ELEMENTI D'INFORMATICA TEORICA

LEGENDA:

- Le descrizioni delle MdT sono scritte con delle parentesi <> simili a queste ma più alte, non le ho trovate, quindi ho usato delle parentesi tonde.
- δ: Q – {q} -> Q e δ: Q\{q} -> Q significa che delta è una funzione che va da Q in Q, ma Q\{q} e Q – {q} significa che Q ≠ q, quindi Q non può essere q.
- (not)MdT significa che la macchina è complementata.
- (un elemento) appartiene a un insieme = ϵ, è incluso in un insieme = ⊆, il prodotto cartesiano di un insieme con l'altro va in Q = $Q \times \sum \rightarrow Q$.

Capitolo 0 (conoscenza generale, semi-facoltativo)

0.1:

L'informatica teorica si concentra su 3 aree tradizionali della teoria della computazione: automi, compatibilità e complessità.

Teoria della complessità: Ci sono vari tipi di problemi per i computer: certi sono semplici, altri difficili. Ad esempio, un problema di riordinamento è semplice al confronto con uno di programmazione degli orari (per delle lezioni, per il lavoro, ecc...).

"Cosa rende dei problemi computazionalmente difficili e altri semplici?" è questa una domanda cardine di questa teoria, anche se attualmente non si ha risposta.

Un obiettivo importante della teoria della complessità è stato quello di schematizzare i problemi in base alla loro difficoltà computazionale. Usando questo schema si può dimostrare che c'è un metodo per dimostrare che dei problemi sono computazionalmente difficili, anche se non si è capaci di dimostrarlo.

Teoria della computabilità: Dei matematici hanno scoperto che certi problemi di base non possono essere risolti da computer. Un esempio di questo fenomeno è il problema di determinare se un'affermazione matematica sia vera o falsa. Le teoria della computabilità e complessità sono molto collegate. Nella teoria della complessità, l'obiettivo è classificare i problemi come semplici e difficili, mentre la teroria della computabilità, la classificazione dei problemi è per quelli solvibili e non. La teoria della computabilità introduce diversi concetti usati nella teoria della complessità.

Teoria degli automi: La teoria degli automi segue le definizioni e le proprietà dei modelli matematici per la computazione. La teoria degli automi è un posto eccellente per iniziare a studiare la teoria della computazione. Le teorie della computabilità e complessità richiedono una definizione precisa di un computer. La teoria degli automi permette di praticare con le definizioni formali di computazione e introduce concetti rilevanti ad altre aree non-teoriche dell'informatica teorica.

Capitolo 1:

1.1:

Automi a stati finiti: Gli automi a stati finiti sono buoni modelli di computer con una memoria estremamente limitata. Gli automi sono delle rappresentazioni grafiche di certi problemi e sono rappresentati da un grafo orientato, ogni automa ha un diagramma degli stati, descritto da uno stato iniziale e da stati accettanti e rigettanti.

Definizione formale di automa a stati finiti: Un'automa a stati finiti ha varie parti. Ha un insieme di stati e regole per passare da uno stato all'altro, dipende dal simbolo in entrata. Ha uno stato iniziale e un insieme di stati accettanti. La definizione formale dice che un'automa a stati finiti è una lista di 5 oggetti: insieme di stati, alfabeto in entrata, funzione per spostarsi, stato iniziale e stati accettanti. Questa lista, in matematica, è definita 5-tupla (quintupla).

Definizione (1.5): Un'automa a stati finiti è una 5-tupla (Q, ∑, δ, q_0, F):

1) Q è un insieme finito per gli stati;

2) ∑ è un insieme finito contentente l'alfabeto;

3) δ: $Q \times \sum \rightarrow Q$ è la funzione di transizione;

4) $q_0 \epsilon$ (appartiene) Q è lo stato iniziale;

5) F ⊆ (è incluso in) Q ed è l'insieme degli stati accettanti.

La definizione formale descrive precisamente cosa s'intende per automa a stati finiti.

Definizione formale di computazione: Finora abbiamo descritto gli automi a stati finiti informalmente, usando i diagrammi degli stati*, e con una definizione formale, come una 5-tupla.

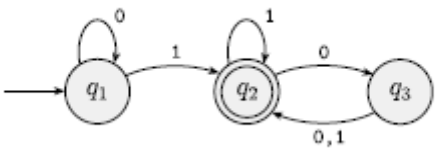
*Diagramma degli stati:

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

Sia M = (Q, ∑, δ, q_0, F) un'automa a stati finiti e sia w = $w_1w_2...w_n$ una stringa dove ogni w_i sia un membro dell'alfabeto. Allora M accetta w se una sequenza di stati r_0, ..., r_n in Q esiste con 3 condizioni:

1) $r_0 = q_0$
2) $\delta(r_i, w_{i+1}) = r_{i+1}$ per ogni i = 0,1, ..., n-1
3) r_n € F

La prima condizione dice che la macchina inizia nello stato iniziale. La seconda condizione dice che la macchina va dallo stato corrente allo stato successivo in accordo con la funzione di transizione. La terza condizione dice che la macchina accetta i suoi input se finisce in uno stato accettante. Dirò che M riconosce A se A = {w|M accetta w}.

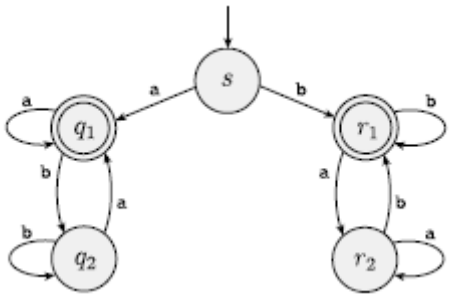


Esempio della definizione 1):

1. Q = {q_1, q_2, q_3}
2. ∑ = {0, 1}
3. $\delta(q_1, 0) = q_1$
   $\delta(q_1, 1) = q_2$
   $\delta(q_2, 0) = q_3$
   $\delta(q_2, 1) = q_2$
   $\delta(q_3, 0) = q_2$
   $\delta(q_3, 1) = q_2$

4. q_1 è lo stato iniziale
5. F = {q_2}

2)



1. Q = {s, q_1, q_2, r_1, r_2}
2. ∑ = {a, b}
3. s è lo stato iniziale
4. F = {q_1, q_2}
5. δ(s, a) = q_1
   δ(s, b) = r_1
   δ(q_1, a) = q_1
   δ(q_1, b) = q_2
   δ(q_2, a) = q_1
   δ(q_2, b) = q_2
   δ(r_1, a) = r_2
   δ(r_1, b) = r_1
   δ(r_2, a) = r_2
   δ(r_2, b) = r_1

Definizione (1.16): Un linguaggio è definito regolare se un'automa a stati finiti lo riconosce.

Rappresentare un automa a stati finiti: Provo a rappresentare l'automa a stati finiti usando "l'automa lettore" come esempio.

Immmagina di essere l'automa che legge la stringa simbolo per simbolo:

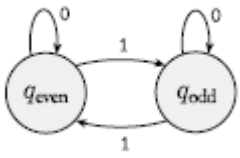
- che informazione sulla stringa letta finora hai bisogno per fare una decisione su cosa accettare o rigettare;
- puoi aggiornare quest'informazione se un altro simbolo d'input arriva?
- gli stati immagazzineranno quest'informazione (non devi immagazzinare tutte le informazioni, solo quelle cruciali).

Ad esempio, si consideri l'alfabeto {0,1} e il linguaggio richiede che tutte le stringhe abbiano un numero dispari di 1. Vuoi costruire l'automa a stati finiti E_1 capace di riconoscere questo linguaggio. Immagina di essere l'automa, inizi ad avere una stringa in input (simbolo per simbolo) composta da 0 e 1. Semplicemente ricordi quanti 1 hai visto per ora, se sono pari o dispari e tieni conto di quest'informazione per quando leggi un nuovo simbolo. Se leggi un 1, cambia la risposta, se leggi 0 lasci invariato. Dunque, le possibilità sarebbero:

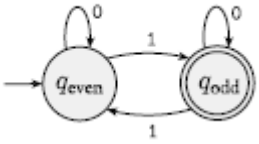
1. pari
2. dispari

Allora assegni uno stato per ogni possibilità.

Dopo, assegni le transizioni vedendo in che stato vai da una possibilità a un'altra leggendo un simbolo.



Infine imposti lo stato iniziale in corrispondenza alle possibilità associate, avendo visto 0 simboli quello sarà lo stato iniziale, ossia corrisponde allo stato q_{even} perché 0 è un numero pari. Lo stato accettante, invece, sarà q_{odd} e come si può vedere lo si raggiunge solo se si legge 1.



Le operazioni regolari: Nella teoria della computazione, gli oggetti sono i linguaggi e gli attrezzi includono operazioni specifiche designate per manipolarli. Definiamo 3 operazioni sui linguaggi, chiamate operazioni regolari e le usiamo per studiare le proprietà dei linguaggi regolari.

Definizione 1.23: Siano A e B linguaggi. Definiamo le operazioni regolari di unione, concatenazione e star nel modo seguente:

- Unione: A U B = {x|x ∈ A o x ∈ B}, prende tutte le stringhe presenti in A e B, unendole in un solo linguaggio;
- Concatenazione: A + B – A o B {xy| x ∈ A e y ∈ B} (si può usare anche il + al posto di quella o), pone una stringa di A prima di una stringa di B in ogni modo possibile, per ottenere le nuove stringhe del linguaggio;
- Star: A* = {x_1... x_k|k ≥ 0 e ogni x_i ∈ A}, quest'operazione è diversa dalle altre 2 perché si applica a un unico linguaggio, infatti è un'operazione unaria. Lavora attaccando ogni numero di stringhe in A insieme per ottenere una stringa nel nuovo linguaggio. Perché "ogni numero" include 0 come una possibilità, la stringa vuota ε è sempre un membro di A*, non importa che linguaggio sia A.

Esempio delle operazioni:

Let the alphabet Σ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$. If $A = \{\mathtt{good}, \mathtt{bad}\}$ and $B = \{\mathtt{boy}, \mathtt{girl}\}$, then

$A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\}$,

$A \circ B = \{\mathtt{goodboy}, \mathtt{goodgirl}, \mathtt{badboy}, \mathtt{badgirl}\}$, and

$A^* = \{\varepsilon, \mathtt{good}, \mathtt{bad}, \mathtt{goodgood}, \mathtt{goodbad}, \mathtt{badgood}, \mathtt{badbad}, \mathtt{goodgoodgood}, \mathtt{goodgoodbad}, \mathtt{goodbadgood}, \mathtt{goodbadbad}, \ldots\}$.

Teorema 1.25: La classe dei linguaggi regolari è chiusa sotto l'operazione di unione. In altre parole, se A_1 e A_2 sono linguaggi regolari, anche $A_1 \cup A_2$

Idea della prova: So che A_1 e A_2 sono regolari perché c'è un certo automa M_1 che riconosce A_1 e un certo automa M_2 che riconosce A_2. Per dimostrare che $A_1 \cup A_2$ è regolare dimostriamo che un certo automa M riconosce il linguaggio. Questa è una prova per costruzione. Si costruisce M dagli automi che riconoscono i linguaggi, quindi M dovrebbe accettare gli stessi input di entrambe le macchine e lavora simulandole, accettando se ognuna delle simulazioni lo fa.

Dimostrazione: Consideriamo due linguaggi regolari A e B. Per definizione, l'unione insiemistica dei due linguaggi sarà L = A ∪ B = {x | x ∈ A o x ∈ B}.

Dato che i due linguaggi sono regolari, esisteranno anche due automi a stati finiti L e N.

Per dimostrare l'esistenza di un automa in grado di riconoscere il linguaggio L, in questo caso gli stati iniziali di accettazione dovranno essere o uno solo o entrambi stati finali.

Grazie a questa considerazione è possibile identificare le caratteristiche dell'automa M = (Q, I, δ, q_0, F) in grado di riconoscere il linguaggio L:

- $Q = Q_1 \times Q_2$;
- $\delta = Q_1 \times Q_2 \times I \rightarrow Q_1 \times Q_2$;
- $q_0 = (q_{0A}, q_{0B})$
- $F = \{(q_A, q_B) \mid (q_A \in F_A) \text{ o } (q_B \in F_B)\} = (F_A \times Q_1) \cup (F_B \times Q_2)$

In conclusione, siccome L è un linguaggio formale associato ad un accettore a stati finiti, dev'essere necessariamente un linguaggio regolare.

Teorema 1.26: La classe di linguaggi regolari è chiusa sotto l'operazione di concatenazione.

In altre parole, se A_1 e A_2 sono regolari anche $A_1 \circ A_2$.
Dimostrazione: Dati due linguaggi regolari L_1 e L_2, la loro concatenazione L $L_1 \circ L_2$ è un linguaggio regolare. Infatti, siano A_1 $\langle \Sigma_1, Q_1, \delta_{N1}, q_{01}, F_1 \rangle$ e A_2 $\langle \Sigma_2, Q_2, \delta_{N2}, q_{02}, F_2 \rangle$, due DFA che accettano L_1 e L_2. Costruiamo da A_1 e A_2 un automa A $\{\Sigma, Q, \delta_N, q_0, F\}$ che riconosce il linguaggio $L_1 \circ L_2$.

Costruzione:

1. $\Sigma = \Sigma_1 \cup \Sigma_2$;
2. $Q = Q_1 \cup Q_2$;
3. $F \{ F_2 \text{ se } \varepsilon \neq L(A_2)\} = F_1 \cup F_2$;
4. $q_0 = q_{01}$;
5. δ_N è definita come:
   - $\delta_N(q, a) = \delta_1(q, a)$, $\forall q \in Q_1$ escluso F_1, $a \in \Sigma_1$
   - $\delta_N(q, a) = \delta_1(q, a) \cup \delta_2(q_{02}, a)$, $\forall q \in F_1$, $a \in \Sigma$
   - $\delta_N(q, a) = \delta_2(q, a)$, $\forall q \in Q_2$, $a \in \Sigma_2$.

1.2:

Non-determinismo: Il nondeterminismo è un concetto utile che ha avuto un grande impatto sulla teoria della computazione. Fino ad adesso abbiamo avuto delle macchine che leggendo uno specifico input sarebbero andate in uno stato successivo ben specifico (determinismo), ora avermo una macchina in cui certe scelte possono esistere in un punto qualsiasi, significa che non si saprà a forza quello che sarà lo stato successivo (nondeterminismo).

Il nondeterminismo è una generalizzazione del determinismo, quindi ogni automa a stati finiti (DFA) è automaticamente un'automa finito non determinisco (NFA). Gli NFA hanno caratteristiche in più:

- Ogni stato di un DFA ha esattamente una sola freccia uscente di transizione per ogni simbolo dell'alfabeto, un NFA può averne più di una per ogni simbolo dell'alfabeto, anzi, in certi casi possono esserci NFA con 0 stati.
- I DFA sulle frecce hanno a forza i simboli dell'alfabeto, invece, gli NFA possono avere sia i simboli dell'alfabeto sia ε.

Un NFA computa nel seguente modo: riceve in input una stringa e arriva in uno stato con più strade per arrivare al prossimo. Ad esempio, il mio NFA si trova nello stato q_1 e il prossimo simbolo che entra è 1. Dopo averlo letto la macchina si divide in più copie di sè e segue tutte le possibilità in parallelo. Se uno stato con la ε viene incontrato da una freccia uscente succede qualcosa di simile. Senza leggere alcun input la macchine si divide in molteplici copie, una segue ognuna delle frecce-ε mentre una rimane allo stato corrente. Poi la macchina prosegue nondeterministicamente come prima.

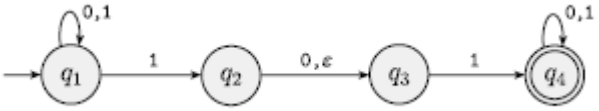
Un altro modo per pensare alla computazione non deterministica è vederla come un albero di possibilità. La radice dell'albero corrisponde all'inizio della computazione. Ogni punto di salto nell'albero corrisponde a un punto nella computazione in cui la macchina ha più scelte. La macchina accetta se almeno uno dei salti di computazione finisce in uno stato accettante.

Gli NFA sono utili in vari aspetti. Ogni NFA può essere convertito in un DFA e costruire un NFA è qualche volta più semplice che costruire direttamente un DFA. Un NFA può essere più piccolo della sua controparte in DFA.

Definizione formale di automa non deterministico a stati finiti: La definizione formale di questi automi è simile a quella degli DFA. L'unica grande differenza è che la funzione di transizione accetta la stringa vuota e nell'alfabeto è accettato anche ε, infatti l'alfabeto sarà: $\sum_\varepsilon = \sum \cup \{\varepsilon\}$. P(Q) (si chiama powerset) corrisponde all'insieme di tutti i sottostati possibili di Q (definizione di powerset), dunque: $\delta: Q \times \sum_\varepsilon \rightarrow P(Q)$.

(1.37): Un'automa non deterministico a stati finiti è una 5-tupla $(Q, \sum, \delta, q_0, F)$, dove:

- Q è l'insieme finito di stati;
- ∑ è l'alfabeto finito;
- $\delta: Q \times \sum_\varepsilon \rightarrow P(Q)$ è la funzione di transizione;
- q_0 è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati accettanti.



Esempi 1):

1. $Q = \{q_1, q_2, q_3, q_4\}$
2. $\sum = \{0, 1\}$
3. q_1 è lo stato iniziale
4. $F = \{q_4\}$
5. δ è dato da:

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

Equivalenza tra NFA e DFA: DFA e NFA riconoscono le stesse classi di linguaggio. Quest'equivalenza è sia soprendente sia utile. Sorprende perché gli NFA appaiono più potenti dei DFA, quindi potremmo aspettarci che gli NFA riconoscono più linguaggi. Utile perché descrivere un NFA da un dato linguaggio è più semplice che descriverlo con un DFA. 2 macchine sono equivalenti se reconoscono lo stesso linguaggio.

Teorema 1.39: Ogni NFA ha un'equivalente DFA.

Idea della dimostrazione: Se un linguaggio è riconosciuto da un NFA, allora dobbiamo mostrare l'esistenza di un DFA che riconosca lo stesso linguaggio. L'idea è convertire l'NFA in un'equivalente DFA che simula l'NFA.

Se k è il numero di stati, allora l'NFA ha 2^k sottoinsieme di stati. Ogni sottinsieme corrisponde a una delle possibilità che il DFA deve ricordare, così il DFA simulante l'NFA avrà 2^k stati.

Dimostrazione: Sia N = (Q, ∑, δ, q_0, F) l'NFA che riconosce il linguaggio A, costruisco il DFA M = (Q', ∑', δ', q_0', F') che riconosce A. Prima di costruirlo interamente consideriamo il caso in cui N non abbia delle frecce ε.

1. Q' = P(Q). Ogni stato di M è un insieme di stati di N.
2. Per R ∈ Q e a ∈ ∑, sia δ'(R,a) = {q ∈ Q| q ∈ δ(r,a) per qualche r ∈ R}.
3. q_0' = {q_0}. M inizia nello stato corrispondente alla collezione che contiente lo stato iniziale di N.
4. F' = {R ∈ Q'| R contiene gli stati accettanti di N}. La macchina M accetta se uno degli stati possibili che N può accettare.

