Elementi Di Informatica Teorica

Capitolo 1

Mio Padre:

https://diario.softml.it/category/laurea/calcolabilita-e-linguaggi-formali/

Definizioni: DFA e NFA

Possiamo definire un **DFA** come una quintupla $(Q,\Sigma, \delta, q_0, F)$ dove:

1. **Q** è l'insieme finito di stati
2. ***Σ*** è l'insieme finito chiamato alfabeto
3. ***δ:* Q x *Σ* -> Q** è la funzione di transizione
4. **q0**$\in$**Q** è lo stato iniziale
5. **F**$\subseteq$**Q** è l'insieme finito di stati accettanti

Possiamo definire un **NFA** come una quintupla $(Q,\Sigma, \delta, q_0, F)$ dove:

1. **Q** è l'insieme finito di stati
2. ***Σ*** è l'insieme finito chiamato alfabeto
3. ***δ:* Q x Σε ->P(Q)** è la funzione di transizione
4. **q0**$\in$**Q** è lo stato iniziale
5. **F**$\subseteq$**Q** è l'insieme finito di stati accettanti

Definizioni di un Esecuzione(Run)

- Una **esecuzione** (nota anche come **computazione**) di M su w = w1w2 · · · wn è una sequenza di transizioni q0
  w1 −→ q1 w2 −→ q2 w3 −→ · · · wn −−→ qn
- L'esecuzione è **accettante** se qn ∈ F.
- Se w ha una esecuzione accettante, diciamo che M **accetta** w.

- L'insieme L(M) = {w ∈ Σ: M accetta w} è il **linguaggio riconosciuto da M** (noto anche come **linguaggio di M**).

Differenza tra NFA e DFA

Un **NFA (automa a stati finiti non deterministico)** e un **DFA (automa a stati finiti deterministico)** sono entrambi automi a stati finiti, ma ci sono alcune differenze chiave tra loro:

- Un **NFA può avere più transizioni uscenti da uno stato per lo stesso simbolo dell'alfabeto**, mentre un DFA deve avere una e una sola transizione uscente da uno stato per ogni simbolo dell'alfabeto.

- un NFA può avere più di uno stato finale, mentre un DFA ha solo uno stato finale. Ciò significa che un **NFA può essere in più di uno stato finale alla fine dell'input** e considerare l'input come accettato, mentre un DFA deve essere in un solo stato finale per considerare l'input come accettato.

- Un **NFA può avere transizioni vuote (ε-transizioni)** , mentre un DFA non può.

- Un **NFA può essere utilizzato per riconoscere linguaggi non regolari**, mentre un DFA può essere utilizzato solo per riconoscere linguaggi regolari.
  Per quanto riguarda l'accettazione di linguaggi non regolari da parte degli NFA, un NFA è in grado di accettare linguaggi non regolari poiché può utilizzare transizioni ε e può avere più di uno stato finale, permettendo di seguire più di un cammino per l'input.

Equivalenza tra NFA e DFA

L'equivalenza tra un **DFA** (deterministic finite automaton) e un **NFA** (nondeterministic finite automaton) si basa sul fatto che entrambi sono in grado di riconoscere **gli stessi linguaggi regolari.** In altre parole, un DFA e un NFA sono equivalenti se entrambi riconoscono lo stesso linguaggio regolare.

Per dimostrare questa equivalenza, è possibile utilizzare una tecnica chiamata **"conversione"** che consiste nel trasformare un DFA in un NFA e viceversa. Questa conversione è possibile poiché entrambi i tipi di automi **seguono le stesse regole di transizione** e utilizzano gli stessi simboli dell'alfabeto di input.

TEOREMA:
Per ogni automa finito non deterministico esiste un automa finito deterministico equivalente

COROLLARIO:
Un linguaggio è regolare se e solo se qualche automa finito non deterministico lo riconosce.

Linguaggi regolari

TEOREMA:

Un linguaggio è chiamato linguaggio regolare se un automa a stati finiti lo riconosce

TEOREMI DA TENERE A MENTE:

Un linguaggio finito è sempre regolare (senza loop).

Un linguaggio con loop è non regolare se contraddice il pumping lemma.

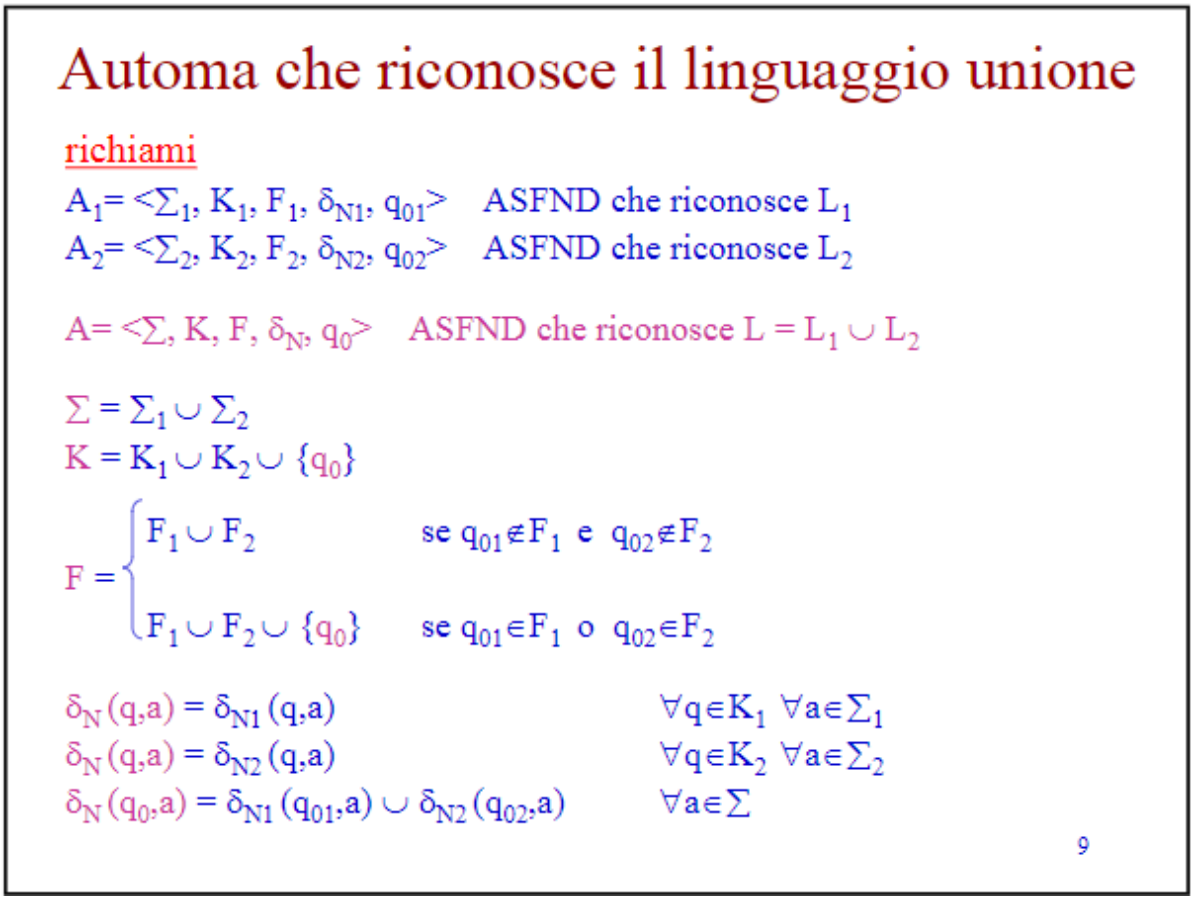
Un linguaggio con loop che rispetta il pumping lemma può essere o non essere regolare.

Chiusura Operazioni

Teorema: se L1 ed L2 sono due linguaggi regolari ⇒ anche i seguenti linguaggi sono regolari

- **Unione:**

L'unione tra due linguaggi regolari L1 e L2, denotata **L1 U L2**, è il linguaggio che contiene **tutte le stringhe** appartenenti a L1 o a L2 o a entrambi.



a

- **Concatenazione:**

La concatenazione tra due linguaggi regolari L1 e L2, denotata **L1•L2**, è il linguaggio che contiene tutte le stringhe ottenute **concatenando** una stringa del linguaggio L1 con una stringa del linguaggio L2.
Ovvero,si può creare un nuovo linguaggio composto da **tutte le possibili combinazioni di stringhe** tra L1 e L2 concatenando le stringhe una dopo l'altra.

