Appunti di Automi e linguaggi formali

Ejo Grejo

6 agosto 2024

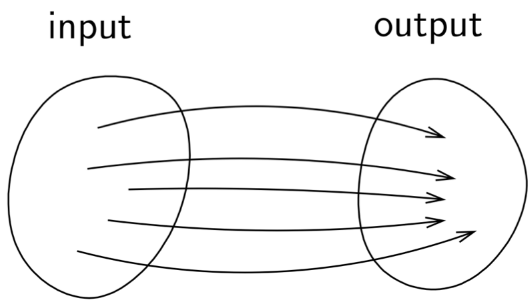
Indice

1 Introduzione

1.1 Problemi

Per descrivere un **problema** bisogna specificare:

- Insieme dei possibili Input
- Insieme dei possibili Output
- La relazione tra Input e Output



Algoritmo: procedura meccanica che esegue delle computazioni (e può essere eseguita da un calcolatore).
Un algoritmo **risolve** un dato problema se:

- Per ogni input, il calcolo dell'algoritmo si interrompe dopo un numero finito di passaggi.
- Per ogni input, l'algoritmo produce un output corretto.

Correttezza di un algoritmo: Verificare che l'algoritmo risolva realmente il problema dato
Complessità computazionale di un algoritmo:

- **complessità temporale**: come varia il tempo di esecuzione rispetto alla dimensione dei dati di input
- **complessità spaziale**: come varia la quantità di memoria utilizzata rispetto alla dimensione dei dati di input

1.2 Linguaggi Formali

- Astrazione della nozione di problema
- I problemi sono espressi come **linguaggi** (insieme di stringhe)
- Le soluzioni determinano se una determinata stringa è nell'insieme o no
  - Esempio: *un certo intero n è un numero primo?*
- Oppure, come *trasformazioni tra linguaggi*
  - Le soluzioni trasformano la stringa di input in una stringa di output
    - Esempio: *quanto fa* $3 + 5$*?*

Quindi in sostanza tutti i processi computazionali possono essere ridotti ad uno tra:

- Determinazione dell'*appartenenza* a un insieme (di stringhe)
- *Mappatura* tra insiemi (di stringhe)

Formalizzeremo il concetto di computazione meccanica:

- Dando una definizione precisa del termine "algoritmo"
- Caratterizzando i problemi che sono o non sono adatti per essere risolti da un calcolatore

1.3 Automi

Gli *automi* sono dispositivi matematici astratti che possono:

- Determinare l'appartenenza di una stringa ad un insieme di stringhe
- Trasformare una stringa in un'altra stringa

Hanno tutti gli *aspetti* di un computer
Il tipo di **memoria** è cruciale:

- memoria finita
- memoria infinita:
  - con accesso limitato
  - con accesso illimitato

Abbiamo diversi tipi di automi per diversi classi di linguaggi
I diversi tipi di automi si differenziano per

- La quantità di memoria (finita vs infinita)
- Il tipo di accesso alla memoria (limitato vs illimitato)

2 Linguaggi Regolari

Un FA (NFA "[[Automi a Stati Finiti Non Deterministici]]" o DFA "[[Automi a Stati Finiti Non Deterministici]]) è un metodo per costruire una macchina che riconosce linguaggi regolari.
Un'*espressione regolare* è un modo dichiarativo per descrivere un linguaggio regolare Esempio:

$$01 * +10*$$

Le espressioni regolari sono usate ad esempio in:

- comandi UNIX (*grep*)
- strumenti per l'analisi lessicale di UNIX (`lex` *Lexical analyzer generator*) e `flex` (*Fast Lex*)
- editor di testo

2.1 Espressioni regolari

sono costruite utilizzando:

- un insieme di *costanti* di base:
  - ϵ per la stringa vuota
  - $\varnothing$ per il linguaggio vuoto
  - $a, b, ...$ per i simboli $a, b, ... \in \sum$
- collegati da *operatori*:
  - $+$ per l'unione
  - . per la concatenazione
  - $*$ per la chiusura di Kleene
- Raggrupati usando le *parentesi* ()

Se E è un espressione regolare, allora $L(E)$ è il *linguaggio rappresentato da E*. La definizione di $L(E)$ è induttiva:

- **Caso Base**:
  - $L(\epsilon) = \{\epsilon\}$
  - $L(\varnothing) = \varnothing$
  - $L(a) = \{a\}$
- **Caso induttivo**:
  - $L(E + F) = L(E) \cup L(F)$
  - $L(EF) = L(E).L(F)$
  - $L(E*) = L(E)^*$
  - $L((E)) = L(E)$

Esempio
$L = \{w \in \{0, 1\}^* : 0 \text{ e } 1 \text{ alternati in } w\}$
$(01)^* + (10)^* + 1(01)^* + 0(10)^*$ oppure $(\epsilon + 1)(01)^*(\epsilon + 0)$

2.1.1 Precedenza

Come per le espressioni aritmetiche, anche per le espressioni regolari ci sono delle *regole di precedenza* degli operatori

1. Chiusura di Kleene
2. Concatenazione (punto)
3. Unione

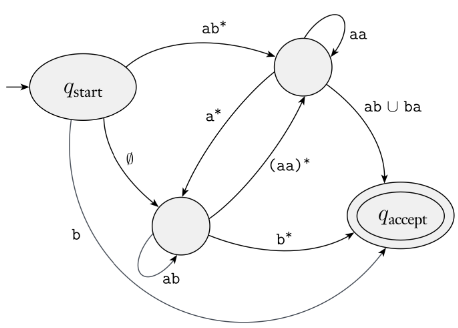
Esempio
$01^* + 1$ è raggruppato in $(0(1)^*) + 1$
e denota un linguaggio *diverso* da $(01)^* + 1$

2.2 Conversione per eliminazione di stati

La procedura che vedremo è in grado di convertire un *qualsiasi automa* (DFA o NFA) in un'espressione regolare equivalente.
Si procede per eliminazione di stati.
Quando uno stato q viene eliminato i cammini che passano per q scompaiono.
Si aggiungono nuove *transizioni etichettate con espressioni regolari* che rappresentano i cammini eliminati.
Alla fine otteniamo un'espressione regolare che rappresenta *tutti i cammini* dallo stato iniziale ad uno stato finale.
→ cioè il linguaggio riconosciuto dall'automa

2.2.1 Da NFA a GNFA

1. Nuovo stato iniziale q_{start} con transizione ε verso il vecchio q_0
2. Nuovo stato finale q_{accept} con transizione ε da tutti i vecchi stati finali $q \in F$
3. Rimpiazzo transizioni multiple tra due stati con l'unione delle etichette
4. Aggiungo transizioni etichettate con $\varnothing$ tra stati non collegati da transizioni



2.3 Definizione formale di GNFA

Un Automa a Stati Finiti Non Deterministico Generalizzato (GNFA) è una quintupla: $A = (Q, \sum, \delta, q_{start}, q_{accept})$

- Q è un insieme finito di *stati*
- $\sum$ è un *albero finito*

- $\delta : Q\backslash\{q_{accept}\} \times Q\backslash \mapsto R$ è una *funzione di transizione* che prende in input *2 stati* e restituisce una *espressione regolare* su $\sum$
- $q_{start} \in Q$ è lo *stato iniziale*
- q_{accept} è lo *stato finale*

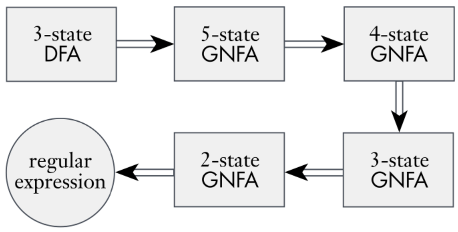
2.3.1 Computazione di un GNFA

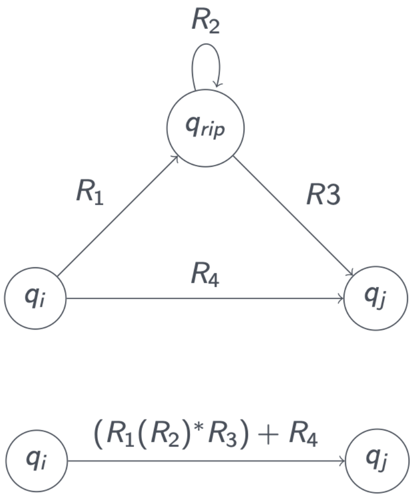
Data una parola $w = w_1 w_2 ... w_m$, dove $w_i \in \sum^*$ Una *computazione* di un GNFA A con input w è una sequenza di stati $r_0 r_1 ... r_m$ che rispetta *2 condizioni*:

1. $r_0 = q_{start}$ (inizia dallo stato iniziale)
2. Per ogni $i, w_i \in L(R_i), dove R_i = \delta(r_{i-1}, r_i)$ (*rispetta la funzione di transizione*)

Una computazione *accetta* la parola w se *termina nello stato finale* ($r_m = q_{accept}$)

- Partiamo da un GNFA con k stati, dove $k \geq 2$
- Se $k > 2$, eliminiamo uno stato q_{rip} per ottenere un GNFA con $k - 1$ stati.
- Quandk $k = 2$, l'etichetta della transiione da q_{start} a q_{accept} è l'espressione regolare equivalente.





Se, nel GNFA:

1. q_i va in q_{rip} con etichetta R_1
2. q_{rip} ha un self loop con etichetta R_2
3. q_{rip} va in q_j con etichetta R_3
4. q_i va in q_j con etichetta R_4

dopo l'eliminazione di q_{rip}, q_i va in q_j con etichetta

$$(R_1(R - 2) * R_3) + R_4$$

Algoritmo di conversione `Convert(A)`

1. Sia k il numero di stati di A
2. Se $k = 2$, ritorna l'espressione R che collega q_{start} con q_{accept}

3. Se $k > 2$, scegli $q_{rip} \in Q \backslash \{q_{start}, q_{accept}\}$ e costruisci un GNFA $A' = (Q', \sum, \delta', q_{start}, q_{accept})$ come segue:

   - $Q' = Q \backslash \{q_{rip}\}$
   - per ogni $q_i \in Q' \backslash \{q_{accept}\}, q_j \in Q' \backslash \{q_{start}\}$, sia $\delta'(q_i, q_j) = (R_1(R_2) * R_3) + R_4$

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

4. Ritorna il risultato calcolato da `Convert(A')`

3 Linguaggi Non Regolari

Di quanti stati necessita l'automa che riconoscere linguaggio $\{0^n1^n|n \geq 0\}$? Occorrono $2n$ stati per poter riconoscere il linguaggio. Essendo n infinito non è possibile determinare un numero finito di stati, pertanto il linguaggio $\{0^n1^n|n \geq 0\}$ non è regolare, proprio perché non può essere riconosciuto da un automa a stati finiti.

Dimostrazione È necessario ragionare *per assurdo*, in quanto siamo costretti a dimostrare che non esiste un automa a stati finiti che riconosce un linguaggio che ipotizziamo essere non regolare. Supponiamo che $L_{01} = \{0^n1^n|n \geq 0\}$ sia regolare, allora esiste un DFA A che accetta L_{01} con k stati. Cerchiamo ora di dimostrare che esiste una parola appartenente a L_{01} che non viene riconosciuta dall'automa DFA. L'automa segue una computazione $r_0, r_1, r_2, \ldots, r_k$ ovvero di lunghezza $k+1$, questo comporta che all'interno della sequenza c'è uno stato che si ripete, supponiamo che si tratti di $r_i = r_j$ per qualche a. Posso sfruttare questo fatto per far sbagliare l'automa: considero la parola 0^i1^i, la computazione ha la forma

$$r_0, r_1, \ldots, r_i, s_1, \ldots, s_i$$

s_i è uno stato finale? Sì, s_i deve essere finale. Cosa succede se a questo automa forniamo in input la parola 0^j1^i ? Abbiamo detto che $r_j = r_i$ quindi l'automa terminerà nello stesso stato finale s_i e dunque verrebbe riconosciuta una parola che non appartiene a L_{01}.

3.1 SBRODEGHI

- supponiamo che $l_0 = \{0^n1^n|n \geq 0\}$
- ...
- cosa succede quando l; automa A legge 1^i partendo da q
- se l'automa finisce la lettura in uno stato finale
- allora accetta, sbagliando la parola 0^j1^j
- se l'automa finisce la lettura in uno stato non finale
- allora rifiuta sbagliando la parola 0^i1^i
- in entrambi i casi abbiamo ingannato l'automa, quindi L_{01} *non può essere regolare*

3.1.1 Proprietà dei linguaggi regolari

Possono essere utilizzate per dimostrare che un DFA effettua un loop per accettare un linguaggio regolare. La dimostrazione è più facile rispetto a quella precedente.