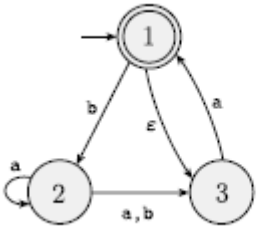
Considerando le frecce ε, invece, si deve aggiungere un'altra notazione. Per ogni stato R di M, definisco E(R) come la collezione di stati che possono essere raggiunti dai membri di R andando solamente nelle frecce ε, includendo i membri di R stesso. Formalmente, per R ⊆ Q ho:

E(R) = {q|q può essere raggiunto da R viaggiando attraverso 0 o più frecce ε}.

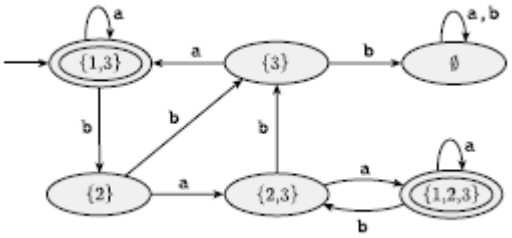
Anche la funzione di transizione è diversa, rimpiazzando E(δ(r,a)) si ottiene:

δ'(R,a) = {q ∈ Q| q ∈ E(δ(r,a)) per qualche r ∈ R}

Si deve anche modificare lo stato iniziale nel caso in cui avesse delle frecce ε che lo raggiungono. Cambiando q_0' con E({q_0}) si ottiene il possibile stato iniziale. Ho costruito il DFA M che simula l'NFA N, dunque la dimostrazione è completa.



Esempio 1): È un NFA, la sua conversione in DFA è la seguente:



Corollario 1.40: Un linguaggio è regolare se e solo se degli NFA lo riconoscono.

Chiusura sotto le operazioni regolari: I linguaggi regolari hanno una chiusura per le operazioni regolari (unione, concatenazione e star), il tutto vale anche per il nondeterminismo.

Teorema 1.45: La classe di linguaggi regolari è chiusa sotto l'operazione di unione.

Idea della dimostrazione: Ho i linguaggi A_1 e A_2 e voglio provare che A_1 U A_2 è regolare. L'idea è prendere 2 NFA N e M per i linguaggi e combinarli nell'NFA K. K deve accettare i suoi input se N o M accetta quegli input. La nuova macchina ha un nuovo stato iniziale che salta agli stati iniziali delle vecchie macchine con le frecce ε. In questo modo, la nuova macchina indovina non deterministicamente quale delle 2 macchine accetta l'input. Se una di loro è accettante, anche K accetterà.

Dimostrazione: Sia N = (Q', ∑', δ', q', F') riconosce A_1 e M = (Q'', ∑'', δ'', q'', F'') riconosce A_2. Costruisco K = (Q, ∑, δ, q, F) riconosce A_1 U A_2:

1. Q = {q} U Q' U Q'', gli stati di K sono tutti gli stati di N e M, con l'aggiunta di un nuovo stato iniziale q;
2. Lo stato q è lo stato iniziale di K;
3. L'insieme di stati accettanti F = F' U F'', gli stati accettanti di K sono tutti gli stati accettanti di N e M.
4. Definisco δ così che per ogni $q_n \in Q$ e ogni $a \in \sum_\varepsilon$
   - $\delta(q_n,a) \to \delta_{1/2}(q_n,a)$ $q_n \in Q'/Q''$
   - $\delta(q_n,a) \to \{q',q''\}$ $q_n = q$ e $a = \varepsilon$
   - $\delta(q_n,a) \to 0$ $q_n = q$ e $a \neq \varepsilon$

Teorema 1.47: La classe dei linguaggi regolari è chiusa sotto l'operazione di concatenazione.

Idea della dimostrazione: Ho i linguaggi A e B e voglio dimostrare che A°B è regolare. L'idea è prendere 2 NFA N e M per A e B e combinarli nel nuovo NFA K. Gli stati accettanti di N diventano stati con frecce ε che saltano allo stato iniziale di M. Gli stati accettanti, invece, sono solo quelli di M.

Dimostrazione: Sia N = (Q', ∑', δ', q', F') riconosce A_1 e M = (Q'', ∑'', δ'', q'', F'') riconosce A_2. Costruisco K = (Q, ∑, δ, q, F) riconosce A_1 o A_2:

1. Q = Q' U Q'', gli stati di K sono tutti gli stati di N e M;
2. Lo stato q' è lo stato iniziale di N;
3. Gli stati accettanti F'' sono gli stati accettanti di K, tutti gli stati accettanti di M.
4. Definisco δ così che per ogni $q_n \in Q$ e ogni $a \in \sum_\varepsilon$:
   - $\delta(q_n,a) \to \delta'(q_n,a)$ $q_n \in Q'$ e $q_n \notin F'$
   - $\delta(q_n,a) \to \delta'(q_n,a)$ $q_n \in F'$ e $a \neq \varepsilon$
   - $\delta(q_n,a) \to \delta'(q_n,a)$ U $\{q''\}$ $q_n \in F'$ e $a = \varepsilon$
   - $\delta(q_n,a) \to \delta''(q_n,a)$ $q_n \in Q$

Teorema 1.49: La classe dei linguaggi regolari è chiusa sotto l'operazione star.

Idea della dimostrazione: Ho il linguaggio A' e voglio dimostrare che anche A'* è regolare. Prendo l'NFA N per A' e lo modifico in modo che riconosca A'*.

Costruisco N* come N, però ha delle frecce ε in più che portano dagli stati accettanti allo stato iniziale di N, in questo modo letta una parola l'automa N può tornare all'inizio e leggerne un'altra. Comunque si deve modificare N e per farlo si dovrebbe aggiungere una possibilità di accettare ε nel linguaggio, l'alternativa migliore è aggiungere un nuovo stato iniziale per N*.

Dimostrazione: Sia N = (Q', ∑', δ', q', F') che riconosce A'. Costruisco N* = (Q, ∑, δ, q, F) che riconosce A'*.

1. Q = {q} U Q'. Gli stati di N* sono gli stessi di N + un nuovo stato iniziale.
2. q è il nuovo stato iniziale
3. F = {q} U F'. Gli stati accettanti sono uguali ai vecchi + un nuovo stato iniziale e accettante.
4. Definisco δ per ogni q_n ∈ Q e ogni a ∈ $\sum_\varepsilon$
   - δ(q_n,a) -> δ'(q_n,a) q_n ∈ Q' e q_n non ∈ F'
   - δ(q_n,a) -> δ'(q_n,a) q_n ∈ F' e a ≠ ε
   - δ(q_n,a) -> δ'(q_n,a) U {q'} q_n ∈ F' e a = ε
   - δ(q_n,a) -> {q'} q_n = q e a = ε
   - δ(q_n,a) -> stato morto q_n = q e a ≠ ε

1.3

Espressioni regolari: Posso usare le operazioni regolare per costruire espressioni che descrivono linguaggi e (le espressioni) vengono chiamate "espressioni regolari", un esempio è (0 U 1)0*. Le espressioni regolari hanno un ruolo importante nelle applicazioni dell'informatica teorica.

Il valore di un'espressione regolare è un linguaggio. Nel caso sopraccitato il valore consiste di tutte le stringhe che iniziano con 0 o 1 seguite da ogni numero di 0. Si ottiene il risultato dissezionando l'espressione nelle sue parti. Innanzitutto, i simboli 0 e 1 sono abbreviano gli insiemi {0} e {1}, quindi (0 U 1) = ({0} U {1}) = {0,1}. La parte 0* abbrevia {0}* e il suo valore è il linguaggio che consiste di tutte le stringhe contenenti ogni numebro di 0. Seconda cosa, il simbolo di concatenazione ° è spesso implicito, infatti, in questo caso l'espressione sarebbe: (0 U 1)°0*. La concatenazione attacca le stringhe dalle 2 parti per ottenere il valore dell'intera espressione.

Definizione formale di un'espressione regolare (1.52): Dico che R è un'espressione regolare se R è:

1. a per qualche a nell'alfabeto ∑
2. ε
3. Ø
4. R' U R'' dove R' e R'' sono espressioni regolari
5. R' ° R'' dove R' e R'' sono espressioni regolari
6. R'* dove R' è un'espressione regolare.

Non si deve confondere l'espressione regolare ε e Ø. ε rappresenta il linguaggio che contiene una sola stringa, Ø rappresenta il linguaggio che non ne contiene.

Questa definizione è induttiva, in quanto R' e R'' sono più piccoli di R, quindi non potrebbe esserci una definizione circolare (da R definisco R).

Equivalenza con gli automi a stati finiti: Le espressioni regolari e gli automi a stati finiti sono equivalenti nella loro potenza descrittiva. Questo fatto è sorprendente perché gli automi a stati finiti e le espressioni regolari sembrano superficialmente diversi.

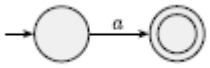
Teorema 1.54: Un linguaggio è regolare se e solo se qualche espressione regolare lo descrive. Questo teorema ha 2 direzioni. Ogni direzione verrà posta e dimostrata come lemma differente.

Lemma 1.55: Se un linguaggio è descritto da un'espressione regolare, allora è regolare.

Idea della dimostrazione: Dico che ho un'espressione regolare R che descrive qualche linguaggio A. Mostro come convertire R in un NFA che riconosce A.

Dimostrazione: Converto R in un NFA N. Considero i 6 casi della definizione formale di espressione regolare:

1. R = a per qualche a nell'alfabeto ∑. Allora L(R) = {a}, l'NFA riconoscerà L(R)



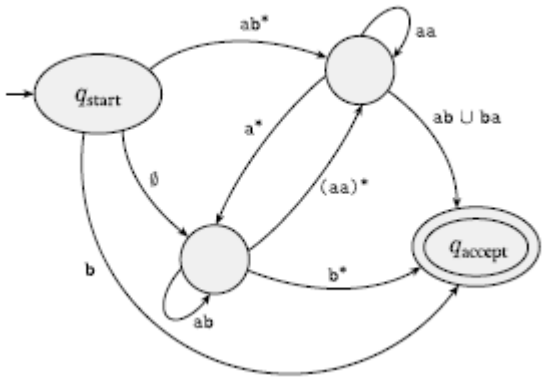
   Formalmente N = ({q_1, q_2}, ∑, δ, q_1, {q_2}), δ(q_1, a) = {q_2} e δ(r,b) = ∅ per r ≠ q_1 o b ≠ a
2. R = ε, allora L(R) = {ε} e l'NFA riconoscerà L(R), formalmente N = ({q_1}, ∑, δ, q_1, {q_1}) dove δ(r,b) = ∅ per ogni r e b
3. R = ∅, allora L(R) = {∅} e l'NFA riconoscerà L(R), formalmente N = ({q}, ∑, δ, q, ∅) dove δ(r,b) = ∅ per ogni r e b.
4. R = R' U R'' (dimostrazione teorema 1.45)
5. R = R' ° R'' (dimostrazione teorema 1.47)
6. R'* (dimostrazione teorema 1.49)

Questi ultimi 3 punti si dimostrano con le rispettive dimostrazioni per la chiusura.

Lemma 1.60: Se un linguaggio è regolare, allora è descritto da un'espressione regolare.

Idea della dimostrazione: Devo mostrare che se un linguaggio A è regolare, un'espressione regolare lo descrive. Perché A è regolare, è accettato da un DFA. Descrivo una procedura per convertire i DFA in espressioni regolare equivalenti. Rompo questa procedura in 2 parti, usando un nuovo tipi di automa a stati finiti, un'automa generalizzato non deterministico a stati finiti (GNFA). Prima mostro come convertire un DFA in un GNFA. I GNFA a differenza degli NFA riescono a leggere segmenti di parole, che riescono a farlo muovere da uno stato all'altro, ogni stringa letta è descritta da un linguaggio regolare (solitamente rappresentati sulla freccia per passare da uno stato all'altro).

Un GNFA è non deterministico e può avere diversi modi di processare la stessa stringa in input. Un GNFA deve avere minimo 2 stati, uno iniziale e uno accettante (quindi non possono essere lo stesso stato).



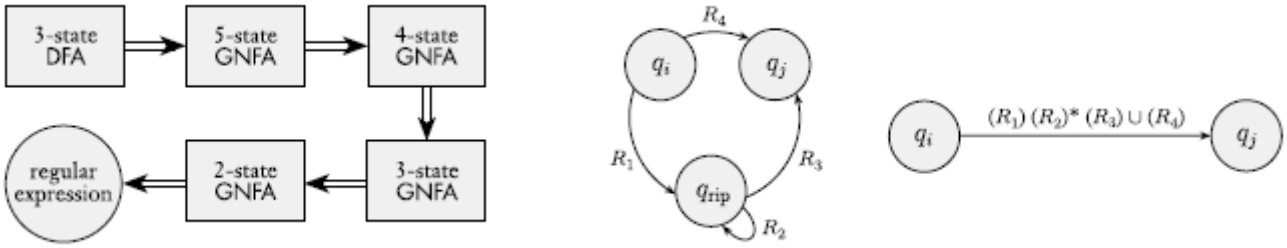
Per convenienza, richiedo che il GNFA abbia sempre una forma speciale che incontra le seguenti condizioni:

- Lo stato iniziale ha delle frecce di transizione che vanno in ogni altro stato nessuna freccia che ritorna allo stato iniziale.
- C'è solo uno stato accettante, e ha frecce che arrivano da ogni altro stato, mentre dallo stato accettante non escono frecce.
- Tranne per lo stato iniziale e lo stato accettante, una freccia va in ogni stato, c'è anche la possibilità che la freccia ritorni nello stesso stato.

Si può convertire facilmente un DFA in un GNFA nella forma speciale. Semplicemente si aggiunge un nuovo stato iniziale con una freccia ε per il vecchio stato iniziale e si raggiunge lo stato accettante con una freccia ε. Se una freccia ha più livelli si rimpiazza con una singola freccia, che unisce i vari livelli. Finalmente si aggiunge una freccia segnata con Ø tra gli stati che non hanno frecce. Quest'ultimo step non cambierà il linguaggio riconosciuto perché questo genere di transizione non può mai essere usato.

Ora si mostra come convertire un GNFA in un'espressione regolare. Dico che il GNFA ha k stati, allora un GNFA ha sicuramente uno stato inziale e uno accettante, diversi tra di loro, allora devo avere $k \geq 2$. Se $k > 2$ costruisco un GNFA equivalente con k-1 stati. Questo step può essere ripetuto finchè il GNFA non è ridotto fino a 2 stati. Se k = 2 il GNFA ha una sola freccia che va dallo stato iniziale allo stato accettante. Questa freccia di transizione sarà segnata proprio come l'espressione regolare.

Lo step cruciale è costruire un'equivalente GNFA con uno stato in meno quando $k > 2$. Si svolge selezionando uno stato, toglierlo dalla macchina e riparando il resto affinché il linguaggio rimanga lo stesso.



Dimostrazione: Per rendere l'idea formale devo, innanzitutto, facilitare la dimostrazione. Avendo introdotto un nuovo tipo di automa, il GNFA, e so che è simile alla funzione di transizione di un NFA, (la funzione) che ha la forma: $\delta: (Q - \{q_{acc}\}) \times (Q - \{q_{ini}\}) \rightarrow R$.

R è la collezione di tutte le espressioni regolari finite nell'alfabeto ∑. q_{acc} e q_{ini} sono, rispettivamente, lo stato accettante e lo stato iniziale. Se $\delta(q_i,q_j) = R'$ la freccia da q_i a q_j è segnata dall'espressione regolare R'. Il dominio della funzione di transizione è $(Q - \{q_{acc}\}) \times (Q - \{q_{ini}\})$ perché una freccia connette ogni stato con l'altro, tranne per lo stato iniziale e accettante.

Definizione (1.64): Un GNFA è una quintupla $(Q, \sum, \delta, q_{ini}, q_{acc})$ dove:

1. Q è l'insieme finito di stati
2. ∑ è l'alfabeto in input
3. $\delta: (Q - \{q_{acc}\}) \times (Q - \{q_{ini}\}) \rightarrow R$ è la funzione di transizione
4. q_{ini} è lo stato iniziale
5. q_{acc} è lo stato accettante

Un GNFA accetta una stringa in ∑* se $w = w_1w_2...w_k$ dove ogni w_i è in ∑* e una sequenza di stati $q_0q_1...q_k$ esiste tale che:

1. $q_0 = q_{ini}$
2. $q_k = q_{acc}$
3. per ogni i, ho $w_i \in L(R_i)$, dove $R_i = \delta(q_{i-1},q_i)$: in altre parole R_i è l'espressione sulla freccia da q_{i-1} a q_i

Sia M il DFA per il linguaggio A. Allora converto M in un GNFA G aggiungendo un nuovo stato iniziale e un nuovo stato accettante, in aggiunta un freccia di transizione se necessario (segnata dalla parola ε). Uso la procedura CONVERT(G), che prende un GNFA e restituisce un'equivalente espressione regolare. Questa procedura usa la ricorsione (significa che richiama se stessa). Un loop infinito è evitato perché la procedura richiama se stessa per processare solo un GNFA che ha uno stato in meno. Il caso in cui il GNFA ha 2 stati è trattato senza ricorsioni.

Cosa fa CONVERT(G):

1. Sia k il numero di stati di G
2. Se k = 2, allora G deve consistere solo di stato iniziale e accettante, con un'unica freccia che va da uno stato all'altro etichettata con un'espressione regolare R. Restituisce l'espressione.
3. Se k > 2, seleziono ogni stato $q_{rip} \in Q$ diverso da q_{ini} e q_{acc}, sia G' il GNFA $(Q', \sum', \delta', q_{ini}, q_{acc})$ dove $Q' = Q - \{q_{rip}\}$ per ogni $q_i \in Q' - \{q_{acc}\}$ e $q_j \in Q' - \{q_{ini}\}$ sia: $\delta'(q_i, q_j) = (R')(R'')^*(R''')U(R'''')$ per $R' = \delta(q_i, q_{rip})$, $R'' = \delta(q_{rip}, q_{rip})$, $R''' = \delta(q_{rip}, q_j)$ e $R'''' = \delta(q_i, q_j)$.
4. Computo CONVERT(G) e restituisco questo valore.

Affermazione (1.65): Per ogni GNFA G, CONVERT(G) è equivalente a G. Dimostro quest'affermazione per induzione su k, il numero di stati del GNFA.

Base: Dimostro l'affermazione vera per k = 2 stati. Se G ha solo 2 stati può avere solo una sola freccia che va dallo stato iniziale a quello finale. L'espressione regolare è l'etichetta della freccia che descrive tutte le stringhe che G può accettare.

Passaggio induttivo: Si consideri la tesi vera per k -1 stati e si assuma che la tesi è vera per k stati. Prima mostro che G e G' riconoscono lo stesso linguaggio. Suppongo che G accetta un input w. Allora in un salto accettante della computazione di G' entra una sequenza di stati: $q_{ini}, q_1, ..., q_{acc}$

Se nessuno degli stati è lo stato rimosso q_{rip}, chiaramente G' accetta anche w. La ragione è che ognuna delle nuove espressioni regolari che etichettano le frecce di G' contengono le vecchie espressioni regolari sotto il simbolo di unione.

Se q_{rip} appare, allora sarà presente tra q_i e q_j nell'automa G, quindi G' accetta w.