Automa che riconosce il linguaggio concatenazione

richiami

$A_1 = <\Sigma_1, K_1, F_1, \delta_{N1}, q_{01}>$ ASFND che riconosce L_1

$A_2 = <\Sigma_2, K_2, F_2, \delta_{N2}, q_{02}>$ ASFND che riconosce L_2

$A = <\Sigma, K, F, \delta_N, q_0>$ ASFND che riconosce $L = L_1 \bullet L_2$

$\Sigma = \Sigma_1 \cup \Sigma_2$

$K = K_1 \cup K_2$

$$F = \begin{cases} F_2 & \text{se } \varepsilon \notin L_2 \\ F_1 \cup F_2 & \text{se } \varepsilon \in L_2 \end{cases}$$

$q_0 = q_{01}$

$\delta_N(q,a) = \delta_{N1}(q,a)$ $\quad \forall q \in K_1 - F_1 \quad \forall a \in \Sigma_1$

$\delta_N(q,a) = \delta_{N1}(q,a) \cup \delta_{N2}(q_{02},a)$ $\quad \forall q \in F_1 \quad \forall a \in \Sigma$

$\delta_N(q,a) = \delta_{N2}(q,a)$ $\quad \forall q \in K_2 \quad \forall a \in \Sigma_2$

11

- **Chiusura di Kleene:**

la chiusura di Kleene di un linguaggio regolare L, denotata **L***, è il linguaggio che contiene tutte le stringhe ottenute ripetendo zero o più volte una stringa del linguaggio L.

Automa che riconosce il linguaggio chiusura stella

richiami

$A = <\Sigma, K, F, \delta_N, q_0>$ ASFND che riconosce L

$A' = <\Sigma', K', F', \delta'_N, q'_0>$ ASFND che riconosce L*

$\Sigma' = \Sigma$

$K' = K \cup \{q'_0\}$

$F' = F \cup \{q'_0\}$

$\delta'_N(q,a) = \delta_N(q,a)$ $\forall q \in K-F$ e $\forall a \in \Sigma$

$\delta'_N(q,a) = \delta_N(q,a) \cup \delta_N(q_0,a)$ $\forall q \in F$ e $\forall a \in \Sigma$

$\delta'_N(q'_0,a) = \delta_N(q_0,a)$ $\forall a \in \Sigma$

nota: lo stato q'_0 è uno stato finale perché L* contiene sempre la stringa vuota

15

- **Complemento:**

Il complemento di un linguaggio regolare L, denotato **L =∑1-L1**, è il linguaggio che contiene tutte le stringhe che non appartengono al linguaggio L.
L è un linguaggio che contiene tutte le stringhe dell'alfabeto Σ che **non appartengono** al linguaggio L1.

Automa che riconosce il linguaggio complementare

richiami

$A = <\Sigma, K, F, \delta, q_0>$ ASF che riconosce L

$A' = <\Sigma', K', F', \delta', q'_0>$ ASF che riconosce $L' = \Sigma^* - L$

$\Sigma' = \Sigma$

$K' = K \cup \{d\}$ ('d' serve solo se c'è qualche $\delta(q,a)$ indefinito)

$F' = K - F$

$q'_0 = q_{01}$

$\delta'(q,a) = \delta(q,a)$ $\forall q \in K$ e $\forall a \in \Sigma$: $\delta(q,a)$ è definito

$\delta'(q,a) = d$ $\forall q \in K$ e $\forall a \in \Sigma$: $\delta(q,a)$ è indefinito

$\delta'(d,a) = d$ $\forall a \in \Sigma$

nota: si ricordi che dire che $\delta(q,a)$ è indefinito è come dire che esiste uno stato (pozzo) non finale q' tale che $\delta(q,a)=q'=\delta(q',x)$ $\forall x \in \Sigma$

13

- **Intersezione:**

L'intersezione tra due linguaggi regolari L1 e L2, denotata **L1 ∩ L2**, è il linguaggio che contiene tutte le stringhe appartenenti a **entrambi** i linguaggi L1 e L2.

- **Differenza:**

La differenza tra due linguaggi, nota come **L1 - L2,** è l'insieme delle stringhe che **appartengono al linguaggio L1 ma non appartengono al linguaggio L2**.

Automi che riconoscono intersezione e differenza

richiami

$A_1 = <\Sigma_1, K_1, F_1, \delta_1, q_{01}>$ ASF che riconosce L_1

$A_2 = <\Sigma_2, K_2, F_2, \delta_2, q_{02}>$ ASF che riconosce L_2

- ASFND che riconosce $L = L_1 \cap L_2$ (intersezione)

  $A = A_1 \cap A_2 = c\,(\,c(A_1) \cup c\,(A_2)\,)$

- ASFND che riconosce $L = L_1 - L_2$ (differenza)

  $A = A_1 - A_2 = c\,(c\,(A_1) \cup A_2)$

17

Espressioni regolari

In aritmetica possiamo usare le operazioni + e x per costruire espressioni come (5+3)*4

Analogamente possiamo utilizzare le operazioni regolari per costruire **espressioni regolari**, un esempio è: (0 U 1)0*. Il valore dell'espressione aritmetica è 32, il valore di un'espressione regolare è un linguaggio regolare. In questo caso il valore è un linguaggio che inizia con 0 o 1 seguito da un numero qualsiasi di simboli uguali a 0. Dove 0 e 1 sono abbreviazioni degli insiemi {0} e {1} e quindi ({0}} U {1}). Il valore in questa parte è il linguaggio {0,1}. Come in algebra per x, la concatenazione • è sottintesa.

Definiamo un'espressione regolare se ***R*** è:

1. *a* per qualche *a* nell'alfabeto Σ
2. ε

3. ∅
4. (R_1 U R_2) dove R_1 e R_2 sono espressioni regolari
5. (R_1 • R_2) dove R_1 e R_2 sono espressioni regolari, o
6. (R_1*) dove R_1 è un'espressione regolare

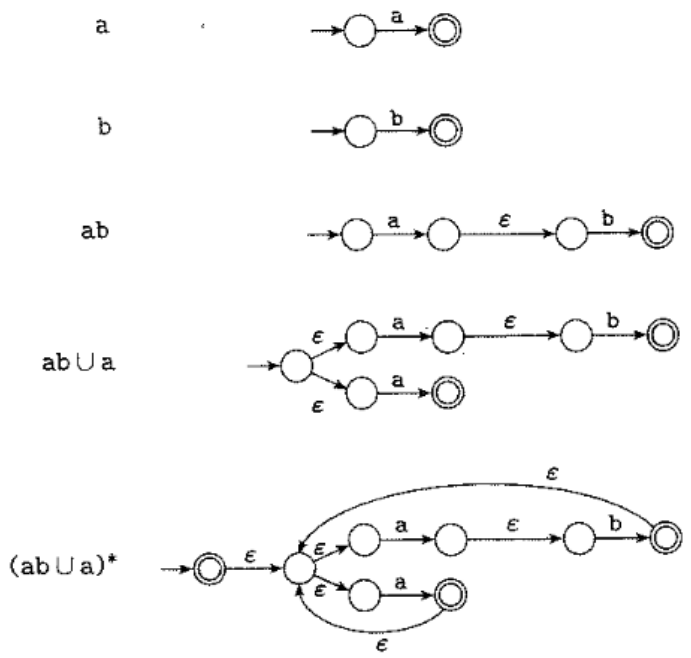
I punti 1, 2, 3 rappresentano rispettivamente i linguaggi {*a*} {ε} {∅}, mentre i punti 4, 5, 6 rappresentano i linguaggi ottenuti con le operazioni regolari di unione, concatenazione e star del linguaggio R_1. I linguaggi {ε} {∅} non sono uguali in quanto l'espressione ε rappresenta il linguaggio con una sola stringa- ovvero la stringa vuota - mentre ∅ rappresenta il linguaggio che non contiene nessuna stringa.

- Se un linguaggio è descritto da un'espressione regolare, allora esso è regolare

IDEA: supponiamo di avere un'espressione regolare R che descrive un linguaggio A. Mostriamo come trasformare R in un NFA che riconosce A.
DIMOSTRAZIONE: trasformiamo R in un NFA N, considerando i **sei casi della definizione di espressione regolare.**
Quindi un NFA per *a,* ε, ∅ e per gli ultimi tre casi utilizziamo la costruzione per dimostrare la chiusura rispetto alle operazioni regolari.

ESEMPIO 1.56

Trasformiamo l'espressione regolare $(\mathtt{ab} \cup \mathtt{a})^*$ in un NFA in una sequenza di passi. Costruiamo l'automa dalle sottoespressioni più piccole alle sottoespressioni più grandi finché abbiamo un NFA per l'espressione iniziale, come mostrato nel diagramma seguente. Nota che questa procedura generalmente non dà l'NFA con il minor numero di stati. In questo esempio, la procedura dà un NFA con otto stati, ma il più piccolo NFA equivalente ha solo due stati. Riesci a trovarlo?



- Se un linguaggio è regolare, allora è descritto da un'espressione regolare.

IDEA: dobbiamo mostrare che se un linguaggio A è regolare, allora un'espressione regolare lo descrive. Poichè A è regolare, esso è accettato da un DFA. Descriviamo una procedura per trasformare i DFA in espressioni regolari equivalenti. Per fare ciò utilizziamo un altro tipo di automa chiamato automa finito non deterministico generalizzato.

Un GNFA è tale se soddisfi tale descrizione:

➔ Lo stato iniziale ha archi di transizione uscenti verso un qualsiasi altro stato ma nessun arco entrante proveniente da un qualsiasi altro stato.

