponiamo di lanciare D sulla descrizione di sé stesso:

$$D(\langle D\rangle) = \begin{cases} accetta & \Leftrightarrow & H \text{ rifiuta } \langle D, \langle D\rangle\rangle & \Leftrightarrow & D \text{ non accetta } \langle D\rangle \\ rifiuta & \Leftrightarrow & H \text{ accetta } \langle D, \langle D\rangle\rangle & \Leftrightarrow & D \text{ accetta } \langle D\rangle \end{cases}$$

Ad esempio:

	$\langle M_1\rangle$	$\langle M_2\rangle$	$\langle M_3\rangle$	$\langle M_4\rangle$	...	$\langle D\rangle$	...
M_1	**acc**	*rej*	*acc*	*rej*	...	*acc*	...
M_2	*acc*	**acc**	*acc*	*acc*	...	*acc*	...
M_3	*rej*	*rej*	**rej**	*rej*	...	*rej*	...
M_4	*acc*	*acc*	*rej*	**rej**	...	*rej*	...
⋮							
D	*rej*	*rej*	*acc*	*acc*	...	???	...
⋮							

Il comportamento anomalo di D su $\langle D\rangle$ ci porta a un'evidente contraddizione. Quindi H non può esistere. E quindi A_{TM} non può essere deciso. Tuttavia A_{TM} è un linguaggio Turing-riconoscibile.

7.4 Teorema: A_{TM} è Turing-riconoscibile

Abbiamo mostrato che A_{TM} non è decidibile, tuttavia il linguaggio è Turing-riconoscibile. Per dimostrarlo, utilizziamo una TM U che implementa il seguente algoritmo:
$U =$ su input $\langle M, w\rangle$, dove M è una TM e w è una stringa:

1: Simula M con input w;
2: Se M durante la computazione M accetta *accetta*, se M rifiuta *rifiuta*.

U su input $\langle M, w\rangle$ entra in un loop infinito se M entra in un loop infinito su w. Quindi U non è un decisore, ma solo un riconoscitore.
Il teorema ci sta dicendo che è impossibile determinare se una TM entra in loop. La macchina di Turing che implementa lo pseudocodice appena mostrato viene detta *macchina di Turing universale*, perché simula una qualsiasi macchina di Turing M a partire da una sua descrizione.

Macchina di Turing universale Vediamo in dettaglio l'implementazione della macchina di Turing universale. U utilizza due nastri:

- N_1 contiene la codifica $\langle M, w\rangle$;
- N_2 contiene la codifica $\langle (a, q, b)\rangle$ della configurazione corrente dove:
  1. ab è il contenuto del nastro;
  2. q è lo stato in cui si trova M;
  3. b dove si trova la testina.

7.5 Chiusura dei linguaggi Turing-riconoscibili

Claim: I linguaggi turing-riconoscibili sono chiusi rispetto a unione e intersezione, non complemento.

8 Linguaggi non-Turing riconoscibili

Abbiamo già dimostrato che esistono linguaggi non Turing-riconoscibili; ora possiamo definire qualche altra proprietà che li riguarda.

8.1 Def: linguaggio coTuring-riconoscibile

Diciamo che un linguaggio L è coTuring-riconoscibile quando $\overline{L}$ è Turing-riconoscibile. Alternativamente, $\overline{L}$ è Turing-riconoscibile $\equiv L$ è coTuring-riconoscibile.

8.2 Teorema: L decidibile $\Leftrightarrow L, \overline{L}$ Turing-riconoscibili

L decidibile $\Rightarrow L, \overline{L}$ Turing-riconoscibili e se $L, \overline{L}$ sono Turing-riconoscibili $\Rightarrow$ L decidibile.

8.2.1 Dim ($\Rightarrow$):

Se L è decidibile $\Rightarrow L$ è anche Turing-riconoscibile per definizione.