In caso G' accettasse w, allora ogni freccia tra i 2 stati q_i e q_j in G' descrive la collezione di stringhe comprese tra q_i e q_j in G, direttamente o non per q_{rip}, G deve accettare w, dunque G e G' sono equivalenti. L'ipotesi induttiva era che G' dovesse avere k-1 stati e che dovesse riconosce un'espressione regolare, anche G riconosce quest'espressione regolare e l'algoritmo è dimostrato.

1.4:

Linguaggi non regolari: Per capire la potenza degli automi a stati finiti si devono capire anche i loro limiti. Si consideri il linguaggio $B = \{0^n1^n \mid n \geq 0\}$, se provassi a cercare un DFA che riconosce B si scopre che la macchina sembra avere il bisogno di ricordare quanti 0 ha visto prima di leggere l'input, perché il numero di 0 non è finito, dunque non si può avere un numero finito di stati. Linguaggi come quello descritto sono linguaggi non regolari. Sembra che i linguaggi richiedano una memoria illimitata, ma non significa che sia realmente necessario. Capita che sia necessario per linguaggi come il linguaggio B, ma ci sono altri linguaggi che nonostante chiedano un numero illimitato di possibilità sono comunque regolari, ad esempio:

C = {w|w ha numero uguale di 0 e 1 (sarebbe 0^n1^n)}

D = {w|w ha numero uguale di casi di 01 e 10 come sottostringhe}

A primo acchito entrambi possono sembrare non regolari, ma dando uno sguardo più preciso solo C è un linguaggio non-regolare, D non lo è, invece.

Pumping lemma per i linguaggi regolari: Per dimostrare la non-regolarità di un linguaggio si userà il pumping lemma. Questo teorema afferma che tutti i linguaggi regolari hanno una proprietà speciale. La proprietà afferma che tutte le stringhe nel linguaggio possono essere "pompate" se sono lunghe almeno quanto un certo valore, chiamata lunghezza di pumping. Significa che una certa stringa contiene una sezione che può essere ripetuta un numero di volte e comunque questa nuova stringa con le ripetizioni appartiene al linguaggio.

Teorema 1.70: Se A è un linguaggio regolare, allora c'è un numero p (lunghezza pumping) dove se a è ogni stringa in A di lunghezza almeno p, allora a può essere divisa in 3 pezzi xyz che soddisfano le seguenti condizioni:

1. Per ogni i ≥ 0 , $xy^iz \in A$
2. $|y| > 0$
3. $|xy| \leq p$

Richiamo la notazione $|a|$ che rappresenta la lunghezza di a, y^i significa che y si ripete un numero di i volte. Quando a è divisa in xyz sia x, sia z possono essere ε, mentre per la condizione 2 y non può esserlo, inoltre senza la condizione 2 il teorema sarebbe vero. La condizione 3 afferma che i pezzi x e y insieme hanno una lunghezza al più p.

Dimostrazione: Sia $M=(Q,\Sigma,\delta,q_1,F)$ un DFA che riconosca il linguaggio A, e sia p il numero di stati di M.
Sia $s=s_1s_2..s_n$ una stringa in A di lunghezza n, dove $n\geq p$. Sia infine r_1, ... , r_{n+1} la sequenza di stati che M assume durante la computazione di s, tale che $r_{i+1}=\delta(r_i,s_i)$ per $1 \leq i \leq n$:

- se la lunghezza della stringa è n, allora il numero di stati che M attraversa è n+1;
- il numero di stati di M è p, e come abbiamo già detto $n\geq p$;
- ma se $n\geq p$, allora sicuramente $n+1>p$, quindi la sequenza di stati deve avere una ripetizione

Se chiamiamo il primo degli stati che si ripetono r_j e il secondo r_l, possiamo suddividere la stringa in ingresso in questo modo:

- $x = s_1...s_{j-1}$
- $y = s_j...s_{l-1}$
- $z = s_l...s_n$

Dato che la sottostringa x tiene occupato M dallo stato r_1 a r_j, la sottostringa y lo tiene occupato da r_j a r_j, e la sottostringa z da r_j a r_{n+1}, che è lo stato accettante, allora M deve accettare xy^iz per qualsiasi $i\geq 0$ (prima condizione del teorema).
Verifichiamo invece la seconda condizione del teorema osservando che $j\neq l$, allora per forza di cose $|y|>0$, perché nel peggiore dei casi $|y|$ è pari a 1. Infine, dato che tra i primi p+1 elementi della sequenza di stati deve esserci almeno una ripetizione, ovvero $l \leq p+1$, allora $|xy|\leq p$ (terza condizione del teorema).

Capitolo 2:

Un modo più potente per descrivere i linguaggi sono le context-free grammars (grammatiche context-free). Un insieme di linguaggi associati alle grammatiche context-free sono chiamati "linguaggi context-free".

2.1:

Grammatiche context-free: Il seguente è un esempio di grammatica context-free chiamata G':

A -> 0A1, A -> B, B -> #

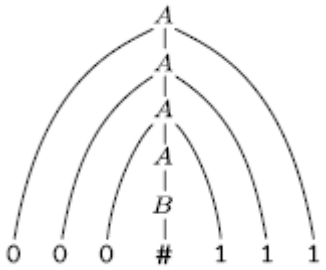
Una grammatica consiste di una collezione di regole di sostituzioni chiamate anche produzioni. Ogni regola appare con una linea nella grammatica, comprendendo un simbolo e una stringa separati da una freccia. Il simbolo è chiamato variabile. La stringa consiste di variabili e altri simboli chiamati terminali. Il simbolo per la variabile è spesso rappresentato da una lettera maiuscola. I terminali sono analoghi agli input e negli alfabeti sono spesso rappresentati da lettere minuscole, numeri o simboli speciali. Una variabile è designata come variabile inziale. Solitamente rappresentata sulla sinistra della regola. Ad esempio, la grammatica G' contiene 3 regole e le variabili sono A e B, A è la variabile iniziale. I terminali di G' sono 0, 1 e #.

Si usa una grammatica per descrivere un linguaggio generando ogni stringa da quel linguaggio nel modo seguente:

1. Si scrive la variabile iniziale;
2. Si trova una variabile che è scritta e una regola che inizia con quella variabile. Si rimpiazzia la variabile scritta con la regola.
3. Si ripete finché non rimangono più variabili.

Ad esempio G' genera la stringa 000#111. Dalla sequenza di sostituzioni si ottiene una stringa chiamata "derivazione". Una derivazione di una stringa 000#111 nella grammatica G' è: A -> 0A1 -> 00A11 -> 000B111 -> 000#111

Si può rappresentare la stessa informazione con un parse tree.



Parse tree di G':

Tutte le stringhe generate in questo modo costituiscono il linguaggio delle grammatiche. Scrivo L(G') per il linguaggio della grammatica G'. Qulche esperimento con la grammatica G' mostra che L(G') è $\{0^n\#1^n \mid \geq 0\}$. Ogni linguaggio che può essere generato da qualche grammatica context-free è chiamato linguaggio context-free (CFL). Per convenienza per presentare una grammatica context-free si abbreviano certe regole con le variabili a sinistra A -> 0A1 e A -> B in una sola linea A -> 0A1|B usando il simbolo "|" come un "o".

Definizione formale di una grammatica context-free (CFG) (2.2) [QUESTO SI DEVE RICORDARE ASSOLUTAMENTE PER GLI ESERCIZI]: Una grammatica context-free è una 4-tupla (V, ∑, R, S):

1. V è un insieme finito chiamato "variabili";
2. ∑ è un insieme finito, disgiunto da V, chiamato "terminali"
3. R è un insieme finito di regole, ogni regola è una variabile e una stringa di variabili e terminali;
4. S ∈ V è la variabile inziale.

Se u, v e w sono stringhe di variabili e terminali e A -> w è una regola della grammatica, dico che uAv -> www. Dico che u deriva v, scritto $u \Rightarrow^{\circ} v$, se u = v o la sequenza v',v'',...v_k per $k \geq 0$ e u -> u' -> u'' -> ... ->

u_k -> v. Il linguaggio della grammatica è $\{w \in \sum^* \mid S \Rightarrow^* w\}$.

Disegnare grammatiche context-free: Disegnare una grammatica context-free richiede creatività. Infatti, queste grammatiche sono molto più difficili da costruire rispetto agli automi a stati finiti. Per disegnare una CFG devo, prima di tutto considerare che molte CFL sono l'unione di CFL più semplici. Per costruire una CFG

da un CFL, devo rompere il CLF in pezzi più semplici, per farlo devo costruire la grammatica un pezzo alla volta. Dunque queste grammatiche possono essere fuse in una grammatica per il linguaggio originale unendo le regole e aggiungendone di nuove: $S \rightarrow S_1|S_2|...|S_k$ dove S_i è l'insieme delle variabili iniziali per le grammatiche singole. Risolvere molti problemi semplici è spesso più facile che risolverne uno complicato.

Ad esempio, per ottenere la grammatica dal linguaggio $\{0^n1^n| \ n \geq 0\} \cup \{1^n0^n \ | \ n \geq 0\}$ prima costruisco la grammatica: $S' \rightarrow 0S'1|\varepsilon$ per il primo linguaggio, poi costruisco $S'' \rightarrow 1S''0|\varepsilon$ per il secondo linguaggio, infine aggiungo le 2 regole tra di loro per ottenere la grammatica:

$S \rightarrow S'|S''$

$S' \rightarrow 0S'1|\varepsilon$

$S'' \rightarrow 1S''0|\varepsilon$

Seconda cosa, costruire una CFG per un linguaggio che sembra essere regolare è semplice se posso prima costruire un DFA per quel linguaggio. Posso convertire ogni DFA in un'equivalente CFG nel modo seguente: faccio una variabile R_i per ogni stato q_i del DFA. Aggiungo la regola $R_i \rightarrow aR_j$ alla CFG se $\delta(q_i, a) = q_j$ è una transizione del DFA. Aggiungo la regola $R_i \rightarrow \varepsilon$ se q_i è uno stato accettante.

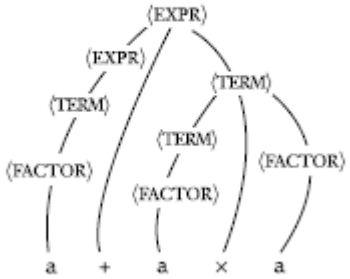
Terzo: certe CFL contengono stringhe con 2 sottostringhe "collegate", nel senso che la macchina per un linguaggio dovrebbe ricordare una quantità illimitata d'informazioni per solo una delle sottostringhe che verificano quella corrispondente proprietà per le altre sottostringa, questa situazione si presenta col linguaggio $\{0^n1^n| \ n \geq 0\}$ perché la macchina dovrebbe ricordare il numero di 0 in ordine di verificare che è uguale al numero di 1.

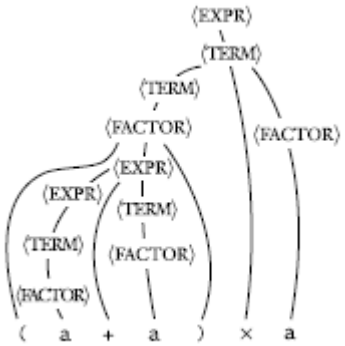
Ambiguità: Qualche volta una grammatica può generare la stessa stringa in molte forme diverse. Quella stringa avrà diversi parse tree e questi avranno significati diversi. Se si ripete una certa stringa significa che viene derivata ambiguamente nella grammatica. Se una grammatica genera delle stringhe in modo ambiguo, si dice che la grammatica è ambigua. Ad esempio la grammatica G^5:

<EXPR> -> <EXPR> + <EXPR> | <EXPR> x <EXPR> | <EXPR> | a

Questa grammatica genera la stringa a + a x a ambiguamente.

Questa grammatica non cattura le solite relazioni precedenti e così può raggruppare il + prima del x o viceversa. La seguente grammatica (G^4) non lo è, perché può generare sia (a + a x a) sia ((a+a) x a). Infine, G^4 non è ambigua, G^5 lo è.





Definizione 2.7: Una stringa w è derivata ambiguamente in una CFG G se ha 2 o più derivazioni differenti a sinistra. La grammatica G è ambigua se genera delle stringhe ambigue.

Qualche volta quando si hanno delle grammatiche ambigue si può trovare una grammatica non ambigua che genera lo stesso linguaggio. Dei CFL, comunque, possono essere generati da grammatiche ambigue, questi linguaggi sono chiamati ereditariamente ambigui.

Forma normale di Chomsky: Quando si lavora con le CFG è spesso conveniente avere delle loro forme semplificate.

Definizione 2.8: Una CFG è nella forma normale di Chomsky (CNF) se ogni regola è della forma: A -> BC, A -> a

Dove a è il terminale, A, B, e C sono delle variabili – B e C che non possono essere la variabile iniziale. In aggiunta è permesso S -> ε, dove S è la variabile iniziale.

Teorema 2.9: Ogni CFG è generata da una CFG nella forma normale di Chomsky.

Idea della dimostrazione: Posso convertire ogni grammatica G nella sua forma normale di Chomsky. La conversione ha vari livelli, dove le regole che violano le condizioni vengono rimpiazzate con regole equivalenti. Prima, si aggiunge una nuova variabile iniziale. Poi, si eliminano tutte le regole-ε della forma

A -> ε, dove A non è la variabile iniziale. Si eliminano anche tutte le regole unità della forma A -> B oppure

A -> A. In entrambi i casi si controlla che la grammatica generi lo stesso linguaggio. Infine si converte le regole restanti nella forma adeguata.

Dimostrazione: Primo, aggiungo una nuova variabile S_0 e la regola S_0 -> S, dove S era la variabile iniziale originale. Questo cambiamento garantisce che la variabile iniziale non sia al lato destro della regola.

Secondo, penso a tutte le regole-ε. Si rimuovono tutte, tranne quella della variabile iniziale. Dunque per ogni volta che si verifica A si aggiunge una nuova regola con quella della verifica rimossa.

Terzo, si pensa alla regole unitarie, si rimuove una regola unitaria A -> B, allora per ogni volta una regola

B -> u appare, si aggiunge la regola A -> u fino a quando non viene rimossa del tutto la regola unitaria. Come prima u è una stringa di variabili e terminali. Si ripete questo passaggio finché non ci sono più regole unitarie.

(quest'ultima parte è più chiara nell'esempio) Infine, si convertono tutte le regole rimanenti nella forma adeguata. Si rimpiazza ogni regola

A -> u'u''...u^k dove k>=3 e ogni u_i è un simbolo della variabile o del terminale, con le regole A -> u'A',

A' -> u''A'', ... , -> $u^{(k-1)}A^{(k-1)}$ -> u^kA^k . Le A_i sono le nuove variabili. Rimpiazzio ogni terminale u^i nelle regole precedenti con U_i e aggiungo la regola U_i -> u_i.

Esempio di conversione in CNF:

1. S -> ASA| aB, A -> B| S, B -> b| ε ⇒ diventa ⇒ S_0 -> S, S -> ASA| aB, A -> B| S, B -> b| ε
2. S_0 -> S, S -> ASA| aB, A -> B| S, B -> b| ε ⇒ rimuovendo le regole-ε B -> ε diventa ⇒ S_0 -> S, S -> ASA| aB| a| SA| AS| S, A -> B| S | ε, B -> b

3a. S_0 -> S, S -> ASA| aB| a| SA| AS| S, A -> B| S | ε, B -> b ⇒ rimuovendo le regole unitarie come (S ->S) diventa ⇒ S_0 -> S| ASA| aB| a| SA| AS, S -> ASA| aB| a| SA| AS, A -> B| S, B -> b

(Si noti che S è diventato come S_0)

3b. S_0 -> S| ASA| aB| a| SA| AS, S -> ASA| aB| a| SA| AS, A -> B| S, B -> b ⇒ rimuovendo regole del tipo A -> S e A -> B diventa ⇒ S_0 -> ASA| aB| a| SA| AS, S -> ASA| aB| a| SA| AS, A -> b| S| ASA| aB| a| SA| AS, B -> b

(Si noti che A ha preso le derivazioni di S e di B)

4. Aggiungo delle nuove variabili per le regole con più di 2 variabili e quelle con un terminale, ottenendo:

S_0 -> AA_1| UB| a| SA| AS, S -> AA_1| UB| a| SA| AS, A -> b| AA_1| UB| a| SA| AS, A_1 -> SA, U -> a, B -> b

(Si noti che A_1 deriva in SA, in quanto prima in S_0, S e A c'era la derivazione ASA – 3 variabili, che non vanno bene per la CNF – e c'è anche la variabile U che deriva nel terminale A, sempre per le derivazioni che hanno S_0, S e A, infatti prima era S_0, S e A -> aB, ora è S_0, S e A -> UB)

2.2:

Automi pushdown (o a pila): Questi automi sono come gli NFA ma hanno una componente in più, lo stack (o pila). Lo stack provvede ad aggiungere una memoria all'insieme di stati finito. Lo stack conferisce all'automa pushdown la possibilità di riconoscere dei linguaggi non regolari. Gli automi pushdown sono equivalenti in potenza alle CFG. Quest'equivalenza è utile perché ci dà 2 opzioni per provare che un linguaggio è un CFL.

Un'automa pushdown (PDA) può scrivere simboli sullo stack e leggerli quando gli servono. Scrivere un simbolo sullo stack è spesso riferito come "spingere" (pushing) il simbolo, rimuovere un simbolo è riferito come "far saltare" (popping). Si noti che tutti gli accessi allo stack possono essere fatti solo a partire dall'alto. In altri termini uno stack è un dispositivo d'immagazzinamento per cui "l'ultimo è dentro, il primo è fuori" ("Last in, First out" = LIFO).

Uno stack è considerevole perché può mantenere una quantità illimitata d'informazioni.

Si consideri il linguaggio L = {$a^n b^n$ con n>1}, non può essere riconosciuto da un DFA o un NFA, perché hanno memoria finita, non sono in grado di riconoscere linguaggi che, per la loro struttura, necessitano di ricordare una quantità di "informazioni" non limitate. Il linguaggio in oggetto, in compenso, può essere accettato da un **Pushdown Automaton** o **PDA.** Il PDA può riconoscere questo linguaggio perché può usare il suo stack per immagazzinare il numero di a che ha visto. Questa natura illimitata dello stack permette al PDA d'immagazzinare numeri di lunghezza illimitata. I PDA possono essere deterministici e non, i PDA deterministici non sono equivalenti ai PDA non-determinsitici in potenza.

Definizione formale di un'automa pushdown: La defizione formale è simile a quella di un'automa a stati a stati finiti, tranne per lo stack. Lo stack è un dispositivo contenente simboli scritti da qualche alfabeto. Al cuore di ogni definizione formale di un'automa c'è la funzione di transizione che descrive il suo comportamento. Richiamo $\sum_\varepsilon = \sum \cup \{\varepsilon\}$ e $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$. Il dominio della funzione di transizione è $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$.

Definizione 2.13: Un PDA nondeterministico è formalmente definito da una 6-pla $(Q,\Sigma,\Gamma,\delta,q_0,F)$ dove

- Q è un insieme finito di stati
- Σ è un insieme finito di caratteri che costituiscono l'alfabeto del nastro
- Γ è un insieme finito di caratteri che costituiscono l'alfabeto dello stack
- $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma^*)$ è la funzione di transizione che associa ad ogni tripla (stato, carattere letto sul nastro, carattere letto sul top dello stack) un insieme di possibili coppie (stato, stringa), dove la stringa sostituisce il top dello stack
- $q_0 \in Q$ è lo stato iniziale
- F sottoinsieme di Q è l'insieme degli stati finali.