- ➔ Esiste un solo stato accettante, ed esso ha archi entranti provenienti da un qualsiasi altro stato ma nessun arco uscente verso un qualsiasi altro stato. Inoltre, lo stato accettante non è uguale allo stato iniziale
- ➔ Eccerto che per lo stati iniziale e lo stato accettante, un arco va da ogni stato ad ogni altro stato e anche da ogni stato in sè stesso

Possiamo facilmente trasformare un DFA in un GNFA aggiungendo un **nuovo stato iniziale** con un **ε-arco** nel vecchio stato iniziale ed un **nuovo stato accettante** con un **ε-archi** provenienti dai vecchi stati accettanti.

- ★ se ci sono più archi che collegano due stati nella stessa direzione, uniamo le etichette dei due stati

Ora mostriamo come trasformare un GNFA in un'espressione regolare.

Supponiamo che il GNFA abbia k stati. Allora dato che sappiamo che un GNFA ha almeno uno stato iniziale e uno accettante sappiamo che k>=2. Se k>2, costruiamo un GNFA equivalente con k-1 stati. Ripetiamo questo procedimento finchè non raggiungiamo k=2 stati, dove l'etichetta dell'arco dallo stato iniziale allo stato accettante è **l'espressione regolare equivalente**.

Equivalenza tra le Espressioni regolari e gli Automi

TEOREMA:

Un linguaggio è regolare se e solo se qualche espressione regolare lo descrive.

LEMMA:

Se un linguaggio è descritto da un'espressione regolare allora esso è regolare.

LEMMA:

Se un linguaggio è regolare, allora è descritto da un'espressione regolare.

Tramite questo teorema sappiamo che è possibile svolgere le **seguenti conversioni**:

Espressione Regolare ⇔ DFA
Espressione Regolare ⇔ NFA
Espressione Regolare ⇔ Linguaggio Regolare

GNFA(Generalized Nondeterministic Finite Automaton)

Possiamo definire un **GNFA** come una quintupla $(Q,\Sigma, \delta, q_{start}, q_{accept})$ dove:

1. **Q** è l'insieme finito di stati,
2. $\boldsymbol{\Sigma}$ è l'alfabeto di input,
3. $\boldsymbol{\delta: Q - \{q_{accept}\} \times (Q - \{q_{start}\}) \to R}$ è la funzione di transizione
4. $\boldsymbol{q_{start}}$ è lo stato iniziale
5. $\boldsymbol{q_{accept}}$ è lo stato accettante

La funzione di transizione **δ: Q -{qaccept} x (Q-{qstart})-> R,** significa che per ogni stato di partenza in Q, **escluso** lo stato finale qaccept e per ogni stato di arrivo in Q escluso lo stato iniziale qstart, esiste una relazione R tra lo stato di partenza e lo stato di arrivo.

Pumping Lemma per i linguaggi non Regolari

Se A è un linguaggio regolare, allora esiste un numero *p* (lunghezza del pumping) tale che, se *s* è una qualsiasi stringa in A di lunghezza almeno *p*, allora s può essere divisa in tre parti *s=xyz* che soddisfano le condizioni:

1. per ogni i>=0, $xy^iz \in A$

2. |y|>0 e
3. |xy|<=*p*

Capitolo 2

Linguaggi Context-Free

Un metodo potente per descrivere i linguaggi come $\{0^n1^n|n >= 0\}$ è quello delle ***grammatiche context-free.*** I linguaggi associati alle grammatiche context-free sono chiamati ***linguaggi context-free***. Un esempio di grammatica context-free, che chiamiamo G_1, è il seguente:

A -> 0A1

A -> B

B -> #

Una grammatica consiste di un insieme di regole di sostituzione, anche chiamate ***produzioni***. Ogni regola appare come una linea nella grammatica, costituita da un simbolo e una stringa separati da una freccia.

Il simbolo è chiamato ***variabile***, spesso è rappresentato da lettere maiuscole. Inoltre solitamente la variabile in alto a sinistra è la ***variabile iniziale***

La stringa consiste di variabili e simboli chiamati ***terminali***.

Per esempio in questa grammatica context-free:

-Le variabili sono: A, B

-I terminali sono: 0, 1, #

Una sequenza di sostituzioni per ottenere una stringa è chiamata ***derivazione***, per esempio possiamo ottenere la stringa 000#111 attraverso questa derivazione:

A -> 0A1 -> 00A11 -> 000A111 -> 000#111

La stessa informazione possiamo rappresentarla anche graficamente tramite un ***albero sintattico*.**

Definizione formale di Grammatica context-free

Possiamo definire una grammatica context-free come una quadrupla (*V, Σ, R, S*), dove

V è un insieme finito i cui elementi sono chiamati ***variabili***

Σ è un insieme finito, disgiunto da V, i cui elementi sono chiamati ***terminali***

R è un insieme finito di ***regole***, dove ciascuna regola è una variabile e una stringa di variabili e terminali
$S \in R$ è la variabile iniziale

Ambiguità

Qualche volta una grammatica può generare la stessa stringa in più modi diversi. Una tale stringa avrà diversi alberi sintattici e quindi diversi significati. Questo risultato può essere indesiderabile per alcune applicazioni, come i linguaggi di programmazione, dove un problema dovrebbe avere un'unica interpretazione. **Quando diciamo che una grammatica genera ambiguamente una stringa, intendiamo che la stringa ha due diversi alberi sintattici, non due differenti derivazioni.** Due derivazioni possono differire solo nell'ordine in cui esse sostituiscono le variabili. Infatti si parla di **derivazione sinistra** se a ogni passo la variabile sostituita è quella che si trova più a sinistra.

Forma normale di Chomsky

Quando si lavora con le grammatiche context-free è spesso conveniente averle in forma semplificata. Una delle forme semplici e utili è chiamata forma di chomsky

Automi a pila

Gli automi a pila sono come gli automi finiti non deterministici ma hanno una componente in più chiamata ***pila*** (stack).

Definiamo quindi gli automi a pila (PDA) è una settupla (Q, Σ, Γ, δ, q0, Z0, F) dove:

- **Q** è un insieme finito di stati
- **Σ** è l'alfabeto di input
- **Γ** è l'alfabeto della pila
- **δ** è la funzione di transizione **(Q x Σ x Γ -> Q x Γ*)**
- **q0** è lo stato iniziale
- **Z0** è il simbolo iniziale nella pila
- **F** è l'insieme degli stati finali.

Equivalenza con le grammatiche context-free

Un linguaggio è context-free se e solo se esiste un automa a pila che lo riconosce

Lemma 2.21
Se un linguaggio è context-free, allora esiste un automa a pila che lo riconosce.
IDEA: Sia A un CFL, allora esiste una CFG G che genera A. Mostriamo come trasformare G in un PDA equivalente, che chiamiamo P

Parsing

Il **parsing** per i CFL (Context-Free Language) è il processo di analisi di una stringa in un linguaggio context-free (ovvero generato da una grammatica context-free), al fine di determinare la sua struttura sintattica secondo la grammatica.

In pratica, il parsing si occupa di **verificare se una data stringa può essere generata dalla grammatica context-free associata** al linguaggio in questione e di costruire un albero di parsing che rappresenti la derivazione della stringa in questione.

Il parsing è un'operazione importante perché permette di verificare la correttezza sintattica di una stringa, ovvero se essa rispetta la grammatica associata al linguaggio. Inoltre, il parsing è un passo

fondamentale in molti processi di elaborazione del linguaggio naturale, come l'analisi sintattica di frasi e testi in linguaggio naturale.

Il parsing per le DCFL:

Il parsing per i DCFL (Deterministic Context-Free Language) è il processo di analisi di una stringa per verificare se appartiene al linguaggio generato da una grammatica context-free deterministica (DCFG).

A differenza dei CFG, **le DCFG** sono grammatiche in cui ogni produzione ha al **massimo un simbolo non-terminale** a sinistra e al massimo una stringa di simboli, terminale e non terminale, a destra. Inoltre, una DCFG deve essere **priva di ambiguità.**

L'analisi della stringa viene effettuata attraverso l'utilizzo di uno stack e di una tabella di parsing, che contiene le possibili transizioni tra gli stati dello stack e i simboli di input. Durante il parsing, gli elementi della stringa vengono caricati nello stack e il parsing procede in base alle regole di transizione presenti nella tabella.
Se il parsing termina con lo stack vuoto e la stringa interamente letta, allora la stringa appartiene al linguaggio generato dalla DCFG.

Il parsing per i DCFL può essere effettuato in tempo lineare rispetto alla lunghezza della stringa di input, rendendolo un metodo molto efficiente per verificare l'appartenenza di una stringa ad un linguaggio generato da una DCFG.

