Appunti di Informatica Teorica

Riccardo Lo Iacono & Stefano Graffeo

Dipartimento di Matematica & Informatica
Università degli studi di Palermo
Sicilia
a.a. 2022-2023

Indice.

– 1 – Teoria degli automi: introduzione e concetti base.

Si consideri il caso di un interruttore. Grazie agli automi è possibile rappresentare facilmente il passaggio tra i due stati come mostrato in 1.

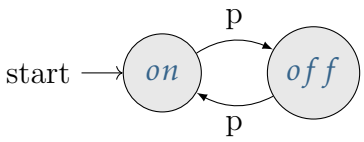


Figura 1: Automa rappresentante uno switch.

Dando una breve definizione di automa: questi è un sistema automatico, rappresentato da un grafo i cui nodi rappresentano gli stati e gli archi le transizioni tra stati.

L'utilizzo degli automi è da ricercare nello studio dei limiti computazionali, cui si legano

1. lo studio della decidibilità, che stabilisce cosa possa fare un computer in assoluto;
2. lo studio della trattabilità che stabilisce cosa possa fare un compute efficientemente.

Agli automi sono inoltre legati due importanti nozioni, quali le grammatiche e le espressioni regolari, che si discuteranno nelle successive sezioni.

– 1.1 – Concetti centrali.

Concetti centrali della teoria degli automi sono gli alfabeti, le stringhe e i linguaggi.

- *Gli alfabeti:* si definisce alfabeto Σ un insieme finito di caratteri.
- *Le Stringhe:* dato Σ un alfabeto, si definisce stringa ω una sequenza di simboli scelti dall'alfabeto.

  Caso particolare è la stringa vuota ε: una stringa composta da zero simboli.

  Data ω una stringa, di questa è possibile stabilirne la lunghezza: ossia il numero di caratteri di cui si compone.

  Infine, considerate $\omega_1 = a_1 \cdots a_k$ e $\omega_2 = b_1 \cdots b_j$ due stringhe, si definisce $\omega_1 \circ \omega_2 = \omega_1\omega_2 = a_1 \cdots a_k b_1 \cdots b_j$ concatenazione di ω_1 e ω_2.
- *I Linguaggi:* dato Σ un alfabeto, si definisce linguaggio L su Σ un sottoinsieme delle stringe ottenibili con l'alfabeto.

– 2 – Automi DFA e NFA.

Come anticipato in *Sezione* (1): un automa è un sistema automatico, rappresentato da un grafo.
Si tenga presente che esistono due classi di automi

- deterministici o DFA;
- non deterministici o NFA.

– 2.1 – I DFA.

Definizione: Si definisce $A = (Q, \Sigma, \delta, q_0, F)$ DFA, ove

- Q rappresenta l'insieme di stati dell'automa;
- Σ è l'alfabeto utilizzato dall'automa;
- δ definisce le transizioni tra gli stati;
- q_0 indica lo stato iniziale;
- F definisce l'insieme di stati finali;

se considerata δ, per ciascun simbolo dell'alfabeto e per ciascuno stato esiste un'unica transizione per quel carattere.

– 2.1.1 – Funzione di trasizione e funzione di transizione estesa.

Dato un automa A, la funzione di transizione δ stabilisce il comportamento dell'automa in ogni suo stato, per ogni simbolo dell'alfabeto.

Esempio: Sia considerato l'automa di *Figura* (1).
La funzione di transizione dello stesso, definisce le seguenti transizioni

$$\delta(on, p) = (off)$$
$$\delta(off, p) = (on)$$

ossia: letto p dallo stato on passa allo stato off , da questi letto p passa a on.

Definizione: Sia $\omega = a_1 \cdots a_n$ una stringa e δ la funzione di transizione di un dato DFA: si definisce funzione di transizione estesa δ^* la funzione che, letta ω a partire da q_0, stabilisce lo stato di arrivo q_f. Cioè

$$\delta^*(q_0, \omega) = (q_f)$$

Osservazione: Dato un automa, la funzione di transizione estesa δ^*, può essere intesa come la sequenziale applicazione della funzione di transizione δ, per ogni simbolo in ω a partire dallo stato q_0.

– 2.2 – Gli NFA.

Definizione: Si definisce $A = (Q, \Sigma, \delta, q_0, F)$ NFA, ove

- Q rappresenta l'insieme di stati dell'automa;
- Σ è l'alfabeto utilizzato dall'automa;
- δ definisce le transizioni tra gli stati;
- q_0 indica lo stato iniziale;
- F definisce l'insieme di stati finali;

se considerata δ, per ciascun simbolo dell'alfabeto e per almeno uno stato esistono più transizioni per quel carattere.

Esempio: Sia considerato l'automa in *Figura* (1), questi può essere rappresentato come NFA dall'automa in *Figura* (2).

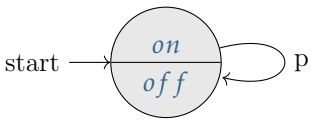


Figura 2: Automa rappresentante uno switch come NFA.

– 2.2.1 – Funzione di transizione estesa.

Definizione: Sia $\omega = a_1 \cdots a_n$ una stringa e δ la funzione di transizione di un dato NFA: si definisce funzione di transizione estesa δ^* la funzione che, letta ω a partire da q_0, stabilisce lo stato di arrivo q_f.

$$\text{Per induzione si ha } \begin{cases} \delta^*(q_0, \varepsilon) = \{q_0\} & \text{base} \\ \delta^*(q_0, \omega) = \bigcup_{q_x \in \delta^*(q_0, \omega)} \delta(q_x, a) \end{cases}$$

– 2.3 – DFA e NFA: linguaggi e proprietà dei linguaggi.

Definizione: Sia A un automa. Si definisce linguaggio di $A, L(A)$, l'insieme delle stringhe ω che accettate da A. Cioè

$$\begin{cases} L(A) = \{\omega : \delta^*(q_0, \omega) \in F\} & \text{se } A \text{ è un DFA} \\ L(A) = \{\omega : \delta^*(q_0, \omega) \cap F \neq \emptyset\} & \text{se } A \text{ è un NFA} \end{cases}$$

– 2.3.1 – Proprietà dei linguaggi.

Sia L il linguaggio riconosciuto da un automa; su di questi è possibile applicare le seguenti operazioni.

- *Potenza n-sima:* si intende la concatenazione di L un certo numero n di volte.

  Esempio: Sia $L = \{\omega : \omega \in \Sigma = \{a, b\}\}$, sia $n = 2$. Segue

  $$L^2 = L \circ L = \{aaaa, aaab, aabb, aaba, abaa, abab, abbb, abba, \dots\}$$

  Osservazione: Se $n = 0$ si ha che $L^0 = \{\varepsilon\}$.