Come nel caso degli NFA, il concetto di configurazione è utile per descrivere la situazione in cui si trova la macchina complessivamente:

- lo stato,
- l'input ancora da esaminare e
- il contenuto della pila.

Quindi una configurazione è un elemento di $Q \times \Sigma^* \times \Gamma^*$.

La configurazione iniziale è (q_0,x,ε)

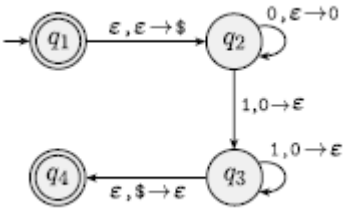
Un PDA $M = (Q,\Sigma,\Gamma,\delta,q_0,F)$ accetta una stringa $w= w_1w_2 \ldots w_n$, con $w_i \in \Sigma_\varepsilon$, se esistono una sequenza di stati, $r_0, r_1,\ldots,r_n$ di Q e una stringa $s_0, s_1, \ldots, s_m$ di Γ^* che soddisfino le seguenti tre condizioni:

- $r_0 = q_0$ e $s_0 = \varepsilon$.
- Per $i = 0, \ldots, m - 1$, si ha che $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, dove $s_i = a \cdot t$ e $s_{i+1} = b \cdot t$, per qualche a,b in Γ_ε e t in Γ^*.
- r_n appartiene a F.

Se $(q,B) \in \delta(p,a,A)$, con $a \in \Sigma_\varepsilon$, $A \in \Gamma_\varepsilon$ e $B \in \Gamma_\varepsilon$ allora è possibile che sia eseguito uno dei seguenti passi di calcolo,

- (POP-PUSH) se $a \in \Sigma$, $A, B \in \Gamma$ con a in lettura sul nastro di input e A in cima alla pila passa nello stato q, rimpiazza A con B in cima alla pila e muove la testina di lettura del nastro di input di una cella verso destra.
- (PUSH) se $a \in \Sigma$, $A = \varepsilon$, $B \in \Gamma$ allora, con a in lettura sul nastro di input e indipendentemente dal simbolo in cima alla pila, passa nello stato q, impila B in cima alla pila e muove la testina di lettura del nastro di input di una cella verso destra.
- (POP) se $a \in \Sigma$, $A \in \Gamma$, $B = \varepsilon$ allora, con a in lettura sul nastro di input e A in cima alla pila, passa nello stato q, elimina A dalla cima della pila e muove la testina di lettura del nastro di input di una cella verso destra.

Se $a = \varepsilon$, i tre tipi di mosse avvengono senza che la testina di lettura si sposti.



Esempio: Il PDA riconosce $\{0^n1^n | n \geq 0\}$. La definizione formale di PDA non contiene un meccanismo esplicito che permetta al PDA di fare test per uno stack vuoto, questo PDA, infatti, è capace di ottenere lo stesso effetto inizializzando lo stack piazzando $ sullo stack. Allora se vede di nuovo $ sa che lo stack è effettivamente vuoto.

Similmente, i PDA non possono testare esplicitamente per aver raggiunto la fine della stringa in input. Questo PDA è capace di ottenere quell'effetto perché lo stato accettante ha effetto solo quando la macchina è alla fine dell'input.

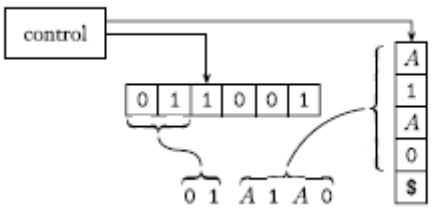
(Equivalenza con le CFG) Teorema 2.20: Un CFL è tale se e solo se qualche PDA lo riconosce.

Lemma 2.21: Se un linguaggio è context free, allora qualche automa pushdown lo riconosce.

Idea della dimostrazione: Sia A un CFL. Dalla definizione di A si sa che A ha una CFG G che lo genera. Si deve mostrare come convertire G in un PDA P equivalente. P lavorerà accettando un suo input w, se G genera w, si determina che c'è una derivazione per w. Ogni passaggio della derivazione cade in una stringa di mezzo di variabili e terminali. Si disegna P per determinare se alcune serie di sostituzioni, usando le regole di G, possono portare dalla variabile iniziale a w.

Una delle difficoltà è testare se c'è una derivazione per w che mostri che sostituzioni si devono fare. Il non determinismo del PDA permette di indovinare la sequenza corretta di sostituzioni. A ogni passaggio della derivazione una delle regole per una variabile particolare è selezionata non deterministicamente ed è usata per sostituire per quella variabile. La seguente è una descrizione informale di P.

1. Piazzio il simbolo di marcature $ e la variabile iniziale sullo stack.
2. Ripeto i seguenti step per sempre:
   a. In cima allo stack ho una variabile di simbolo A, seleziono non deterministicamente una delle regole per A e sostistuisco A con la stringa più a destra della regola;
   b. In cima allo stack ho un terminale di simbolo a, leggo il prossimo simbolo dall'input per confrontarlo con a. Se sono uguali, ripeto. Se non lo sono, rifiuto su questo ramo del non determinismo;
   c. In cima allo stack ho il simbolo $, inserisco lo stato accettante. Accetto l'input se è stato letto tutto.



Dimostrazione: Avendo i dettagli formali per costruire un PDA $P = (Q,\Sigma,\Gamma,\delta,q_0,F)$. Per rendere più chiara la costruzione userò la notazione abbreviata della funzione di transizione. Questa notazione provvede un modo per scrivere un'intera stringa sullo stack in un passaggio della macchina. Si può simulare quest'azione introducendo stati per scrivere la stringa un simbolo alla volta, come implementato nella seguente

costruzione formale. Siano q e r degli stati del PDA e sia a presente in Σ_ε e s in Γ_ε. Voglio che il PDA vada da q a r quando legge a e caccia s (pop). Inoltre, voglio spingere (push) l'intera stringa u = u'u''...w sullo stack nello stesso momento. Posso implementare quest'azione introducendo dei nuovi stati q', ..., q_{t-1} e impostando la funzione di transizione nel modo seguente:

- $\delta(q, a, s)$ contiene (q', u_l)
- $\delta(q', \varepsilon, \varepsilon) = \{(q'', u_{l-1})\}$
- ...
- $\delta(q_{t-1}, \varepsilon, \varepsilon) = \{(r, u')\}$

Uso la notazione $(r,u) \in \delta(q, a, s)$ significa che quando q è lo stato dell'automa, a è il prossimo simbolo e s è il simbolo in cima allo stack, il PDA può leggere a ed espellere s, allora spinge u sullo stack e va allo stato r.

Gli stati di P sono $Q = \{q_0, q_{stop}, q_{acc}\} \cup E$ (U = unione), dove E è l'insieme di stati di cui ho bisogno per implementare l'abbreviazione appena descritta. Lo stato iniziale è q_0, l'unico stato accettante è q_{acc}.

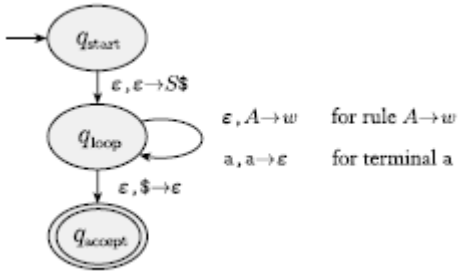
La funzione di transizione è definita come segue. Inizializzo lo stack per contenere i simbolo $ e N, implementando lo step 1 nella descrizione informale: $\delta(q_0, \varepsilon, \varepsilon) = \{(q_{loop}, N\$)\}$. Allora inserisco delle transizioni per il loop principale dello step 2.

Prima, gestisco il caso (a) dove in cima allo stack c'è la variabile. Sia $\delta(q_{loop}, \varepsilon, A) = \{(q_{loop}, w \mid$ dove

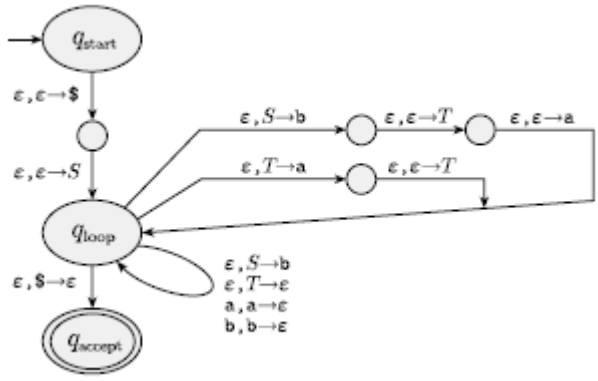
A -> w è una regola in R)}. Secondo, gestisco il caso (b) dove in cima allo stack c'è il terminale.

Sia $\delta(q_{loop}, a, a) = \{(q_{loop}, \varepsilon)\}$. Infine, gestisco il caso (c) dove il marcatore di stack vuoto $ è in cima allo stack. Sia $\delta(q_{loop}, \varepsilon, \$) = \{(q_{acc}, \varepsilon)\}$.

La dimostrazione è finita.



Esempio 1): Ho la grammatica S -> aTb | b, T -> Ta | ε, diventa il seguente PDA:



Lemma 2.27: Se un PDA riconosce qualche linguaggio, allora è context-free.

Idea della dimostrazione: Ho un PDA P e voglio fare una CFG G che genera tutte le stringhe che P accetta. In altre parole, G dovrebbe generare una stringa se quella stringa fa andare il PDA dallo stato iniziale a uno accettante.

Per ottenere questo obiettivo si disegna una grammatica che fa qualcosa in più. Per ogni coppia di stati p e q in P, la grammatica avrà variabili A_{pq}. Questa variabile genera tutte le stringhe che P può prendere da p con uno stack vuoto a q con uno stack vuoto. Osservo che queste stringhe possono anche prendere P da p a q, senza considerare il contenuto in p, lasciondo lo stack a q nella stessa condizione di p.

Primo, semplifico la nostra task modificando P leggermente per dargli le 3 seguenti caratteristiche:

1. Ha un solo stato accettante q_{acc};
2. Svuota lo stack prima di accettare;
3. Ogni transizione spinge un simbolo sullo stack (push) o lo fa uscire dallo stack (pop), ma non fa entrambe le cose allo stesso momento.

Dare a P le caratteristiche 1 e 2 è semplice. Per dargli 3, si deve rimpiazzare ogni transizione che simultaneamente caccia e spinge con una transizione a 2 sequenze che va attraverso uno stato nuovo, e rimpiazza ogni transizione che non caccia o spinge con una transizione a 2 sequenze che spinge a caccia un simbolo arbitrario dello stack.

Dimostrazione: Dico che $P = (Q,\Sigma,\Gamma,\delta,q_0,\{q_{acc}\})$ e costruisco G. Le variabili di G sono $\{A_{pq} \mid p, q \in Q\}$. La variabile iniziale $A_{(q0, qacc)}$. Ora descrivo le regole di G in 3 parti:

1. Per ogni $p, q, r, s \in Q$, $u \in \Gamma$ e $a, b \in \sum_\varepsilon$, se $\delta(p, a, \varepsilon) \subseteq (r, u)$ e $\delta(s, b, u) \subseteq (q, \varepsilon)$, metto la regola $A_{pq} \rightarrow aA_{rs}b$ in G;
2. Per ogni $p, q, r, s \in Q$, metto la regola $A_{pq} \rightarrow A_{pr}A_{rq}$ in G;
3. Infine, per ogni $p \in Q$ metto la regola $A_{pp} \rightarrow \varepsilon$ in G.

(Non-CFL)Teorema 2.34 (pumping lemma per CFL): Se A è un CFL allora c'è un numero p (lunghezza di pumping) dove, se s è una stringa in A di almeno lunghezza p, allora s può essere divisa in 5 pezzi s = uvxyz che soddisfano le condizioni:

1. Per ogni $i \geq 0$, $uv^ixy^iz \in A$
2. $|vy| > 0$
3. $|vxy| \leq p$

Idea della dimostrazione: Sia A un CFL e sia G una CFG che lo genera. Deco mostrare che ogni stringa sufficentemente lunga s in A può essere pompata e rimane in A. L'idea dietro questo approccio è semplice.

Sia s una stringa molto lunga in A, perché è in A, è derivabile in G e quindi ha un parse tree. Il parse tree per s deve essere molto alto perché s è molto lunga.

Su questo lungo percorso, delle variabili di simbolo R si ripetono per il principio dei cassetti. Questa ripetizione ci permette di rimpiazzare dei sottoalberi sotto la seconda ripetizione di R con il sottoalbero sotto la prima ripetizione di R e ottenere ancora un parse tree legale. Intanto, posso tagliare s in 5 pezzi uvxyz e posso ripetere il secondo e quarto pezzo per ottenere una stringa ancora presente nel linguaggio, insomma $uv^ixy^iz \in A$.

Dimostrazione: Sia G una CFG per il CFL A. Sia b il massimo numero di simboli a destra della regola (almeno 2). In altre parole, al più le foglie di b sono 1 passo dalla variabile iniziale; al più le foglie di b^2 sono ad almeno 2 passaggi dalla variabile iniziale; e al più le foglie di b^h sono ad almeno h passaggi dalla variabile iniziale. Quindi, se l'altezza del parse tree è al più h, la lunghezza della stringa generata è b^h. Se una stringa generata è almeno b^{h+1}, ognuno dei suoi parse tree deve essere almeno h+1 alto.

Sia |V| il numero di variabili in G. Imposto $p = b^{|V|+1}$. Ora se s è una stringa in A e la sua lunghezza è p o più il suo parse tree dev'essere |V|+1 alto, perché $b^{|V|+1} \geq b^{|V|} + 1$.

Per vedere come vengono pompate certe delle stringhe s, sia τ uno dei suoi parse tree.

Divido s in uvxyz, ogni ripetizione di R ha un sottoalbero sotto, che genere una parte della stringa s. La ripetizione superiore di R ha un sottoalbero più largo e genera vxy, dove la ripetizione più in basso genera solo x con un sottoalbero più piccolo. Ognuno di questi sottoalberi sono generati dalla stessa variabile, quindi posso sostituire l'una con l'altra e ottenere ancora un parse tree valido. Rimpiazzo dal più piccolo al più grande ripetutamente dando parse tree per le stringhe uv^ixy^iz per ogni i > 1. Rimpiazzo dal più grande al più piccolo, che genera la stringa uxz, ciò stabilisce la condizione 1 del lemma.

Per ottenere la condizione 2, devo essere sicuro che v e y non sono entrambe ε. Se fossero stati i parse tree ottenuti dalla sostituzione del sottoalbero più piccolo per il più grande avrebbe avuto meno nodi di τ e generebbe ancora s. Questo risultato non è possibile perché ho appena scelta τ essere un parse tree per s con il minore numero di nodi. Quella è una ragione per aver selezionato τ in quel modo.

In ordine per ottenere la condizione 3, si deve essere sicuri che vxy ha al più lunghezza p. Nel parse tree per s la ripetizione superiore di R genera vxy. Ho scelto R così che ogni sua ripetizone finisca sotto |V| + 1 variabili sul percorso, e scelgo il percorso più lungo nel parse tree, così il sottoalbero dove R genera vxy è al più |V| + 1 alto. Un albero di quest'altezza può generare una stringa di lunghezza al più $b^{|V|+1} = p$.

7Capitolo 3:

3.1:

Macchine di Turing: La macchina di Turing (MdT), è simile a un'automa a stati finiti ma con una memoria illimitata e senza restrizioni, una MdT è un modello più preciso di un computer a scopo generale. Gli output accettati e rigettati sono ottenuti inserendo degli stati designati per accettare e rigettare. Se non è inserito uno stato accettante o uno rigettante andrà avanti per sempre, senza fermarsi.

Tra le macchine computazionali, la **Macchina di Turing** (**MdT**) è senza dubbio la più potente

Tesi di church-turing: Per ogni problema calcolabile esista una MdT in grado di risolverlo**.** La MdT come modello di calcolo è stato introdotto da Alan Turing.

L'utilità di un tale algoritmo sta nel fato che sarebbe in grado di risolvere tutti i problemi matematici e ancora più importante che ogni ragionamento umano poteva essere ridotto a mero calcolo meccanizzabile.

E' stata una prima risposta negativa al problema della decidibilità, Gödel dimostrò che la semplice coerenza di un sistema formale non può garantire che ciò in esso presente viene dimostrato sia vero oppure falso.

Una macchina di Turing opera su un nastro (potenzialmente infinito) che come per gli NFA si presenta come una sequenza di caselle nelle quali possono essere registrati simboli di un ben determinato alfabeto finito. A differenza degli NFA, la testina di una MdT può, oltre che leggere, anche scrivere sul nastro. Per questo motivo la testina viene chiamata di I/O (input-output). Inoltre, la testina può spostarsi a destra o a sinistra.

Un algoritmo si può definire come una collezione di istruzioni semplici per la realizzazione di un determinato compito (sono utili in molti ambiti). Il matematico David Hilbert identificò 23 problemi matematici difficili e li consegnò alla comunità scientifica come "problemi competitivi" per il secolo che st-ava per iniziare. Alcuni problemi sono stati risolti in pochi anni, altri hanno visto la soluzione solo pochi anni fa, ed altri ancora non hanno avuto tuttora risposta .

Il decimo problema della lista riguardava gli algoritmi ed in particolare chiedeva di definire un algoritmo che "dato in input un polinomio, si è in grado di testare se quel polinomio ammette o meno radice intera".

Una radice di un polinomio è un assegnamento di valori alle sua variabili in modo tale che il valore del polinomio diventi 0. Un polinomio è la somma di termini, dove ciascun termine è il prodotto di una costante (chiamato coefficiente) e di alcune variabili.

C'è da dire che Hilbert non usò il termine algoritmo per questo problema, ma quello di "processo" che, per definizione, è determinato da un numero finito di operazioni.

Dunque, Hilbert non chiedeva di controllare l'esistenza di un algoritmo ma solo di implementarlo. Questo perché era convinzione comune che ogni problema dovesse ammettere una procedura risolutiva.

I primi risultati sul decimo problemi di Hilbert arrivarono negli anni '30 e passarono per una definizione rigorosa del concetto di algoritmo.

Tale definizione arrivò ad opera di Alonzo Church and Alan Turing:

- Church usò un sistema notazionale chiamato λ-calculus.
- Turing usò un modello di macchina computazionale (la macchina di Turing)

Queste due definizioni furono poi mostrate essere equivalenti. Questa equivalenza diede origine alla seguente tesi chiamata tesi di ***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.

Il logico americano Alonzo Church, in seguito alle sue ricerche sulla computabilità effettiva, propose di identificare la classe delle funzioni calcolabili mediante un algoritmo (o funzioni effettivamente calcolabili) con una partico-lare classe di funzioni aritmetiche (funzioni ricorsive). Tale identificazione è oggi nota col nome di **Tesi di Church**.

È possibile dimostrare l'equivalenza tra la classe delle funzioni ricorsive e la classe delle funzioni **Turing-computabili** (decidibili), in quanto ogni funzione Turing-computabile è ricorsiva, e viceversa.