Conversione da CFG a PDA

Il modo pratico di convertire una grammatica context-free in un PDA è quello di seguire questi 5 step fondamentali:

1. **ε, ε→$** ovvero pushiamo lo stack symbol allo stato iniziale $\mathbf{Q_{start}}$

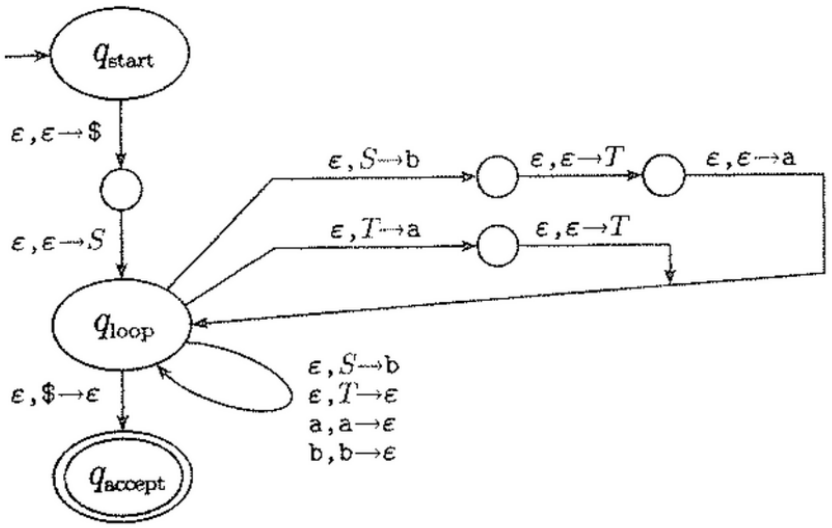
2. **ε, ε→S** pushiamo la variabile iniziale al top dello stack
3. Nello stato $\mathbf{Q_{loop}}$ pushiamo i simboli e le variabili partendo dal simbolo più a destra fino ad arrivare alla fine. Prima di tutto sostituiamo lo start symbol con il simbolo più a destra, e poi sostituiamo ε con il resto dei simboli partendo da destra verso sinistra appunto.
4. Poppiamo i terminali (ex. a, b) di tutte le produzioni
5. $\mathbf{Q_{accept}}$ poppa lo stack symbol, quindi **ε, $→ε**

Ricapitolando, gli stati fondamentali sono Q_{start}, Q_{loop}, Q_{accept}.
In Q_{start} semplicemente pushiamo il simbolo dello stack **$**
Nel prossimo stato pushiamo il simbolo della variabile iniziale.
In Q_{loop} noi pushiamo le produzioni partendo dal simbolo più a destra, da destra verso sinistra ricordando che sostituiamo la variabile da cui deriva solo con il simbolo più a destra, mentre il resto sostituiamo **ε** con gli altri simboli.
Successivamente poppiamo i terminali(sempre nel Q_{loop}).
E in Q_{accept} poppiamo il simbolo dello stack

Usiamo la procedura sviluppata nel Lemma 2.21 per costruire un PDA P_1 dalla seguente CFG G.

$$S \to \mathtt{a}T\mathtt{b} \mid \mathtt{b}$$
$$T \to T\mathtt{a} \mid \varepsilon$$

La funzione di transizione è mostrata nel diagramma seguente.



Pumping Lemma per i linguaggi non context-free

Se A è un linguaggio context-free, allora esiste un numero *p* (lunghezza del pumping) tale che, se *s* è una qualsiasi stringa in A di lunghezza almeno *p*, allora s può essere divisa in cinque parti *s=uvxyz* che soddisfano le condizioni:

1. per ogni i>=0, $uv^ixy^iz \in A$
2. |vy|>0 e
3. |vxy|<= *p*

Capitolo 3

Macchine di Turing Deterministica:

Una **Macchina di Turing (TM)** è un modello matematico di un calcolatore astratto, introdotto da Alan Turing nel 1936. Esistono due tipi di Macchina di Turing: deterministica e non-deterministica.
Una **Macchina di Turing Deterministica (DTM)** è una TM che, dato uno stato corrente e un simbolo sulla testina di lettura, esegue un'unica transizione di stato, legge un simbolo sul nastro e scrive un simbolo sul nastro.
La definizione formale di una **DTM è una 7-tupla (Q, Σ, Γ, δ, q0, qaccept, qreject)**, dove:

- **Q** è un insieme finito di stati,
- **Σ** è un alfabeto di input,
- **Γ** è un alfabeto di nastro che include Σ e un simbolo speciale "blank"
- **δ: Q x Γ → Q x Γ x {L, R}** è una funzione di transizione che descrive la prossima azione della TM in base allo stato corrente e al simbolo sulla testina di lettura
- **q0** ∈ **Q** è lo stato iniziale
- **qaccept** ∈ **Q** è lo stato di accettazione e
- **qreject** ∈ Q è lo stato di rifiuto, con **qreject ≠ qaccept**.

Macchine di Turing Non Deterministica:

Una **Macchina di Turing non deterministica (NTM)** è una TM che, dato uno stato corrente e un simbolo sulla testina di lettura, può eseguire più di una transizione di stato.
La definizione formale di una NTM è una **7-tupla (Q, Σ, Γ, δ, q0, qaccept, qreject)**, dove:

- **Q** è un insieme finito di stati,
- **Σ** è un alfabeto di input,
- **Γ** è un alfabeto di nastro che include Σ e un simbolo speciale "blank"
- **δ: Q x Γ → P(Q x Γ x {L, R})** è una funzione di transizione che descrive le possibili prossime azioni della TM in base allo stato corrente e al simbolo sulla testina di lettura, dove P(Q x Γ x {L, R}) è l'insieme delle parti del prodotto cartesiano Q x Γ x {L, R}.
- **q0** ∈ **Q** è lo stato iniziale
- **qaccept** ∈ **Q** è lo stato di accettazione e
- **qreject** ∈ Q è lo stato di rifiuto, con **qreject ≠ qaccept**.

La differenza principale tra una DTM e una NTM è che una **DTM segue sempre una transizione unica** in base allo stato corrente e al simbolo sulla testina di lettura, mentre una **NTM può seguire più di una transizione**. Ciò significa che una NTM può esplorare più possibilità contemporaneamente, ma questo rende il processo di esecuzione più complesso e meno prevedibile rispetto a una DTM.

Configurazione Macchina di Turing:

Una **configurazione** di una MdT è data dalle seguenti informazioni:

1. Stato corrente;
2. Contenuto del nastro (i simboli appartenenti a Σ)
3. La posizione della testina.

Una configurazione può essere rappresentata come una tupla **(u,q,v)** o con la stringa **uqv** dove:

- q è stato corrente;
- u ϵ Σ* è la stringa di simboli di Σ che si trova a sinistra della testina;
- v ϵ Σ* è la stringa di simboli di Σ che si trova a destra della testina;

Per esempio, $11q_7011$ rappresenta la configurazione dove il nastro è 11011, lo stato corrente è q_7 e la testina è sullo zero.

Una configurazione u accept v è detta di **accettazione**; u reject v è invece di **rifiuto**

La relazione di esecuzione tra configurazioni è (→ si legge “porta a”) :

–$C_1 \rightarrow C_2$: La MdT esegue un passo da C1 a C2;

–$C_1 \rightarrow^* C_2$: chiusura transitiva - La MdT esegue in 0,1,2 o più passi uno spostamento da C_1 a C_2;

Macchina di Turing Multinastro:

Una **macchina di Turing multinastro** è paragonabile ad un'ordinaria Macchina di Turing a singolo nastro ma con più nastri. Ogni nastro ha la sua **testina** per leggere e scrivere.

Inizialmente l’input risiede sul I° nastro e gli altri sono inizializzati con caratteri di blank.

La funzione di transizione cambia per permettere di leggere,scrivere e muovere le testine su alcuni o tutti i nastri simultaneamente. Formalmente è descritta in questo modo:

$$\delta : Q\backslash\{accept,reject\} \times \Gamma^k \rightarrow Q \times \Sigma^k \times \{L,R,S\}^k$$

dove **k** è il numero di nastri;

L'espressione $\delta(q_i,a_1,\ldots, a_k)=(q_j,b_1,\ldots, b_k,L,R,\ldots,L)$ sta a significare che:

> "Se la macchina è nello stato q_i e le testine da 1 a k stanno leggendo i simboli a_1 … a_k, la macchina va nello stato q_j, scrive i simboli b_1 … b_k, e direziona ogni testina per muoversi a sinistra o destra, o rimanere ferma, in base a quanto specificato dalla relazione di transizione."

Equivalenza tra TM multinastro e TM nastro singolo:

*"Per ogni **Macchina di Turing multinastro** esiste una **macchina di Turing a singolo nastro** equivalente."*

Due Macchine di Turing sono equivalenti se riconoscono lo stesso linguaggio.

Conversione:

Conversione da una **TM multinastro**, chiamata M, ad una **TM a nastro singolo**, chiamata S.