3.2 Pumping Lemma

Sia L un linguaggio regolare. Allora

- **esiste una lunghezza** $k > 0$ tale che
- **ogni parola** $w \in L$ di lunghezza $|w| \geq k$

- **può essere spezzata** in $w = xyz$ tale che

1. $y \neq \varepsilon$ (il secondo pezzo è non vuoto)
2. $|xy| \leq k$ (i primi due pezzi sono lunghi al max k)
3. $\forall i \geq 0, xy^i z \in L$ (possiamo "pompare" y rimandandolo in L)

3.2.1 Dimostrazione

- Supponiamo che L sia un linguaggio regolare
- Allora è riconosciuto da un DFA con, supponiamo, k stati
- Consideriamo una parola $w = a_1, a_2, \ldots, a_n \in L$ di lunghezza $n \geq k$
- Consideriamo gli stati nella computazione di A per w

$$p_0, p_1, p_2, \ldots p_k \ldots p_n$$

Siccome in $p_0, p_1, \ldots p_k$ ci sono $k+1$ stati ne esiste uno che si ripete:
Esistono $l < m$ tali che $p_l = p_m$ e $m \leq k$

3.2.2 Pumping lemma come gioco

Esiste una strategia che ci consente di vincere sempre su un certo linguaggio.

1. $\exists k > o$ (giocatore 1 sceglie la dimensione di k)
2. $\forall w \in L, |w| \geq k$ (giocatore 2 sceglie una parola)
3. $\exists xyz \mid w = xyz \; y \neq \varepsilon \; |xy| \leq k$ (giocatore 1 sceglie come partizionare la parola)
4. $\forall i \geq 0 xy^i z \in L$ (giocatore 2 sceglie una potenza affinché la condizione sia verificata)

Se giocatore 2 vince allora il linguaggio non è regolare.

3.2.3 Esistono linguaggi non regolari che rispettano il Pumping Lemma

Sia L_{ab} il linguaggio delle stringhe sull'alfabeto (a, b) dove il numero di a è uguale al numero di b. L_{ab} è regolare? Per assurdo ipotizzo L_{ab} regolare.
Se lo è, allora rispetta il Pumping Lemma $\rightarrow$ deve esistere una lunghezza $k > 0$ che rende vero il Pumping Lemma.
Consideriamo la parola $w = a^k b^k$, w appartiene al linguaggio e $|w| > k$.
Per ogni suddivisione $w \in xyz$ dove $y \neq \varnothing$ e $|xy| \leq k$

$$x = a^P$$
$$y = a^Q$$
$$z = a^{K-P-Q} b^K$$

Se prendiamo come esponente $i = 2$, $xy^2 z = a^P a^{2Q} a^{K-P-Q} b^K = a^{Q+K} b^K$ la parola non fa parte del linguaggio.
Conclusione: per assurdo non è regolare.

Il linguaggio $L_{rev} = \{ww^R : w \in \{a,b\}*\}$ è regolare?

$w^R = w$ scritto alla rovescia
$w = w_1, w_2, \ldots, w_k$
$w^R = w_n, w_{n-1}, \ldots, w_1$

1. Suppongo L_{rev} sia regolare,allora esiste $k > 0$ che rende vero il Pumping Lemma
2. considero la parola $w = a^k b^2 a^k$
3. Per ogni suddivisione di xyz dove $y \neq \varepsilon$ e $|xy| \leq k$

$$x = a^P$$
$$y = a^Q$$
$$z = a^{K-P-Q} b^2 a^K$$

Se prendiamo come esponente $i = 2$, $xy^2z = a^P a^{2Q} a^{K-P-Q} b^2 a^K = a^{Q+K} b^2 a^K$ la parola non fa parte del linguaggio.
Conclusione: per assurdo non è regolare.

Il linguaggio $L_p = \{1^p : p$ è primo$\}$ è regolare?

1. Assumo che L_p sia regolare e che $k > 0$ sia la lunghezza che rende vero il Pumping Lemma.
2. Considero $w = 1j$ dove j è il primo numero primo $> k$
3. Per ogni suddivisione $w = xyz$ dove $y \neq \varnothing$ e $|xy| \leq k$
   (a) $x = 1^P$
   (b) $y = 1^Q$
   (c) $z = 1^{J-P-Q}$ dove $Q > 0$ e $P + Q \leq k$

$i = 2$
$xy^2z = 1^P 1^{2Q} 1^{J-P-Q} = 1^{Q+J}$

$i = 2J$
$xy^{2J}z = 1^P 1^{2jQ} 1^{J-P-Q} = 1^{(2J)Q+J-Q}$
$= 1^{(2J-1)Q+J} = 1^{(Q+1)J} \notin L_p$
La parola non è contenuta nel linguaggio in quanto la sua lunghezza è un numero scomponibile in fattori.

4 Linguaggi Context-Free

Abbiamo visto che esistono *linguaggi non regolari* (ad esempio $\{0^n1^n|n \geq 0\}$), Consideriamo ora una classe più grande di linguaggi, i **Linguaggi Context-free (CFL)**, usati nello studio dei linguaggi naturali dal 1950, e nello studio dei compilatori dal 1960. Vedremo dunque due metodi per descrivere un linguaggio CFL:

- Grammatiche Context-free
- Automi a pila (pushdown automata)

4.1 Grammatiche Context-free

Le grammatiche context-free sono un metodo più potente per descrivere i linguaggi, si prestano particolarmente bene a descrivere comportamenti ricorsivi all'interno di un linguaggio.
Inizialmente sono state utilizzate per descrivere i **linguaggi naturali** (vedi Una grammatica per l'inglese), oggi sono fondamentali per lo sviluppo dei *parser*.

Vediamo ad esempio la grammatica G_1:

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

Quello che possiamo notare è la presenza di

- un insieme di **regole di sostituzione** (o *produzioni*)
- alcune **variabili** (A e B)
- i **terminali** ossia i simboli dell'alfabeto $(0, 1, \#)$
- una **variabile iniziale** A

1. Scrivi la variabile iniziale
2. Trova una variabile che è stata scritta e una regola che inizia con quella variabile. Sostituisci la variabile con il lato destro della regola.
3. Ripeti 2. fino a quando non ci sono più variabili.

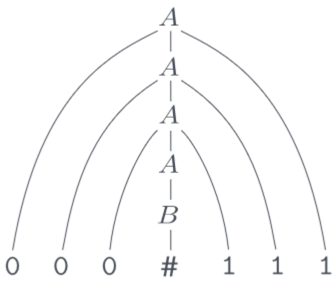
Esempio per G_1

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

La sequenza di sostituzioni si chiama **Derivazione di 000#111**

4.2 Albero sintattico

Una derivazione definisce un **albero sintattico (parse tree)**



- la radice è la variabile iniziale
- i nodi interni sono variabili
- le foglie sono terminali

Tutte le stringhe generate in questo modo costituiscono il *linguaggio della grammatica* G_1, ossia $L(G_1)$.

4.3 Una grammatica per l'inglese

Definiamo G_2 nel seguente modo:
$\langle SENTENCE\rangle \rightarrow \langle NOUN\text{-}PHRASE\rangle\langle VERB\text{-}PHRASE\rangle$
$\langle NOUN\text{-}PHRASE\rangle \rightarrow \langle CMPLX\text{-}NOUN\rangle$
$\langle NOUN\text{-}PHRASE\rangle \rightarrow \langle CMPLX\text{-}NOUN\rangle\langle PREP\text{-}PHRASE\rangle$
$\langle VERB\text{-}PHRASE\rangle \rightarrow \langle CMPLX\text{-}VERB\rangle$
$\langle VERB\text{-}PHRASE\rangle \rightarrow \langle CMPLX\text{-}VERB\rangle\langle PREP\text{-}PHRASE\rangle$
$\langle PREP\text{-}PHRASE\rangle \rightarrow \langle PREP\rangle\langle CMPLX\text{-}NOUN\rangle$
$\langle CMPLX\text{-}NOUN\rangle$ß$\langle ARTICLE\rangle\langle NOUN\rangle$
$\langle CMPLX\text{-}VERB\rangle$ß$\langle VERB\rangle|\langle VERB\rangle\langle NOUN\text{-}PHRASE\rangle$
$\langle ARTICLE\rangle \rightarrow$ `a | the`
$\langle NOUN\rangle \rightarrow$ `boy | girl | flower`
$\langle VERB\rangle \rightarrow$ `touches | likes | sees`
$\langle PREP\rangle \rightarrow$ `with`

La grammatica G_2 ha 10 variabili ($SENTENCE$, $NOUN\text{-}PHRASES$, eccetera...), 27 terminali, ovvero i simboli dell'alfabeto inglese standard, e 18 regole. Tra le stringe di $L(G_2)$ vi sono

```
a boy sees
the boy sees a flower
a girl with a flower likes the boy
```

Ognuna di queste stringhe ha una derivazione nella grammatica G_2, ad esempio
$\langle SENTENCE\rangle \rightarrow \langle NOUN\text{-}PHRASE\rangle\langle VERB\text{-}PHRASE\rangle$
$\rightarrow \langle CMPLX\text{-}NOUN\rangle\langle VERB\text{-}PHRASE\rangle$

$\rightarrow \langle ARTICLE \rangle \langle NOUN \rangle \langle VERB\text{-}PHRASE \rangle$
$\rightarrow$ `a` $\langle NOUN \rangle \langle VERB\text{-}PHRASE \rangle$
$\rightarrow$ `a boy` $\langle VERB\text{-}PHRASE \rangle$
$\rightarrow$ `a boy` $\langle CMPLX\text{-}VERB \rangle$
$\rightarrow$ `a boy` $\langle VERB \rangle$
$\rightarrow$ `a boy sees`

4.4 Definizione di grammatica context-free

Una grammatica context-free è una quadrupla (V, Σ, R, S) dove

- V è un insieme finito di **variabili**
- Σ è un insieme finito di **terminali** disgiunto da V
- R è un insieme di **regole**, dove ogni regola è una variabile e una stringa di variabili e terminali
- $S \in V$ è la **variable iniziale**

Se u, v, w sono stringhe di variabili e terminali e $A \rightarrow w$ è una regola:

- uAv **produce** uwv : $uAv \Rightarrow uwv$
- u **deriva** v : $u \Rightarrow^* v$ se:
  - $u = v$, oppure
  - esiste una sequenza $u1, u2, \ldots, uk$ tale che $u \Rightarrow u1 \Rightarrow u2 \Rightarrow \cdots \Rightarrow uk \Rightarrow v \Rightarrow$
- il **linguaggio della grammatica** è $L(G) = \{w \in \Sigma^* \mid S \Rightarrow^* w\}$

4.5 Esempi

4.5.1 Parentesi

Consideriamo la grammatica $G_3 = \langle \{S\}, \{a, b\}, R, S \rangle$. L'insieme delle regole della grammatica G_3 , ossia R, è

$$S \rightarrow aSb \mid SS \mid \varepsilon$$

Questa grammatica genera stringhe come

- `abab`: $S \rightarrow SS \rightarrow abS \rightarrow abab$
- `aaabbb` $S \rightarrow aSb \rightarrow aaSbb \rightarrow aaaSbbb \rightarrow aaabbb$
- `aababb`: $S \rightarrow aSb \rightarrow aSSb \rightarrow aabSb \rightarrow aababb$

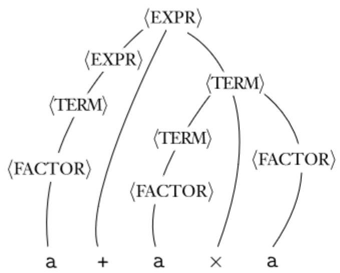
Si può più dedurre più facilmente di che linguaggio si tratta pensando ad `a` come ad una parentesi aperta `"("` e a `b` come ad una parentesi chiusa `")"`.
Visto in questo modo, $L(G_3)$ è il linguaggio di tutte le stringhe di **parentesi correttamente annidate**.
Si osservi che il lato destro di una regola può essere la parola vuota ε

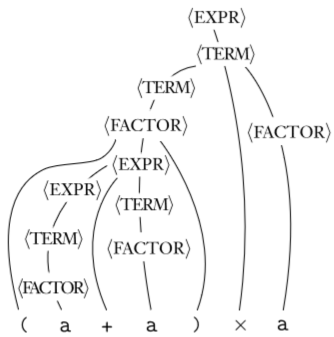
4.5.2 Operazioni aritmetiche

Si consideri la grammatica $G_4 = (V, \Sigma, R, \langle EXPR \rangle)$
V è $\{\langle EXPR \rangle, \langle TERM \rangle, \langle FACTOR \rangle\}$ e Σ è $\{a, +, \times, (,)\}$
Le regole sono:

$$\begin{aligned}
&\langle EXPR \rangle \to \langle EXPR \rangle + \langle TERM \rangle \mid \langle TERM \rangle \\
&\langle TERM \rangle \to \langle TERM \rangle \times \langle FACTOR \rangle \mid \langle FACTOR \rangle \\
&\langle FACTOR \rangle \to (\langle EXPR \rangle) \mid a
\end{aligned}$$

Le due stringhe `a+axa` e `(a+a)xa` possono essere generate dalla grammatica G_4, e questi sono i loro alberi sintattici.