- *Stella di Kleene:* rappresenta l'unione di tutte le potenze di L. Cioè

  $$L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$$

  Osservazione: Se $L = \emptyset$ allora $L^* = \{\varepsilon\}$.

- *Croce:* indica l'unione di tutte le potenze di L , meno L^0. Cioè

  $$L^+ = L^1 \cup L^2 \cup \cdots$$

  Vale dunque

  $$L^+ = L \circ L^*$$

– 2.3.2 – Linguaggio universale e complemento.

Definizione: Sia Σ un alfabeto. Si definisce linguaggio universale Σ^*, l'insieme di tutte le parole applicando all'alfabeto Kleene.

Definizione: Sia L un linguaggio su un alfabeto Σ. Si definisce complemento di L, L^C, l'insieme di stringhe che appartengono a Σ^* ma non a L.

– 2.4 – Equivalenza tra NFA e DFA.

Si potrebbe erroneamente pensare che NFA e DFA riconoscano linguaggi diversi, ma si dimostra che non è così.
Prima di dimostrare il teorema di equivalenza tra NFA e DFA, è necessario parlare di *subset construction*.

– 2.4.1 – Subset construction.

Sia $N = (Q = \{q_0, q_1, \ldots, q_k\}, \Sigma, \delta_N, q_0, F_N)$ un NFA.
Per ogni $q_i \in Q, i = 0, 1, \ldots, k$ e per ogni $x \in \Sigma$, si definisco gli stati di un DFA D, dati dall'insieme degli stati definite da δ_N. Inoltre, uno stato di D sara accettante se, almeno uno, degli sti di N da cui è definito è accettante. In fine le transizioni di D, sono analoghe a quelle di N.

Esempio: Sia considerato l'NFA di *Figura* (3)

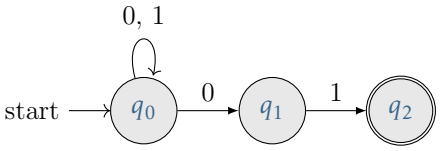


Figura 3: Automa per il linguaggio delle parole che terminano con 01.

Considerando δ si ha

$$\delta(q_0, 0) = (q_0)$$
$$\delta(q_0, 1) = (q_0)$$
$$\delta(q_0, 0) = (q_1)$$
$$\delta(q_1, 1) = (q_2)$$

da ciò segue l'automa di *Figura* (4).

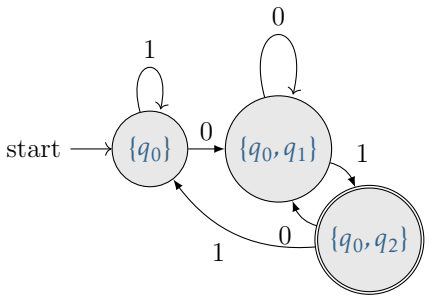


Figura 4: Subset construction dell'automa di *Figura* (3).

Poiché gli stati $\{q_1\}, \{q_2\}$ sono inaccessibili da $\{q_0\}$, questi sono stati trascurati.

– 2.4.2 – Teorema di equivalenza tra NFA e DFA.

Teorema 2.1.
Sia D un DFA ottenuto per subset construction da un NFA N, allora $L(D) = L(N)$.

Dimostrazione: Per dimostrare che $L(D) = L(N)$, si procederà per induzione su $|\omega|$ che

$$\delta_D^*(\{q_0\}, \omega) = \delta_N^*(q_0, \omega) \tag{1}$$

Base: Sia $|\omega| = 0$, ossia $\omega = \varepsilon$.
Per definizione di δ^* , segue che $\delta_D^*(\{q_0\}, \omega) = \delta_N^*(q_0, \omega) = \{q_0\}$.

Induzione: Supposto che quanto detto finora sia vero per $|\omega| = n$, si consideri $|\omega| = n + 1$. Sia posta $\omega = xa$, ove a è l'ultimo carattere della stringa.
Per ipotesi induttiva $\delta_D^*(\{q_0\}, x) = \delta_N^*(q_0, x) = \{p_1, \ldots, p_k\}$, segue dalla definizione induttiva di δ^* per gli NFA

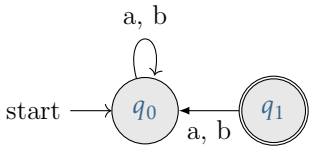
$$\delta_N^*(q_0, \omega) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$$

ma $\bigcup_{i=1}^{k} \delta_N(p_i, a) = \delta_D(\{p_1, \ldots, p_k\}, a)$, segue pertanto

$$\delta_D^*(\{q_0\}, \omega) = \delta_D(\{p_1, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$$

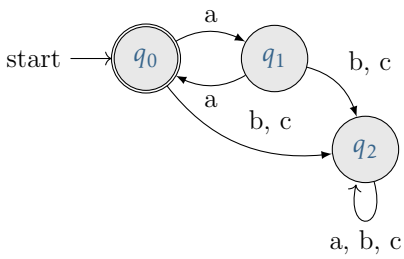
– 2.5 – Esercizi su DFA e NFA.

1. Sia $L = \{\}$ definito su $\Sigma = \{a, b\}$. Si realizzi un automa che lo riconosca.

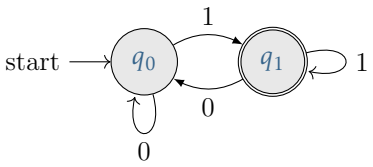


Nota: Soluzione al problema è un qualsiasi DFA, o NFA che sia, al cui stato accettante non è possibile accedere.

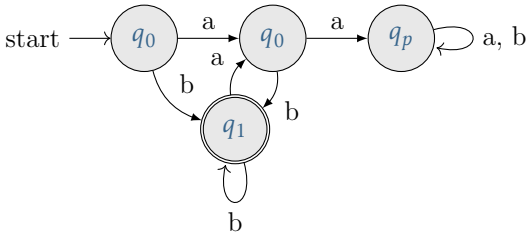
2. Sia $L = \left\{a^{2n} : n \geq 0\right\}$ definito su $\Sigma = \{a, b, c\}$. Si realizzi un automa che lo riconosca.



3. Sia $L = \{\omega : \omega = \Sigma^* 1\}$ definito su $\Sigma = \{0, 1\}$. Si realizzi un automa che lo riconosca.



4. Sia $L = \left\{\omega : \omega = (\Sigma^* aa\Sigma^*)^C\right\}$. Si realizzi un automa che lo riconosca.



– 3 – Proprietà dei linguaggi REC.

Definizione: Sia L un linguaggio. Questi si definisce regolare se accettato da un DFA.

I linguaggi regolari sono chiusi, cioè rimangono regolari, rispetto operazioni quali

- *intersezione*;
- *unione*;
- *complemento*;
- *Kleene*;
- *croce*.

– 3.1 – Chiusura per intersezione.

Teorema 3.1.
Siano L_1 e L_2 linguaggi REC. Allora $L = L_1 \cap L_2$ è REC.