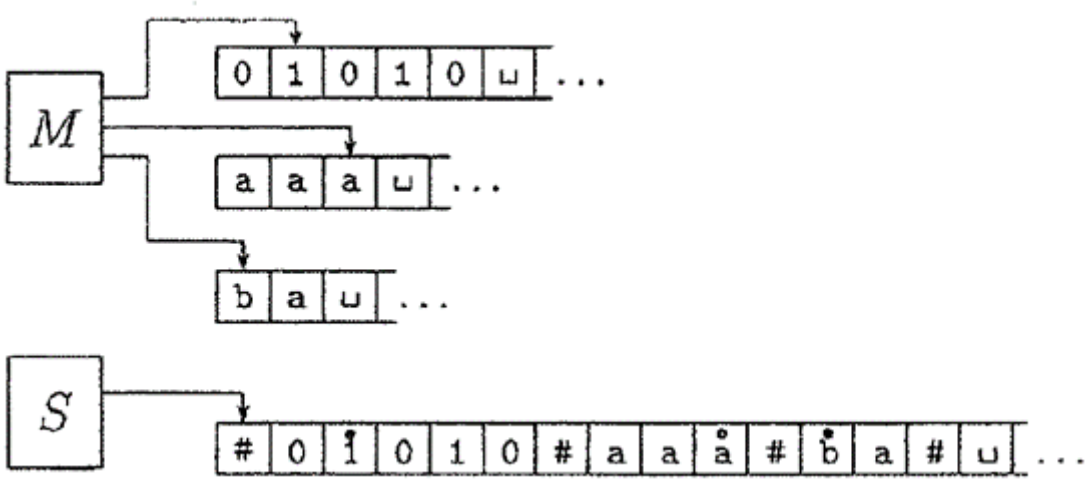
Assumiamo che **M abbia k nastri.**

Allora **S simula l'effetto dei k nastri di M** memorizzando il loro contenuto sul suo unico nastro.

S utilizza il **simbolo # come delimitatore** per separare i contenuti dei diversi nastri.

Inoltre, **S deve tener traccia delle posizioni** delle varie testine. Per fare questo, per ogni simbolo nella posizione di una testina dei k nastri, **S scrive lo stesso simbolo con l'aggiunta di un punto sopra.**

Questo nuovo simbolo sarà poi aggiunto all'alfabeto del nastro di S.



S= Su input $w=w_1 \ldots w_n$:

$$\#\dot{w}_1 w_2 \cdots w_n \#\dot{\sqcup}\#\dot{\sqcup}\# \cdots \#$$

1. **S** in prima istanza **copia il contenuto dei k nastri** della TM M sul suo nastro:

2. Per simulare un **singolo movimento**, S scandisce il nastro dal **primo simbolo #** il quale indica il limite sinistro, fino al (k+1)-esimo #, il quale indica il **limite destro**, per individuare il simbolo che rappresenta la testina virtuale. Come secondo passo S **aggiorna i nastri**, in base a come è definita la funzione di transizione di M.

3. Se in qualche punto **S muove una delle testine virtuali a destra di un simbolo #**, questa azione indica che M ha mosso la testina corrispondente in una porzione del nastro dov'è presente un simbolo di **blank**. Così **S scrive un simbolo blank** su questa cella del nastro e **trasla di un unità** i contenuti del nastro, da questa cella fino al simbolo # più a destra. La simulazione continua poi **ciclicamente**.

Corollario:

*“Un linguaggio è **Turing-Riconoscibile** se e solo se esiste almeno una macchina di Turing Multinastro che lo riconosce.”*

Macchina di Turing come Trasduttore:

Una macchina di Turing che calcola una funzione (anche parziale), può essere vista come un Trasduttore.

A partire da un input sul nastro, se la macchina ad un certo punto si ferma, l’output è la stringa presente sul nastro in quel momento.

Questo in macchina di Turing Multinasto è ancora più evidente:

- Un nastro per l’input
- Un nastro per le configurazioni
- Un nastro per l’output

Linguaggi Palindromi per le TM:

Un linguaggio palindromo è un linguaggio che letto in qualsiasi verso resta identico.

Macchina di Turing per l’accettazione del linguaggio di palindrome:

Muovi la testina avanti e indietro in modo che:

- verifica il primo simbolo con l’ultimo simbolo e marcali, ad esempio sostituendoli il primo con ▷ e l’ultimo con Ц ;
- verifica il secondo simbolo con il penultimo, e cosi via…

	▷	0	1	␣
s	s,▷,R	q0,▷,R	q1,▷,R	yes,␣,S
q0	q0,▷,R	q0,0,R	q0,1,R	p0,␣,L
q1	q1,▷,R	q1,0,R	q1,1,R	p1,␣,L
p0	yes,▷,S	r,␣,L	no,1,S	r,␣,S
p1	yes,▷,S	no,0,S	r,␣,L	r,␣,S
r	s,▷,R	r,0,L	r,1,L	r,␣,S

Enumeratori:

“Un linguaggio è decidibile se esiste una macchina di turing non deterministica che lo decide.”

Il termine linguaggio turing riconoscibile deriva da una variante di macchina di turing chiamato **Enumeratore**. Definito in modo informale un enumeratore è una macchina di turing con una **stampante** collegata.

La macchina di turing può utilizzare la stampante come dispositivo di **output per stampare le stringhe.** Ogni volta che la macchina di turing aggiunge una stringa alla lista, la invia alla stampante.

Un **enumeratore E** inizia con un nastro di **input vuoto**.

Se l’enumeratore non si ferma può stampare un elenco infinito di stringhe. Il linguaggio enumerato da E, è la collezione di tutte le stringhe che esso stampa.

Inoltre E potrebbe generare le stringhe del linguaggio in qualsiasi ordine, anche con ripetizioni.

I linguaggi Turing-Riconoscibili sono chiamati anche **Ricorsivi enumerabili** proprio da questa variabile della Macchina di Turing.

Teorema:

“Un linguaggio è turing-riconoscibile se e solo esiste un enumeratore che lo enumera.”

Macchina di Turing Universale:

Perché studiamo le macchine di turing:

L'interesse nella macchina di Turing (TM) per gli informatici risiede soprattutto nel fatto che essa rappresenta un **modello di calcolo algoritmico**, di un tipo di calcolo cioè che è automatizzabile in quanto eseguibile da un dispositivo meccanico.

Ogni TM è il **modello astratto di un calcolatore** - astratto in quanto prescinde da alcuni vincoli di limitatezza cui i calcolatori reali devono sottostare;

Per esempio, **la memoria** di una TM (vale a dire il suo nastro) è potenzialmente estendibile **all'infinito** (anche se, in ogni fase del calcolo, una TM può sempre utilizzarne solo una porzione finita), mentre un calcolatore reale ha sempre **limiti ben definiti di memoria.**

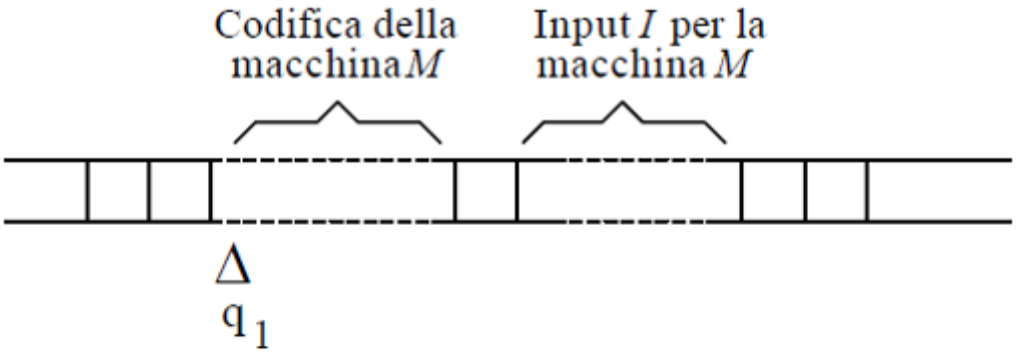
Dunque, una TM M che accetta un linguaggio è **analoga a un programma che implementa un algoritmo.**

Come è strutturata la macchina di Turing universale:

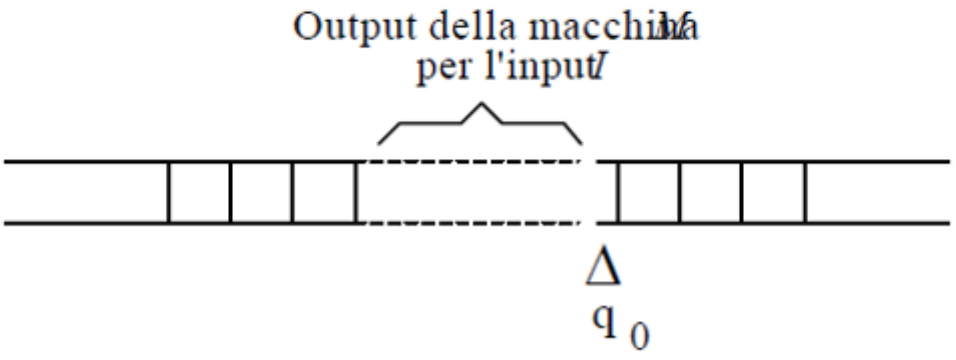
E' possibile definire una TM, detta **Macchina di Turing Universale (MTU)**, in grado di simulare il comportamento di ogni altra TM.
Si può dimostrare che esiste un TM (la MTU appunto) che, preso in input un opportuno codice effettivo delle componenti di un'altra macchina, ne simula il comportamento.
Più formalmente, la **MTU U** è una macchina il cui input è composto dalla concatenazione di due elementi:

1. la **codifica della tavola di transizioni** di una TM M;
2. un input I per M.