4.6 Come si progetta una grammatica context-free

Abbiamo appena visto la grammatica G_4. Tale grammatica potrebbe benissimo costituire una sottogrammatica di un *linguaggio di programmazione*, in quanto le operazioni aritmetiche sono una componente fondamentale dei linguaggi di programmazione.
In questo e in altri svariati contesti è probabile imbattersi in una CFG (context-free grammatic) composta da altre grammatiche meno complesse. Per unire queste grammatiche è necessario creare una regola $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, dove $S_1 \dots S_k$ sono le variabili iniziali di ognuna delle grammatiche comprese nel linguaggio.

Un esempio banale: Se volessimo costruire la grammatica $\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$ è necessario costruire la prima grammatica

$$S_1 \to 0S_11 \mid \varepsilon$$

poi quella per la seconda

$$S_2 \to 1S_20 \mid \varepsilon$$

La grammatica che comprende entrambe sarà quindi l'insieme di queste tre regole:

$$S \to S_1 \mid S_2$$

$$S_1 \to 0S_11 \mid \varepsilon$$

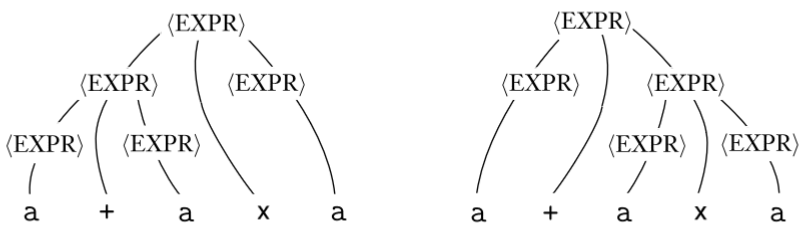
$$S_2 \to 1S_20 \mid \varepsilon$$

Sebbene si tratti di un linguaggio non regolare è solitamente più facile costruire una grammatica per un linguaggio regolare partendo dal suo DFA.

4.7 Forma normale di Chomsky

4.7.1 Ambiguità

Una grammatica CFG può generare una stessa stringa a partire dall'applicazione di derivazioni diverse, ad esempio la grammatica G_4 è in grado di produrre la stringa `a + a x a` con i seguenti alberi di derivazione:



Questa due casi costituiscono un **ambiguità**.

- Una stringa w è derivata ambiguamente dalla grammatica G se esistono due o più alberi sintattici che la generano
- **Equivalentemente**: Una stringa w è derivata ambiguamente dalla grammatica G se esistono due o più derivazioni a sinistra che la generano
- Una grammatica è ambigua se genera almeno una stringa ambiguamente

Spesso dunque è preferibile ricavare una forma normalizzata di una CFG per poterla analizzare meglio.
Una delle forme più semplici ed utili è appunto la forma normale di Chomsky.

4.7.2 Definizione della forma normale di Chomsky

Una grammatica context-free è in forma normale di Chomsky se ogni regola è della forma

$$A \to BC$$
$$A \to a$$

Dove a è un terminale e B, C non possono essere la variabile iniziale. Inoltre, ci può essere la regola iniziale $S \to \varepsilon$ per la variabile iniziale S.

Ogni CFG è generata da una forma normale di Chomsky
di conseguenza
Ogni CFG è riducibile ad una forma normale di Chomsky

4.7.3 Come passare ad una forma normale di Chomsky

1. aggiungiamo una nuova variabile iniziale
2. eliminiamo le ε-regole $A \to \varepsilon$
3. eliminiamo le regole unitarie $A \to B$
4. trasformiamo le regole rimaste nella forma corretta

Esempio Trasformiamo la grammatica G_6 in forma normale di Chomsky:

$$S \to ASA \mid aB$$

$$A \to B \mid S$$

$$B \to b \mid \varepsilon$$

1. Aggiungiamo una nuova variabile iniziale $S_0 \to S$ *Questo ci serve ad evitare che la variabile iniziale compaia a* ***destra*** *di una regola di derivazione*

2. Eliminiamo le ε-regole $A \to \varepsilon$

   - Se $A \to \varepsilon$ è una regola dove A non è la variabile iniziale.
   - Per ogni regola del tipo $R \to uAv$ aggiungiamo la regola

     $$R \to uv$$

     Attenzione: Nel caso di più occorrenze di A consideriamo tutti i casi: per le regole come $R \to uvAw \mid uAvw \mid uvw$
   - Nel caso di regole $R \to A$ aggiungiamo $R \to \varepsilon$ solo se non abbiamo già eliminato $R \to \varepsilon$
   - ripetere fino a che non sono state eliminate tutte le ε-regole

3. Eliminare le **regole unitarie** $A \to B$:

   - Se $A \to B$ è una regola unitaria
   - Per ogni regola del tipo $B \to u$, aggiungiamo la regola unitaria eliminata in precedenza
   - ripetere finché non sono state eliminate tutte le regole unitarie

4. Trasformiamo le regole rimaste nella forma corretta:

   - Se $A \to u_1u_2 \dots u_k$ è una regola tale che
     - Ogni u_i è una variabile o un terminale
     - $k \geq 3$
   - sostituisci la regola con la catena di regole $A \to u_1A_1$, $A_1 \to u_2A_2$, $A_@ \to u_3A_3$ $\dots$ $A_{k-2} \to u_{k-1}u_k$
   - rimpiazza ogni terminale u_i; sul lato destro di una regola con una nuova variabile U_i e aggiungi la regola

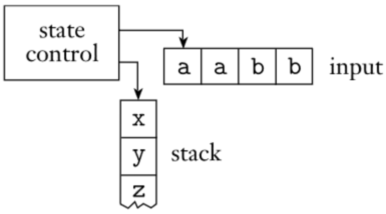
     $$U_i \to u_i$$

   - ripetere per ogni regola non corretta

5 Automi a Pila

Come preannunciato in [[Linguaggi Context-free]], gli automi a pila sono un metodo per definire alcuni *linguaggi non regolari*. Gli automi a pila (*pushdown automata*, o *PDA*) sono degli automi a cui viene affiancata una memoria detta **stack** (pila), e sono **computazionalmente equivalenti alle grammatiche context-free**. Un PDA può scrivere simboli nella pila e rileggerli in seguito. Scrivere un simbolo comporta "*spingere giù*" i simboli della pila. In qualsiasi momento il simbolo nella cima può essere letto e rimosso, esattamente come si comporta una memoria di tipo *stack*, e dunque **LIFO** (last in first out). Un automa a pila è composto da:

- **Input**: stringa di caratteri dell'alfabeto
- **Memoria**: stati + pila
- **Funzioni di transizione**: dato lo stato corrente, un simbolo di input ed il *simbolo in cima alla pila*, stabilisce quali possono essere gli stati successivi e i *simboli da scrivere sulla pila*



Le operazioni di scrittura sono

- **push**: inserimento di un simbolo nella pila
- **pop**: lettura e rimozione di un simbolo dalla pila

Questo permette agli automi a pila di avere *memoria infinita* **MA** *con accesso limitato*. Esempio: $\{0^n 1^n | n \geq 0\}$

Un PDA usa la pila per contare 0 e 1:

- legge i simboli in input, e *scrive* ogni 0 letto sulla pila
- non appena vede gli 1, *cancella* uno 0 dalla pila per ogni 1 letto
- se l'input termina esattamente quando la pila si svuota, **accetta**
- se ci sono ancora 0 nella pila al termine dell'input, *rifiuta*
- se la pila si svuota prima della fine dell'input, *rifiuta*
- se qualche 0 compare nell'input dopo gli 1, *rifiuta*

Attenzione

Gli automi a pila possono essere **non deterministici**. Automi a pila deterministici e non deterministici *non* sono computazionalmente equivalenti, diversamente da quanto avviene tra DFA e NFA.

5.1 Definizione di PDA

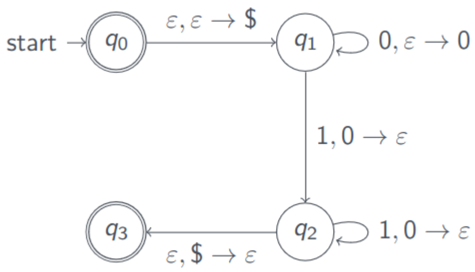
La definizione è simile a quella di un automa finito, tranne che per la pila. Tale pila *non utilizza necessariamente lo stesso linguaggio della stringa in input* Per tanto un automa a pila è una sestupla $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$:

- Q è l'insieme finito di *stati*
- Σ è l'alfabeto di *input*
- Γ è l'alfabeto della *pila*
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \mapsto 2^{Q \times \Gamma_\varepsilon}$ è la funzione di transizione
- $q_0 \in Q$ è lo *stato iniziale*
- $F \subseteq Q$ è l'insieme di *stati accettanti* (dove $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$)

Esempio PDA per $\{0^n 1^n \mid n \geq 0\}$:

- $P = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \{0, \$\}, \delta, q_0, \{q_0, q_3\})$
- con δ descritta dalla tabella:

Input:	0			1			ε		
Pila:	0	\$	ε	0	\$	ε	0	\$	ε
q_0									$\{(q_1, \$)\}$
q_1			$\{(q_1, 0)\}$	$\{(q_2, \varepsilon)\}$					
q_2				$\{(q_2, \varepsilon)\}$				$\{(q_3, \varepsilon)\}$	
q_3									



o dalla transizione

5.2 Computazione e linguaggi di un PDA

Data una parola w, un PDA accetta la parola se:

- possiamo scrivere $w = w_1 w_2 \dots w_m$ dove $w_i \in \Sigma \cup \{\varepsilon\}$
- esistono una sequenza di stati $r_0, r_1, \dots, r_m \in Q$ e
- *una sequenza di stringhe* $s_0, s_1, s_2, \dots, s_m \in \Gamma^*$

tali che

1. $r_0 = q_0$ e $s_0 = \varepsilon$ (inizia dallo stato iniziale e pila vuota)
2. per ogni $i = 0, \dots, m-1, (r_i + 1, b) \in \delta(r_i, w_{i+1}, a)$ con $s_i = at$ e $s_{i+1} = bt$ per qualche $a, b \in \Gamma_\varepsilon$ e $t \in \Gamma^*$ (l'automa rispetta la funzione di transizione)
3. $r_m \in F$ (la computazione *termina in uno stato finale*)

5.3 Accettazione per pila vuota

La nostra definizione di PDA accetta le parole *per stato finale.* L'accettazione per pila vuota costituisce un altro un altro modo per definire la *condizione di accettazione*: Un PDA accetta la parola w per pila vuota se esiste una computazione che

- consuma tutto l'input
- termina con la pila vuota ($s_m = \varepsilon$)

Equivalenza

Theorem 5.1 *Per ogni linguaggio accettato da un PDA per stato finale esiste un PDA che accetta per pila vuota, e viceversa*

Theorem 5.2 (Equivalenza con le grammatiche context-free) *Un linguaggio è context-free se e solo se esiste un PDA che lo riconosce.*

Sappiamo che un linguaggio è context-free se esiste una CFG che lo genera Mostreremo come trasformare la grammatica in un PDA e, viceversa, come trasformare un PDA in una grammatica.

Lemma 5.3 *Se un linguaggio è context-free, allora esiste un PDA che lo riconosce*

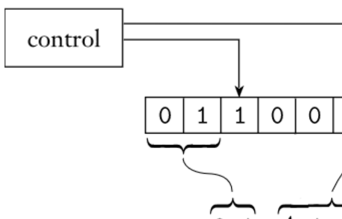
Per fare convertire una CFG in PDA è importante tenere a mente che ogni passo di derivazione produce una **stringa intermedia** contenente variabili e terminali. Data una stringa w il PDA dev'essere progettato in modo da stabilire se una serie di sostituzioni effettuate secondo le regole della CFG possa condurre dalla variabile iniziale a w. Per fare questo è fondamentale sfruttare il **non determinismo** del PDA per poter effettuare le sostituzioni che portano a w. Il PDA inizia scrivendo la variabile iniziale sulla pila.