Dimostrazione: Sia A_1 un automa che riconosce L_1 , sia A_2 un automa che riconosce L_2.

$$A_1 = (Q_1, \Sigma, \delta_1, q_{0_1}, F_1) \qquad A_2 = (Q_2, \Sigma, \delta_2, q_{0_2}, F_2)$$

Sia $A = (Q, \Sigma, \delta, q_0, F)$ un automa che riconosce L. Ponendo

- $Q = \{(q_1, q_2) : q_1 \in Q_1 \wedge q_2 \in Q_2\}$ o analogamente $Q = Q_1 \times Q_2$;
- $q_0 = (q_{0_1}, q_{0_2})$;
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ per ogni a tale che la transizione sia definita sia in A_1 che A_2;
- $F = \{(q_1, q_2) : q_1 \in F_1 \wedge q_2 \in F_2\}$ o analogamente $F = F_1 \times F_2$.

Sia $\omega \in L$, segue

$$\begin{aligned} \omega \in L &\iff \omega \in L_1 \wedge \omega \in L_2 \\ &\iff \delta_1^*(q_{0_1}, \omega) \in F_1 \wedge \delta_2^*(q_{0_2}, \omega) \in F_2 \\ &\iff \delta^*((q_{0_1}, q_{0_2}), \omega) \in F \end{aligned}$$

– 3.2 – Chiusura per unione.

Teorema 3.2.
Siano L_1 e L_2 linguaggi REC. Allora $L = L_1 \cup L_2$ è REC.

Dimostrazione: Sia A_1 un automa che riconosce L_1 , sia A_2 un automa che riconosce L_2.

$$A_1 = (Q_1, \Sigma, \delta_1, q_{0_1}, F_1) \qquad A_2 = (Q_2, \Sigma, \delta_2, q_{0_2}, F_2)$$

Sia $A = (Q, \Sigma, \delta, q_0, F)$ un automa che riconosce L. Ponendo

- $Q = \{(q_1, q_2) : q_1 \in Q_1 \wedge q_2 \in Q_2\}$ o analogamente $Q = Q_1 \times Q_2$;
- $q_0 = (q_{0_1}, q_{0_2})$;
- $\delta((q_1, q_2), a) = (\delta_1(q_1, a), \delta_2(q_2, a))$ per ogni a tale che la transizione sia definita sia in A_1 che A_2;
- $F = \{(q_1, q_2) : q_1 \in F_1 \wedge q_2 \in F_2\}$.

Sia $\omega \in L$, segue

$$\begin{aligned} \omega \in L &\iff \omega \in L_1 \vee \omega \in L_2 \\ &\iff \delta_1^*(q_{0_1}, \omega) \in F_1 \vee \delta_2^*(q_{0_2}, \omega) \in F_2 \\ &\iff \delta^*((q_{0_1}, q_{0_2}), \omega) \in F \end{aligned}$$

Nota: Similarmente si dimostra anche la chiusura per complemento, Kleene, croce.

– 4 – Espressioni regolari.

Definizione: Si definisce *espressione regolare*, (o RegEx) e, la descrizione algebrica delle stringhe di un dato linguaggio.

– 4.1 – Costruzione di una RegEx.

Sia e una RegEx. La costruzione di e è di tipo ricorsivo.

Base:

- ε e $\emptyset$ sono espressioni regolari, ove $L(\varepsilon) = \{\varepsilon\}, L(\emptyset) = \{\}$.
- Se α è un simbolo, allora questi è una RegEx, ove $L(\alpha) = \{\alpha\}$.

Induzione:

- Siano e ed f due RegEx. Allora $e + f$ è un'espressione regolare.
- Siano e ed f due RegEx. Allora ef è un'espressione regolare.
- Sia e RegEx. Allora e^* è un'espressione regolare.
- Sia e RegEx. Allora (e) è un'espressione regolare.

– 4.2 – Precedenza nelle RegEx.

Quando si opera con due o più espressioni regolari, bisogna prestare attenzione agli operatori che le lega. In generale, la priorità massima è assegnata alla Stella di Kleene, a seguire la concatenazione e in ultimo la croce.

– 4.3 – Linguaggi locali.

Definizione: Sia L un linguaggio. Questi dicasi locale se esprimibile tramite la seguente quadrupla.

$$(Ini(L), Fin(L), Dig(L), Null(L))$$

ove

- $Ini(L)$ stabilisce l'insieme di caratteri con cui $\omega \in L$ può iniziare.
- $Fin(L)$ stabilisce l'insieme di caratteri con cui $\omega \in L$ può terminare.
- $Dig(L)$ stabilisce l'insieme di tutte le possibili coppie di caratteri che $\omega \in L$ può contenere.
- $Null(L)$ stabilisce se l'insieme contiene o meno la parola vuota.

– 4.3.1 – Calcolo ricorsivo di Ini, Fin, Dig, Null.

Sia L un linguaggio locale. La costruzione della quadrupla, che è ricorsiva, è descritta a seguire.

- **Ini:** considerando la parte ricorsiva
  - $Ini(e+f) = Ini(e) \cup Ini(f)$;
  - $Ini(ef) = Ini(e) \cup Null(e)Ini(f)$;
  - $Ini(e^*) = Ini(e)$.
- **Fin:** considerando la parte ricorsiva
  - $Fin(e+f) = Fin(e) \cup Fin(f)$;
  - $Fin(ef) = Fin(f) \cup Fin(e)Null(f)$;
  - $Fin(e^*) = Fin(e)$.
- **Null:** considerando la parte ricorsiva
  - $Null(e+f) = Null(e) \cup Null(f)$;
  - $Null(ef) = Null(e) \cap Null(f)$;
  - $Null(e^*) = \varepsilon$.
- **Dig:** considerando la parte ricorsiva
  - $Dig(e+f) = Dig(e) \cup Dig(f)$;
  - $Dig(ef) = Dig(e) \cup Dig(f) \cup Fin(e)Ini(f)$;
  - $Dig(e^*) = Dig(e) \cup Fin(e)Ini(e)$.

Nota: Nel descrivere il calcolo di $Ini, Fin, Dig, Null$, è stata trascurata la base. Infatti $*(\varepsilon) = \varepsilon, *(\emptyset) = \emptyset$, ove $*$ sostituisce $Ini, Fin, Dig, Null$, escluso $Dig(\varepsilon) = \emptyset$. Inoltre $Ini(a) = Fin(a) = a$ se a è un carattere, mentre $Dig(a) = Null(a) = \emptyset$.

– 4.3.2 – Automi locali.

Definizione: Sia L un linguaggio locale. Si definisce *automa locale* un DFA che riconosce L.
La costruzione dell'automa locale è realizzata secondo quanto segue.