La Tesi di Church può quindi essere formulata come segue: **"una funzione è effettivamente calcolabile se solo se è Turing-computabile".**

Definizione formale di una MdT (3.3): Il cuore della definziione di una MdT è la funzione di transizione δ, perché spiega come la macchina va da un passaggio all'altro. Per una MdT, δ ha la forma Q x Γ -> Q x Γ x {L, R}.

Una MdT (deterministica) è formata da una 7-pla (Q, Σ, Γ, δ, q_0, accept, reject):

1. **Q** è un insieme finito di **stati**;
2. **STATI SPECIALI**: q_0 è lo **stato iniziale**
3. "accept" è lo **stato accettante**;
4. "reject" è lo **stato rifiutante**;
5. **Γ** è l'**alfabeto del nastro**. Include il simbolo speciale "-" oppure "μ" (simbolo di blank);
6. **Σ** ⊆ **Γ** \{-} è l'**alfabeto di input**;
7. **δ** : Q\{accept, recject} · Γ □ Q· Γ ·{L,R} è la **funzione di transizione** (parziale).

Nota: accept ≠ reject e sono degli **halting states,** cioè non esistono transizioni da questi stati.

Osservazione: nella definizione classica, la MdT ha anche un simbolo speciale di starting sul nastro, uno stato speciale aggiuntivo "halt" di halting e la testina ha la possibilità di rimanere ferma in una transizione. Queste modifiche non alterano il potere computazionale della macchina. Talvolta, noi useremo queste

varianti per semplicità. La funzione di transizione $\delta : Q\backslash\{accept, recject\} \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$ permette alla MdT, a partire da un certo stato e leggendo un simbolo sul nastro, di transire in un nuovo stato, sostituire il simbolo letto sul nastro, e muovere la testina di lettura a sinistra o a destra.

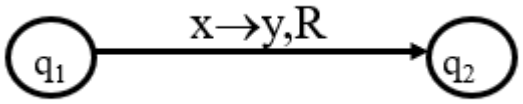
INIZIALMENTE: Stato=q_0 ; nastro in input … - - $w_1 \ldots w_n$ - -… testina posizionata su w_1 (su ▷ se esiste, che si troverà subito a sinistra di w_1);

AD OGNI PASSO: nuova configurazione conforme alla funzione di transizione δ; La testina non può muoversi a sinistra del primo simbolo della parola in input. Se si fa uso del marcatore di inizio stringa "▷", questo simbolo non viene mai sovrascritto e la testina non va mai oltre questo simbolo. Per semplicità, assumiamo che quando la testina si trova negli stati "accept" o "reject" (anche "halt" se previsto) non si muove.

La rappresentazione grafica in figura rappresenta la situazione in cui:

- il simbolo corrente sul nastro è x,
- Lo stato corrente è q_1,
- La testina legge x, lo sostituisce con y, si sposta a destra e transisce nello stato q_2

Se x=y, allora l'etichettatura x→x,R si può semplicemente scrivere come x→R.



Se invece di R, ci fosse stato L, allora la testina si sposta a sinistra, a meno che si tratti della prima cella non blank, nel qual caso, la testina non si sposta.

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") :

- $C_1 \square C_2$: La MdT esegue un passo da C1 a C2;
- $C_1 \square^* C_2$: chiusura transitiva - La MdT esegue in 0,1,2 o più passi uno spostamento da C_1 a C_2;

Una MdT M ***accetta*** una stringa in input se, prima o poi incontra lo stato "**accept**". Altrimenti la stringa è non accettata. $L(M)=\{x \in \Sigma^* \mid M \text{ su input x raggiunge lo stato "accept"}\}$

Dunque, l'input è non accettato da M in due casi:

- M incontra ad un certo punto lo stato reject;
- M non si ferma mai (non incontra mai uno stato di halting)

Il linguaggio **riconosciuto** (accettato) da una MdT è l'insieme delle stringhe che esso accetta

Una MdT è detta **decisore** se si ferma su ogni input. Se la macchina è un **decisore**, allora diremo che non solo accetta un linguaggio, ma lo decide anche. Formalmente, M decide un linguaggio se L ⊆ ∑*, L=L(M) e per ogni x in Σ^*–L, M termina in uno stato "reject";

Definizione 3.5: Un linguaggio è detto **Turing-riconoscibile** (o **ricorsivo-enumerabile**) se esiste una MdT che lo riconosce.

Definizione 3.6: Un linguaggio è detto **Turing-decidibile** (o **ricorsivo**) se esiste una MdT che lo decide.

LINGUAGGI RICORSIVI ⊆ LINGUAGGI RICORSIVI ENUMERABILI.

Consideriamo la variante della MdT con lo stato aggiuntivo "halt". Una funzione computabile con M è data dalla seguente descrizione:" su input x, se M termina in uno stato di "accept" allora M(x)="accept", altrimenti se termina in uno stato di "reject" M(x)="reject", se invece termina in uno stato "halt" allora M(x)= contenuto del nastro che si trova tra il primo simbolo "blank" (o il simbolo speciale "▷") e l'insieme infinito di simboli blank "-". Su altri input x (es. quando M gira all'infinito): M(x)= indefinito.

Per le funzioni possiamo avere per input e output di alfabeti differenti. Una funzione è ricorsiva se è computabile da qualche Macchina di Turing.

Come abbiamo detto in precedenza, la classe dei linguaggi Turing-riconoscibile (Turing-recognizable) è anche chiamata *ricorsiva enumerabile* (*recursively enumerable).*

Questo termine ha origine da una variante delle macchine di Turing: l'***enumeratore.***

Un enumeratore è una macchina di Turing che può stampare stringhe. Ogni qualvolta la macchina di Turing vuole aggiungere una stringa alla lista, semplicemente manda la stringa alla stampante.

Nella figura è mostrato lo schema dell'enumeratore. Un enumeratore E inizia a lavorare con un nastro vuoto. Se l'enumeratore lavora per sempre, può produrre una lista infinita di stringhe Il linguaggio enumerato da E è l'insieme di tutte le possibili stringhe che E riesce a stampare. In particolare, è utile notare che E può generare le stringhe in qualsiasi ordine, anche con ripetizione.

Corollario (3.18): Un linguaggio è Turing-riconoscibile (Turing-recognizable) se e solo se esiste un enumeratore che lo enumera.

Varianti delle MdT: Definizioni alternativi delle MdT includono le versioni multinastro e nondeterministica. Sono chiamate varianti del modello della MdT. Il modello originale e le sue varianti hanno la stessa potenza di calcolo/riconoscono la stessa classe di linguaggi.

MdT multinastro: Una macchina di Turing multinastro è paragonabile in potenza a un'ordinaria MdT a singolo nastro, solo che ha più nastri. Ogni nastro ha la sua testina per leggere e scrivere. Inizialmente l'input risiede sul O° 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:

- **δ** : $Q\backslash\{accept,reject\} \times \Gamma^k \rightarrow Q \times \Sigma^k \times \{L,R,S\}^k$ dove k è il numero di nastri;

L'espressione $\delta(q_i,a_1,\ldots, a_k)=(q_j,b_1,\ldots, b_k,L,R,\ldots,L)$ significa che se la macchina è nello stato q_i e le testine da 1 a k stanno leggendo i simboli $a_1 \ldots a_k$, la macchina va nello stato q_j, scrive i simboli $b_1 \ldots b_k$, e direziona ogni testina per muoversi a sinistra o destra, o rimanere ferma, in base a quanto specificato dalla relazione di transizione. Sebbene le Macchine di Turing multinastro sembrano essere più potenti delle classiche TM,

possiamo facilmente dimostrare che sono equivalenti. Ricordiamo che due Macchine di Turing sono equivalenti se riconoscono lo stesso linguaggio.

TEOREMA (3.13): Per ogni Macchina di Turing multinastro esiste una macchina di Turing a singolo nastro equivalente.

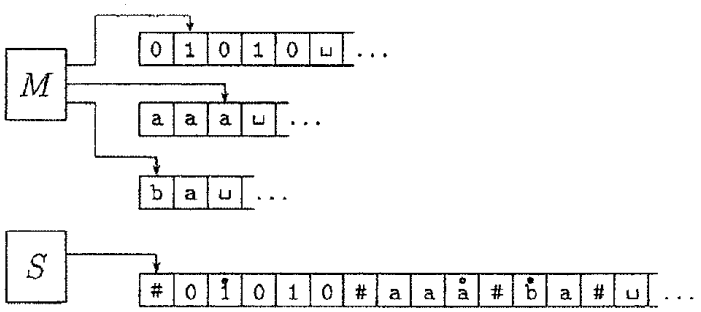
DIMOSTRAZIONE: Mostriamo come convertire una TM multinastro M in una TM S a singolo nastro. L'idea chiave è quella di mostrare come simulare M con S.

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. Per maggiore chiarezza di seguito è rappresentata una TM a 3 nastri e la sua corrispondente riduzione in una una TM a singolo nastro:

Di seguito è descritto il funzionamento della TM S:

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

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. Cosi 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 (3.15): Un linguaggio è Turing-Riconoscibile se e solo se esiste almeno una macchina di Turing Multinastro che lo riconosce.

DIMOSTRAZIONE:

->: Un linguaggio Turing riconoscibile è riconosciuto da una normale Macchina di Turing (a singolo nastro), la quale è un caso particolare di una Macchina di Turing multinastro.

<-: Questo si dimostra attraverso il teorema 3.13.

Si ricordi che una MdT che calcola una funzione (anche parziale) può essere vista come un **Trasduttore**

Questo è ancora più evidente nel caso di una macchine di Turing a più nastri, dove un nastro può essere usato per l'input, uno per l'output ed eventualmente altri nastri possono essere usati per l'esecuzione della macchina

MdT nondeterministiche: Con le macchine di Turing deterministiche è possibile caratterizzare la classe di complessità polinomiale P come la classe dei linguaggi che possono essere decisi da una macchina di Turing in tempo polinomiale.

Con le macchine di Turing nondeterministiche, introdurremo le classi di complessità di tempo e spazio nondeterministiche. In particolare introdurremo la classe NP, come la classe dei linguaggi che

- possono essere decisi in tempo esponenziale,
- L'appartenenza di una stringa al linguaggio può essere verificata in tempo deterministico polinomiale

In una macchina di Turing nondeterministica (NMdT), la funzione di transizione δ associa ad una coppia (stato, simbolo letto sul nastro della macchina) un insieme di (0, 1, o più) di triple (stato, nuovo simbolo da scrivere, direzione della testina). Una MdT nondeterministica è definita nel modo seguente: a ogni punto in una computazione, la macchina può procedere secondo diverse possibilità. La funzione di transizione per una MdT nondeterministica ha la forma: δ: Q x Γ -> P(Q x Γ x {L, R}). La computazione di una MdT nondeterministica è un albero di cui i salti corrispondono a possibilità differenti per la macchina.

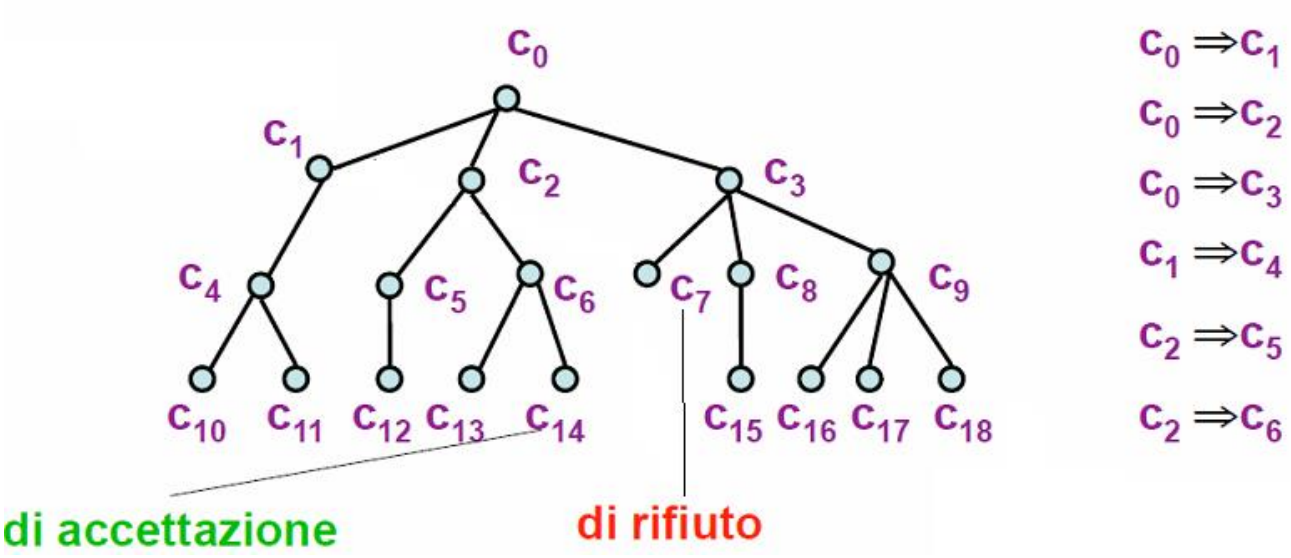
Una macchina di Turing non deterministica (NMdT) è definita dalla tupla N= (Σ, Γ, Q, δ, q_0, accept, reject) dove Γ contiene i caratteri speciali "blank" (ed eventualmente lo "starting tape") e δ è la relazione di transizione definita nel seguente modo: δ : (Q\{accept, reject} × Γ) → P(Q × Σ × {l, r}). Cioè, da ogni configurazione si può transire in una o più configurazioni simultaneamente. La configurazione successiva non è univocamente determinata. In realtà si possono avere 0, 1 o più mosse che permettono una possibile configurazione. La computazione non è più una successione di configurazioni ma un albero di configurazioni. Il grado di non determinismo corrisponde al massimo numero di configurazioni generate dalla funzione. Una NMdT si comporta come se ad ogni passo instanziasse nuove NMdT, ognuna delle quali elabora una delle configurazione ottenute dalla funzione di transizione.

Una stringa in input è accettata da una NMdT M se dalla configurazione iniziale può raggiungere una configurazione di accettazione (cioè, una configurazione il cui stato è "accept")

Il linguaggio L(M) accettato da una NMdT M è l'insieme delle stringhe in input che sono accettate da M

- x ϵ L(M) significa che c'è **una** computazione di M sulla stringa x che porta ad uno stato accept. Si noti che esiste anche la possibilità in questo caso che altre computazioni su x non portano a nessuno stato (interruzione della computazione per mancanza di mosse possibili) oppure ci sono computazioni che continuano per sempre.
- x ∉ L(M) significa che **nessuna** computazione di M su x porta ad uno stato "accept", ovvero che **tutte** le computazioni sono non accettanti.

Esempio di albero di computazione:



Osservazioni:

- Si osservi che i due concetti sono molto "asimmetrici" tra di loro: il primo richiede che esista almeno una computazione accettante, quindi se abbiamo a disposizione una soluzione è facile certificarla come tale.
- Viceversa una stringa è rifiutata se tutte le possibili computazioni sono rifiutanti e questo può essere molto più difficile da verificare.

Teorema (3.19): Un linguaggio è decidibile se e solo se qualche MdT nondeterministica lo decide.

Teorema (3.16): Per ogni NMdT M esiste una MdT (deterministica) $M^{(3)}$ a 3 nastri equivalente (= Ogni NMdT ha un'equivalente MdT). La simulazione di M tramite $M^{(3)}$ si ottiene visitando l'albero delle computazioni di M utilizzando l'algoritmo che può **visitare in ampiezza** gli alberi. Non si può **visitare in profondità** poiché si potrebbe visitare un ramo corrispondente ad una computazione infinita.

Ad ogni passo di computazione di M si possono generare un numero massimo di scelte **d** pari al grado di non determinismo della macchina M. Supponiamo di numerare con numeri compresi tra 1 e d le scelte derivanti dalla funzione di transizione di M.

In tal modo ogni computazione potrà essere identificata come una sequenza di numeri compresi tra 1 e d, ognuno dei quali identifica una delle possibili d scelte generate dalla funzione di transizione di M.

Non tutte le combinazioni saranno valide poiché non è detto che ad ogni passo la funzione di transizione generi esattamente d scelte.

Dopo "i" passi di computazione della macchina M, esistono al più d^i stringhe di lunghezza "i" che rappresentano particolari computazioni di M.

La macchina $M^{(3)}$ è organizzata nel seguente modo:

- primo nastro contiene la stringa di input
- il secondo nastro contiene, in ordine lessicografico, **tutte** le sequenze finite composte da cifre comprese tra 1 e d. Ogni sequenza di lunghezza s sul secondo nastro è in corrispondenza con una computazione di M di **s** passi. Dunque per s passi avremo al più $\mathbf{d^s}$ stringhe sul secondo nastro.
- il terzo nastro eseguirà la simulazione vera e propria

La simulazione avviene nel seguente modo:

- Al passo s-esimo di computazione di M, si generano sul secondo nastro stringhe di lunghezza s corrispondenti a possibili sequenze di scelte (nondeterministiche) per computazioni di lunghezza s
- per ogni sequenza di lunghezza s

    a. si copia il contenuto del primo nastro sul terzo nastro

    b. si scandisce il secondo nastro, e per ogni j corrispondente all'indice di una possibile scelta si applica la j-esima scelta al terzo nastro

Esempio: Se s=4 e d=2 e la sequenza è 2112, $M^{(3)}$ sceglie per la prima mossa la seconda transizione disponibile, per la seconda mossa la prima, ecc....

Se esiste un cammino di lunghezza s che porta M in uno stato finale, esiste sicuramente una fase di calcolo di $M^{(3)}$ che percorre tale cammino (in tempo finito). Se viceversa tale cammino non esiste allora anche $M^{(3)}$ non raggiungerà mai lo stato finale. Tutte le regole di transizione si numerano da 1 a d.

Visita livello 1:

- L'input è sul nastro 1. Si genera la stringa di indirizzo sul nastro 2 e si copia l'input sul nastro 3.
- Per la sequenza 1, c'è solo una configurazione possibile: quella iniziale. Si copia la stringa dal nastro 1 al nastro 3.

Visita livello 2:

- Si generano in sequenza le stringhe di lunghezza 2 sul nastro 2, per esempio 11 e 12.
- Per ogni stringa prodotta, si copia l'input dal primo al terzo nastro e si modifica in base all'applicazione della giusta regola di transizione (prima la 1 e poi la 2). Poi si ricopia il tutto nel nastro 1
- La stringa sul nastro 1 corrispondente alle configurazioni successive possibile della NMdT, separate da #

Visita livello 3:

- Si generano in sequenza le stringhe di lunghezza 3 sul nastro 2, per esempio 111, 112, 121 e 122.
- Per ogni stringa prodotta, si copia la configurazione da modificare dal primo al terzo nastro (per 111 e 112 fino al # e poi per 121 e 122 la parte dopo #) e si modifica in base alla regola di transizione scelta.
- La stringa sul nastro 3 corrisponde a quattro configurazioni della NMdT, tutte separate da #. Si sposta questa stringa dal terzo al primo nastro e si passa al livello successivo ….

Ad ogni passo j della computazione di M, la macchina deterministica $M^{(3)}$ deve compiere esattamente $\mathbf{jd}^{\mathbf{j}}$ passi, cioè la lunghezza del cammino per il numero dei cammini.