Per ogni M e per ogni I, la MTU **"decodifica"** le tuple che definiscono M, e le applica ad I, ottenendo lo stesso output che M avrebbe ottenuto a partire da I.
Formalmente si dice che **U(M;I)=M(I)**



Inizio della simulazione di M



Fine della simulazione di M

Siccome U deve poter simulare qualsiasi TM M, **non può essere considerato un limite superiore "a priori"** per il numero di stati e di simboli di M che U deve considerare.

Per questo motivo si assume che stati e simboli di M sono numeri interi.

In particolare, **si assume che:**

- **L'alfabeto** Σ di M sia {1, 2, … , |Σ|},
- **L'insieme di stati K** sia {|Σ|+1, |Σ|+2, … , |Σ|+|K|},
- **Lo stato iniziale** s=|Σ|+1,
- Gli stati "**accept", "reject"** sono codificati con |Σ|+2 e |Σ|+3

- I numeri |Σ|+|K|+1 e |Σ|+|K|+2 codificano gli spostamenti **“left”** e **“right”**
- La funzione di transizione è ottenuta in modo ovvio, rappresentano le regole come tuple di numeri.
- Tutti i numeri saranno **codificati come numeri binari** di lunghezza ⌈log(|K| + |Σ|)⌉

Svolgimento:

La codifica della TM M in input per U comincerà con il numero |K| e poi |Σ| entrambi in binario e separati da virgole.

Segue poi una **descrizione di δ** in termini di quintuple ((q,a),(p,b,d)), con d in {left, right}.

Poi segue un **“;”** che ha il compito di **segnalare la fine della descrizione di M**.

Ancora, **si inserisce la codifica in binario della parola input** $x = x_1, ..., x_k$, con la virgola usata come separatore degli interi binari che codificano i singoli simboli.

Gli **oggetti aggiuntivi** (parentesi, virgola, punto e virgola, ecc.) possono anche essere codificati con altri interi successivi a quelli utilizzati.

Nota: Ogni codifica “algoritmica” effettiva va bene.
Nota: La rappresentazione di M e la rappresentazione del suo input possono anche essere messi su **due nastri differenti**, vista l’equivalenza (polinomiale) tra una macchina di Turing a più nastri e una ad un solo nastro.

La MTU sull’input $<M,x_1 ... x_k>$ ha due nastri:
- Il primo contiene l’input $<M,x_1 ... x_k>$
- Il secondo **contiene la (codifica della) configurazione corrente di M,** nella forma (q,u,v) dove **uv** è il contenuto del nastro di M ad

un certo punto della sua computazione, **q** è lo stato in cui si trova M e il simbolo in lettura è il primo di v.

I primi passi della MTU **servono per scrivere sul secondo nastro la codifica della configurazione iniziale**, $(s,x_1...x_k)$.

Simulazione di un passo di M:

- **U esamina il secondo nastro** fino a trovare la codifica binaria dello stato corrente q (ricordiamo che è un numero tra $\{|\Sigma|+1, |\Sigma|+2, \dots , |\Sigma|+|K|\}$)
- Cerca sul primo nastro una **regola per q**
- Poi muove la testina del secondo nastro per individuare **il simbolo in lettura** per M e controlla se la regola in lettura sul primo nastro coinvolge lo stesso simbolo input; **se sì la regola viene implementata** (cambiando la configurazione sul secondo nastro in corrispondenza) altrimenti si controlla la regola successiva
- Quando **M** si ferma, anche **U** si ferma.

Architettura di Von Neumann:

Il modo in cui la Macchina di Turing universale codifica le altre macchine di turing e le interpreta, è analogo a come le moderne cpu utilizzano i programmi.

La **MTU tratta i programmi** (cioè la codifica delle tuple della TM da simulare) e i dati (l'input della TM da simulare) in maniera sostanzialmente **analoga**: essi vengono memorizzati sullo stesso supporto (il nastro), rappresentati utilizzando lo stesso **alfabeto di simboli ed elaborati in modo simile**.

Queste caratteristiche sono **condivise** dagli attuali calcolatori, che presentano la struttura nota come ***architettura di von Neumann*** (dal nome dello scienziato di origine ungherese John von Neumann che la ideò).

La **caratteristica più importante** della macchina di von Neumann è costituita dal fatto che sia dati che programmi vengono trattati in modo **sostanzialmente omogeneo**, ed immagazzinati nella stessa unità di memoria.

Ad esempio, poiché dati e programmi sono oggetti di natura omogenea, è possibile costruire programmi che **prendano in input altri programmi** e li elaborino, e che producano programmi in output.

Queste possibilità sono ampiamente sfruttate negli attuali calcolatori digitali, e da esse deriva gran parte della loro potenza e della loro facilità d'uso (ad esempio, **un compilatore o un sistema operativo** sono essenzialmente programmi che operano su altri programmi).

Anche la **potenza computazionale** tra il calcolatore di von Neumann e la macchina di Turing è la stessa:

se si suppone che il calcolatore di von Neumann è dotato di memoria e tempi di calcolo illimitati, esso è in grado di calcolare tutte le funzioni computabili **secondo la Tesi di Church** (per questo si dice che una macchina di von Neumann è un *calcolatore universale*).

La MTU costituisce quindi un **modello astratto** degli attuali calcolatori digitali (elaborato prima della loro realizzazione fisica).

Algoritmi e Tesi Church-Turing:

Un problema (o una funzione) è calcolabile (o decidibile) in modo algoritmico (o calcolabile in modo effettivo, o effettivamente calcolabile) se esiste un algoritmo che consente di calcolarne i valori per tutti gli argomenti.

“La classe delle funzioni calcolate da una MdT è la classe delle funzioni che informalmente sono considerate effettivamente calcolabili, o equivalentemente la classe dei linguaggi decisi da una MdT corrisponde alla classe dei problemi intuitivamente effettivamente risolvibili.”

Capitolo 4

Cos’è la decidibilità?

In teoria della computazione, la decidibilità è un concetto che descrive la capacità di un algoritmo o di una macchina **di calcolare la risposta a una domanda** o di determinare se una determinata proprietà è vera o falsa.

In generale, un problema è detto decidibile se esiste un algoritmo in grado di determinare la soluzione in un **numero finito di passi**, indipendentemente dalle dimensioni del problema. In altre parole, se **esiste un algoritmo che può rispondere "sì" o "no" a una domanda in modo corretto in un tempo finito**, allora il problema è decidibile.

Al contrario, un problema è detto indecidibile se non esiste alcun algoritmo in grado di determinare la soluzione in un tempo finito. In altre parole, se **non esiste alcun algoritmo in grado di rispondere "sì" o "no"** a una domanda in modo corretto in un tempo finito, allora il problema è indecidibile.

Definizione formale decisore:

M decide un linguaggio **L** $\subseteq$ **Σ*** se **L = L(M)** e per ogni **x** in **Σ* - L**, **M** termina in uno stato *“reject”*.

Differenze tra Turing-Riconoscibile e Decidibile

In una **TM riconoscibile** i possibili esiti sono solo: accetta; rifiuta/loop.

Un **decisore** è una macchina di Turing che termina sicuramente in uno stato accept or reject e **non ha mai loop infiniti**.

Perché un **decisore nondeterministico** accetti un linguaggio, bisogna che almeno uno dei **branch termini con uno stato accettato**; viceversa perché sia rifiutato bisogna **che tutti i branch finiscano in uno stato rifiutato.**

(Per Branch si intendono i rami dell'albero di una TM NonDeterministica.)

Definizioni:

*"Un linguaggio è detto **Turing-riconoscibile** (o **ricorsivo-enumerabile**) se esiste una macchina di Turing che lo riconosce"*

*"Un linguaggio è detto **Turing-decidibile** (detto anche **decidibile** o **recursive**) se esiste una macchina di Turing che lo decide."*

*"Un linguaggio è **co-Turing-riconoscibile** (co-Turing-recognizable) se e solo se il suo complemento è **Turing-riconoscibile** (Turing-recognizable)."*

*"Un linguaggio è decidibile se e solo se è **Turing-riconoscibile** e **co-Turing-riconoscibile**."*

Decidibilità, Teoremi

A_{DFA} è decidibile

L'idea è quella di creare una macchina di Turing che dati (B, w), B automa e w stringa, simuli il DFA. Se la simulazione finisce in uno stato di accettazione accetta, altrimenti rifiuta

A_{NFA} è decidibile

Si potrebbe procedere come per la DFA e creare una TM che decide simuli il NFA ma c'è un'altra strada. Ovvero convertire il NFA in un DFA e procedere come per il punto precedente

A_{REX} è decidibile

Anche qui, si converte l'espressione regolare in un NFA e si procede come per il punto precedente

E_{DFA} è decidibile

Vogliamo verificare se un DFA può raggiungere uno stato accettante dal suo stato iniziale percorrendo le frecce del DFA. Per fare ciò costruiamo una TM M che utilizza un algoritmo di marcatura. Ovvero su input <A> marca lo stato iniziale S e a sua volta marca ogni stato che ha una transizione con uno stato già marcato. Accetta quando nessun stato accettante è marcato, rifiuta quando marcato

Indecidibilità, Teoremi

A_{TM} è indecidibile

A_{TM} prende (M, w), M è una TM e w una stringa.