Idea:

- Se L è context-free, allora esiste una CFG G che lo genera
- Mostriamo come trasformare G in un PDA equivalente P
- P è fatto in modo da *simulare* i *passi di derivazione* di G
- P accetta w se esiste una derivazione di w in G

5.3.1 Rappresentare le stringhe intermedie

- La **pila** memorizza le stringhe intermedie
- P trova le variabili nella stringa intermedia e fa le sostituzioni seguendo le regole di G



- **Idea**: metto nella pila solo **i simboli dalla prima variabile in poi**

- Ogni derivazione è una sequenza di **stringhe intermedie**

$S \to 0S1 \mid \varepsilon$ $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$ L'automa partirà dalla variabile iniziale, poi supponiamo che l'automa si trovi in $00S11$ Grazie alle transizioni elaborate avremo la pila

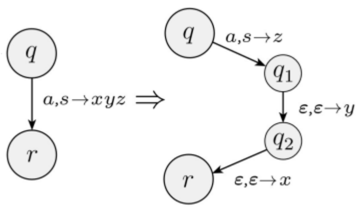
0 0 S In questo caso l'automa trova una variabile e applica una regola $(0S1 \mid \varepsilon)$ 1 1 \$ Ad indicare la fine della pila

5.4 Definizione informale del PDA

1. Inserisci il simbolo marcatore \$ e la variabile iniziale S sulla pila
2. Ripeti i seguenti passi:
   (a) Se la cima della pila è la variabile A: scegli una regola $A \to u$ e scrivi u sulla pila (qui si sfrutta il non determinismo)
   (b) Se la cima della pila è un terminale a: leggi il prossimo simbolo di input.
      - se sono uguali, procedi
      - se sono diversi, rifiuta
3. Se la cima della pila è \$: vai nello stato accettante

5.5 Notazione compatta

Supponiamo che il PDA vada da q z r quando legge a e fa il pop di s ... inserendo la **stringa** di tre caratteri $u = xyz$ sulla pila, per farlo sono necessarie 3 transizioni come



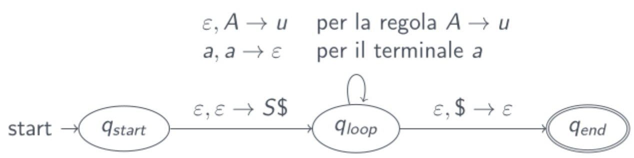
nell'immagine

Per implementare il push multiplo dobbiamo **aggiungere stati ausiliari**

5.6 Dimostrazione

Data $G = (V, \Sigma, R, S)$ definiamo $P = (Q, \Sigma, \Gamma, q_{start}, F)$

- $Q = \{q_{start}, q_{loop}, q_{end}\}$
- $\Gamma = \Sigma \cup V \cup \{\$\}$
- $F = \{q_{end}\}$
- funzione di transizione



5.7 Da PDA a grammatica Context-Free

Abbiamo un PDA P che riconosce il linguaggio Mostriamo come trasformare P in una CFG equivalente G

- Una stringa w accetta da P se fa andare P dallo stato iniziale a quello finale
- Progetteremo una grammatica che **fa un po' di più**
  - Una variabile A_{pq} per ogni coppia di stati p, q di P
  - A_{pq} genera tutte le stringhe che portano **da** p **con pila vuota** a q **con pila vuota**

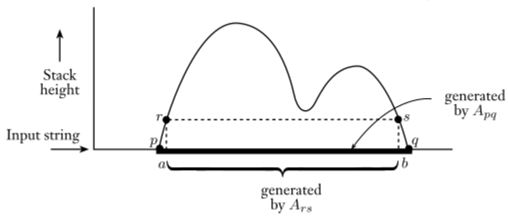
Come prima cosa, semplifichiamo P in modo che rispetti tre condizioni:

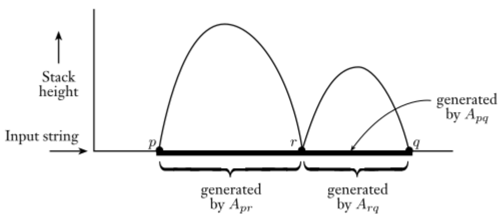
1. ha un unico stato accettante q_f
2. svuotiamo la pila prima di accettare
3. Ogni transizione **inserisce un unico simbolo sulla pila (push)** oppure **elimina un simbolo dalla pila (pop)**, ma non fa entrambe le cose contemporaneamente.

Andare da P a $q(1)$ Per andare da p con pila vuota a q con pila vuota:

- la prima mossa deve essere necessariamente un push
- l'ultima mossa deve essere un pop

Ci sono due casi:



1. Il simbolo inserito all'inizio viene eliminato alla fine:
   Per ogni coppia di transizioni che effettuano il push e il pop di un determinato simbolo andremo ad aggiungere una regola: $A_{pq} \to aA_{rs}b$
2. Oppure no, il simbolo inserito all'inizio viene rimosso prima di terminare la computazione.  in questo caso dobbiamo aggiungere la regola $\forall p, q, r \in Q$ $A_{pq} \to A_{pr}A_{rq}$ $A_{pq} \to \varepsilon$ per ogni $q \in Q$

Le regole di G Formalmente

- sia $P = (Q, \Sigma, \Gamma, \delta, q_0, \{q_r\})$
- costruiamo $G = (V\Sigma, R, A_{q0}, q_r)$ tale che
  - $V = \{A_{pq}, p, q \in Q\}$
  - Per ogni $p, q, r, s, \in Q$, $u \in \Gamma$ e $a, b \in \Sigma$, se $\delta(p, a, \varepsilon)$ contiene (r, u) e $\delta(s, b, u)$ contiene $(q, \varepsilon\ )$, aggiungi la regola $A_{pq} \to aA_{rs}b$
  - Per ogni $p, q, r, \in Q$, aggiungi la regola $A_{pq} \to A_{pr}A_{rq}$
  - Per ogni $p \in Q$, aggiungi la regola $A_{pp} \to \varepsilon$

Due dimostrazioni induttive

Lemma Se A_{pq} genera la stringa x, allora x può portare P da p con pila vuota a q con pila vuota

Lemma Se la stringa x può portare P da p con pila vuota a q con pila vuota, allora A_{pq} genera x.

Theorem 5.4 *Un linguaggio è context-free se e solo se esiste un PDA che lo rionosce*

- Sappiamo che un linguaggio è context-free se esiste una CFG che lo genera
- Abbiamo mostrato come trasformare una CFG in un PDA
- E viceversa, come trasformare un PDA in grammatica

6 Macchine di Turing

6.1 La Tesi di Church-Turing

Finora abbiamo visto DFA con una quantità finita di memoria. I PDA che hanno memoria illimitata ma ad accesso limitato (abbiamo solo operazioni *push* e *pop*). Questo comporta dei limiti alla loro capacità di computazione.

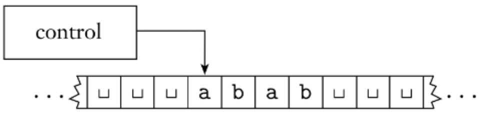
6.2 Macchina di Turing

È un modello proposto da Alan Turing nel 1936 per risolvere problemi matematici.

- Ha memoria illimitata, utilizza un nastro infinitoj/
- Non ci sono restrizioni di accesso alla memoria

Una macchina di Turing è un modello molto più preciso di un computer.
Tuttavia...

- Ci sono problemi che una Macchina di Turing **non può risolvere**
- questi problemi *vanno oltre le capacità di un computer*



la macchina di Turing è composte da

- un nastro infinito come **memoria illimitata**
- una testina che **legge e scrive** simboli sul nastro
- all'inizio il nastro contiene l'input
- per memorizzare informazione si **scrive sul nastro**
- la testina si può muovere **ovunque sul nastro**
- stati speciali per **accetta** e **rifiuta**

6.3 Primo esempio di TM

Costruiamo una macchina di Turing per il linguaggio

$$B = \left\{ w\#w \mid w \in \{0,1\}^* \right\}$$

- M_1deve accettare se l'input sta in B, e rifiutare altrimenti
- M_1= "su input w:

1. Si muove a zig-zag lungo il nastro, raggiungendo posizioni corrispondenti ai due lati di # per controllare se contengono lo stesso simbolo. In caso negativo, o se non trovi #, **rifiuta**. Barra gli elementi già controllati

2. Se tutti i simboli a sinistra di # sono stati controllati, verifica i simboli a destra di #. Se c'è qualche simbolo ancora da controllare **rifiuta**, altrimenti **accetta**

Questa descrizione della macchina di Turing M_1 ne illustra il funzionamento ma non ne mostra tutti i dettagli.

6.4 Automi finiti vs Macchine di Turing

1. Una TM può sia scrivere che leggere sul nastro
2. Una TM può muoversi si a a destra che a sinistra
3. Il nastro è infinito
4. Gli stati di rifiuto e accettazione hanno effetto immediato

6.5 Definizione formale

Una macchina di Turing è una tupla $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\}$

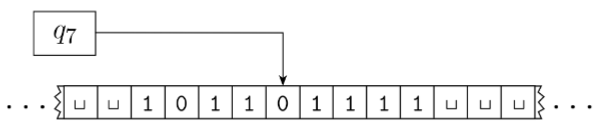
- Q è l'insieme finito di **stati**
- Σ è l'**alfabeto di input** che non contiene il simbolo **blank**
- Γ è l'**alfabeto del nastro** che contiene e Σ
- $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ è la **funzione di transizione**
- $q_0 \in Q$ è lo **stato iniziale**
- $q_{accept} \in Q$ è lo **stato di accettazione**
- $q_{reject} \in Q$ è lo **stato di rifiuto** (necessariamente diverso da q_{accept})

Nella macchina di Turing la funzione di transizione è la parte fondamentale.

6.6 Configurazioni

Lo stato corrente, la posizione della testina e il contenuto del nastro formano a **configurazione** di una TM. Dalla configurazione possiamo sapere la **prossima mossa** Le configurazioni sono rappresentate da una tripla uqv:

- q è lo **stato corrente**
- u è il contenuto del **nastro prima della testina**
- v è il contenuto del **nastro dalla testina in poi**
- la testina si trova **sul primo simbolo di** v



(la configurazione in figura è $1011q_701111$)

6.7 Computazione

La configurazione C_1 produce C_1 se la TM può passare da C_1 a C_2 in un passo Se $a, b, c \in \Gamma$, $u, v \in \Gamma^*$ e q_i, q_j sono stati, allora uaq_ibv produce uq_jacv se $\delta(q_i, b) = (q_i, c, L)$ uaq_ibv produce $uacq_jv$ se $\delta(q_i, b) = (q_i, c, R)$

- la **configurazione iniziale** con input w è q_0w
- in una **configurazione di accettazione** lo stato è q_{accept}
- in una **configurazione di rifiuto** lo stato è q_{reject}

6.8 Linguaggi Turing-riconoscibili

Una TM M **accetta** l'input w se esiste una **sequenza di configurazioni** $C_1, C_2, \ldots, C_k$ tale che:

- C_1 è la configurazione iniziale con input w
- Ogni C_i produce C_{i+1}
- C_k è una configurazione di accettazione

Il **Linguaggio riconosciuto da** M è l'insieme delle stringhe accettate da M

Theorem 6.1 *Un linguaggio è <i>Turing-riconoscibile</i> (o anche <i>ricorsivamente enumerabile</i>) se esiste una macchina di Turing che lo riconosce*

</center>

6.9 Linguaggi Turing-decidibili

Se forniamo un input ad una TM, ci sono **tre risultati possibili**:

- la macchina **accetta**
- la macchina **rifiuta**
- la macchina va in **loop** e non si ferma mai

La TM può non accettare sia rifiutando che andando in loop. Un TM che termina sempre la computazione è un **decisore** Un decisore **decide** un linguaggio se lo riconosce

Theorem 6.2 *Un linguaggio è <i>Turing-decidibile</i> (o anche <i>ricorsivo</i>) se esiste una macchina di Turing che lo decide*

6.10 Esempi

6.10.1 Esempio 1

TM che **decide** il linguaggio di tutte le stringhe di 0 la cui lunghezza è una potenza di 2:

$$A = \{0^{2^n} \mid n \geq 0\}$$

$M_1 =$ "su input w:

1. Scorri il nastro da sinistra a destra, cancellando ogni secondo 0
2. Se il nastro conteneva un solo 0, **accetta**
3. Se il nastro conteneva un numero dispari di 0, *rifiuta*
4. Ritorna all'inizio del nastro
5. Vai al passo 1"

6.10.2 Esempio 2

vedi Primo esempio di TM

6.10.3 Esempio 3

TM che esegue operazioni aritmetiche. Decide il linguaggio $C = \{a^i b^j c^k \mid k = i \cdot j$ e $i, j, k \geq 1\}$

$M_3 =$ "su input w:

1. Scorri il nastro da sinistra a destra e controlla se l'input sta in $a^+b^+c^+$. **rifiuta** se non lo è
2. Ritorna all'inizio del nastro
3. Barra una a e scorri a destra fino a trovare una b. Fai la spola tra b e c, barrando le b e le c fino alla fine delle b. Se tutte le c sono barrate e rimangono ancora b, *rifiuta*.
4. Ripristina le b barrate e ripeti
5. finché ci sono a da barrare.
6. Quando tutte le a sono barate, controlla se tutte le c sono barrate: se sì, **accetta**; altrimenti *rifiuta*."