- $Q = \{q_0\} \cup \Sigma$;
- Se $Null(L) = \varepsilon$ allora q_0 è accettante;
- $\forall a_i \in \Sigma$, ogni arco etichettato a_i, entra nello stato q_{a_i};
- Da q_0 escono le transizioni definite da Ini;
- Le altre transizioni sono definite da Dig;
- Gli stati finali sono indicati da Fin.

– 4.4 – RegEx e automi locali.

Definizione: Sia e una RegEx. Questa si dice lineare se nessun carattere in e è ripetuto.

Sia e una RegEx non lineare. Questa può essere linearizzata semplicemente ridefinendo le occorrenze multiple, così che l'espressione e' ottenuta dalla linearizzazione sia definito su un nuovo alfabeto Σ'.

Algoritmo .
Sia e espressione regolare. La costruzione di un'automa locale che riconosce e è realizzata come segue.

1. *Se e è regolare si al punto 2, altrimenti si procede alla linearizzazione.*
2. *Si definisce da quadrupla $(Ini(L), Fin(L), Dig(L), Null(L))$, procedendo ricorsivamente al calcolo degli insiemi.*
3. *Si costruisce l'automa locale seguendo le transizioni della quadrupla.*
4. *Se la quadrupla è definita dopo aver linearizzato e, allora si procede rimuovendo la ridefinizione dei caratteri.*
5. *Si procede alla subset construction.*

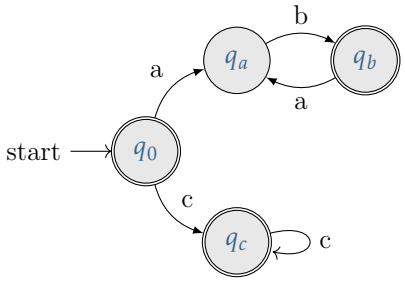
Esempio: Sia $e = (ab)^* + c^*$. Si costruisce un automa che riconosca e.

Svolgimento: Procedendo applicando l'algoritmo si ha quanto segue.

1. Si osserva che e è lineare, si passa dunque alla costruzione della quadrupla, da cui

   - $Ini(e) = \{a, c\}$
   - $Fin(e) = \{b, c\}$
   - $Dig(e) = \{ab, ba, cc\}$
   - $Null(e) = \varepsilon$

2. Procedendo alla costruzione dell'automa, segue



Osservazione: Poiché e è lineare si ha che non è necessario procedere alla subset construction. Infatti l'automa di cui sopra è gia un DFA.

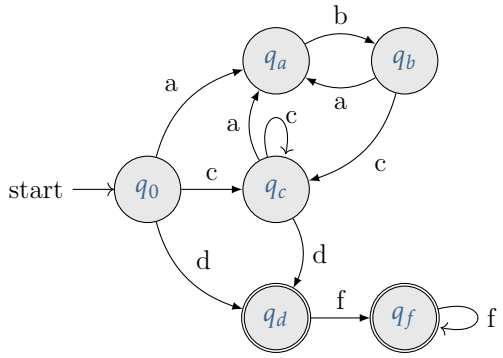
Esempio: Sia $e = (ab + a)^* ba^*$. Si costruisca un automa che riconosca e.

Svolgimento: Procedendo applicando l'algoritmo si ha quanto segue.

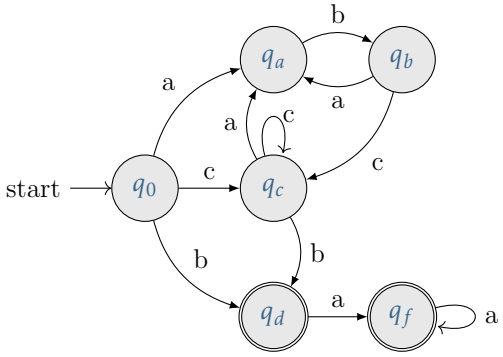
1. Si osserva che e non è lineare, si procede alla sua linearizzazione. Segue che
$$e = (ab + a)^* ba^* \quad \text{diventa} \quad e' = (ab + c)^* df^*$$

2. Considerando la quadrupla, segue
   - $Ini(e') = \{a, c, d\}$
   - $Fin(e') = \{d, f\}$
   - $Dig(e') = \{ab, bc, ca, ba, \ldots\}$
   - $Null(e') = \emptyset$

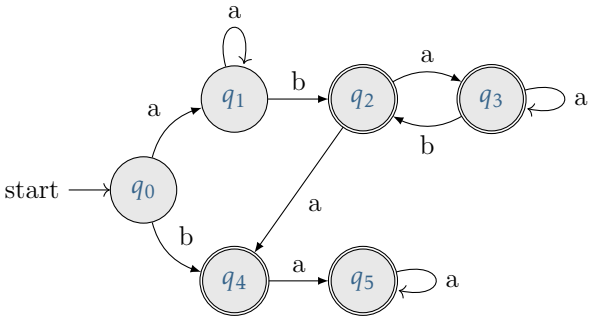
3. Passando all'automa, segue



4. Procedendo rimuovendo la ridefinizione dei caratteri, si ha



5. Concludendo con la subset construction, segue



– 4.5 – Da DFA a RegEx.

Si è finora dimostrato che per ogni RegEx esiste un automa che lo riconosce. Si può dimostrare il viceversa, se il seguente teorema è soddisfatto.

Teorema di Kleene.
Sia L un linguaggio regolare, sia A un DFA che lo riconosce. Allora esiste un'espressione regolare e equivalente.

Più in generale

$$REC = REG$$

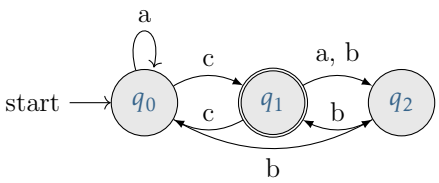
ove REG indica l'insieme dei linguaggi regolari.

Algoritmo .
Sia A un DFA che riconosce un certo linguaggio L. La costruzione della RegEx equivalente è realizzata come segue.

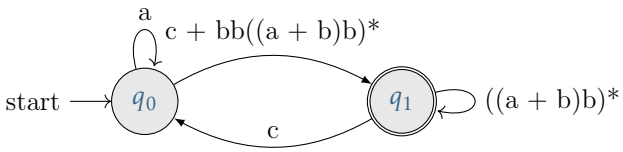
- *Se A ha più stati accettanti, si creano tante copie quante gli stati finali, ciascuno con un solo stato accettante.*
- *Per ciascuna delle copie:*
  - *si eliminano le transizioni intermedie, fino ad ottenere automi con un solo stato accettante e uno finale;*
  - *si determina la RegEx e_i per la copia.*
- *L'espressione regolare sarà data come*

$$e = e_1 + \cdots + e_k$$

Esempio: Sia A il seguente DFA. Si trovi la RegEx equivalente.