L'idea è di avere un decisore H(<M, w>) per A_{TM} che accetta se M accetta e rifiuta se M rifiuta. Successivamente creiamo una TM D che chiama H per verificare cosa fa M quando il suo input è la stessa descrizione <M>.

La tm D fa esattamente il contrario di M. Accetta quando M rifiuta e rifiuta quando M accetta. Ora chiediamoci cosa succedesse se usassimo di su D medesima? Indipendentemente da cosa fa D, essa è costretta a fare il contrario, il che è una contraddizione. Quindi è facile intuire che non possono esistere nè la tm H e nè la tm D.

Capitolo 5

$HALT_{TM}$ è indecidibile

Questo problema è noto come **problema della fermata**. Usiamo l'indecibilità di A_{TM} per dimostrare che $HALT_{TM}$ è a sua volta indecidibile, riducendo A_{TM} ad $HALT_{TM}$.

Assumiamo che $HALT_{TM}$ sia decidibile. $HALT_{TM}$ prende in input (M, w), dove M è una TM e M si ferma su w. L'idea è quella di prendere un decisore R per $HALT_{tm}$ e un decisore S per A_{tm}. Se utilizzassimo S vedremo che, accetta quando M accetta e rifiuta quando M rifiuta o entra in un ciclo. Vediamo che non possiamo utilizzare S perchè essendo un decisore non è concesso andare in loop, la simulazione non sarà in grado di terminare. Utilizzando invece R saremo in grado di sapere se M si ferma su w. Accetta quando M si ferma e rifiuta quando M non si ferma, in quanto <M, w> non è in A_{TM}. Se R esistesse potremmo decidere A_{TM}, ma questo va in contraddizione in quanto A_{TM} è indecidibile, pertanto $HALT_{TM}$ è indecidibile.

E_{TM} è indecidibile

Vogliamo dimostrare che E_{TM} è indecidibile.

L'idea è quella di partire assumendo che E_{TM} sia indecidibile e mostriamo che A_{TM} è decidibile, una contraddizione.

Sia R che decide E_{TM}, usiamo R per costruire la TM S che decide A_{TM}. S esegue R su input <M> e vede se R accetta. Se accetta L(M) è vuoto e M non accetta w. Ma se R rifiuta sappiamo che L(M) non è vuoto e M accetta qualche stringa ma non sappiamo se sia w.

Quindi eseguiamo R su una modifica di <M>. Quindi M_1 rifiuta se la stringa è diversa da w e accetta se è uguale a w. Quindi utilizziamo R per vedere se la macchina riconosce il linguaggio vuoto.

Se R accetta, rifiuta, se R rifiuta, accetta. poichè se R accetta vuol dire che M accetta la stringa w, e quindi rifiuta perchè il linguaggio non è vuoto. Se R rifiuta vuol dire che il linguaggio è vuoto e quindi accetta. Ma se R fosse un decisore per E_{TM}, S sarebbe un decisore per $A_{TM.}$. Ma sappiamo che non esistono decisori per A_{TM}, quindi $E_{T;}$ è indecidibile

Capitolo 6

Cose fuori dal libro

Linear Bounded Automaton (LBA)

Le **Linear Bounded Automaton (LBA)** sono un tipo di automi a nastro limitato. Sono simili alle macchine di Turing standard, ma hanno un nastro di input limitato in lunghezza. In altre parole, hanno una **dimensione finita di memoria**.

Una LBA è composta da un insieme di stati, un alfabeto di simboli, una funzione di transizione, un **simbolo speciale di "fine nastro"**, uno stato iniziale e un insieme di stati finali.

A differenza delle macchine di Turing standard, una LBA **non può spostare la testina di lettura oltre i bordi del nastro**, quindi non può accedere a una quantità illimitata di memoria.

Le LBA sono utilizzate per modellare problemi computazionali a memoria limitata, come ad esempio, **problemi di decidibilità** in teoria della complessità computazionale. Inoltre, le LBA sono utilizzate per dimostrare la non regolarità di alcuni linguaggi.

Le LBA sono l'**intermezzo tra i PDA e le Macchine di Turing**, visto che sono più potenti di un PDA ma meno potenti delle TM. Le LBA generalmente accettano linguaggi **"Context-Sensitive"**, si può anche dimostrare che ogni CFL può essere deciso da un LBA.

DEFINIZIONE FORMALE DI LBA:

Una Linear Bounded Automaton (LBA) è formalmente definita come un sistema a **5-tuple** (Q, Σ, δ, q0, F) dove :

- **Q** è un insieme finito di stati
- **Σ** è un alfabeto finito di simboli
- **δ : Q x Σ → Q x Σ x {L, R}** è la funzione di transizione, che specifica l'azione successiva della LBA in base allo stato corrente e al simbolo letto dalla testina di lettura.
- **q0** è uno stato iniziale
- **F** è un insieme di stati finali

Inoltre, una LBA ha anche un nastro di input finito, su cui la testina di lettura può spostarsi a sinistra o a destra, e un simbolo speciale di "fine nastro" che indica i bordi del nastro.
L'esecuzione termina quando la LBA raggiunge uno stato finale o incontra il simbolo di fine nastro.

TEOREMI E DIMOSTRAZIONI:

$\mathbf{A_{LBA}}$ = {<M,w> | M è un LBA che accetta una stringa w}.

Il teorema analogo per le MT, ovvero A_{TM}, è indecidibile.

$\mathbf{A_{LBA}}$ **è decidibile** e per dimostrarlo abbiamo bisogno di un lemma, il quale dice che il **numero di configurazioni di un LBA è limitato**.

LEMMA:

Preso un LBA "M" con "q" stati e "g" simboli nell'alfabeto del nastro. Ci sono esattamente " qng^n "configurazioni distinte di M per nastro di lunghezza n.

DIMOSTRAZIONE:

Una configurazione di M non è nient'altro che una fotografia di un istante di tempo nel mezzo della computazione.

- **M** ha q stati
- La lunghezza del nastro è **n**, così la testina può essere al più in n posizioni differenti
- $\mathbf{g^n}$ sono le possibili stringhe di simboli del nastro che si possono trovare sul nastro

Il prodotto di queste tre quantità è il numero totale di configurazioni differenti di M con un **nastro di lunghezza** $\mathbf{qng^n}$**.**

TEOREMA:

A_{LBA} è decidibile.

IDEA PER LA DIMOSTRAZIONE:

Bisogna **simulare** $\mathbf{A_{LBA}}$, chiamata M, su una macchina di Turing M', per dimostrare **la decidibilità** dobbiamo come prima cosa individuare quando la macchina **va in loop**.

L'idea per capire quando M va in loop è questa:

- Se, quando M computa w, ripete **più volte la stessa configurazione** allora va in loop.
- Visto che M è un LBA possiamo, tramite il Lemma precedente, dire che ha un **numero limitato di volte che può ripetere la stessa configurazione**. Di conseguenza, tramite M', possiamo dire che se M non si ferma dopo un certo numero di passi significa che è in loop.

DIMOSTRAZIONE:

L'algoritmo che decide $\mathbf{A_{LBA}}$:

M’:=“Su input <M,w>, dove M è un LBA e w è una stringa:

1. Simula M su w finché non si ferma o, alternativamente, per qng^n passi.
2. Se M si ferma, abbiamo i seguenti casi:
   I. Se M accetta allora M’ accetta
   II. Se M rifiuta allora M’ rifiuta
3. Se M non si ferma, allora M’ rifiuta.”

La macchina può rifiutare se non si ferma in qng^n, poiché per il lemma precedente, **M deve aver incontrato almeno una volta una configurazione precedente**, dunque si trova in loop.

Questo teorema è Decidibile per gli LBA ma non per le TM.

Computation Historie(CH):

Una computation history di una macchina di Turing è u**na descrizione dettagliata dell'esecuzione di una computazione** su una specifica input. In particolare, una computation history descrive la sequenza di stati attraversati dalla macchina di Turing durante la sua esecuzione, nonché i simboli scritti e letti dalla testina di lettura e le azioni di movimento della testina di lettura.

Una computation history può essere utilizzata **per dimostrare l'accettazione o il rifiuto** di un input da parte della macchina di Turing, in base all'ultimo stato raggiunto e alla posizione della testina di lettura alla fine dell'esecuzione. Inoltre, può essere utilizzata **per analizzare la complessità computazionale** di un problema, valutando il numero di passi necessari per la macchina di Turing per completare una computazione.

Una computation history **non ha una definizione formale** precisa, ma può essere descritta in modo formale come una **sequenza di tuple** contenenti informazioni sullo stato corrente, simboli scritti e letti, posizione della testina di lettura e azioni di movimento della testina di lettura.

Sia “M” una MT e “w” un input per M. Una **CH accettante** per M su w è una sequenza di configurazioni C_1 , C_2, …, C_l dove:

- C_1 è la configurazione di partenza di M su w
- C_l è una configurazione accettante
- Per ogni C_l, segue C_{l-1} secondo le regole di M

Una CH non accettante ha C_l come **configurazione non accettante**.
Le Macchine di Turing Deterministiche hanno una sola Ch per input mentre quelle non deterministiche possono averne diverse.