6.10.4 Esempio 4

TM che risolve il problem degli **elementi distinti**. Prende in input una sequenza di stringhe separate da # e accetta se tutte le stringhe sono diverse. Decide il linguaggio: $D = \{\#x_1\#x_2\# \ldots \#x_l \mid x_j \in \{0,1\}^*$ e $x_i \neq x_j$ per ogni $i \neq j\}$

$M_4 =$ "su input w

1. Mette un segno sul simbolo del nastro più a sinistra. Se è un blank, **accetta**. Se è un #, continua con

2. Altrimenti, *rifiuta*.

3. Scorre a destra fino al successivo # e vi mette sopra un secondo segno. Se nessun # viene trovato, allora era presente solo x_1 : **accetta**.

4. Procede a zig-zag confrontando le due stringhe a destra dei # segnati. Se sono uguali, **rifiuta**

5. Sposta il segno più a destra sul successivo # alla sua destra.Se non trova nessun #, sposta il segno più a sinistra sul successivo # alla sua destra, e sposta il segno più a destra sul successivo #. Se on c'è un # dopo il segno più a destra, allora tutte le stringhe sono state confrontate: **accetta**

6. Vai alla fase 3.

6.11 Conclusioni

- I linguaggi A, B, C e D sono *decidibili*
- Tutti i linguaggi Turing-decidibili sono anche Turing-riconoscibili
- I linguaggi A, B, C e D sono anche *Turing-riconoscibili*
- Vedremo che ci sono linguaggi *Turing-riconoscibili ma non decidibili*

7 Varianti di Macchine di Turing

Esistono definizioni alternative delle macchine di Turing, chiamiamo **varianti** queste alternative. Tutte le varianti "ragionevoli" riconoscono *la stessa classe di linguaggi* Le Turing machine sono un modello *robusto.*

7.1 Macchine a nastro semi-infinito

- È una Turing Machine con un nastro *infinito solo verso destra.*
- L'input si trova *all'inizio del nastro*
- La testina parte dalla *posizione più a sinistra del nastro*
- Se M tenta di spostare la testina a sinistra quando si trova nella prima cella del nastro, allora *la testina rimane ferma.*

Theorem 7.1

Per ogni TM a nastro semi-infinito esiste una TM a nastro infinito equivalente

Per ogni TM a nastri infinito esiste una TM a nastro semi-infinito equivalente

7.2 Macchine Multinastro

È una TM con k nastri semi-infiniti, k testine di lettura e scrittura, l'input si trova sul nastro 1. Ad ogni passo scrive e si muove simultaneamente su tutti i nastri. Funzioni di transizione:

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

$$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, L, R, \ldots, L)$$

se lo stato è q_i e le testine leggono $a_1, \ldots, a_k$ allora scrivi $b_1, \ldots, b_k$ sui k nastri. Muovi ogni testina a sinistra o a destra come specificato.

Theorem 7.2 *Equivalenza Per ogni TM multinastro esiste una TM a singolo nastro equivalente*

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

1. Inizializza il nastro per rappresentare i k nastri:

$$\#w_1 w_2 \ldots w_n \#\#\# \ldots \#$$

2. Per simulare una mossa di M, scorri il nastro per determinare i simboli puntati dalle testine virtuali
3. Fai un secondo passaggio del nastro per aggiornare i nastri virtuali secondo la funzione di transizione di M
4. Se S sposta una testina virtuale a destra su un $\#$, allora M ha spostato la testina sulla parte vuota del nastro. Scrivi un e sposta il contenuto del nastro di una cella a destra.

5. Se si raggiunge una configurazione di accettazione, **accetta***, se si raggiunge una configurazione di rifiuto, *rifiuta*, altrimenti ripeti da 2.

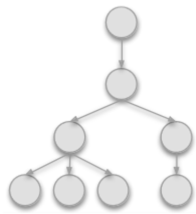
Corollario 7.2.1 *Un linguaggio è Turing-riconoscibile* ***se e solo se*** *esiste una macchina di Turing multinastro che lo riconosce.* $\Rightarrow$
Un linguaggio è Turing-riconoscibile se è riconosciuto da una TM con un nastro, che è un caso particolare di TM multinastro. $\Leftarrow$
Costruzione precedente

7.3 Macchine non deterministiche

Una TM non deterministica ha **più strade possibili** durante la computazione. Consideriamo macchine con un solo nastro semi-infinito. La funzione di transizione è:

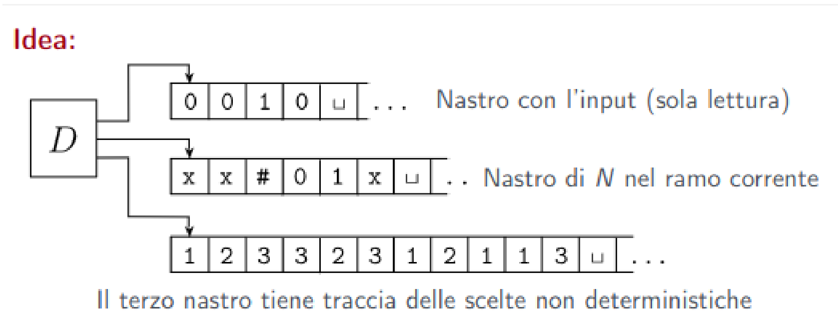
$$\delta : Q \times \Gamma \rightarrow 2^{(Q \times \Gamma \times \{L,R\})}$$

La computazione è un **albero** che descrive le scelte possibili. La macchina accetta se **esiste un ramo** che porta allo stato di accettazione



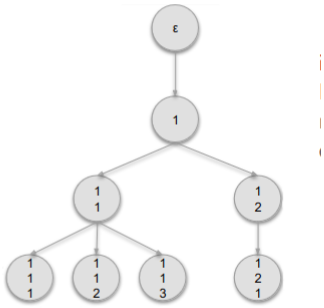
Tutti i rami devono essere esaminati fino a quando non viene trovato uno **stato di accettazione**. Come esaminare l'albero: *in ampiezza* o *in profondità*?

Theorem 7.3 *Per ogni TM* ***non deterministica*** *esiste una TM* ***deterministica*** *equivalente*

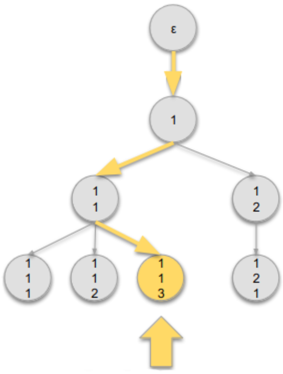


7.4 Come funziona il terzo nastro

Ad ogni nodo viene assegnato un **indirizzo**: una stringa sull'alfabeto $\Gamma_b = \{1, 2, \dots, b\}$, dove b è il massimo numero di figli dei nodi dell'albero:



Il nodo 113 si raggiunge prendendo il **primo** figlio della radice, seguito dal **primo** figlio di quel modo ed infine dal **terzo** figlio. Questo ordinamento può essere utilizzato per attraversare in modo efficiente l'albero in ampiezza.



7.5 Come funziona D

1. Inizialmente il nastro 1 contiene l'input w e i nastri 2 e 3 sono vuoti.

2. Copia il nastro 1 sul nastro 2 e inizializza la stringa sul nastro 3 a ε

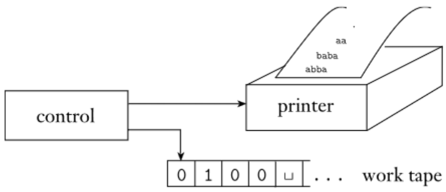
3. Usa il nastro 2 per simulare N con input w su un ramo di computazione. Prima di ogni passo di N, consulta il simbolo successivo sul nastro 3 per determinare quale scelta fare (tra quelle consentite). Se non rimangono più simboli sul nastro 3, o se questa scelta non è valida, interrompi questo ramo e vai alla fase 4. Vai ala fase 4. anche se si incontra una configurazione di rifiuto. Se viene trovata una configurazione di accettazione, **accetta**.

4. Sostituire la stringa sul nastro 3 con la stringa successiva nell'ordine delle stringhe. Simula il ramo successivo di N andando alla fase 2.

7.6 Conclusione

Corollario 7.3.1 *Un linguaggio è Turing-riconoscibile se e solo se esiste una macchina di Turing* ***non deterministica*** *che lo riconosce.*
$\Rightarrow$ *Un linguaggio è Turing-riconoscibile se è riconosciuto da una TM deterministica, che è un caso particolare di TM non deterministica.*
$\Leftarrow$ *Costruzione precedente*

7.7 Enumeratori



L'enumeratore è una macchina di Turing che dispone di una stampante.

- Un emumeratore E inizia con **nastro vuoto**.
- Di tanto in tanto, **invia una stringa alla stampante**
- Linguaggio **enumerato** da E : tutte le stringhe stampate
- E può generare le stringhe in qualsiasi ordine, anche con ripetizioni.

7.8 Equivalenza

Theorem 7.4 *Un linguaggio è Turing-riconoscibile se e solo se esiste un enumeratore che lo enumera*

Idea: dobbiamo mostrare che

- se esiste un enumeratore E, allora esiste una TM M che riconosce lo stesso linguaggio
- se esiste una TM M che riconosce il linguaggio, allora possiamo costruire un enumeratore

7.9 Macchine di Turing monodirezionali

- Una macchina di Turing con "resta ferma" invece di "muovi a sinistra"
- Funzione di transizione: $\delta : Q \times \Gamma \rightarrow Q \times \Gamma\{S, R\}$
- Ad ogni passo, la TM può lasciare ferma la testina o muoverla a destra
- *Non può muoversi a sinistra*

Quale classe di linguaggi riconosce?

7.10 Equivalenza con altri modelli

- Esistono altri modelli di computazione universali
- Alcuni sono molto simili alle macchine di Turing
- Altri sono molto diversi
- Hanno tutti una caratteristica comune: **accesso senza restrizioni** ad una **memoria illimitata**
- Sono **tutti equivalenti tra loro**

8 Varianti di Macchine di Turing

Contesto storico
David Hilbert, discorso al Secondo Congresso Internazionale di Matematica, Parigi, 1900

- Definisce 23 problemi matematici come sfida per il nuovo secolo
- **Decimo problema**: creare un algoritmo per determinare se un polinomio ha una radice intera
- Il presupposto era che **l'algoritmo dovesse esistere**, e bastava trovarlo
- Ora sappiamo che questo problema è **non risolvibile algoritmicamente**

8.1 Cos'è un algoritmo

La **nozione intuitiva** di algoritmo esiste da migliaia di anni, mentre la **definizione formale** di algoritmo è stata data per la prima volta nel XX secolo. Senza una definizione formale, è quasi **impossibile provare** che un algoritmo non può esistere.

8.1.1 Tesi di Church-Turing

1936 Church pubblica un formalismo chiamato λ-calcolo per definire algoritmi.

1936 Turing pubblica le specifiche per una *macchina astratta* per definire algoritmi

1952 Kleene mostra che i due modelli sono **equivalenti**

1970 Matiyasevich dimostra che l'algoritmo per stabilire se un polinomio ha radici intere **non esiste**

8.1.2 Il decimo problema di Hilbert

Il decimo teorema di Hilbert con la nostra terminologia $D = \{p \mid p \text{ è un polinomio avente radice intera}\}$

- Il problema diventa "**D è un insieme decidibile?**"
- Possiamo mostrare che D è **Turing-riconoscibile**
- Partiamo da un problema più semplice

$D_1 = \{p \mid p \text{ è un polinomio su } x \text{ avente radice intera}\}$

8.2 Come descrivere una Turing Machine

- Descrizione formale
  - Dichiara esplicitamente tutto quanto
  - Estremamente dettagliata
  - Da evitare a tutti i costi!!!
- Descrizione implementativa

  - Descrive a parole il movimento della testina e la scrittura sul nastro
  - Nessun dettaglio sugli stati
- Descrizione di alto livello
  - Descrizione a parole dell'algoritmo
  - Nessun dettaglio implementativo
  - Da utilizzare sempre, se non indicato altrimenti

8.2.1 Notazione formale per macchine di Turing

- L'input è sempre una **stringa**
- Se l'input e un oggetto, deve essere rappresentato come una stringa
  - Polinomi, grammatiche, automi, ecc...
  - L'input può essere una combinazione di diversi tipi di oggetti.
- Un oggetto O codificato come stringa è $\langle O \rangle$.
- Una sequenza di oggetti $O_1, O_2, \ldots, O_k$ è codificata come $\langle O_1, O_2, \ldots, O_k \rangle$
- L'algoritmo viene descritto con un **testo**, indentato e con struttura a blocchi.
- La prima riga dell'algoritmo descrive l'**input** macchina

Esempio: un problema di grafi I grafi sono strutture dati che vengono usate estensivamente in informatica.
Ci sono migliaia di problemi computazionali che sono importanti per le applicazioni e che si possono modellare con i grafi.
Vedremo ora che cos'è un grafo, e studieremo alcuni problemi sui grafi che sono interessanti per la loro **classe di complessità**.