Svolgimento: Poiché q_2 è l'unico stato che non è ne finale ne iniziale, si procede alla sua eliminazione. Da cio segue quanto nella figura a seguito.



Da cui l'espressione equivalente è

$$e = a^*(c + bb((a+b)b)^*)(c(c + bb((a+b)b)^*) + ((a+b)b)^*)$$

– 5 – Proprietà dei linguaggi regolari.

Siano A , B due DFA. Si può dimostrare che

$$A = B \iff L(A) = L(B)$$

– 5.1 – Minimizzazione di DFA.

Sia A un DFA. A seguito di quanto detto sopra, ne consegue che è possibile realizzare un DFA B equivalente ad A, ma con un numero minimo di stati. Analogo ragionamento è estensibile agli NFA, sebbene per questi non è sempre vero.

– 5.1.1 – Relazione di indistinguibilità.

Sia A un DFA, siano p e q suoi stati. Si ha che

$$p \,|\, q \iff (\delta^*(q,\omega) \wedge \delta^*(q,\omega)) \in F, \quad \forall \omega \in \Sigma^*$$

oppure

$$p \,|\, q \iff (\delta^*(q,\omega) \wedge \delta^*(q,\omega)) \notin F, \quad \forall \omega \in \Sigma^*$$

Cioè p e q sono indistinguibili se per ogni parola del linguaggio universale si ha che, calcolando la funzione di transizione estesa per i due stati, entrambi conducono ad uno stato accettante/rifiutante per ω.

– 5.1.2 – Algoritmo riempi-tabella.

Uno strumento utile alla minimizzazione è l'algoritmo riempi-tabella, con il quale è possibile stabilire ricorsivamente gli stati equivalenti.

Base: Se p è accettante e q non lo è, allora la coppia (p,q) è distinguibile.

Induzione: Se p,q sono stati tali che, per un simbolo di input α, si ha che

$$\delta(p,\alpha) \wedge \delta(q,\alpha)$$

conducono a stati noti come distinguibili, allora (p,q) sono distinguibili.

Teorema 5.1.
Se due stati non sono distinti dall'algoritmo riempi-tabella, alloro sono equivalenti.

– 5.2 – Pumping Lemma.

Capita spesso di perdere molto tempo nello stabilire se un linguaggio L è regolare o meno. Per semplificare tale processo è possibile utilizzare uno strumento molto potente: il *pumping lemma*.

Lemma 5.1.
Sia L un linguaggio. Questi non è regolare se

$$\exists n : \forall \omega \in L, |\omega| \leq n, \exists x, y, z : \omega = xyz$$

per cui almeno una delle seguenti proprietà non è soddisfatta.

- $y \neq \varepsilon$
- $\left|xy\right| \leq n$
- $\forall k \geq 0, xy^k z \in L$

Esercizio: Sia $L = \{\omega : \omega = a^n b^n, n \geq 0\}$. Stabilire se L è regolare.

Svolgimento: Sia supposto L non regolare, segue

$$\forall n \quad a^n b^n = \underbrace{a \cdots a}_{n \text{ volte}} \underbrace{b \cdots b}_{n \text{ volte}} \in L$$

Procedendo col considerare alcune partizioni

1. Sia $x = a^i, y = a^j, z = b^n$ tale che $i + j = n$: si osserva che, posto $k = 0$, segue $a^i b^n \notin L$, poiché $i < n$.
2. Sia $x = a^i, y = a^j b^h, z = b^l$ tale che $i + j = n$, $h + l = n$: si osserva però che $\left|xy\right| = \left|a^{i+j} b^h\right| > n$.

Nota: Le partizioni non riportate sono state trascurate, in quanto ovvio non soddisfacenti almeno una delle proprietà.

Osservazione: Sia il pumping lemma per i linguaggi REC, sia quello per i linguaggi CF in *Sezione* (6.5), garantiscono esclusivamente la non appartenenza ad una data famiglia di linguaggi. Cioè, se un linguaggio soddisfa il pumping lemma, non è certo che questi sia regolare (o CF).

– 6 – Grammatiche Context-Free.

Definizione: Dato Σ un certo alfabeto, si definisce grammatica G la seguente quadrupla.

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

ove

- Σ è l'alfabeto di simboli terminali;
- V è l'alfabeto dei simbolo non terminali;
- S è un simbolo non terminale detto assioma;
- P è l'insieme delle regole di produzione.

Esempio: Sia $\Sigma = \{a, b\}$, sia $V = \{S\}$. Stabilire una grammatica che generi il linguaggio $a^n b^n$.

Svolgimento: Poiché $V = \{S\}$, sia S l'assioma, segue che le regole di produzione sono le seguenti.

$$\textcircled{1} \quad S \to ab$$
$$\textcircled{2} \quad S \to aSb$$

Considerando ad esempio la parola $a^3 b^3$, questa è ottenuta applicando due volte ② di produzione, seguite da ①.

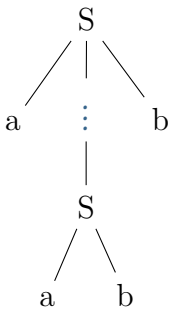
– 6.1 – Alberi sintattici.

Definizione: Sia G una grammatica CF. Si definisce albero sintattico di G una struttura ad albero che

- abbia ogni nodo etichettato da un carattere non terminale;
- abbia ogni foglia etichettata da un simbolo terminale.

Esempio: Sia G la grammatica per il linguaggio $a^n b^n$ di cui sopra. Se ne stabilisca l'albero sintattico.

Svolgimento: Ponendo il ripetersi della seconda regola di produzione con dei puntini, segue



– 6.2 – Proprietà delle CFG.

Sia G una grammatica CF. Allora questa è chiusa rispetto l'unione e la concatenazione, ma non per l'intersezione o il complemento.

– 6.2.1 – Unione.

Siano G_1 e G_2 due CFG. Siano L_1, L_2 rispettivamente i linguaggi generati da G_1, G_2. Sia $G = G_1 \cup G_2$, segue che questa generi $L = L_1 \cup L_2$. Infatti

$$S \to S_1 \mid S_2$$

ove S è assioma per G, S_1 e S_2 lo sono rispettivamente per G_1 e G_2.

– 6.2.2 – Concatenazione.

Siano L_1, L_2 linguaggi generati rispettivamente da $G_1 = (\Sigma_1, V_1, S_1, P_1), G_2 = (\Sigma_2, V_2, S_2, P_2)$. Sia $L = L_1 L_2$. Poiché $\omega \in L = uv : u \in L_1 \wedge v \in L_2$, segue che la grammatica che genera L sarà $G = (\Sigma_1 \Sigma_2, V_1 V_2, S_1 S_2, P_1 P_2)$

– 6.2.3 – Intersezione.

Si è detto che le CFG non sono chiuse per l'intersezione. Per dimostrare che sia effettivamente così basta considerare il seguente esempio.