TEOREMA:

$\mathbf{E_{LBA}}$ = {<M> | M è un LBA dove L(M)= Ø}.

è indecidibile.

DIMOSTRAZIONE:

Iniziamo con $\mathbf{A_{TM}}$ (Accettazione di una macchina di Turing) che è un decidibile, ovvero un problema di decidibilità che chiede se una macchina di Turing M accetta una stringa w.

Il nostro obiettivo è dimostrare che $\mathbf{E_{LBA}}$ (LBA con Linguaggio vuoto) è indecidibile, ovvero un problema di decidibilità che chiede se un LBA M ha un linguaggio vuoto.

Per farlo, utilizzeremo la tecnica di riduzione, ovvero dimostriamo che esiste un algoritmo che, dato un problema A ($\mathbf{A_{TM}}$) può generare un'istanza del problema B ($\mathbf{E_{LBA}}$) in modo che la soluzione di B corrisponda alla soluzione di A.

Per ogni Macchina di Turing “M” e ingresso “w”, definiamo un LBA “B” tale che
L(B)= { CH | CH è accettante per una data MT M e un dato ingresso w}

Per verificare che una CH è accettante devono essere soddisfatte le seguenti condizioni:

- Se L(B)= Vuoto vuol dire che M non accetta w e dunque (M,w) non è in $\mathbf{A_{TM}}$
- Viceversa se L(B) non è vuoto allora M accetta w e dunque (M,w) è in $\mathbf{A_{TM}}$

Costruiamo adesso il decisore S per $\mathbf{A_{TM}}$, supponendo che esiste un decisore R per L(B):
S:= su ogni ingresso <M,w>,

- costruisce un LBA B da (M,w) come definito sopra.
- Esegue R su ingresso (B)
- se R accetta -> S rifiuta (L(B) è vuoto -> M non accetta w)
- se R non accetta -> S accetta (L(B) è non vuoto -> M ha una CH accettante per w)

Quindi, l'esistenza di un decisore per $\mathbf{E_{LBA}}$ implica un decisore per $\mathbf{A_{TM}}$.
Sappiamo però che $\mathbf{A_{TM}}$ è indecidibile, quindi non esiste un decisore per $\mathbf{E_{LBA}}$.

Note Importanti

Questi sono i linguaggi con le loro rispettive macchine:

PRIMO CAPITOLO

Espressione Regolare ⇔ DFA
Espressione Regolare ⇔ NFA
Espressione Regolare ⇔ Linguaggio Regolare
DFA ⇔ Linguaggio Regolare

NFA ⇔ Linguaggio Regolare
NFA -> DFA
ε - NFA -> NFA

SECONDO CAPITOLO

Linguaggi Context-Free ⇔ Grammatiche Context-Free

TERZO CAPITOLO

Turing-Riconoscibile == Recursively Enumerable Languages
Macchine di Turing ⇔ Linguaggio Turing-Riconoscibile
Macchine di Turing Non Deterministiche⇔ Linguaggio Turing-Riconoscibile
Macchine di Turing Multinastro ⇔ Linguaggio Turing-Riconoscibile

Operazioni Linguaggi

M romp o cazz e tradurr

Linguaggi Turing-Riconoscibili:

UNIONE E INTERSEZIONE:

A TM that recognizes **L1** ∪ **L2**:
On input x, run M1 and M2 on x in parallel, and accept iff either accepts.
(Similarly for intersection; but no need for parallel simulation)

COMPLEMENTAZIONE:

I linguaggi Turing-Riconoscibili **non sono chiusi** rispetto alla complementazione.
Basti pensare che ogni linguaggio Turing-riconoscibile a cui riusciamo a provare l'indecidibilità non è Co-Turing-Riconoscibile.
Prova:

A_{TM} è Turing Riconoscibile

Dimostrazione:
Una **macchina di Turing U** capace di accettare **A_{TM}** è la seguente:

Su un input (M, w), dove **M** è la codifica di una **TM** e **w** è una stringa:

1. **U** simula M su w.
2. Se M **non entra mai** in uno stato di accettazione, **allora U accetta**; se M non entra mai in uno stato di rifiuto, allora U accetta.

Si noti che **U cicla su un input (M, w)** se M cicla su w, il che dimostra perché U non decide A_{TM}.

CONCATENAZIONE:

A TM to recognize **L1•L2**:
On input x, do in parallel, for each of the |x| + 1 ways to divide x as yz:
run M1 on y and M2 on z, and accept if both accept. Else reject.

KLEENE CLOSURE(STAR):

A TM to recognize L1* :
On input x, if x = ε accept.
Else, do in parallel, for each of the $2^{|x|-1}$ ways to divide x as w1 . . . wk (wi ≠ ε): run M1 on each wi and accept if M1 accepts all. Else reject.

Linguaggi Decidibili:

UNIONE E INTERSEZIONE:

A TM that decides L1 ∪ L2:
On input x, run M1 and M2 on x, and accept iff either accepts.
(Similarly for intersection.)

COMPLEMENTAZIONE:

A TM that decides comp(L1):
On input x, run M1 on x, and accept if M1 rejects, and reject if M1 accepts.

CONCATENAZIONE:

A TM to decide **L1•L2**:
On input x, for each of the |x| + 1 ways to divide x as yz: run M1 on y and M2 on z, and accept if both accept. Else reject.

KLEENE CLOSURE(STAR):

A TM to decide L ∗ 1 :
On input x, if x = accept. Else, for each of the $2^{|x|-1}$ ways to divide x as w1 . . . wk (wi 6=): run M1 on each wi and accept if M1 accepts all. Else reject.

Compito Gennaio

Prima Parte:

1. (4pt) Siano L1 e L2 due linguaggi regolare su Z, si può dire che anche L1\L2 è regolare?

2. (12 pt) Si definisca un NFA sull'alfabeto Z=(a, b, c) che accetti il linguaggio L di tutte le parole dove la lettera a o la lettera b sono presenti almeno due volte. Per esempio, L contiene aba e bacbbbab ma non abc, ne abcc. Si scriva anche il DFA equivalente, usando la costruzione per sottoinsiemi.

3. (8 pt) Provare che il linguaggio L = (x^k^3 I k € (0, 1, 2, 3, ...), ovvero L contiene X, XXX, XXXXXXXX, non e regolare.

4. (8 pt) Scrivere una CFG che generi parole su 2 = [0,1) che contengano più occorrenze di 1 rispetto a 0. Per esempio 100111010

Seconda Parte:

1. (12pt) Rispondere alle seguenti domande (argomentando, un semplice si/no non sarà accettato):
   - **a.** Dare la definizione di Turing riconoscibile e co-Turing riconoscibile, Che succede se un linguaggio appartiene ad entrambi?
   - **b.** Esistono linguaggi che non sono né Turing e né co-Turing riconoscibili, se si indicarne uno.
   - **c.** I linguaggi decidibili sono chiusi rispetto all'intersezione?
   - **d.** Sia M un automa linear bounded (LBA). Quante configurazioni sono possibili su un input w, con lw| = n?
   - **e.** AlbA = ((M, w) | M is an LBA that accepts w) è decidibile?

2. (10pt) Sia Z = (0, 1, -, a), e L= (w-a"|we (0, 1)* tale per cui w è una rappresentazione binaria di n), per esempio, 00011-aaa E L ma non le parole 101-aa ne 001-aaa. Si costruisca una macchina di Turing che decida L

3. (10pt) Si provi che il seguente linguaggio è indecidibile. Si può usare una riduzione da ATM or da altri problemi noti essere indecidibili L=<M>: M è una TM e M accetta la stringa 001

Soluzioni:

Seconda Parte:

Esercizio 1:

a. **Dare la definizione di Turing riconoscibile e co-Turing riconoscibile, Che succede se un linguaggio appartiene ad entrambi?**

b. **Esistono linguaggi che non sono né Turing e né co-Turing riconoscibili, se si indicarne uno?**

Non esiste un linguaggio che non sia né Turing né co-Turing riconoscibile, poiché secondo la teoria della computazione, ogni linguaggio computabile può essere rappresentato da un modello di macchina di Turing e, quindi, può essere riconosciuto da un modello di macchina di Turing. Inoltre, ogni linguaggio non computabile può essere rappresentato da una macchina di Turing che non accetta la stringa e quindi può essere riconosciuto come non computabile.

c. **I linguaggi decidibili sono chiusi rispetto all'intersezione?**

d. **Sia M un automa linear bounded (LBA). Quante configurazioni sono possibili su un input w, con |w| = n?**

e. **AlbA = ((M, w) | M is an LBA that accepts w) è decidibile?**

Esercizio 3:

Si provi che il seguente linguaggio è indecidibile. Si può usare una riduzione da ATM or da altri problemi noti essere indecidibili L=<M>: M è una TM e M accetta la stringa 001:

Un'idea per dimostrare che il linguaggio è indecidibile è quello di ridurre A_{TM}
ad $HALT_{TM}$, e mostrare che essa è indecidibile. Prendiamo un decisore R per $HALT_{TM}$ e un decisore S per A_{TM}. Vediamo che se usassimo S su M saremmo in grado di decidere se