Definizione di base Un grafo **non orientato** (detto anche **indiretto**) G è una coppia (V, E) dove

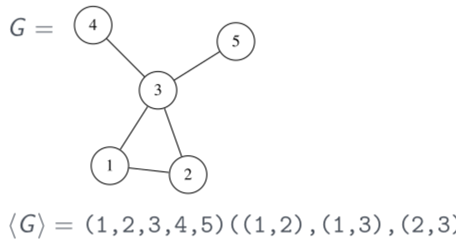
- $V = \{v_1, v_2, \ldots, v_n\}$ è un insieme finito e non un vuoto di vertici
- $E \subseteq \Big\{\{u, v\} \mid u, v \in v\Big\}$ è un insieme di **coppie non ordinate**, ognuna delle quali corrisponde ad un **arco non orientato** del grafo.

Definizione Un grafo è **connesso** se ogni nodo può essre raggiunto da ogni altro nodo tramite gli archi del grafo.

Problema Il linguaggio $A = \{\langle G \rangle \mid G \text{ è un grafo connesso }\}$ è decidibile?
Definiamo una Turing Machine che decide A

Descrizione di alto livello
$M =$ "Su input $\langle G \rangle$, la codifica di un grafo G:

1. **Seleziona** il primo nodo G e lo marca.
2. **Rpeti** la fase seguente fino a quando non vengono marcati nuovi nodi:
3. Per ogni nodo in G, **marcalo** se è connesso con un arco ad un nodo già marcato
4. **Esamina** tutti i nodi di G: se sono tutti marcati, **accetta**, altrimenti *rifiuta*



Codifica del grafo Codifica di G: lista dei nodi + lista degli archi
M verifica che l'input **sia una codifica di un grafo**

- Se l'input non è nella forma corretta, **rifiuta**
- Se l'input codifica un grafo, prosegue con la fase 1

9 Linguaggi decidibili

Obiettivi

- Studiare il potere degli algoritmi
- Capire quali problemi sono risolvibili da un algoritmo e quali no
- In questa lezione iniziamo considerando problemi **decidibili**

9.1 Problemi sui linguaggi regolari

9.1.1 Problema dell'accettazione

Ovvero, testare se un DFA accetta una stringa $A_{DFA} = \{\langle B, w\rangle \mid B$ è un DFA che accetta la stringa $w\}$ B accetta w **se e solo se** $\langle B, w\rangle$ appartiene ad A_{DFA} Mostrare che il linguaggio è **decidibile** equivale a mostrare che il problema computazionale è**decidibile**.

Theorem 9.1 *A_{DFA} è decidibile*

Idea: definire una TM che decide A_{DFA} $M =$ "Su input $\langle B, w\rangle$, dove B è un DFA e w una stringa:

1. Simula B su input w
2. Se la simulazione termina in uno stato finale, **accetta**. Se termina in uno stato non finale, *rifiuta*."

Dimostrazione

- la codifica di B è una lista dei componenti: Q, Σ, δ, q_0 e F
- fare la simulazione è facile

Theorem 9.2 *A_{NFA} è decidibile*

$A_{NFA} = \{\langle B, w\rangle \mid B$ è un ε-NFA che accetta la strina $w\}$ **Idea**: usiamo la TM M che decide A_{DFA} come subroutine.

Dimostrazione: $N =$ "Su input $\langle B, w\rangle$, dove B è un ε-NFA e w una stringa:

1. Trasforma B in un DFA equivalente C usando la costruzione per sottoinsiemi
2. Esegui M con input $\langle C, w\rangle$
3. Se M accetta, **accetta**; altrimenti, *rifiuta*."

N è un decisore per A_{NFA}, quindi A_{NFA} è **decidibile**.

Theorem 9.3 *A_{REX} è decidibile*

$A_{REX} = \{\langle R, w\rangle \mid R$ è una espressione regolare che genera la stringa w $\}$ **Idea**: usiamo la TM N che decide A_{NFA} come subroutine

Dimostrazione: $P =$ "Su input $\langle R, w\rangle$, dove R è una espressione regolare e w una stringa:

1. Trasforma Rin un ε-NFA equivalente C usando la procedura di conversione
2. Esegui N con input $\langle C, w\rangle$
3. Se N accetta, **accetta**; altrimenti *rifiuta*."

P è un decisore per A_{REX}, quindi A_{REX} è **decidibile**
Riassumendo

- Ai fini della **decidibilità**, è equivalente dare in input alla TM un DFA, un ε-NFA o una espressione regolare
- La TM è in grado di con ertire una odifica nell'altra
- **Ricorda**: mostrare che il linguaggio è **decidibile** equivale a mostrare che il problema computazionale è **decidibile**.

9.2 Test del vuoto

Negli esempi precedenti dovevamo decidere se una stringa appartenesse o no ad un linguaggio. Ora vogliamo determinare se un automa finito accetta una **qualche** stringa $E_{DFA} = \{\angle A\rangle \mid A \text{ è un DFA e } L(A) = \varnothing\}$ Puoi descrivere un algoritmo per eseguire questo test?

9.3 E_{DFA} è decidibile

Dimostrazione: verifica se c'è uno stato finale che può essere raggiunto a partire dallo stato iniziale. $T =$ "Su input $\langle A\rangle$, la codifica di un DFA A:

- **Marca** lo stato iniziale di A.
- **Ripeti** la fase seguente fino a quando non vengono marcati nuovi stati:
- **marca** ogni stato di A che ha una transizione proveniente da uno stato già marcato
- Se nessuno degli stati finali è marcato, **accetta**; altrimenti *rifiuta*."

9.4 Test di equivalenza

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

- costruiamo una DFA C che accetta solo le stringhe che sono accettate da A o da B, ma non da entrambi
- se $L(A) = L(B)$ allora C non accetterà nulla
- il linguaggio di C è la **differenza simmetrica** di A e B

9.5 EQ_{DFA} è decidibile

Dimostrazione:

- la **differenza simmetrica** di A e B è $L(C) = \Big(L(A) \cap \overline{L(B)}\Big) \cup \Big(\overline{L(A)} \cap L(B)\Big)$
- i linguaggi regolari sono **chiusi** per unione, intersezione e complementazione
- $F =$ "Su input $\langle A, B \rangle$ dove A e B sono DFA:
  1. Costruisci il DFA C per differenza simmetrica
  2. Esegui T, la TM che decide E_{DFA} con input $\langle C \rangle$
  3. Se T accetta, **accetta**; altrimenti *rifiuta*."

9.6 Problemi per linguaggi Context-free

$A_{CFG} = \{\langle G, w \rangle \mid G$ è una CFG che genera la stringa $w\}$
Idea: costruiamo una TM che provi tutte le derivazioni di G per trovarne una che genera w
Perché questa strategia non funziona?

9.7 A_{CFG} è decidibile

Se la CFG è in forma normale di Chomsky, allora ogni derivazione di w è lunga **esattamente** $(2|w| - 1)$ **passi**.
Le TM possono convertire le grammatiche nella forma normale di Chomsky!
Dimostrazione: $S =$ "Su input $\langle G, w \rangle$, dove G è una CFG è w una striga:

1. Converti G in forma normale di Chomsky
2. Elenca tutte le derivazioni di $2|w| - 1$ passi. Se $|w| = 0$, elenca tutte le derivazioni di lunghezza 1
3. Se una delle derivazioni genera w, **accetta**; altrimenti *rifiuta*."

9.8 Test del vuoto

$E_{CFG} = \{\langle G \rangle \mid A$ è una CFG ed $L(G) = \varnothing\}$

- **Problema**: non possiamo usare S del teorema precedente. **Perché no?**
- Bisogna procedere in modo diverso!

Idea: stabilisci per ogni variabile se è in grado di generare una stringa di terminali
$R =$ "Su input $\langle G \rangle$, la codifica di una CFG G:

1. **Marca** tutti i simboli terminali di G
2. **Ripeti** la fase seguente fino a quando non vengono marcate nuove variabili:
3. **Marca** ogni variabile A tale che esiste una regola $A \to U_1 \dots U_k$ dove ogni simbolo $U_1 \dots U_k$ è già stato marcato.

4. Se la variabile iniziale non è marcata, **accetta**; altrimenti *rifiuta*."

Theorem 9.4 *EQ_{CFG} è decidibile*

$EQ_{CFG} = \{\langle G, H \mid G$ e H sono CFG e $L(G) = L(H)\}$ **Idea**:

- Usiamo la stessa tenica di EQ_{DFA}
- Calcoliamo la **differenza simmetrica** di G e H per provare l'equivalenza

STOP!

- Le CFG non sono chiuse per complementazione ed intersezione
- EQ_{CFG} **non è decidibile!**

9.9 Relazioni tra classi di linguaggi

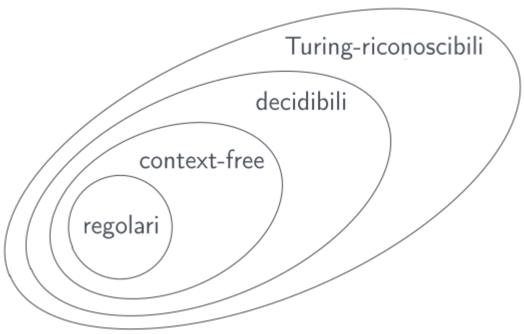
Theorem 9.5 *ogni CFL è decidibile*

Domanda:

- è facile simulare la pila con una TM
- sappiamo che le TM nondeterministiche possono essere simulate da una TM deterministica

Non basta semplicemente simulare un PDA con una TM?
Quali altre opzioni abbiamo?

- Dato un CFG L, sia G la grammatica per L
- Costruiamo la TM S che decide A_{CFG}
- La TM che decide L è
  $M_G =$ "Su input w:
  1. Esegui la TM S con un input $\langle G, w\rangle$
  2. se S accetta, **accetta**; altrimenti, *rifiuta*



Queste non sono solo classi di linguaggi, ma anche **classi di capacità computazionale**.

10 Indecidibilità

10.1 Metodo della diagonalizzazione

Tale metodo è un metodo scoperto da Cantor nel 1873, e serve per confrontare le dimensioni di **insiemi infiniti**.

due insiemi finiti hanno la stessa dimensione se gli elementi di un insieme possono essere accoppiati agli elementi dell'altro insieme

Corrispondenze Abbiamo due insiemi A e B e una funzione $f : A \to B$

- f è **iniett** se non mappa mai elementi diversi nello stesso punto: $f(a) \neq f(b)$ ogniqualvolta che $a \neq b$
- f è **suriettiva** se tocca ogni elemento di B: Per ogni $b \in B$ esiste $a \in A$ tale che $f(a) = b$
- Una funzione iniettiva e suriettiva è chiamata **biettiva**: è un modo per **accoppiare** elementi di A con elementi di B

Definizione A e B hanno la **stessa cardinalità** se esiste una funzione **biettiva** $f : A \to B$

10.2 Esempio: naturali vs numeri pari

- $\mathbb{N} = \{0, 1, 2, \dots\}$, insieme dei numeri naturali.
- $\mathbb{E} = \{0, 2, 4, \dots\}$, insieme dei numeri pari.

Quale dei due insiemi è il più grande?

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

Altri esempi

- $\mathbb{Q}$ è numerabile?
- $\mathbb{R}$ è numerabile?
- Dato un alfabeto finito Σ, Σ^* è numerabile?
- L'insieme di **tutte le macchine di Turing** è numerabile?
- L'insieme di **tutte le sequenze binarie infinite** è numerabile?
- Dato un alfabeto finito Σ, l'insieme di **tutti i linguaggi** su Σ^* è numerabile?