Esempio: Sia $L = \{a^n b^n c^n : n \geq 0\}$, siano

$$L_1 = \left\{a^m b^m c^k : m, k \geq 0\right\}$$
$$L_2 = \left\{a^i b^l c^l : i, l \geq 0\right\}$$

stabilire se $L_1 \cap L_2 = L$ è generabile tramite una CFG.

Svolgimento: Siano posti S, T rispettivamente gli assiomi di produzione per L_1, L_2, con le regole di produzione di seguito riportate.

$$\begin{aligned} S &\to S_1 S_2 & \qquad T &\to T_1 T_2 \\ S_1 &\to a S_1 b \mid \varepsilon & \qquad T_1 &\to a T_1 \mid \varepsilon \\ S_2 &\to c S_2 \mid \varepsilon & \qquad T_2 &\to b T_2 c \mid \varepsilon \end{aligned}$$

Si osserva che L_1, L_2 sono generati da CFG, metre

$$L_1 \cap L_2 = L = \{a^n : n \geq 0\} \circ \{b^n : n \geq 0\} \circ \{c^n : n \geq 0\}$$

non lo è.

Nota: L'assenza di chiusura per il complemento è da ricondurre alla mancata chiusura per l'intersezione, segue da De Morgan, supponendo G_1, G_2 due CFG, che

$$(G_1 \cup G_2)^C = G_1^C \cap G_2^C$$

– 6.3 – Grammatiche unilaterali.

Sia G una CFG. Questa si dice unilaterale se le sue regole di produzione sono definite come segue

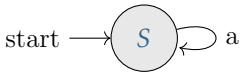
$$S \to \alpha B$$
$$S \to \alpha$$

con α simbolo terminale e B non terminale.

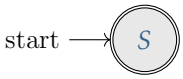
– 6.3.1 – Dalle CFG unilaterali destre agli automi.

Sia G una CFG unilaterale destra, sia S il suo assioma. Allora il passaggio ad automa è definito come segue

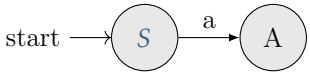
- Se $S \to aS$ allora



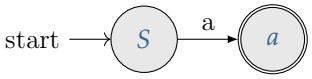
- Se $S \to \varepsilon$ allora



- Se $S \to aA$ allora



- Se $S \to a$ allora



– 6.4 – Forma normale di Chomsky.

Definizione: Sia G una grammatica CF. Questa dicasi in forma normale di Chomsky se, per ciascuno dei simboli non terminali, questi produce una coppia di non terminali o un non terminale.

Esempio: Sia G una CFG in forma normale di Chomsky, sia S il suo assioma, allora S sarà del tipo

$$S \to AB$$
$$A \to a$$
$$B \to b$$

– 6.5 – Pumping Lemma per le CFG.

Così come per i linguaggi REC, anche i linguaggi CF soddisfano il Pumping Lemma, seppure con alcune differenze.

Lemma 6.1.
Sia L un linguaggio. soddisfa il pumping lemma se

$$\exists n \in \mathbb{N} : \forall \omega \in L, |\omega| \geq n, \exists u, v, w, x, y : \omega = uvwxy$$

per cui tutte le seguenti proprietà sono soddisfatte.

- $|vwx| \leq n$
- $v, x \neq \varepsilon$
- $\forall i \geq 0, uv^i wx^i y \in L$

Esercizio: Sia $L = \{\omega : \omega = a^n b^n c^n, n \geq 0\}$. Stabilire se n è Context-Free.

Svolgimento: Sia supposto L non CF, segue

$$\forall n \quad a^n b^n c^n = \underbrace{a \cdots a}_{n \text{ volte}} \underbrace{b \cdots b}_{n \text{ volte}} \underbrace{c \cdots c}_{n \text{ volte}} \in L$$

Procedendo col considerare alcune partizioni

1. Siano $u = a^j, v = a^k, w = a^l, x = a^m, y = b^n c^n$ tali che $j + k + l + m = n$: si osserva che, posto $i = 0$, segue $uv^0 wx^0 y \notin L$, poiché $j + l < n$.
2. Siano $u = a^j, v = a^k, w = a^l, x = b^m, y = b^o c^n$ tali che $k+l+m \leq n, j+k+l = n$, $o + m = n$: si osserva, posto $i = 0$, che $uv^0 wx^0 y \notin L$, poiché $j + l < n, o < n$.

Nota: Le partizioni non riportate sono state trascurate, in quanto ovvio non soddisfacenti almeno una delle proprietà.

– 6.6 – Gerarchia di Chomsky.

La gerarchia di Chomsky stabilisce un ordine delle grammatiche, stabilito sulla base della loro capacità di produrre linguaggi. Tale classificazione divide le grammatiche in

- *Tipo 0*: le grammatiche riconosciute da una macchina di Turing[1].
- *Tipo 1*: grammatiche Context-Sensitive.
- *Tipo 2*: grammatiche Context-Free.
- *Tipo 3*: grammatiche "riconosciute" da DFA.

Se vista in altro modo, la gerarchia di Chomsky stabilisce inoltre il grado di decidibilità delle grammatiche. Si osservi la tabella sotto riportata.

	Membership Problem	Emptiness Problem	Finiteness Problem	Equivalence Problem
Tipo 0		NO		NO
Tipo 1		NO		NO
Tipo 2	SI	SI	SI	NO
Tipo 3	SI	SI	SI	SI

Considerando ciascuna colonna della tabella di cui sopra

- *Membership Problem*: riguarda la possibilità di stabilire se, considerata un certa grammatica G, una parola ω appartenga al linguaggio generato da G.
- *Emptiness Problem*: riguarda la possibilità di verificare se, considerata un certa grammatica G, il linguaggio generato da G è vuoto.
- *Finiteness Problem*: riguarda la possibilità di controllare se, considerata un certa grammatica G, il linguaggio generato da G è finito.
- *Equivalence Problem*: riguarda la possibilità di determinare se, considerate due grammatiche G_1, G_2, i linguaggi generati da G_1 e G_2 sono uguali.

– 6.6.1 – Ambiguità di una CFG.

Definizione: Sia G una CFG, sia $L(G)$ il linguaggio generato dalla CFG. Si dirà G ambigua se

$$\exists \omega \in L(G) : \exists 2 \text{ o più alberi sintattici di } \omega$$

Cioè G è ambigua se, il linguaggio da essa generato contiene almeno una parola che, attraverso le regole di produzione, può essere generata in più modi.

.

[1] Vedi *Sezione* (8)

– 7 – Automi a pila.