8.2.2 Dim ($\Leftarrow$):

Siano M_1, M_2 TM che riconoscono $L, \overline{L}$ rispettivamente. Costruisco M che decide L:
M = su input w:

1: Esegue M_1, M_2 in parallelo su input w;
2: Se M_1 accetta *accetta*. Se M_2 accetta *rifiuta*.

Abbiamo quindi definito una TM M che decide L:

$$\forall w, w \in L \text{ oppure } w \in \overline{L} \Rightarrow M \text{ accetta solo } w \in L \Rightarrow M \text{ decide } L$$

8.2.3 Corollario

Se un linguaggio L non è decidibile, allora uno tra L e $\overline{L}$ non è Turing-riconoscibile.

8.3 Teorema: $\overline{A_{TM}}$ non è Turing-riconoscibile

Il linguaggio $\overline{A_{TM}}$ non è Turing-riconoscibile.

8.3.1 Dim

Se per contraddizione $\overline{A_{TM}}$ fosse Turing-riconoscibile, allora visto che A_{TM} è Turing-riconoscibile seguirebbe dal corollario 8.2.3 che A_{TM} sia anche decidibile, ma nella sezione 7.3 abbiamo dimostrato che non lo è.

9 Riducibilità

La riducibilità è una tecnica matematica che ci permette di dimostrare agevolmente problemi. Siano due problemi A, B diciamo che A *si riduce* a B ($A \leq B$) quando possiamo usare una soluzione di B per risolvere A. Una riduzione non dice nulla su come risolvere né A né B, ma solo su come *usare* B per *risolvere* A.
Possiamo applicare la tecnica della riducibilità all'indecibilità:

$$A \leq B \text{ e } B \text{ indecidibile} \rightarrow A \text{ indecidibile}$$

9.1 Teorema: $HALT_{TM}$ non è decidibile

Il linguaggio $HALT_{TM}$ non è decidibile.

$$HALT_{TM} = \{\langle M, w\rangle : M \text{ è una TM e } M \text{ si ferma su input } w\}$$

9.1.1 Dim per riduzione

Dimostro che $A_{TM} \leq HALT_{TM}$: facendo ciò, se per assurdo $HALT_{TM}$ fosse decidibile, la riduzione implicherebbe che anche A_{TM} lo sia, ma abbiamo già dimostrato che non lo è. Se ciò accade avremmo una contraddizione, quindi $HALT_{TM}$ non può essere decidibile.
Sia per assurdo $HALT_{TM}$ decidibile con un certo decisore R. Costruisco S che decide A_{TM}:
S su input $\langle M, w\rangle$:

1: Esegue su R su $\langle M, w\rangle$;
2: Se R rifiuta, *rifiuta*;
3: Se R accetta, simula M su w finché non si ferma;
4: Se M ha accettato, *accetta*. Se M ha rifiutato, *rifiuta*.

9.2 Teorema: E_{TM} non è decidibile

Il linguaggio E_{TM} non è decidibile.

$$E_{TM} = \{\langle M\rangle : M \text{ è una TM e } L(M) = \emptyset\}$$

9.2.1 Dim per riduzione

Per assurdo, sia R TM decisore per E_{TM}. Costruisco S che decide A_{TM}.
Idea: S lancia $R(\langle M\rangle)$. Se R accetta, allora rifiuta, altrimenti eseguo M; ma M può entrare in loop. Allora modifico M in M_w che rifiuta tutto tranne w, ovvero:
M_w su input x:

1: Se $x \neq w$ rifiuta;
2: Se $x = w$, simula M su w e *accetta* se M accetta.

Posso modellare quindi la TM S che decide A_{TM}: la TM trasforma M in M_w che rifiuta tutto tranne w, quindi possiamo eseguire il test del vuoto per confermare se $L(M_w) = \varnothing$, quindi per costruzione M non accetta w, oppure se $L(M_w) = \{w\}$, quindi M accetta w.
$S =$ su input $\langle M, w\rangle$:

1: Usa $\langle M\rangle$ per costruire $\langle M_w\rangle$;
2: Esegue R su input $\langle M_w\rangle$;
3: Se R accetta, *rifiuta*; ▷ Il linguaggio è vuoto.
4: Se R rifiuta *accetta*. ▷ Il linguaggio contiene esclusivamente w.