Se M termina in k passi allora $M^{(3)}$ esegue al più in un numero di passi pari al più a $\sum_{j=0}^{k} \quad j * d^j$, che è esponenziale in k. Quindi una macchina di Turing non deterministica può essere simulata da una macchina di Turing deterministica multi-nastro in un tempo esponenziale.

Una macchina non deterministica può "risolvere" un problema (ad esempio, accettare un linguaggio) in tempo polinomiale rispetto alla lunghezza della stringa mentre la simulazione effettuata da una macchina deterministica richiede tempo esponenziale. È chiaramente possibile l'aggiunta simultanea di più nastri e del nondeterminismo alla definizione di base di una MdT.

La macchina ottenuta ha sicuramente una rappresentazione più succinta della macchina classica, ma non ha un maggiore potere computazionale:

- È possibile trasformare una MdT multinastro non deterministica in una mono-nastro, applicando la trasformazione vista per le MdT deterministiche;
- Poi si può rimuovere il nondeterminismo;
- Le due trasformazioni richiedono un numero di passi esponenziali rispetto alla taglia della macchina di partenza.

Corollario 3.18: Un linguaggio è Turing-riconoscibile se e solo se qualche NMdT lo riconosce (è una conseguenza del teorema 3.16).

Capitolo 4:

4.1:

Ripartiamo da due concetti fondamentali della macchina di Turing. Una macchina di Turing M accetta un input w se esiste una sequenza di configurazioni $c_1,c_2,\ldots c_k$ tale che:

1. c_1 è la configurazione iniziale di M su w
2. Ogni c_i porta a c_{i+1} (tramite la funzione di transizione), e
3. c_k è una configurazione di accettazione.

L'insieme di stringhe che M accetta è il linguaggio riconosciuto da M e denotato L(M).

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

Quando avviamo una macchina di Turing su un dato input, se la macchina accetta l'input prima o poi si fermerà sullo stato di accettazione "yes", altrimenti può fermarsi sullo stato di rifiuto "no" o ciclare per sempre.

Spesso, distinguere una TM in loop da una che sta solo facendo una lunga computazione non è un compito facile. Per questa ragione, è preferibile usare Macchine di Turing che si fermano su ogni possibile input. Queste macchine sono dette **decisori**, perché **decidono** un linguaggio (decidono cioè per ogni *stringa se è accettata o meno).*

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

Abbiamo già detto che le macchine di Turing sono modelli computazionali più potenti degli NFA. In particolare è possibile definire degli algoritmi per testare se:

- Un DFA/NFA accetta una data stringa
- Un linguaggio di un DFA/NFA è vuoto o meno
- Due automi finiti sono equivalenti (cioè accettano lo stesso linguaggio).

Tutti questi problemi possono essere rifrasati in termini di linguaggi. Per esempio, il problema di accettazione per un DFA può essere espresso tramite un linguaggio A_{DFA} contenente la codifica di tutti i possibili DFA e le stringhe che essi accettano nel modo seguente:

A_{DFA} = { (B, w) | B è un DFA che accetta la stringa in input w}

Il problema di verificare se UN DFA *B* accetta un input w equivale al problema di verificare se ***(B, w) è un elemento di*** A_{DFA}. Similmente, possiamo riformulare gli altri problemi in termini di verifica di appartenenza di un elemento in un linguaggio. Mostrare che il linguaggio A_{DFA} è decidibile equivale a mostrare che il relativo problema computazionale è decidibile.

Teorema 4.1: A_{DFA} è un linguaggio decidibile.

Idea della dimostrazione: Semplicemente si deve presentare una MdT M che decida A_{DFA}.

M = "Su input (B, w) dove B è un DFA è una stringa:

1. Simula B su input w.
2. Se la simulazione finisce in uno stato accettante, accetta, se finisce in uno non-accettante, rigetta."

Dimostrazione: Prima, esamino l'input (B,w). È una rappresentazione di un DFA B insieme a una stringa w. Una rappresentazione ragionevole di B è semplicemente una lista dei suoi 5 componenti: Q, ∑, δ, q_0, F. Quando M riceve questi input, M determina prima quale rappresenta propriamente un DFA B e una stringa w. Senno, M rigetta. Allora M termina direttamente la simulazione. Tiene traccia dello stato corrente di B e della posizione corrente di B nell'input w scrivendo quest'informazione sul suo nastro. Inizialmente, lo stato corrente di B è q_0 e la posizione corrente di B è il simbolo più a sinistra di w. Gli stati e la posizione sono aggiornati secondo la funzione di transizione. Quando M finisce di processare l'ultimo simbolo di w, M accetta l'input se B è uno stato accettante; M rigetta l'input se B è uno stato non-accettante.

Teorema 4.2: A_{NFA} = {(B, w)| B è un NFA che qaccetta in input la stringa w} è un linguaggio decidibile.

Dimostrazione: Una MdT N che decide A_{NFA} deve operare in modo simile alla MdT precedente, simulando un NFA al posto di un DFA. Per farlo dichiaro N come utilizzatore di M come una subroutine, perché M lavora coi DFA, N converte prima l'NFA riceve come input da un DFA prima di passarlo a M.

N = "Su input (B, w), dove B è un NFA e w è una stringa:

1. Converto l'NFA B in un DFA C equivalente usando il teorema 1.39;
2. Eseguo l'MdT M dal teorema 4.1 per input (C, w);
3. Se M accetta, stato accettante, altrimenti stato rigettante."

Eseguire l'MdT M nel 2° passaggio significa incorporare M nel design di N come una sottoprocedura. Similmente si può determinare in che punto un'espressione regolare genera una data stringa.

Teorema 4.3: A_{REX} = {(R. w) | R è un'espressione regolare che genera w} è un linguaggio decidibile.

Dimostrazione: La seguente MdT P decide A_{REX}.

P = "Su input (R, w), dove R è un'espressione regolare e w è una stringa:

1. Converte l'espressione regolare R in un NFA A equivalente usando la procedura data dal Teorema 1.54;
2. Eseguo l'MdT N su input (A, w);
3. Se N accetta, stato accettante; se N rigetta, stato rigettante."

Teorema 4.4: E_{DFA} = {A | A è un DFA e L(A) = Φ} è un linguaggio decidibile.

Dimostrazione: Una macchina di Turing in grado di decidere questo linguaggio è quella mostrata nelle lezioni precedenti per il calcolo della raggiungibilità. In pratica, presa una codifica del DFA, la macchina di Turing verifica se seguendo le regole di transizione è possibile raggiungere uno stato finale da uno iniziale. L'algoritmo opera con un marcatore degli stati raggiungibili. Se ad un certo punto non è possibile marcare

nuovi stati e non si è ancora raggiunto uno stato si accettazione per il DFA allora la macchina di Turing non accetta.

(Parte di EQ_{DFA}) La decidibilità del vuoto per un DFA permette di decidere anche l'equivalenza di due DFA. Infatti, ricordando che i DFA sono chiusi rispetto al complemento, all'unione e all'intersezione, l'equivalenza di due DFA A e B equivale a decidere il vuoto di

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$ = differenza simmetrica

Formalmente la descrizione di E_{DFA} è la seguente: "Su input (M), dove M è un DFA:

1. Segna lo stato iniziale di M;
2. Ripete finché non vengono segnati dei nuovi stati iniziali;
3. Segna ogni stato che ha una transizione che arriva da un altro stato, già marcato;
4. Se nessuno stato è segnato, accetta, altrimenti rigetta."

Problemi decidibili che riguardano i linguaggi context-free: Verranno descritti gli algoritmi che determinano in che modo una CFG genera una stringa in particolare e determinano in che modo il linguaggio di una CFG è vuoto.

Teorema 4.7: A_{CFG} = {(G, w) | G è una CFG che genera la stringa w} è un linguaggio decibile.

Idea della dimostrazione: Per la CFG G e la stringa w voglio determinare in che modo G genera w. Un'idea è usare G attraverso tutte le derivazioni per determinare in che modo ognuna è una derivazione di w.

Quest'idea dà una MdT che è un riconoscitore, ma non un decisore per A_{CFG}. Per rendere questa MdT un decisore ci si deve assicurare che l'algoritmo prova solo molte derivazioni finite. Nella CNF è stato mostrato che se G era in CNF, ogni derivazione di w ha 2n – 1 passaggi, dove n è la lunghezza di w. In quel caso, controllando solo le derivazioni con 2n – 1 passaggi per determinare in che modo G genera w sarebbe sufficiente.

Dimostrazione: L'MdT S per A_{CFG} è definita:

S = "Su input (G, w) dove G è una CFG e w è una stringa:

1. Converto G a una grammatica equivalente in CNF;
2. Listo tutte le derivazioni con 2n – 1 passaggi, dove n è la lunghezza di w, tranne se n = 0, allora listerò tutte le derivazioni con un passaggio;
3. Se ognuna di queste derivazioni genera w, accetta, se non, rigetta".

Teorema 4.9: Ogni CFL è decidibile.

Idea della dimostrazione: Sia A un CFL. L'obiettivo è mostrare che A è decidibile. Una (pessima) idea è convertire un PDA per A direttamente in una MdT, che non è così difficile da simulare, perché simulare uno stack con i nastri delle MdT è molto semplice. I PDA per A possono essere nondeterminsitici, ma sembra essere okay perché si può convertire in una MdT nondeterminstica. Comunque c'è una difficoltà, certi salti della computazione del PDA possono andare avanti per sempre, leggendo e scrivendo lo stack senza mai fermarsi. L'MdT simulante allora dovrebbe avere anche dei salti non-halting nella sua computazione e così l'MdT non sarebbe un decisore. Un'idea differente è necessaria. Invece, si dimostrerà questo teorema con l'MdT S.

Dimostrazione: Sia G una CFG per A e disegno una MdT M_G che decide A. Si costruisce una copia di G in M_G. Funziona nel modo seguente:

M_G = “Su input w:

1. Eseguo MdT S su input (G, w);
2. Se questa macchina accetta, stato accettante; se rigetta, stato rigettante.”

Macchine di Turing Universali: 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.

Sino ad ora abbiamo considerato TM in grado di effettuare un solo tipo di calcolo, sono cioè dotate di un insieme di oggetti che consente loro di calcolare una singola funzione (ad esempio la somma, il prodotto, ecc.).

Tuttavia, è possibile definire una TM, detta **Macchina di Turing Universale** (MTU), in grado di simulare il comportamento di ogni altra TM. Questo è reso possibile dal fatto che gli oggetti che definiscono ogni TM possono essere rappresentati in modo tale da essere scritte sul nastro di una TM. In particolare, è possibile sviluppare un metodo per codificare mediante numeri naturali la tavola di transizione di una qualsiasi TM.

In questo modo, il codice di una TM può essere scritto sul nastro e dato in input a un'altra TM.

La codifica può essere definita nel modo seguente: dato un codice, si possa ottenere la tavola di transizione corrispondente e viceversa mediante un procedimento algoritmico (una codifica che gode di questa proprietà è detta una **codifica effettiva**).

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 (si veda la figura in alto a lato):

- la codifica della tavola di transizioni di una TM M;
- 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 (come mostrato nella figura a lato in basso).

Formalmente si dice che U(M ; I) = M(I). 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|)⌉

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.

4.2:

Definizione 4.12: Si assuma di avere due insiemi A e B, e una funzione f da A a B. La funzione f è detta corrispondenza (biettiva) se

- f(a) ≠ f(b) se e solo se a ≠ b
- per ogni b ϵ B c’è un a ϵ A tale che f(a)=b.

Due insiemi A and B sono della stessa taglia se c’è una corrispondenza f:A ⍰ B. In una corrispondenza, ogni elemento di A corrisponde ad un solo elemento di B e ogni elemento di B ha un unico elemento di A che corrisponde ad esso. Con il metodo di Cantor, è possibile mostrare, ad esempio, che l’insieme dei numeri naturali e l’insieme dei numeri naturali pari hanno la stesa taglia. Infatti la corrispondenza f tra i due insiemi è semplicemente *f(n)* = 2n.

Definizione (4.14): Un insieme è contabile se è finito oppure se ha la stessa taglia dell’insieme dei numeri naturali N. Si consideri l’insieme dei numeri razionali positivi Q = {m/n | m,n ϵ N}. Questo insieme è contabile? La risposta è **sì** perché esiste un modo per associare tutti gli elementi di Q agli elementi di N tramite una corrispondenza. Il metodo è il seguente: si dispongono tutti i numeri di Q in una matrice infinita dove la i-esima riga contiene tutti i numeri con numeratore i e la j-esima colonna tutti i numeri con denominatore j.

La matrice si può trasformare in una lista leggendo i suoi elementi in diagonale. Si noti che non leggiamo gli elementi per riga perché le righe sono infinite, altrimenti partendo dalla prima riga non visiteremmo mai la seconda. Non tutti gli insiemi infiniti possono essere messi in corrispondenza con N. Questi insiemi sono detti non contabili.

Metodo della diagonalizzazione: Per provare la non decidibilità di un linguaggio, si può ricorrere al metodo della diagonalizzazione introdotto da Georg Cantor. Confrontare due insiemi finiti è semplice: basta contare gli elementi che essi contengono. Nel caso di insiemi infiniti, invece, non possiamo utilizzare questo metodo, perché chiaramente non finirebbe mai.

Cantor propose un metodo alternativo basato sull’osservazione che due insiemi (finiti o infiniti) hanno la stessa taglia se gli elementi di un insieme possono essere accoppiati con gli elementi dell’altro.

(Sarebbe anche il teorema 4.17 + dimostrazione) Per esempio, l'insieme dei numeri reali R non è contabile. La prova procede per assurdo, mostrando che esiste un termine che non ha corrispondenza in N. Il numero si costruisce nel seguente modo: da tutti i numeri f(i), si costruisce il termine t= $0,c_1c_2$…. dove ogni c_i è tale da essere differente dalla i-esima cifra decimale di f(i). Per esempio se f(1)= 4,14… e f(2)=6,45… allora t può essere 0,27… Questa costruzione assicura che t non corrisponde a nessun f(i).

Corollario 4.18: Trasportando questo risultato nella teoria della computazione, se uno mostra che **i linguaggi non sono contabili**, mentre **le macchini di Turing sono contabili**, segue che alcuni linguaggi non sono decidibili o **addirittura non sono Turing-riconoscibili**.

Per mostrare che l'insieme delle macchine di Turing è contabile si osservi che l'insieme di tutte le stringhe **Σ* è contabile**. La funzione di corrispondenza è tra le stringhe di lunghezza i e l'insieme dei numeri necessari per listare tutte le parole di quella lunghezza. L'insieme delle macchine di Turing è contabile perchè ogni macchina è una codifica su Σ.

Per mostrare che l'insieme dei linguaggi è non contabile, prima si osservi che l'insieme B delle sequenze infinite binarie è non contabile (prova simile al caso dei reali). Sia C l'insieme di tutti i linguaggi sull'alfabeto Σ. Mostriamo adesso che anche C è non contabile, dando una corrispondenza con B.

Sia Σ* ={s_1, s_2, s_3, ... }. Adesso associamo i linguaggi di C alle stringhe di B in modo che ciascun linguaggio A in C abbia un'unica sequenza in B.

La sequenza binaria è così costituita: l'i-esimo bit della sequenza è 1 se s_i ϵ A e 0 altrimenti (formalmente questa sequenza è chiamata la X_A caratteristica di A). Per esempio, se A è il linguaggio delle stringhe che iniziano per 0 sull'alfabeto Σ = {0,1}, la sua caratteristica X_A è la seguente:

$$\begin{aligned}
\Sigma^* &= \{\ \epsilon\ ,\ 0\ ,\ 1\ ,\ 00\ ,\ 01\ ,\ 10\ ,\ 11\ ,\ 000\ ,\ 001\ ,\ \cdots\ \}\ ; \\
A &= \{\ \quad 0\ ,\ \quad 00\ ,\ 01\ ,\ \quad\quad 000\ ,\ 001\ ,\ \cdots\ \}\ ; \\
\chi_A &= \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad \cdots \quad .
\end{aligned}$$

Non decibilità: In un tipo di problema insolvibile viene dato un programma del computer e una specificazione precisa di cosa dovrebbe fare. C'è la necessità di verificare che il programma performi come specificato, perché sia il programma sia la specificazione sono oggetti matematici precisi, si spera di automatizzare il processo di verificazione rendendo questi oggetti dei computer opportunamente programmati. Il problema generale della verificazione del software non è un solvibile per un computer.

Sia A_{TM} (non decidibile) analogo ad A_{DFA} e A_{CFG} (decidibili).

<u>Teorema (4.11):</u> Il problema dell'appartenenza è indecidibile per la macchina di Turing. In seguito mostriamo la prova della sua indecidibilità così come di altri problemi. Questo al fine di introdurre tecniche di prova di indecidibilità.

Il problema di determinare se una data macchina di Turing accetta una data stringa si può ridurre al problema di decisione del linguaggio seguente A_{TM}, in linea con quanto fatto per gli NFA:

A_{TM} = {(M, w) | M è una macchina di Turing e M accetta w}

Prima di mostrare che A_{TM} è indecidibile, mostriamo che è Turing-riconoscibile (Turing-recognizable).

Questo mostra formalmente che i riconoscitori sono più potenti dei decisori. 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:

- U simula M su w.

- Se M non entra mai in uno stato di accettazione, allora U accetta; se M entra in uno stato di rifiuto, allora U rigetta."

Si noti che U cicla su un input (M, w) se M cicla su w, il che dimostra perché U non decide A_{TM}. Se U avesse un modo per determinare che M non si fermerà mai su un certo input w, allora U potrebbe non accettare un tale input.

Come vedremo in seguito, A_{TM} permette di dimostrare che il **problema della fermata** (**halting problem)** è indecidibile. Mostreremo cioè che nessuna macchina di Turing è in grado di determinare se un'altra macchina di Turing si fermerà prima o poi, o meno.

Un linguaggio non decidibile: Supponiamo per assurdo che il problema dell'appartenenza è decidibile. Dimostreremo che tale ipotesi produce una contraddizione.

Dimostrazione: Se A_{TM} è decidibile, per l'universalità esiste una macchina di Turing H che decide M, cioè, data M (più precisamente, una sua codifica) e un qualsiasi input w di M, la macchina H si ferma e accetta se M accetta w, mentre H si ferma e non accetta se M non riesce ad accettare w. Schematicamente H(M,w) si comporta nel modo seguente:

- accetta se la computazione di *M* con ingresso *w* termina con accettazione,
- rifiuta se la computazione di *M* con ingresso *w* non accetta (cicla o rifiuta).

Adesso, si consideri una nuova macchina di Turing D che utilizzi H come subroutine, tale che D chiama H per stabilire come si comporta M su input M e restituisce l'opposto del valore ottenuto. In pratica, D si comporta come segue:

- rifiuta se la computazione di *M* con ingresso *M* termina con accettazione,
- accetta se la computazione di *M* con ingresso *M* non accetta.

Nota: Il fatto di eseguire una macchina sulla propria descrizione non deve preoccupare. Questo è simile al caso in cui un programma lavora con se stesso come input. Per esempio, un compilatore è un programma che trasforma altri programmi ed entrambi possono essere scritti con lo stesso linguaggio.

Cosa succede se a D diamo in input la stessa macchina D (in pratica la sua codifica).