Corollario

- L'insieme di tutte le macchine di Turing è **numerabile**
- L'insieme di tutti i linguaggi è **non numerabile**
- **devono** esistere linguaggi non riconoscibili da una macchina di Turing

10.3 Un risultato fondamentale

*Esiste un problema specifico che è algoritmicamente **irrisolvibile***

- Problemi di interesse non solo teorico, ma anche pratico.
- Esempio: Verifica del software:

Verificare che un programma è corretto non è risolvibile algoritmicamente

Theorem 10.1 A_{TM} *è indecidibile*

$A_{TM} = \{\langle M, w\rangle \mid M$ è una TM che accetta la stringa $w\}$

- **Chiarimento**: A_{TM} è Turing-riconoscibile
- Conseguenza: i riconoscitori **sono più potenti** dei decisori
- U = "Su input $\langle M, w\rangle$, dove M è una TM e w una stringa: 1. Simula M su input w 2. Se la simulazione raggiunge lo stato di accettazione, **accetta**; se raggiunge lo stato di rifiuto, *rifiuta*."
- U è un **riconoscitore**. Perché non è un **decisore**?

10.4 Macchina Universale di Turing

U è un esempio di **Macchina Universale di Turing** Introdotta da Alan Turing nel 1936, può simulare **qualsiasi macchina di Turing** a partire dalla sua descrizione.

Theorem 10.2 A_{TM} *è indecidibile*

$A_{TM} = \{\langle M, w\rangle \mid M$ è una TM che accetta la stringa $W\}$

Dimostrazione

- Per contraddizione. Assumiamo A_{TM} decidibile per poi trovare una contraddizione
- Supponiamo H decisore per A_{TM}
- Cosa fa H con input $\langle M, w\rangle$?

$H(\langle M, w\rangle)$ = **accetta** se M accetta w, *rifiuta* altrimenti

- Definiamo una TM D che usa H come subroutine
- D = "Su input $\langle M\rangle$, dove M è una TM:

1. Esegue H su input $\langle M, \langle M \rangle \rangle$
2. Dà in output l'opposto dell'output di H, se H accetta, *rifiuta*; se H rifiuta, **accetta**

- Cosa fa D con input $\langle D \rangle$? $D(\langle D \rangle)$ = **accetta** se D non accetta $\langle D \rangle$, *rifiuta* altrimenti.

Questa è una contraddizione!

Comprendere la dimostrazione

1. H accetta $\langle M, w \rangle$ esattamente quando M accetta w
   (a) Banale: abbiamo assunto che H esista e decida A_{TM}
   (b) M rappresenta **qualsiasi** TM e w è una **qualsiasi** stringa
2. D rifiuta $\langle M \rangle$ esattamente quando M acetta $\langle M \rangle$
   (a) Cosa è successo a w?
   (b) w è solo una stringa, come $\langle M \rangle$. Tutto ciò che stiamo facendo è definire quale stringa dare in input alla macchina.
3. D rifiuta $\langle D \rangle$ esattamente quando D accetta $\langle D \rangle$
   Questa è la contraddizione
4. Dove si usa la diagonalizzazione?

10.5 Un linguaggio non Turing-riconoscibile

- Abbiamo visto che A_{TM} è **Turing-riconoscibile**
- Sappiamo che l'insieme di **tutte le TM** è **numerabile**
- Sappiamo che l'insieme di **tutti i linguaggi** è **non numerabile**
- Di conseguenza **deve** esistere un linguaggio non **Turing-riconoscibile**

C'è ancora una cosa che dobbiamo fare prima di poter mostrare un linguaggio non **Turing-riconoscibile**.
Mostreremo che se un linguaggio e il suo complementare sono Turing-riconoscibili, allora il linguaggio è decidibile.
Un linguaggio è **co-Turing riconoscibile** se è il complementare di un linguaggio Turing-riconoscibile

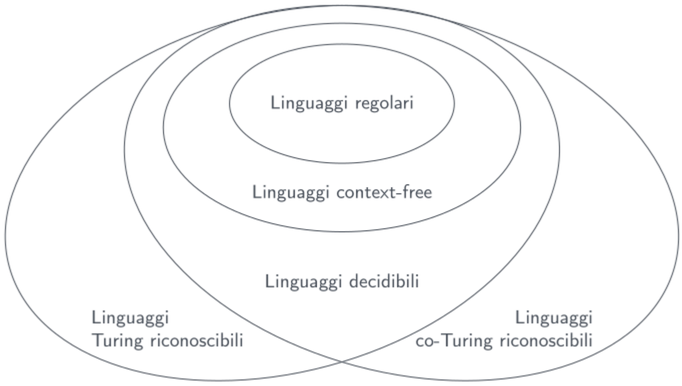
Theorem 10.3 *Un linguaggio è decidibile se e solo se è Turing-riconoscibile e co-Turing riconoscibile.*

Dimostrazione

- Dobbiamo dimostrare entrambe le direzioni
- Se A è decidibile, allora sia A che $\overline{A}$ sono Turing-riconoscibili
  - Il complementare di un linguaggio decidibile è decidibile!
- Se A e $\overline{A}$ sono Turing-riconoscibili, possiamo costruire un decisore per A

10.6 $\overline{A_{TM}}$ non è Turing-riconoscibile

Se il complementare di A_{TM} fosse Turing-riconoscibile, allora A_{TM} sarebbe decidibile. Sappiamo che A_{TM} non è decidibile, quindi il suo complementare non può essere Turing-



riconoscibile!

11 Riducibilità

Vediamo un altro problema indecidibile

Il problema della fermata $HALT_{TM} = \{\langle M, w\rangle \mid M$ è una TM che si ferma su input $w\}$ Come possiamo dimostrare che $HALT_{TM}$ è indecidibile? Possiamo provare con la **diagonalizzazione** Sappiamo che A_{TM} è indecidibile: **possiamo usare questo fatto per semplificare la dimostrazione?**

Riducibilità

- Una **riduzione** è un modo per trasformare un problema in un altro problema
- Una soluzione al secondo problema può essere usata per **risolvere il primo problema**
- Se A è riducibile a B, e B è decidibile, allora A è **decidibile**
- Se A è riducibile a B, e A è decidibile, allora A è **decidibile**

11.1 Dimostrazione per riduzione

Queste dimostrazioni sono usate per dimostrare che un problema è indecidibile:

1. **Assumi** che B sia decidibile
2. **Riduci** A al problema B
   - costruisci una TM che usa B per risolvere A
3. si A è indecidibile, allora questa è una **contraddizione**
4. L'assunzione è sbagliata e B è indecidibile

11.2 Il problema del vuoto

$E_{TM} = \{\langle M\rangle \mid M$ è una TM tale che $L(M) = \varnothing\}$

- La dimostrazione è per contraddizione e riduzione di A_{TM}
- Chiamiamo R la TM che decide E_{TM}
- Useremo R per costruire la TM S che decide A_{TM}

11.3 Stabilire se un linguaggio è regolare

$REGULAR_{TM} = \{\langle M\rangle \mid M$ è una TM tale che $L(M)$ è regolare $\}$

- La dimostrazione è per contraddizione e riduzione di A_{TM}
- Chiamiamo R la TM che decide $REGULAR_{TM}$
- Useremo R per costruire la TM S che decide A_{TM}
- Capire come possiamo usare R per implementare S è meno ovvio di prima

11.4 Il problema dell'equivalenza

$EQ_{TM} = \{\langle M_1, M_2 \mid M_1, M_2 \text{ TM tali che } L(M_1) = L(M_2)\}$

- La dimostrazione è per contraddizione e riduzione di E_{TM} (problema del vuoto)
- Chiamiamo R la TM che decide EQ_{TM}
- Useremo R per costruire la TM S che decide E_{TM}

11.5 Riducibilità mediante funzione

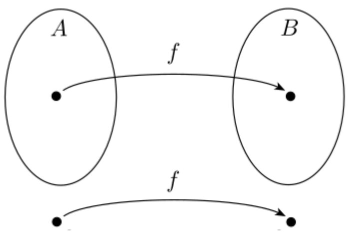
Trasforma istanze del problema A in istanze del problema B mediante una **funzione calcolabile**

- Chiarisce e formalizza la riducibilità

Definizione $f : \Sigma^* \to \Sigma^*$ è una **funzione calcolabile** se esiste una TM M che su input w, termina la computazione avendo solo $f(w)$ sul nastro

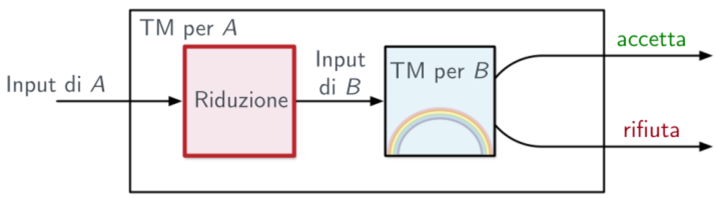
- Le operazioni aritmetiche sugli interi sono funzioni calcolabili
- Le trasformazioni di macchine di Turing possono essere funzioni calcolabili

Un linguaggio A è **riducibile mediante funzione** al linguaggio B ($A \leq_m B$), se esiste una **funzione calcolabile** $f : \Sigma^* \to \Sigma^*$ tale che Per ogni $w : w \in A$ se e solo se $f(w) \in B$



f è la **riduzione** da A a B

Se esiste una **riduzione** da A e B, possiamo risolvere A usando una soluzione per B:



11.6 Proprietà delle riduzioni

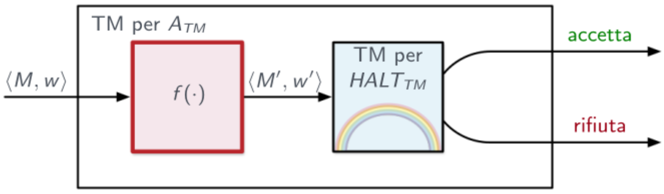
Teorema Se $A \leq_m B$ e B è ???, allora A è ???

Teorema Se $A \leq_m B$ e A è allora B è ???

11.7 Il problema della fermata(2)

$HALT_{TM} = \{\langle M, w\rangle \mid M$ è una TM che si ferma su input $w\}$

- Possiamo dimostrare che $A_{TM} \leq_m HALT_{TM}$?
- Qual è l'input della funzione di riduzione?
- Qual è l'output?
- Quali proprietà devono rispettare?

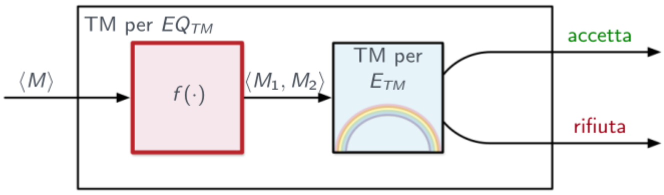


M accetta w se e solo se M' si ferma su w'

11.8 Il problema dell'equivalenza(2)

$$EQ_{TM} = \{\langle M_1, M_2\rangle \mid L(M_1) = L(M_2)\}$$

- Possiamo dimostrare che $E_{TM} \leq_m EQ_{TM}$?
- Qual è l'input della funzione di riduzione?
- Qual è l'output?
- Quali proprietà devono rispettare?

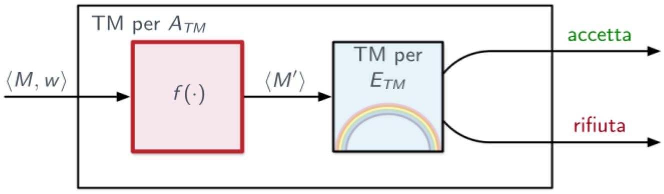


$L(M) = \varnothing$ se e solo se $L(M_1) = L(M_2)$

11.9 Il problema del vuoto(2)

$E_{TM} = \{\langle M\rangle \mid M$ è una TM tale che $L(M) = \varnothing\}$

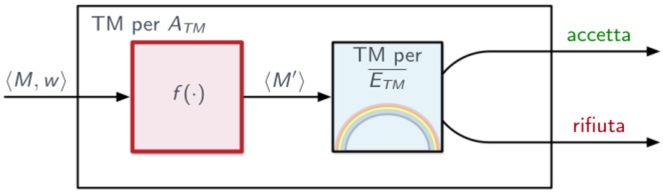
- Possiamo dimostrare che $A_{TM} \leq_m E_{TM}$?
- Qual è l'input della funzione di riduzione?
- Qual è l'output?
- Quali proprietà devono rispettare?



M accetta w se e solo se $L(M') = \varnothing$

STOP!!!

Non sappiamo come ridurre il problema dell'accettazione al problema del vuoto!



M accetta w se e solo se $L(M') \neq \varnothing$

Proprietà delle riduzioni (2)

Teorema Se $A \leq_m B$ e B è ???, allora A è ???

Teorema Se $A \leq_m B$ e A è ??? allora B è ???