9.3 Teorema: REG_{TM} non è decidibile

Il linguaggio REG_{TM} non è decidibile.

$$REG_{TM} = \{\langle M\rangle : M \text{ è TM e } L(M) \text{ è un linguaggio regolare}\}$$

9.3.1 Dim per riduzione

Sia R decisore per REG_{TM}. Costruisco S che decide A_{TM}.
Idea: S prende il suo input $\langle M, w\rangle$ e modifica M in M_2 in modo che M_2 riconosce un linguaggio regolare se e solo se M accetta w. Utilizziamo un generico linguaggio non regolare che sarà riconosciuto da M_2 se M non accetta w, come ad esempio $\{0^n1^n : n \geq 0\}$, e riconosca il linguaggio regolare Σ^* se M accetta w. In particolare, M_2 accetta automaticamente tutte le stringhe in $\{0^n1^n : n \geq 0\}$. Inoltre, se M accetta w, M_2 accetta tutte le altre stringhe.
È da osservare che M_2 non è costruita con lo scopo di essere eseguita. La costruiamo col solo scopo di dare in input la sua descrizione al decisore R di REG_{TM} che abbiamo assunto esistere.

$$S \text{ accetta} \Leftrightarrow R \text{ accetta} \Leftrightarrow L(M_2) \text{ è regolare} \Leftrightarrow M \text{ accetta } w$$

M_2 su input x:

1: Se x ha la forma 0^n1^n, *accetta*;
2: Se x non ha tale forma, esegue M su input w e *accetta* se M accetta

Definiamo quindi S che decide A_{TM}:
$S =$ su input $\langle M, w\rangle$:

1: Costruisce la TM M_2
2: Esegue R su input $\langle M_2\rangle$; ▷ Se R accetta, $w \in L(M)$
3: Se R accetta, *accetta*. Se R rifiuta, *rifiuta*.

9.4 Teorema: EQ_{TM} non è decidibile

Il linguaggio EQ_{TM} non è decidibile.

$$EQ_{TM} = \{\langle M_1, M_2\rangle : M_1, M_2 \text{ TM e } L(M_1) = L(M_2)\}$$

9.4.1 Dim per riduzione

Sia R decisore per EQ_{TM}. Costruisco S che decide E_{TM}, che abbiamo già dimostrato nella sezione 9.2 essere non decidibile.
Idea: costruisco M_{rej} tale che $L(M_{rej}) = \varnothing$ per definizione.
$M_{rej} =$ su input x:

1: *rifiuta*.

$S =$ su input $\langle M\rangle$:

1: Costruisco M_{rej};
2: Eseguo $R(\langle M, M_{rej}\rangle)$;
3: Se R accetta *accetta*. Se R rifiuta *rifiuta*.

10 Riducibilità mediante funzione

Abbiamo mostrato come utilizzare la tecnica della riducibilità per dimostrare che alcuni problemi sono indecidibili. Ora possiamo formalizzare il concetto di

riducibilità e utilizzarlo in modo più raffinato. La nozione di ridurre un problema a un altro può essere definita formalmente in vari modi, la nostra scelta ricade sulla *riducibilità mediante funzione*. Informalmente, essere in grado di ridurre un problema A a un altro problema B utilizzando una riduzione mediante funzione significa che esiste una funzione *calcolabile* che trasforma istanze del problema A in istanze del problema B. Se abbiamo tale funzione, detta riduzione, siamo in grado di risolvere A risolvendo istanze di B attraverso la trasformazione da A in B.

10.1 Def: Funzione calcolabile

Una funzione $f : \Sigma^* \to \Sigma^*$ *è calcolabile* se esiste una TM t.c. $\forall w \in \Sigma^*$ questa termina con $f(w)$ sul nastro.