Definizione: Sia G una CFG, sia L il linguaggio da essa generato. Si definisce *automa a pila*, (o PDA), l'automa capace di riconoscere L.
Come gli altri automi questi è descrivibile tramite una tupla, che è la seguente

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

ove Q, Σ, q_0, F hanno la stessa funzionalità di quelle in un DFA, metre

- Γ è l'insieme dei simboli di pila;
- $Z_0 \in \Gamma$ è il simbolo di pila vuota;
- $\delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$: cioè preso $q \in Q, a \in \Sigma, w \in \Gamma$ si ha $\delta(q, a, w) \to (p, y)$ con $p \in Q, y \in \Gamma$.

Per analogia, un PDA è un NFA con una pila che ad ogni transizione

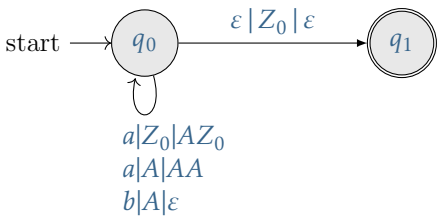
1. legge un simbolo in input;
2. cambia (o meno) stato;
3. rimpiazza (o meno) il top della pila.

Passando alla progettazione di un PDA, questa può essere realizzata in modo che la computazione accettata sia dovuta

- al passaggio in uno stato accettante;
- alla pila vuota.

Esempio: Sia $L = \{\omega : \omega = a^n b^n, n \geq 0\}$. Costruire il PDA che lo riconosce.

Svolgimento: Ponendo Z_0 simbolo di pila vuota, il PDA riconoscente L è il seguente.



– 7.1 – Dalle CFG ai PDA.

Sia G una CFG, questa può essere trasformata in PDA applicando il seguente algoritmo.

Algoritmo .

1. *L'alfabeto del PDA sarà l'alfabeto della CFG, cioè* $\Sigma_P = \Sigma_G$.
2. *L'alfabeto di pila è dato dall'unione dell'alfabeto della CFG, dei simboli non terminali e del simbolo di pila vuota, cioè* $\Gamma = \Sigma_G \cup V_G \cup \{Z_0\}$.
3. *Il PDA ha due soli stai:* q_0 *iniziale e* q_1 *finale. Se realizzato con pila vuota, in un certo senso,* $q_0 = q_1$.
4. *Allo stato* q_1 *si arriva solo per transizioni da* q_0*, con mosse del tipo*
$$\delta(q_0, \varepsilon, Z_0) \to (q_1, Z_0)$$
5. *Per ogni carattere* $x \in \Sigma_P$ *si definisce una transizione del tipo*
$$\delta(q_0, x, x) \to (q_0, \varepsilon)$$
6. *Le altre transizioni sono definite a partire dalle regole di produzione.*
   - *Se la regola è del tipo* $A \to BA_1 \cdots A_n, n \geq 0$*, si aggiunge una transizione del tipo* $\delta(q_0, \varepsilon, A) \to (q_0, A_n \cdots A_1 B)$.
   - *Se la regola è del tipo* $A \to bA_1 \cdots A_n, n \geq 0$*, si aggiunge una transizione del tipo* $\delta(q_0, \varepsilon, A) \to (q_0, A_n \cdots A_1)$.
   - *Se la regola è del tipo* $A \to \varepsilon$*, si aggiunge una transizione del tipo* $\delta(q_0, \varepsilon, A) \to (q_0, \varepsilon)$.

Esercizio: Sia $L = \left\{\omega : \omega = a^n b^{n+2}, n \geq 0\right\}$. Costruire il PDA che riconosce L.

Svolgimento: Si procede stabilendo per prima cosa la CFG che genera L. Posto S l'assioma, si ha

$$S \to Abb$$
$$A \to ab \,|\, aAb$$

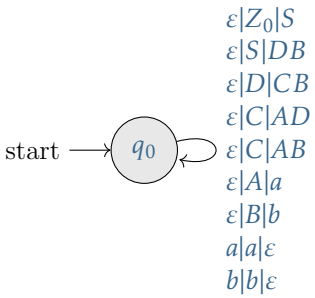
Normalizzando secondo Chomsky, segue

$$S \to DB$$
$$D \to CB$$
$$C \to AD \,|\, AB$$
$$B \to b$$
$$A \to a$$

Considerando adesso le transizioni del PDA, dall'algoritmo precedentemente introdotto segue

$$\delta(q_0, \varepsilon, Z_0) \to (q_0, S)$$
$$\delta(q_0, a, a) \to (q_0, \varepsilon)$$
$$\delta(q_0, b, b) \to (q_0, \varepsilon)$$
$$\delta(q_0, \varepsilon, S) \to (q_0, DB)$$
$$\delta(q_0, \varepsilon, D) \to (q_0, CB)$$
$$\delta(q_0, \varepsilon, C) \to (q_0, AD)$$
$$\delta(q_0, \varepsilon, C) \to (q_0, AB)$$
$$\delta(q_0, \varepsilon, B) \to (q_0, b)$$
$$\delta(q_0, \varepsilon, A) \to (q_0, a)$$

da cui in conclusione si ha il seguente automa, accettante per pila vuota.



– 8 – La macchina di Turing.

Definizione: Si definisce *macchina di Turing*, (o MT), un modello formale di macchina capace di eseguire algoritmi, composti da un numero di passi elementari di calcolo.

– 8.1 – Notazione per le MT.

Una MT può essere descritta in maniera formale dalla seguente set-tupla.

$$M = (Q, \Sigma, \Gamma, \delta, q_o, B, F)$$

ove Q, Σ, q_0, F hanno la stessa funzione di quelle di un DFA, mentre

- Γ è l'insieme di simboli di nastro;
- $B \in \Gamma \setminus \Sigma$ è il blank;
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ è la funzione di transizione.

Da un punto di vista grafico, un MT si compone di tre parti quali

- *controllo:* gestisce il comportamento della MT ad ogni stato;
- *testina:* meccanismo che permette di scorrere tra i vari stati;
- *nastro:* una sequenza infinita di celle.

– 8.2 – Istantanea di una MT.

Quando si parla di *istantanea di una MT*, (o ID), si deve pensare alla descrizione insiemistica dello stato della MT in un dato ciclo. Tale istantanea è descritta dalla seguente tupla

$$\alpha, q, \beta \in \Gamma^* Q \Gamma^*$$

ove

- q è lo stato attuale;
- $\alpha\beta$ rappresenta la stringa compresa tra il primo e l'ultimo carattere non blank.

– 8.3 – Tesi di Turing-Church e codifica binaria di una MT.

A differenza di un automa, le MT hanno la possibilità di non terminare mai la loro esecuzione. Da ciò nasce il *problema di arresto di una MT*.
Parlando ora dei linguaggi validi per una MT, si hanno

- *linguaggi riconosciuti:* linguaggi che per ogni parola appartenente o meno al linguaggio, portano ad un arresto;
- *linguaggi accettati:* linguaggi che portano ad un arresto solamente per parole interne al linguaggio.

– 8.3.1 – Tesi di Turing-Church.