12 Complessità di tempo

12.1 Misure di Complessità

Comsideriamo il linguaggio $A = \{0^k 1^k \mid k \geq 0\}$

- Che tipo di linguaggio è?
- È **decidibile**?
- Quanto **tempo** serve ad una TM a nastro singolo per decidere questo linguaggio?

Definizione

- Sia M una TM **deterministica** che si **ferma** su tutti gli input
- Il **tempo di eseguzione** (o **complessità di tempo**) di M è la funzione $f : \mathbb{N} \mapsto \mathbb{N}$ tale che $f(n)$ è il numero massimo di passi che M utilizza su un input di lunghezza n.
- Se $f(n)$ è il tempo di esecuzione di M, diciamo che M è una TM **di tempo** $f(n)$
- Useremo n per rappresentare la **lunghezza dell'input**
- Ci interesseremo dell'**analisi del caso pessimo**

12.2 Notazione O-grande

- **Analisi asintotica**: valuta il tempo di esecuzione su input grandi
- Considera solo il **termine di ordine maggiore** e **ignora i coefficienti**

  Definizione 1 *Date due funzioni f, g, diciamo che $f(n) = O(g(n))$ se esistono interi positivi c, n_0 tali che per ogni $n \geq n_0$*

  $$f(n) \leq cg(n)$$

  *$g(n)$ è un **limite superiore asintotico** per $f(n)$*

12.3 Analisi di complessità

Analizziamo questa TM per $A = \{0^k 1^k \mid g \geq 0\}$
M_1 = "Su input w:

1. Scorrere il nastro e *rifiuta* se trova uno 0 a destra di un 1
2. Ripete finché il nastro contiene almeno uno 0 e un 1
3. Scorre il nastro cancellando uno 0 e un 1
4. Se rimane almeno uno 0 dopo che ogni 1 è stato cancellato, o se rimane almeno un 1 dopo che ogni 0 è stato cancellato, *rifiuta*. Altrimenti, se non rimangono né 0 né 1 sul nastro, **accetta**"

12.4 Classi di complessità di tempo

Definizione 2 *Sia $t : \mathbb{N} \mapsto \mathbb{N}$ una funzione*
*La **classe di complessità di tempo** $TIME(T(N))$ è l'insieme di tutti i linguaggi che sono **decisi** da una **TM in tempo** $O(t(n))$*

- *Questo è diverso dalle classi di linguaggi discusse in precedenza, che si concentravano sulla **computabilità***
- *Questa classificazione si concentra sul **tempo necessario** per decidere il linguaggio.*

Possiamo fare di meglio? Dall'analisi che abbiamo fatto, sappiamo che

$$A = \{0^k 1^k \mid k \geq 0\}$$

appartiene alla classi di complessità di tempo $TIME(n^2$, perché M_1 decide A in tempo $O(n^2)$

MA Esiste una macchina che decide A in modo **asintoticamente più veloce**?

Miglioriamo M_1 **Idea**: cancelliamo metà degli 0 e metà degli 1 ad ogni scansione $M_2 =$ "Su input W:

1. Scorre il nastro e **rifiuta** se trova uno 0 a destra di un 1
2. Ripete finché il nastro contiene almeno uno 0 ed un 1
3. Scorre il nastro e controlla se il numero totale di 0 e 1 rimasti è pari o dispari. Se è dispari, *rifiuta*
4. Scorre il nastro, cancellando prima ogni secondo 0 a partire dal primo 0, poi cancellando ogni secondi 1 a partire dal primo 1.
5. se nessuno 0 e nessun 1 rimangono sul nastro **acetta**. Altrimenti *rifiuta*

Possiamo fare ancora meglio?

- Possiamo trovare una TM che decide A in $O(n)$?
- **Problema**: non esiste una TM **a nastro singolo** che è in grado di decidere A in tempo $O(n)$.
- Possiamo farlo se la TM **ha un secondo nastro**
- **Importante**: la complessità di tempo dipende dal **modello di calcolo**
- La **Tesi di Church-Turing** implica che tutti i modelli di calcolo "ragionevoli" siano equivalenti.
- **in pratica**: discuteremo qunto questa differenza sia importante (o meno) per il nostro sistema di classificazione più avanti.

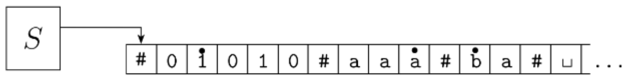
Idea: usiamo il secondo nastro per contare 0 e 1 $M_2 =$ "Su input w:

- Scorre il nastro 1 e *rifiuta* se trova uno 0 a destra di un 1
- Scorre i simboli 0 sul nastro 1 fino al primo 1. Contemporaneamente, copia ogni 0 sul nastro 2.
- Scorre i simboli 1 sul nastro 1 fino alla fine dell'input. Per ogni 1 letto sul nastro 1, cancella uno 0 sul nastro 2. Se ogni 0 è stato cancellato prima di aver letto tutti gli 1, *rifiuta*
- Se tutti gli 0 sono stati cancellati, **accetta**. Se rimane qualche 0, *rifiuta*

12.5 Relazione di complessità tra modelli

12.5.1 Singolo nastro vs. Multinastro

Theorem 12.1 *Sia $t(n)$ una funzione tale che $t(n) \leq n$. Ogni TM **multinastro** di tempo $t(n)$ ammette una TM equivalente a **nastro singolo** di tempo $O(t^2(n))$.*

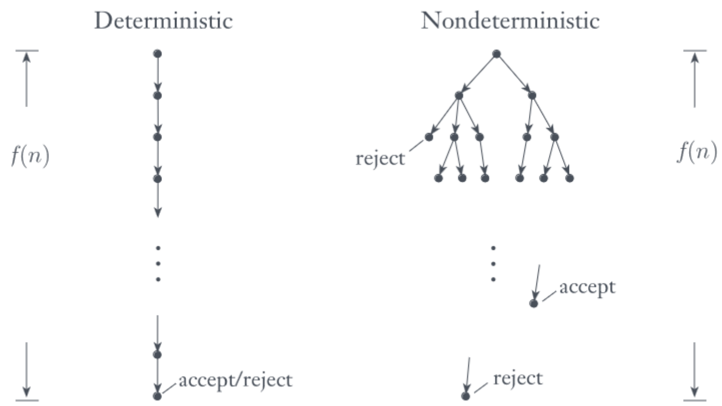
- Ricordiamo la dimostrazione di come convertire una TM da multinastro a nastro 

singolo.
- Dobbiamo determinare quanto tempo ci vuole per simulare ogni passo della macchina multinastro sulla TM a nastro singolo.

12.5.2 TM non deterministiche

Theorem 12.2 *Sia N una TM non deterministica che è anche un **decisore**. Il **tempo di esecuzione** di N è la funzione $f : \mathbb{N} \mapsto \mathbb{N}$ tale che $f(n)$ è il massimo numero di passi che N usa per ognuno dei rami di computazione, su input di lunghezza n.*

Nota Bene la definizione di tempo di esecuzione per le TM non deterministiche non è destinato a corrispondere ad un qualche dispositivo di calcolo reale. È uno strumento teorico che utilizziamo per comprendere i problemi computazionali.



12.5.3 Determinismo vs. Non determinismo

Theorem 12.3 *Sia $t(n)$ una funzione tale che $t(n) \leq n$. Ogni TM **non deterministica** di tempo $t(n)$ ammette una TM equivalente a **nastro singolo** di tempo $2^{O(t(n))}$.*

Dimostrazione

- Sia N una TM non deterministica di tempo $t(n)$
- Costruiamo una TM deterministica D che simula N.
- D è una TM **multinastro**. Qual è la complessità quando viene convertita in una TM a nastro singolo?

13 Classe P

Riassunto:

- Differenza di tempo **polinomiale** tra TM a nastro singolo e multi-nastro
- Differenza di tempo **esponenziale** tra TM deterministichee non deterministiche

Una differenza **polinomiale** è considerata piccola

- Tutti i modelli di calcolo deterministici "ragionevoli" sono **polinomialmente equivalenti**
- "Ragionevole" è definito in modo approssimativo, ma include modelli che assomigliano molto ai computer reali

Una differenza **esponenziale** è considerata grande

Definizione 3 *P è l classe di linguaggi che sono decidibili in **tempo polinomiale** da una TM deterministica a singolo nastro*

$$P = \bigcup_k TIME(n^k)$$

- P è invariante per i modelli di calcolo **polinomialmente equivalenti** ad una TM deterministica
- P corrisponde approssimativamente ai problemi che sono **realisticamente risolvibili** da un computer

Per dimostrare che un problema/algoritmo è in P

- Descrivi l'algoritmo per fasi numerate
- Dai un limite superiore polinomiale al numero di fasi che l'algoritmo esegue per un input di lunghezza n
- Assicurati che ogni fase possa essere completata in tempo polinomiale su un modello di calcolo deterministico ragionevole
- L'input deve essere codificato in modo ragionevole

13.1 Due problemi in P

Raggiungibilità di un grafo $PATH = \{\langle G, s, t\rangle \mid G$ grafo che contiene un cammino da$s$ a $t\}$

Numeri relativamente primi $RELPRIME = \{(x, y) \mid 1$ è il massimo comun divisore di x e $y\}$

Theorem 13.1 *ogni linguaggi context-free è un elemento in P*

- Abbiamo dimostrato che ogni CFL è **decidibile**
  - L'algoritmo nella dimostrazione è **esponenziale**
- La soluzione polinomiale usa la **programmazione dinamica**
- La complessità è $O(n^3)$

14 La classe NP

14.1 Giochiamo a Domino

Disponete in file le **tessere del domino** che vi sono state consegnate in modo da usare tutte le tessere. **È un problema facile o difficile da risolvere?**

Definizione 4 *Un problema è **trattabile** (facile) se esiste un **algoritmo efficiente** per risolverlo*

- Gli algoritmi efficienti sono **algoritmi con complessità polinomiale**, il loro tempo di esecuzione è $O(n^2)$ per qualche costante k.
- Avere complessità polinomiale è una **condizione minima** per considerare un algoritmo efficiente
- Un algoritmo con complessità più che polinomiale (per esempio esponenziale) è un algoritmo **non efficiente** perché non è scalabile.

14.2 Mostriamo che Domino[1] è trattabile

Obiettivo Trovare un algoritmo polinomiale per **Domino[1]**

- Formulazione del problema in termini di **linguaggio**
- Definizione di una Macchina di Turing che lo **decide**
- Analisi di **complessità** della macchina di Turing (o dell'algoritmo)

14.3 Un linguaggio e una riduzione

$D_1 = \{\langle B\rangle \mid B$ è un insieme di tessere del domino, ed esiste un allineamento che usa tutte le tessere $\}$

- Usiamo una **Riduzione mediante funzione** per trovare l'algoritmo polinomiale.
- Riduciamo D_1 ad un **problema su grafi**
- ... per il quale sappiamo che **esiste un algoritmo polinomiale**

14.4 Dalle tessere al grafo

Definizione 5 *Un grafo (non orientato) è una coppia (V,E) dove:*

- $V = \{v_1, v_2, \dots, v_n\}$ *è un insieme finito e non vuoto di **vertici***
- $E \subseteq \big\{\{u, v\} \mid u, v \in V\big\}$ *è un insieme di **coppie non ordinate** ognuna delle quai corrisponde ad un **arco** del grafo.*

Grafo del dominio

- **Vertici**: i numeri che si trovano sulle tessere
  - $V = \{$⚀, ⚁, ⚂, ⚃$\}$
- **Archi**: le tessere del domino
  - $E = \{$⚂⚃, ⚀⚁, ⚃⚁, ⚃⚀, ⚂⚁$\}$

14.5 Dominio[1] è un problema su grafi!

Camino Euleriamo: percorso di un grafo che attraversa **tutti gli archi** una sola volta.

Il problema del Cammino Euleriano

$$EULER = \{\langle G \rangle \mid \text{ è un grafo che possiede un cammino Euleriano } \}$$

- $EULER$ è un problema classico di **teoria dei grafi**
- Esistono **algoritmo polinomiali** per risolverlo.

14.6 Algoritmo di Fleury

- Scegliere un vertice con **grado dispari** (un vertice qualsiasi se tutti pari)
- Scegliere un arco tale che la sua cancellazione **non sconnetta il grafo**
- **Passare** al vertice nell'altra estremità dell'arco scelto.
- **Cancellare** l'arco del grafo
- **Ripetere** i tre precedenti finché non eliminate tutti gli archi

Complessità Su un grafo con n archi, l'algoritmo di Fleury impiega tempo $O(n^2)$

14.7 Complessità di Domino[1]

- L'algoritmo di Fleury risolve $EULER$ in tempo **polinomiale**
- La riduzione ci dice che $D_1 \leq_m EULER$
- Quanto tempo serve per risolvere il problema D_1?