Ci chiediamo cioè se D accetta o meno con input D? Abbiamo che *D(D) accetta se D* non accetta *D. Viceversa, rifiuta D quando D accetta D.* Questa è ovviamene una contraddizione. Dunque, H e D non possono esistere.

In pratica, abbiamo dimostrato, in base alla definizione di D, che D con input D dà origine a un calcolo che termina se e soltanto se il calcolo di D per l'input D non termina, il che è palesemente assurdo.

Ne consegue quindi che una macchina che si comporta come H non può esistere, e che quindi, se è vera la tesi di Church, non può esistere un algoritmo che permette di decidere il problema dell'appartenenza.

Dunque il problema dell'appartenenza è indecidibile (o non ricorsivo). Per meglio rendersi conto di questo risultato, consideriamo il metodo della diagonalizzazione di Cantor. Ancora una volta, si assuma che H sia in grado di decidere A_{TM}, che D, costruita a partire da H, su input M accetta esattamente quando M non accetta l'input M. Infine, si dia in input a D lo stesso D. Allora si ha che:

- H accetta *(M, w) esattamente quando M accetta w*
- *D rifiuta M esattamente quando M accetta M (come input)*
- *D rifiuta D quando D accetta D (contraddizione)*

Vediamo adesso come la diagonalizzazione porta a mostrare un assurdo, sotto l'ipotesi che A_{TM} è decidibile.

Si consideri la tabella dei comportamenti di *H* e *D. In questa tabella riportiamo sulle righe le macchine di Turing M_1, M_2 ecc, e sulle colonne le loro descrizioni.* Ogni combinazione riga i e colonna j dice se una macchina M_i accetta o meno M_j. In particolare in (i,j) scriviamo *accept se* M_i accetta o meno M_j e lasciamo la cella vuota se M_i rifiuta o cicla su quell'input.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

Nella figura (1), ogni cella (i, j) rappresenta il risultato di H sugli input della figura mostrata nella slide precedente. Dunque, se M_3 no accetta M_2, nella cella (3,2) scriviamo reject, perché H rifiuta l'input (M_3, M_2). Nella figura (2), aggiungiamo la riga e la colonna D alla figura in alto.

Si noti che D computa l'opposto dei valori presenti sulla diagonale. La contraddizione arriva in corrispondenza del punto interrogavo dove il valore deve essere l'opposto di se stesso.

1)

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$			$\vdots$		

2)

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

1. Mostriamo una condizione necessaria e sufficiente per un linguaggio affinché sia ricorsivo(recursive)
2. Mostriamo l'esistenza di linguaggi che non sono Turing-riconoscibile (Turing-recognizable).

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

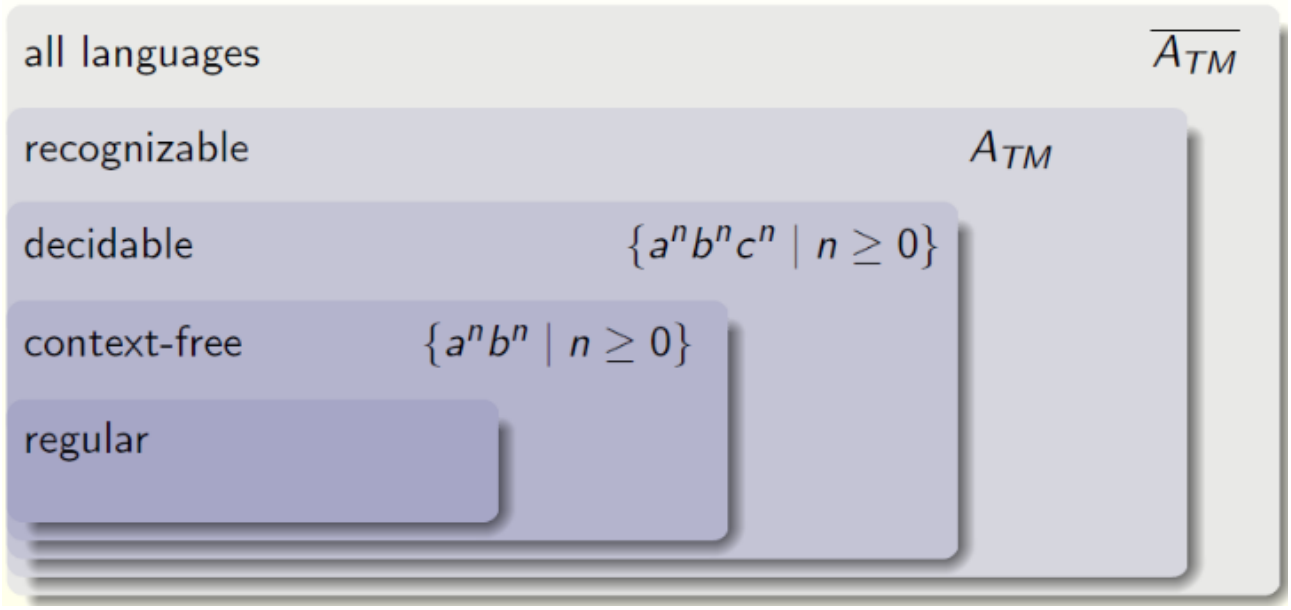
Teorema (4.22): Un linguaggio è decidibile se e solo se è **Turing-riconoscibile** e **co-Turing-riconoscibile**.

Dimostrazione: Ci sono due direzioni da provare. Dapprima assumiamo che A sia decidibile. Questa direzione segue dal fatto che il complemento di un linguaggio decidibile è decidibile e dal fatto che ogni linguaggio decidibile è anche Turing-riconoscibile.

Per l'altra direzione, se A e il suo complemento comp(A) sono Turing-riconoscibile, possiamo usare una Macchina di Turing a 2 nastri dove sul primo decidiamo A e sul secondo decidiamo comp(A). Data una qualsiasi parola w essa appartiene ad A o a comp(A). Qualsiasi sia il caso, si ha che comunque la macchina su w si fermerà, per definizione di accettazione.

Corollario (4.23): Complemento(A_{TM}) non è Turing-riconoscibile.

Dimostrazione: Noi sappiamo che A_{TM} è Turing-riconoscibile. Se anche comp(A_{TM}) fosse Turing-riconoscibile, allora per il teorema precedente anche A_{TM} sarebbe decidibile. Ma noi sappiamo che A_{TM} non è decidibile. Dunque, comp(A_{TM}) non può essere Turing-riconoscibile.



Capitolo 5:

5.1:

Problemi non decidibili dalla teoria del linguaggio: Già è stata stabilita la non decidibilità di A_TM, il problema di determinare se una MdT accetta un input dati. Si consideri un problema collegato, HALT_TM, il problema di determinare se una MdT si ferma (accettando o rigettando) su un dato input. Questo problema è largamente conosciuto come halting problem (problema della fermata). Si usa la non decidibilità di A_TM per dimostrarne la non decidibilità.

Teorema 5.1: $HALT_{TM}$ = {(M, w) | M è una MdT e M si ferma sull'input w} non è decidbile.

Idea della dimostrazione: Questa dimostrazione è per contraddizione. Si assuma che $HALT_{TM}$ è decidibile e si usa la supposizione che anche A_{TM} è decidibile. L'idea chiave è mostrare che A_{TM} è riducibile a $HALT_{TM}$. Si assuma che ci sia una MdT R che decide $HALT_{TM}$. Allora si usa R per costruire S, una MdT che decide A_{TM}. Per avere un'idea del modo di cosruire S, si finga di essere S. L'obiettivo è decidere A_{TM}, ricevo in input (M, w).

Devo dare come output lo stato accettante, quindi M accetta w. Se accetta o rigetta faccio lo stesso. Ma posso non essere capace di determinare se M va in loop, e in quel caso la mia simulazione non termina.

Suppongo di avere l'MdT R che decide $HALT_{TM}$. Con R posso testare se M si ferma su w. Se R indica che M non si ferma su w rigetto perché (M, w) non è A_{TM}. Se R indica che M si ferma su w, io posso fare la simulazione senza alcun problema di poter finire in loop. Dunque, se l'MdT R esiste posso decidere A_{TM}, ma so che A_{TM} non è decidibile. Vado in contraddizione e posso concludere che R non esiste, allora $HALT_{TM}$ non è decidibile.

Dimostrazione: Si assuma che per lo scopo di ottenere una contraddizione tale che MdT R decida $HALT_{TM}$. Costruisco S per decidere A_{TM}, con S che opera nel modo seguente:

S = "Su input (M, w), una codifica di una MdT M e una stringa w:

1. Esegue la MdT R su input (M, w);
2. Se R rigetta, stato rigettante;
3. Se R accetta, simula M su w finché si ferma;
4. Se M ha accettato, stato accettante; se M ha rigettato, stato rigettante."

Chiaramente, se R decide $HALT_{TM}$, allora S decide A_{TM}. Perché A_{TM} è indecidibile, anche $HALT_{TM}$ deve esserlo.

Teorema 5.2: E_{TM} = {(M) | M è una MdT e L(M) = 0} non è decidibile.

Idea della dimostrazione: Si assuma che E_{TM} è decidibile e allora si deve mostrare che A_{TM} è decidibile. Sia R una MdT che decide E_{TM}. Uso R per costruire l'MdT S che decide A_{TM}.

Un'idea è per S di eseguire R su input (M) e si vede se accetta. Se lo fa so che L(M) è vuoto e dunque che M non accetta w. Ma se R rigetta (M), allora so che L(M) non è vuoto e accetta qualche stringa – ma non so ancora se M accetta la stringa w. Quindi devo usare un'idea differente.

Invece di eseguire R su (M), si esegue R su una modificazione di (M). Modifico (M) per garantire che M rigetti tutte le stringhe tranne w, ma su input w lavora come al solito. Allora uso R per determinare se la macchina modificata riconosce il linguaggio vuoto. La sola stringa che la macchina può accettare ora è w, così il suo linguaggio sarà non-vuoto se accetta w. Se R accetta quando viene data una descrizione della macchina modificata, si sa che la macchina modificata non accetta niente e che M non accetta w.

Dimostrazione: Si scriva l'appena descritta macchina modificata usando la solita notazione. Sia M_1:

M_1 = "Su input x:

1. Se $x \neq w$, rigetta;
2. Se x = w, eseguo M su input w e accetta se M lo fa."

Questa macchina ha la stringa w come parte della sua descrizione. Conduce il test se x = w nel modo ovvio, scannerizzando l'input e confrontandolo carattere per carattere con w per determinare se sono uguali.

Mettendo tutto insieme, si assuma che l'MdT R decida E_{TM} e costruisco l'MdT S che decide A_{TM} come segue.

S = "Su input (M, w) una codifica di una MdT M e una stringa w:

1. Uso la descrizione di M e w per costruire M_1 nel modo descritto;
2. Eseguo R su input (M_1);
3. Se R accetta, stato rigettante; se R rigetta, stato accettante."

Si noti che S deve attualmente essere capace di computare una descrizione di M_1 da una descrizione di M e w. È capace di fare così perché ha solo bisogno di aggiungere stati extra a M per performare il test per x=w.

Se R fosse un decisore per E_{TM}, S sarebbe un decisore per A_{TM}, che non può esistere, così so che E_{TM} non è decidibile.

Teorema 5.3: $REGULAR_{TM}$= {(M) | M è una MdT e L(M) è un linguaggio regolare} non è decidibile.

Idea della dimostrazione: Si assuma che $REGULAR_{TM}$ è decidibile per una MdT R e uso questa supposizione per costruire una MdT S che decide A_{TM}. Meno ovvio ora è come usare l'abilità di R per assistere S nel suo intento. L'idea è far di prendere a S il suo input (M, w) e modificare M così che la risultante MdT riconosca un linguaggio regolare se e solo se M accetta w. Chiamo la macchina modificata M_2. Designo M_2 per riconoscere il linguaggio regolare $\{0^n1^n \mid n >= 0\}$ se M non accetta w, e riconosce il linugaggio regolare $\sum^*$ se M accetta w. Si deve specificare come S può costruire M_2 da M e w. M_2 lavora automaticamente per accettare tutte le stringhe in $\{0^n1^n \mid n >= 0\}$. In aggiunta, se M accetta w, M_2 accetta tutte le altre stringhe. Si noti che la MdT M_2 non è costruita per tutti i supposti che vengono eseguiti su certi input. Costruisco M_2 solo per lo scopo di raggiungere la sua descrizione nel decisore per $REGULAR_{TM}$ che è stato supposto di esistere. Una volta che questo decisore restituisce la sua risposta si può usare per ottenere la risposta se M accetta w. Dunque, si può decidere A_{TM}, una contraddizione.

Dimostrazione: Sia R una MdT che decidere $REGULAR_{TM}$ e costruisco l'MdT S per decidere A_{TM}. Allora S lavora nel modo seguente:

S = "Su input (M, w), dove M è una MdT e w è una stringa:

1. Costruisco la seguente MdT M_2.
   M_2 = "Su input x:
   - Se x ha la forma 0^n1^n, accetta;
   - Se x non ha questa forma, eseguo M su input w e accetto se M accetta w."
2. Eseguo su input (M_2);
3. Se R accetta, stato accettante, se R rigetta, stato rigettante."

Teorema 5.4: EQ_{TM} = {(M_1, M_2) | M_1 e M_2 sono MdT e $L(M_1) = L(M_2)$} non è decidibile.

Idea della dimostrazione: Mostro che se EQ_{TM} fosse decidibile anche E_{TM} dovrebbe esserlo, in quanto EQ_{TM} è una riduzione di E_{TM}. L'idea è semplice, E_{TM} è il problema di determinare se un linguaggio di una MdT è vuoto. EQ_{TM} è il problema di determinare se i linguaggi di 2 MdT sono uguali. Se uno di questi linguaggi fosse vuoto, si finirebbe nel problema di determinare se il linguaggio dell'altra macchina è vuoto (problema di E_{TM}). Così in un senso, il problema di E_{TM} è un caso speciale del problema di EQ_{TM} dove una delle macchine è fissata per riconoscere il linguaggio vuoto. Quest'idea si attua rendendo la riduzione semplice.

Dimostrazione: Sia l'MdT R decisore di EQ_{TM} e costruisco l'MdT S per decidere E_{TM} come segue.

S = "Su input (M), dove M è una MdT:

1. Eseguo R su input (M, M_1) , dove M_1 è una MdT che rigetta tutti gli input.
2. Se R accetta, va nello stato accettante; se R rigetta, stato rigettante."

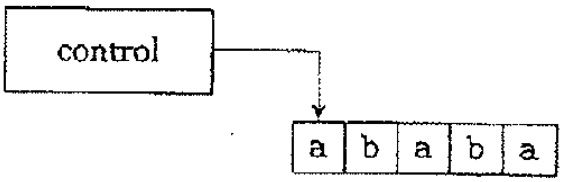
Se R decide EQ_{TM}, S decide E_{TM}, ma E_{TM} non è decidibile, quindi EQ_{TM} dev'essere anche non decidibile.

Riduzione tramite le storie di computazione:

Definizione 5.6: Le linear bounded automaton (LBA) sono Macchine di Turing con una sola limitazione: la testina sul nastro non può muoversi oltre la porzione di nastro contenente l'input.

Se un LBA prova a muovere la sua testina oltre l'input, la testina rimane ferma in quel punto come accade nella Macchina di Turing classica quando la testina tenta di andare più a sinistra del primo simbolo in input

(o il simbolo di start se utilizzato). Un LBA è una Macchina di Turing con memoria limitata, com'è mostrato nella figura sottostante. Può risolvere solo problemi il cui input non superi la capienza del nastro.



Usando un alfabeto del nastro più ampio dell'alfabeto di input si ha che la memoria disponibile viene incrementata di un fattore costante. Perciò per un input di lunghezza n, la quantità di memoria disponibile è lineare in n. Da questo deriva il nome di questo modello.

Nonostante questi vincoli sulla memoria, gli LBA sono piuttosto potenti in termini di accettori di linguaggi. Ad esempio, i decisori per A_{DFA} e E_{DFA} sono tutti LBA. Gli LBA sono strettamente più potenti dei PDA. Infatti, si può provare che ogni CFL può essere deciso da un LBA. Inoltre esistono linguaggi accettati da LBA che non sono context-free come ad esempio $L=\{a^nb^nc^n\}$ e $L=\{a^{n!}\}$.

Gli LBA sono anche meno potenti delle Macchine di Turing (si pensi a un qualsiasi linguaggio che ha bisogno di memoria "non constante" in aggiunta all'input per poterlo processare). I linguaggi accettati dagli LBA sono generati da grammatiche denominate ***"context-sensitive".*** Come accennato, gli LBA sono piuttosto potenti in termini di accettori di linguaggi. Vediamone un esempio:

A_{LBA} è il problema di determinare se un LBA accetta il suo input. Formalmente, A_{LBA} ={(M, w) | M è un LBA che accetta una stringa w}. Ricordiamo che il problema analogo per le MT, ovvero A_{TM}, è indecidibile

Teorema 5.9: A_{LBA} è decidibile.

La dimostrazione della decidibilità di A_{LBA} è semplice qualora si possa asserire che il numero di configurazioni possibili è limitato.

LEMMA 5.8: Preso un LBA M con q stati e g simboli nell'alfabeto del nastro. Ci sono esattamente qng^n configurazioni distinte di M per un nastro di lunghezza n.

DIMOSTRAZIONE (del lemma): Ricordando che una configurazione di M è come uno "snaphshot" nel mezzo di una computazione. Una configurazione consiste dello stato di controllo, posizione della testina e il contenuto del nastro. Per quanto riguarda le "grandezze" degli oggetti coinvolti nelle configurazioni, notiamo che: M ha q stati.

La lunghezza del nastro è n, cosi la testina può essere al più in n posizioni differenti, 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 n, ovvero qng^n.

IDEA PER LA DIMOSTRAZIONE (del teorema): Prima di tutto per decidere se un LBA M accetta l'input w, simuliamo M su w con una MdT M'. Durante il corso della simulazione, se M si ferma e accetta o rigetta, allora la prova termina perché M' si ferma. La difficoltà nasce se M gira all'infinito su w. Questo si può ovviare individuando quando la macchina va in loop, cosi possiamo fermarci e rifiutare.

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

- Quando M computa su w, M passa da una configurazione ad un'altra. Se M ripete più volte una stessa configurazione significa che è entrata in un loop.

- Poiché M è un LBA, sappiamo che la capienza del nastro è limitata. Attraverso il lemma visto nella precedente slide, sappiamo che M può avere solo un numero limitato di configurazioni in base alla capienza del suo nastro.
- Quindi M ha un limite di volte prima di entrare in qualche configurazione che ha già fatto precedentemente. Come conseguenza del lemma, possiamo dire che se M non si ferma dopo un certo numero di passi significa che è in loop. Questo M' è in grado di rilevarlo.

DIMOSTRAZIONE (del teorema): L'Algoritmo che decide A_{LBA} è il seguente.

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 esegue 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."

Abbiamo visto un problema decidibile per un LBA che non lo è per una Macchina di Turing classica. Esistono tuttavia problemi indecidibili per le TM che rimangono tali anche per le LBA. Un esempio è il seguente linguaggio:

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

Definizione 5.5 Una Computation Historie (CH) di una MdT M è la sequenza finita di configurazioni che M attraversa per decidere un input. Sia *M* una MdT e *w* un ingresso 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
- ogni C_i segue C_{i-1} secondo le regole di *M*

Nota: Le MdT deterministiche hanno una sola CH per un dato input, mentre le MdT non deterministiche ne possono avere diverse.