10.2 Def: Linguaggio riducibile mediante funzione

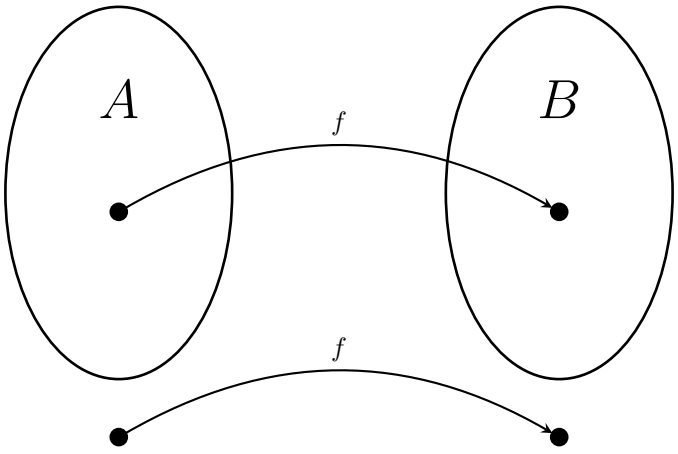
Un linguaggio A è *riducibile mediante funzione* al linguaggio B se esiste una funzione f calcolabile come segue:

$$f : \Sigma^* \to \Sigma^* : \forall w \in \Sigma^*, w \in A \Leftrightarrow f(w) \in B$$

Possiamo quindi dire che f è riduzione da A a B. In particolare, la funzione mappa un $w \in A$ a un $w' \in B$, e tutti $w \notin A$ a $w' \notin B$. Un linguaggio riducibile mediante funzione si denota con:

$$A \leq_m B$$

La m in $\leq_m$ specifica che la riduzione avviene attraverso una *mapping function*.



10.3 Teorema: Se $A \leq_m B$ e B è decidibile, allora A è decidibile

Se A si riduce a B e B è decidibile, allora A è decidibile.

10.3.1 Dim

Sia R la TM che decide B e sia f la riduzione, ovvero la funzione, che riduce A a B. Possiamo costruire un decisore S per A:
$S =$ su input w:

1: Calcola $f(w)$;
2: Esegue R su $f(w)$;
3: Se R accetta *accetta*, altrimenti *rifiuta*.

In particolare:

- Se $w \in A, f(w) \in B \Rightarrow S$ accetta.
- Se $w \notin A, f(w) \notin B \Rightarrow S$ rifiuta.

Quindi S è un decisore per B.

10.3.2 Corollario

Se $A \leq_m B$ e A indecidibile, allora B è indecidibile.

10.4 $HALT_{TM}$ non è decidibile: dimostrazione per mapping function

Abbiamo già dimostrato per riduzione generale che $HALT_{TM}$ non è decidibile, ma ora proviamo a dimostrarlo attraverso una funzione di riduzione.
Effettuiamo una riduzione da A_{TM} a $HALT_{TM}$ modellando una certa f che trasforma ogni istanza del problema A_{TM} in un'istanza del problema $HALT_{TM}$ e ogni non-istanza del problema A_{TM} in una non-istanza del problema $HALT_{TM}$. La funzione quindi prende in input $\langle M, w\rangle \in A_{TM}$ e restituisce in output $\langle M', w\rangle \in HALT_{TM}$.

$$\forall x \in \Sigma^*, x = \langle M, w\rangle \in A_{TM} \Leftrightarrow f(x) = f(\langle M, w\rangle) \in HALT_{TM}$$

Modelliamo la macchina M' che sarà chiamata da F:
$M' =$ su input x:

1: Esegue M su x;
2: Se M accetta, *accetta*; ▷ accetta, quindi $M' \in HALT_{TM}$
3: Se M rifiuta, cicla. ▷ cicla, quindi $M' \notin HALT_{TM}$

Modelliamo la TM F che calcola f:
$F =$ su input $\langle M, w\rangle$:

1: Costruisce M';
2: Output $\langle M', w\rangle$.