La tesi di Turing-Church, che sebbene sia solo una tesi non è mai stata confutata, stabilisce: *una funzione è calcolabile se e solo se una MT la calcola.*

– 8.3.2 – Codifica binaria di una MT.

Sia M una macchina di Turing, siano $q_1,\dots,q_r$ suoi stati, sia q_1 stato iniziale e q_2 finale. Siano $X_1,\dots,X_s$ i simboli di nastro, siano $0 = X_1, 1 = X_2, B = X_3$ siano infine $L = D_1, R = D_2$. Considerando la funzione di transizione

$$\delta(q_i, X_j) = (q_k, X_l, D_m)$$

sia $0^i10^j10^k10^l10^m$ la sua codifica.
Poiché le transizioni sono in numero finito, posta C_t la t-esima transizione, segue che

$$C_1 11 C_2 11 \cdots C_{n-1} 11 C_n$$

sono tutte le transizioni, descrivendo difatti l'intera MT.

Esempio: Sia $M = (\{q_1, q_2, q_3\}, \{0,1\}, \{0,1,B\}, \delta, q_1, B, \{q_2\})$. Stabilire la codifica di M sapendo che le transizioni sono le seguenti.

$$\begin{aligned} T_1 &= \delta(q_1, 1) = (q_3, 0, R) \\ T_2 &= \delta(q_3, 0) = (q_1, 1, R) \\ T_3 &= \delta(q_3, 1) = (q_2, 0, R) \\ T_4 &= \delta(q_3, B) = (q_3, 1, L) \end{aligned}$$

Svolgimento: Siano $q_1 = 0^1, q_2 = 0^2, q_3 = 0^3$, siano $0 = 0^1, 1 = 0^2$ inoltre siano $B = 0^3, L = 0^1, R = 0^2$, separando ogni parte della transizione con un uno, e ogni transizione con due uno segue che

$$M = \underbrace{0100100010100}_{T_1} 11 \underbrace{000101010010011}_{T_2} \underbrace{000100100101001}_{T_3} 11 \underbrace{0001000100010010}_{T_4}$$

– 8.4 – Linguaggio diagonale e Linguaggio universale.

Definizione: Sia M_i una MT, sia ω_i la sua codifica. Si definisce linguaggio diagonale L_D, l'insieme delle coppie (M_i, ω_i) tali per cui M_i non accetta ω_i. Cioè

$$L_D = \{(M_i, \omega_i) : M_i \text{ non accetta } \omega_i\}$$

Teorema 8.1.
Il linguaggio diagonale L_D non è decidibile.

Dimostrazione: Sia M_i una MT che riconosce L_D, sia $\omega_i \in L_D$.
Segue che ω_i è accettata, ma pertanto $\omega_i \notin L_D \implies M_i$ non accetta ω_i.
In breve, si ha che ω_i è accetta da M_i solo se non lo è.

– 8.4.1 – Linguaggio universale e MT universale.

Definizione: Sia M una MT, sia ω una stringa accetta. Di definisce linguaggio universale L_U, l'insieme delle coppie (M, ω), tali che M accetta ω. Cioè

$$L_U = \{(M, \omega) : M \text{ accetta } \omega\}$$

Teorema 8.2.
Sia L un linguaggio ricorsivo, allora L^C è ricorsivo.

Teorema 8.3.
Siano L, L^C ricorsivamente enumerabili, allora L è ricorsivo.

Definizione: Si definisce MT universale, la MT U che riconosce L_U, capace di simulare ogni altra MT.

Nota: Se M accetta ω, questa si arresta e simultaneamente si arresta U.

Da quanto finora detto si può dimostrare che L_U non è ricorsivo. Sia, per assurdo, L_U ricorsivo, ne segue L_U^C ricorsivo.
Sia ora considerata la seguente MT

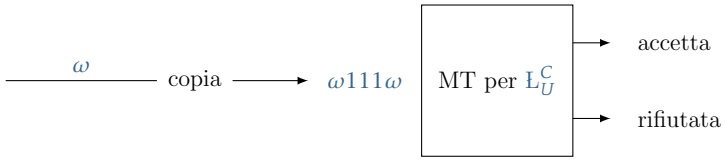


Figura 5: Ipotetica MT riconoscente L_U^C

Supponendo che L_U^C sia decidibile, si sta rendendo decidibile L_D che è assurdo.

– 9 – Teoria della complessità.

La *teoria della complessità* si occupa di studiare la complessità computazionale di un problema. Cioè, dato P un problema, posto che questi sia trattabile, quale funzione di complessità caratterizza l'algoritmo che risolve P?

Come anticipato sopra, esistono problemi *trattabili* e conseguentemente problemi *non trattabili*.

Definizione: Sia P un problema. Si dirà che P è trattabile se è possibile dimostrare che lo stesso è risolvibile da una MT deterministica. Si dirà P non trattabile altrimenti.

– 9.1 – Riduzione polinomiale.

Definizione: Sia P_1 un problema. Si dirà che P_1 è polinomialmente riducibile a un problema P_2 se

$$\exists f : \omega \in P_1 \iff f(\omega) \in P_2$$

con f calcolabile in tempo polinomiale.

– 9.2 – Problemi P e NP.

Definizione: Sia Q un problema. Dicasi Q problema di *classe* P se e solo se Q è risolvibile da una MT polinomialmente deterministica.

Definizione: Sia L un linguaggio. Si dirà L appartenente alla *classe* P se: esiste un polinomio $f(n)$ tale che L sia deciso da una MT polinomialmente deterministica in tempo $f(n)$.

Definizione: Sia L un linguaggio. Si dirà L appartenente alla *classe* NP se: esiste una MT polinomialmente non deterministica che lo riconosce in tempo polinomiale $f(n)$.

Osservazione: Poiché ogni MT deterministica è anche non deterministica, si ha

$$P \subseteq NP$$

Da cio ci si potrebbe chiedere $P = NP$? Ad oggi tale domanda rimane ancora senza una risposta certa.

– 9.2.1 – Problemi NP-completi.

Definizione: Sia P un problema. Dicasi P essere NP-completo se

- $P \in NP$;
- $\forall P^{'} \in NP, \exists f : \omega \in P^{'} \iff f(\omega) \in P$.

Teorema 9.1.
Sia $P_1 \in NP-completi$, sia $P_2 \in NP$. Se P_1 è polinomialmente riducibile a P_2, allora $P_2 \in NP-completi$.

Teorema 9.2.
Se $Q \in NP-completi \wedge Q \in P \implies P = NP$.

Nota: Dal *Teorema* (9.2) segue che, qualora ci si riuscisse, dimostrando che un problema Q_1 appartenente agli NP-completi è riducibile ad un problema Q_2 in P, si dimostrebbe dal *Teorema* (9.1) che ogni problema NP-completo è riducibile a Q_2, dimostrando in tal modo $P = NP$.