Abbiamo trasformato un'istanza di A_{TM} in un'istanza di $HALT_{TM}$ e una non-istanza di A_{TM} in una non-istanza di $HALT_{TM}$ con successo. Per il corollario 10.3.2 ne deduciamo che $HALT_{TM}$ è indecidibile.

10.5 EQ_{TM} non è decidibile: dimostrazione per mapping reduction

Costruiamo una f che trasforma ogni istanza del problema E_{TM} in una istanza del problema EQ_{TM}, e una non-istanza del problema E_{TM} in una non-istanza del problema EQ_{TM}. Con questa f, avendo già dimostrato che E_{TM} non è decidibile, per il teorema, EQ_{TM} non è decidibile.

$$\langle M \rangle \in E_{TM} \Leftrightarrow \langle M, M' \rangle = f(\langle M \rangle) \in EQ_{TM}$$

$F =$ su input $\langle M \rangle$:

1: Costruisce M' codifica di una TM che rifiuta sempre;
2: Output: $\langle M, M' \rangle$

In particolare, se $\langle M \rangle \in E_{TM}$, allora $L(M) = \varnothing$. Quindi, per definizione di M', $L(M) = L(M') \rightarrow \langle M, M' \rangle \in EQ_{TM}$.
Inoltre, se $f(\langle M \rangle) = \langle M, M' \rangle \in EQ_{TM}$, allora $L(M) = L(M')$, ma $L(M) = \varnothing$ $\rightarrow \langle M \rangle \in E_{TM}$.
Abbiamo trasformato un'istanza di E_{TM} in un'istanza di EQ_{TM} e una non-istanza di E_{TM} in una non-istanza di EQ_{TM} con successo. Per il corollario e il teorema, sapendo che E_{TM} non è dedicibile, ne concludiamo che EQ_{TM} non è decidibile.

11 Turing-riconoscibilità e riduzioni

Data una riduzione da A a B e avendo informazioni su uno dei due linguaggi, possiamo ricavare informazioni sull'altro linguaggio.

11.1 Teorema: se $A \leq_m B$ e B è Turing-riconoscibile, allora A è Turing-riconoscibile

Analogamente per la proprietà di decidibilità, se $A \leq_m B$ e B è Turing-riconoscibile, allora A è Turing-riconoscibile.

11.1.1 Corollario

Analogamente per la proprietà di indecidibilità, se $A \leq_m B$ e A non è Turing-riconoscibile, allora B non è Turing-riconoscibile.

11.1.2 Applicazioni: dimostriamo la non Turing-riconoscibilità attraverso $\overline{A_{TM}}$

Sappiamo che $\overline{A_{TM}}$ non è Turing-riconoscibile. Possiamo sfruttare questa conoscenza per dimostrare che altri linguaggi non sono Turing-riconoscibili attraverso una funzione di riduzione.

11.1.3 Teorema: se $A \leq_m B$, allora $\overline{A} \leq_m \overline{B}$

Se $\overline{A}$ non è Turing-riconoscibile e $A \leq_m B$, allora $\overline{B}$ non è Turing-riconoscibile.

11.1.4 EQ_{TM} non è Turing-riconoscibile: dimostrazione per mapping reduction

Sappiamo che $\overline{A_{TM}}$ non è Turing-riconoscibile. Per il teorema precedente dimostro che $A_{TM} \leq_m \overline{EQ_{TM}}$ e da ciò ne deriva anche che $\overline{A_{TM}} \leq_m EQ_{TM}$. Trasformo quindi un'istanza di A_{TM} in un'istanza di $\overline{EQ_{TM}}$ e una non-istanza di A_{TM} in una non-istanza di $\overline{EQ_{TM}}$.

$$\langle M, w\rangle \in A_{TM} \Leftrightarrow f(\langle M, w\rangle) \in \overline{EQ_{TM}}$$

Modelliamo due TM M_1 e M_2:
M_1 = su input x:
1: Rifiuta;
M_2 = su input x:
1: Esegue M su w. Se M accetta, *accetta*.
Ora costruiamo la nostra F che implementa la funzione di riduzione:
F = su input $\langle M, w\rangle$:
1: Costruisco $\langle M_1, M_2\rangle$;
2: Output: $\langle M_1, M_2\rangle$.
M_1 non accetta alcuna stringa. $L(M_1) = \varnothing$. Se M accetta w, M_2 accetta qualsiasi stringa. $L(M_2) = \Sigma^*$, quindi le due macchine non sono equivalenti. Viceversa, se M non accetta w, M_2 non accetta alcuna stringa, ed esse sono equivalenti. Perciò F riduce A_{TM} a $\overline{EQ_{TM}}$.

Part III
Complessità

12 Complessità: introduzione

Finora non ci siamo preoccupati di misurare l'efficienza dei nostri algoritmi; iniziamo a rifletterci.
Dato un problema, come $PATH$, vorremmo un algoritmo, ovvero una TM, che lo risolva in maniera *efficiente*.
L'efficienza può essere espressa in termini di:

1. tempo e spazio;
2. processore;
3. energia

4. randomness;

5. ...

In questo corso ci concentreremo solo sulla complessità in termini di tempo e di spazio.
Osserviamo ora le differenze distintive di calcolabilitità e complessità:
La calcolabilità, in particolare, risponde alla domanda *quali problemi sono risolvibili?*
Abbiamo osservato che la risposta è *non tutti*. In particolare, $HALT_{TM}$ non è risolvibile.
La complessità, invece:

1. classifica problemi in termini di efficienza;
2. $P = NP$? Ovvero, trovare una soluzione è tanto veloce quanto verificarla?
   Intuitivamente sembra molto più facile verificare una soluzione anziché trovarla, ma la domanda non è così banale.
3. $P = NC$? Ogni algoritmo è parallelizzabile efficientemente?
4. $P = L$? Gli algoritmi devono allocare memoria per forza?
5. $P = BPP$? Gli algoritmi efficienti si possono de-randomizzare?
6. $P = PSPACE$? Se posso risolvere un problema con poco spazio, posso risolverlo anche in poco tempo?

Al giorno d'oggi non sappiamo rispondere con certezza a nessuna di queste domande. In questo corso parleremo maggiormente delle classi P, NP e $PSPACE$.

12.1 Def: tempo di esecuzione di una TM

Sia M un decisore. La complessità di tempo M è una funzione

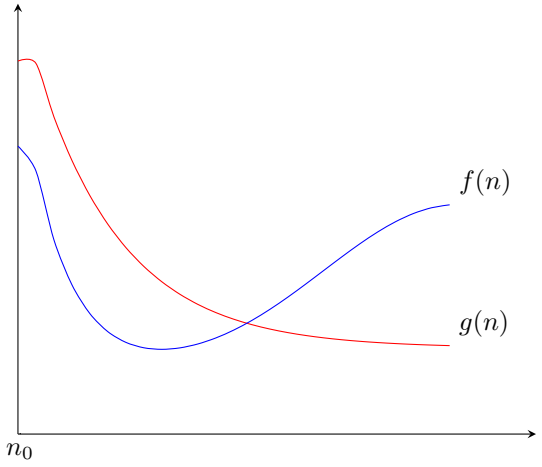
$$T : \mathbb{N} \to \mathbb{N} : T(n) = max(x \in \Sigma^* : |x| = n)^{15}$$

Questa notazione si chiama *worst-case notation*.
Per quello che ci interessa definire in termini di complessità, le costanti non contano.

[15] numero di passi di M con input x

12.2 Def: $O(n)$

Siano $f, g : \mathbb{N} \to \mathbb{R}^+$. Allora $f(n) = O(g(n))$ se $\exists c, n_0 \in \mathbb{N}^+ : \forall n \geq n_0$ si ha $f(n) \leq c \cdot g(n)$.



Es:

$5n = O(n)$;

$4n^3 + 3n^2 + n = O(n^3)$

12.3 Teorema