Teorema 5.10: E_{LBA} è indecidibile.

RIPRENDI QUI. Idea della dimostrazione: Riduciamo A_{TM} a E_{LBA}. Per ogni MdT M e ingresso w, definiamo un LBA *B* tale che

L(B)={CH | CH è accettante per una data MdT M e un dato ingresso w}

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

Per verificare che ogni sequenza in ingresso per *B* è una CH accettante devono essere verificate le condizioni descritte in precedenza.

Dimostrazione: Costruiamo il decisore *S* per 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 E_{LBA} implica un decisore per A_{TM}. Sappiamo però che A_{TM} è indecidibile, quindi non esiste un decisore per E_{LBA}.

5.2:

Un semplice problema non decidibile: Il fenomeno dell'indecidibilità non è un problema che riguarda solo gli automi. Per mostrare un esempio, consideriamo un problema indecidibile riguardante una "semplice" manipolazione di stringhe: il Post correspondence problem (PCP).

Possiamo descrivere questo problema paragonandolo ad un tipo di puzzle. Iniziamo con un insieme di domini, ognuno contenente due stringhe.

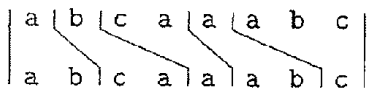
Un pezzo singolo del domino è rappresentato in questo modo: $\left[\frac{a}{ab}\right]$

Mentre un insieme di domini sarà di questo tipo: $\left\{\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right]\right\}$

L'obbiettivo è quello di costruire una lista di questi domini (le ripetizioni sono ammesse) in modo che la stringa dei simboli scritti nella parte superiore dei domini è uguale a quella dei simboli inferiori. Questa lista è chiamata un match.

Ad esempio la seguente lista è un match per questo puzzle: $\left[\frac{a}{ab}\right]\left[\frac{b}{ca}\right]\left[\frac{ca}{a}\right]\left[\frac{a}{ab}\right]\left[\frac{abc}{c}\right]$.

Leggendo l'inizio della stringa otteniamo "abcaaabc", che è uguale alla stringa sotto. Possiamo anche rappresentare questo match deformando i domini in modo che creiamo la corrispondenza da sopra a sotto la linea.



Per alcuni insiemi di domini non è sempre possibile trovare un match. Per esempio l'insieme:

$\left\{\left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{ba}\right]\right\}$ non può contenere un match perché ogni stringa che si trova sopra è più lunga rispetto alla stringa che si trova sotto.

Il **Post correspondence problem** riguarda la possibilità di determinare se una collezione di domini ha un match. Questo problema non è risolvibile con algoritmi. Prima di dare la definizione formale di questo teorema con la sua definizione, formuliamo precisamente il problema e in seguito definiamo il suo linguaggio.

Un instanza del PCP è un insieme P di domini: $\left[\frac{t_1}{b_a}\right]\left[\frac{t_2}{b_2}\right]\left[\frac{t_k}{b_k}\right]$

Un match è una sequenza di interi $i_1, i_2,\ldots, i_l$, dove $t_{i1}, t_{i2},\ldots t_{in} = b_{i1}, b_{i2},\ldots, b_{in}$.

Nota: la stessa coppia può apparire più volte, ovvero, può essere che per certi $m \neq j$, si ha che $i_j = i_m$.

Il problema del PCP è quello di determinare se P ha un match.

Il linguaggio corrispondente al problema è definito nel modo seguente: PCP={<P>| P è un istanza del Post Correspondence Problem con un match}

TEOREMA 5.15: PCP è indecidibile.

Idea della dimostrazione: Per la prova riduciamo A_{TM} al PCP attraverso le Computation Histories.

Dati $\langle M, w \rangle$, costruiamo un insieme P di domino nel modo seguente: Iniziamo con un domino che ha nella parte di sotto la configurazione iniziale di M e sopra una stringa vuota. Aggiungiamo a P dei domino la cui parte superiore corrisponde alla configurazione corrente e la parte inferiore è la configurazione successiva.

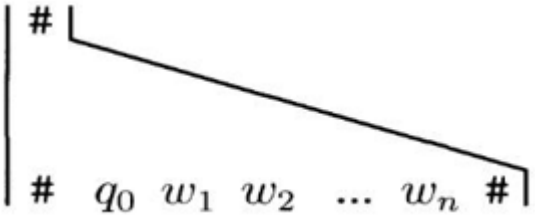
Questi domino devono anche tenere conto di

- Il movimento della testina.
- Aumentare lo spazio del nastro con degli spazi vuoti (quando richiesto).
- Forzare ad usare come primo domino della soluzione del match quello corrispondente alla configurazione iniziale di M.

Per semplicità, assumiamo che ogni soluzione possibile deve sempre iniziare con il primo domino.

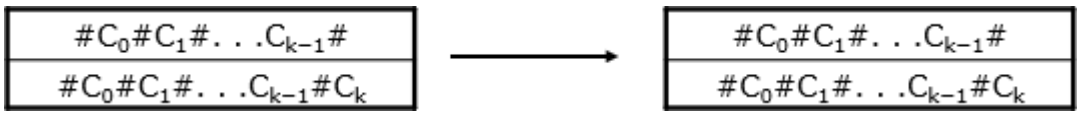
Dimostrazione: Chiameremo il PCP con questa regola aggiuntiva "**modified PCP**" o semplicemente MPCP.

Step 1: Dati $\langle M, w \rangle$, costruiamo il primo domino dell'insieme P come corrispondente alla configurazione iniziale è q_0w nel modo seguente:



Il simbolo # è un nuovo simbolo non appartenente all'alfabeto del nastro di M e serve come marcatore

Step 2: Ad ogni passo di computazione di M facciamo corrispondere dei domino in P in modo da poter copiare la configurazione corrente di M dalla parte di sotto dei domino (delimitata tra #) nella parte superiore e sostituiamo nella parte sottostante corrispondente la prossima configurazione



Per fare questo, abbiamo bisogno di fare più passi elementari nel modo seguente:

Una configurazione racchiusa tra # avrà sempre la forma del tipo α**bqc**β.

Per calcolare la prossima configurazione, occorre copiare α sia nella parte superiore che inferiore

Copiare bqc sulla parte superiore e la prossima configurazione sulla parte inferiore, copiare β sia nella parte superiore che inferiore.

Per copiare α e β, aggiungiamo il sequente domino in P, per ogni a in Γ:

a
a

Step 3: Per ogni transizione (R) del tipo δ(q,c)=(p,d,R), noi aggiungiamo a P il seguente domino

qc
dp

Questo domino ci permetterà di muovere da

#. . . # ab
#. . . #a**bqc**β # ab

→

#. . . # abqc
#. . . #a**bqc**β # abdp

Per il caso speciale in cui c può essere il carattere blank "-", aggiungiamo in P il domino. L'ultimo domino sarà utile quando M leggerà spazi blank a destra della stringa di input

Step 4: Per ogni transizione (L) del tipo δ(q,c)=(p,d,L), noi aggiungiamo a P il seguente domino

bqc
pbd

Questo domino ci permetterà di muovere da

#. . . # a
#. . . #a**bqc**β # a

→

#. . . # abqc
#. . . #a**bqc**β # apbd

Per il caso speciale in cui b può essere il carattere blank "-", aggiungiamo in P il domino

#qc
#pd

Quest'ultimo domino sarà utile quando M è all'inizio dell'input e cerca di andare a sinistra.

Step 5: M accetta w se possiamo raggiungere nel PCP la seguente situazione:

$\#C_0\#\ldots\#C_{n-1}\#$
$\#C_0\#\ldots\#C_{n-1}\#abq_{accept}\beta\#$

Prima di terminare con la riduzione, dobbiamo sistemare il fatto che la stringa nella parte inferiore dei domino è più lunga di una configurazione della parte superiore. Per questo motivo, aggiungiamo a P i domino del tipo:

cq_{accept}
q_{accept}

e

$q_{accept}c$
q_{accept}

Gli ultimi domino permetteranno di scartare uno alla volta i simboli in eccesso, fino ad arrivare alla seguente situazione:

$\#C_0\#\ldots\#q_{accept}c\#$
$\#C_0\#\ldots\#q_{accept}c\#q_{accept}$

Step 6: Per finire dobbiamo aggiungere il seguente ulteriore domino a P:

$q_{accept}\#\#$
#

Con questa costruzione, abbiamo una istanza per MPCP che ammette soluzione se e solo se M accetta w.

Per completare la dimostrazione, dobbiamo forzare il fatto che il primo domino deve essere il primo ad essere usato in ogni possibile soluzione del PCP.

Step 7: Sia "⋆" un nuovo simbolo (cioè non presente in Γ U {#}).

Per ogni stringa s, sia ⋆s la stringa ottenuta inserendo il simbolo "⋆"

Prima di ogni simbolo in s. Per esempio, ⋆(abc) = ⋆a ⋆ b ⋆ c.

Analogamente, definiamo s⋆ e ⋆s⋆ come la stringa ottenuta aggiungendo solo dopo, oppure sia prima che dopo, ogni simbolo di s il simbolo ⋆

Dato l'insieme di domino P costruiti precedentemente, si sostituiscano i domino nel modo seguente:

- t1/b1 = t1⋆/ ⋆ b1⋆
- ogni altro ti/bi= ⋆ ti / b1⋆
- il domino finale q_{accept}##/ # = ⋆q_{accept}# ⋆ #/#

In questo modo, il primo domino deve essere assolutamente il primo altrimenti non ci sarà matching.

5.3:

Mappatura di riducibilità: La nozione di ridurre un problema in una altro può essere definito in uno di più modi. La scelta di quale usare dipende dall'applicazione. La scelta è un semplice tipo di riducibilità chiamata mappatura di riducibilità.

Funzioni computabili: Una MdT computa una funzione iniziando con l'input della funzione sul nastro e fermandosi con l'output della funzione sul nastro.

Definizione 5.17: Una funzione f: ∑* -> ∑* è una funzione computabile se qualche MdT M, su ogni input w si ferma con solo f(w) sul suo nastro.

Definizione formale di mappatura di riducibilità (5.20): Il linguaggio A è riducibile B scritto nel modo seguente $A \leq_m B$ B, se c'è una funzione computabile f: ∑* -> ∑*, dove per ogni w ϵ A <-> f(w) ϵ B.

La funzione f è chiamata la riduzione da A a B.

Una mappatura di riduzione da A a B ci dà un modo di convertire domande sul test di appartenenza di A al test di appartenenza di B. Per testare se w ϵ A si usa la riduzione f per mappare w a f(w) e testare se f(w) ϵ B. Il termine mappatura di riduzione viene dalla funzione o mappatura che dà il significato di fare la riduzione. Se un problema è riducibile per mappatura a un secondo si può ottenere una solutzione al problema originale.

Teorema 5.22: Se $A \leq_m B$ e B è decidibile, allora A è decidibile.

Dimostrazione: Sia M il decisore per B e f la riduzione da A a B. Si descriva un decisore N per A come segue.

N = "Su input w:

1. Computa f(w);
2. Esegue M su input f(w) e dia come output cosa M dà come output."

Chiaramente se w ϵ A, allora f(w) ϵ B perché f è una riduzione da A a B. Dunque M accetta f(w) se w ϵ A. N lavora come desiderato.

Corollario 5.23: Se $A \leq_m B$ e A non è decidibile, allora B non è decidibile.

Teorema 5.28: Se $A \leq_m B$ e B è Turing- riconoscibile, allora A è Turing-riconoscibile. (La prova è uguale a 5.22, solo che M e N sono riconoscitori)

Corollario 5.29: Se $A \leq_m B$ se A non è Turing-riconoscibile, allora B non è Turing-riconoscibile.

In un tipica applicazione di questo corollario, lascio che A sia (not)A_{TM} complemento di A_{TM}. So che (not)A_{TM} non è Turing riconoscibile. La defizione di mappatura di riducibiltà implica che $A \leq_m B$ significa lo stesso di (not) $A \leq_m$ (not)B. Per dimostrare che B non è riconoscibile, si può mostrare che $A_{TM} \leq_m$ (not)B. Si può anche usare la mappatura di riducibilità per mostrare che certi problemi non sono sia Turing-riconoscibili sia co-Turing-riconoscibili.

Teorema 5.30: EQ_{TM} è nè Turing-riconoscibile, né co-Turing-riconoscibile.

Dimostrazione: Prima si mostra che EQ_{TM} non è Turing-riconoscibile. Per farlo si mostra che A_{TM} è riducibile a (not)EQ_{TM}. La funzione di riduzione f funziona nel modo seguente.

F = "Su input (M, w), dove M è una MdT e w una stringa:

1. Costruisco le 2 seguenti macchine:
   M_1 = "Su ogni input: Rifiuta."
   M_2 = "Su ogni input: esegue M su w. Se accetta, accetta anche M_2."
2. Output (M_1, M_2)."

Qui, M_1 accetta niente. Se M accetta w, M_2 accetta tutto e così le 2 macchine non sono equivalenti. In modo converso, se M non accetta w, M_2 accetta niente e le macchine sono equivalenti. Dunque f riduce A_{TM} a (not)EQ_{TM} come desiderato. Per mostrare che (not)EQ_{TM} è non Turing-riconoscibile si dà una riduzione da A_{TM} al complemento (not)EQ_{TM}—ossia, EQ_{TM}. Dunque si mostra che $A_{TM} \leq_m EQ_{TM}$. La seguente MdT G computa la funzione di riduzione g.

G = "Su input (M, w), dive M è una MdT e w una stringa:

1. Costruisco le 2 macchine seguenti, M_1 e M_2:
   M_1 = "Su ogni input: accetta."
   M_2= "Su ogni input:
   1. Eseguo M su w;
   2. Se accetta, M_2 accetta."
2. Output(M_1, M_2)."

L'unica differenza tra f e g è la macchina M_1. In f questa macchina rigetta sempre, mentre in g accetta sempre. In f e g, M accetta w se M_2 accetta sempre. In g, M accetta w se M_1 e M_2 sono equivalenti. Questo è perché g è una riduzione da A_{TM} a EQ_{TM}.

Capitolo 6:

6.2:

Esempi di affermazioni matematiche: Esempi di affermazioni matematiche sono i seguenti:

1. $\forall q\,\exists p\,\forall x,y\,\left[p>q \wedge (x,y>1 \rightarrow xy\neq p)\right]$,
2. $\forall a,b,c,n\,\left[(a,b,c>0 \wedge n>2) \rightarrow a^n+b^n\neq c^n\right]$, and
3. $\forall q\,\exists p\,\forall x,y\,\left[p>q \wedge (x,y>1 \rightarrow (xy\neq p \wedge xy\neq p+2))\right]$

La prima formula asserisce che esistono infiniti numeri primi. Questa affermazione è nota essere vera dal tempo di Euclide (più di 2300 anni fa).

La seconda formula corrisponde all'ultimo teorema di Fermat che è stato dimostrato pochi anni fa ad opera di Andrew Wiles.

La terza formula asserisce che esistono infinite coppie di numeri primi che differiscono di solo due unità. Questa è solo una congettura ed è tuttora non dimostrata ne confutata.

Al fine di automatizzare il processo di determinazione di verità delle affermazioni matematiche è utile considerare queste affermazioni come stringhe e definire un linguaggio formato da tutte le affermazioni vere.

Il problema della determinazione di verità delle affermazioni si riduce a alla decidibilità di questo linguaggio

Definzione di linguaggio: Per la definizione del linguaggio si consideri il seguente alfabeto:

$$\{\wedge, \vee, \neg, (,), \forall, x, \exists, R_1, \ldots, R_k\}$$

Λ, V, e ¬, corrispondo alle operazioni booleane and,or e not*; "(" e* ")" sono le parentesi*;* ∀ *ed* ∃ sono i quantificatori universale ed esistenziale*;* x denota variabili*;* R_1 *, ... ,* R_k sono *relazioni.*

Una formula è una stringa sull'alfabeto dato, una stringa della forma $R_i(x_i, \ldots, x_j)$ *è una formula atomica.* Il valore j è l'arità ("f(x) = x^2 + 3x" ha come argomento x, che è l'unico operatore, infatti ha arità 1) della relazione R_i.

Una formula (ben formata), (in breve fbf)

1. è una formula atomica,
2. è una combinazione booleana di altre formule più piccole
3. è una quantificazione su altre formule f del tipo ∃ x_i [f] oppure ∀ x_i [f]

Nota: I quantificatori legano le variabili all'interno del loro "scope" (parentesi quadre). Se una variabile non è legata in una formula allora la variabile è detta libera. Le formule senza variabili libere sono dette sentenze o *statements*.

Ordini delle logiche: (Primo ordine) Formule ben Formate (fbf):

- $R_1(x1) \wedge R_2(x1, x2, x3)$
- $\forall x_1[R_1(x_1) \wedge R(x_1, x_2, x_3)]$
- $\forall x_1 \exists x_2 \exists x_3[R_1(x_1) \wedge R_2(x_1, x_2, x_3)]$

Osservazione: solo l'ultima fbf è una sentenza che si legge "per ogni x_1 esistono x_2 e x_3 tali che $R_1(x_1)$ e $R_2(x_1, x_2, x_3)$ sono veri".

(Secondo ordine) Costruendo tale sistema possiamo ragionare su sentenze del tipo

- $\forall$ q, $\exists$ x, y[p>q $\wedge$ (x,y>1 $\rightarrow$ xy $\neq$ p)]. (infiniti numeri primi)
- $\forall$ a, b, c, n[(a,b,c>0 $\wedge$ n>2) $\rightarrow$ $a^n + b^n \neq cn$]. (ultimo teorema di Fermat)
- $\forall$ q $\exists$ p $\forall$ x, y[p>q $\wedge$ (x,y>1 $\rightarrow$ (xy $\neq$ p $\wedge$ xy $\neq$ p+2))]. (congettura sui numeri primi gemelli)

(Terzo ordine) Per avere senso, una logica ha bisogno che le venga assegnato un significato. Per fare questo, abbiamo bisogno di assegnare la sintassi a uno specifico costrutto matematico, chiamato modello.

Un modello è composto da un **universo** e un insieme di **relazioni**, una per ogni simbolo di relazione nella logica.

Esempio:

- sia Σ = {$\wedge$, V, e ¬, (,), $\forall$, $\exists$, x,$R_1(\cdot\ ,\ \cdot)$}.
- Un modello per questa logica è M_1 = (N,≤), con x $\rightarrow$ N and R1 $\rightarrow$≤.
- N è l'universo e la relazione ≤ ϵ N × N è l'*interpretazione* per il simbolo di relazione binaria R_1.

(Quarto ordine) Data una logica e un modello, possiamo verificare se una particolare sentenza è vera nel modello.

Esempio 1:

- Data la logica Σ = {$\wedge$, V, e ¬, (,), $\forall$, $\exists$, x,$R_1(\cdot, \cdot)$}, col modello M_1 = (N,≤).
- Possiamo chiederci se la sentenza $\forall$ x $\exists$ y[R_1(x, y) V R_1(y, x)] è vera.
- Chiaramente la sentenza è vera, visto che per ogni assegnamento x $\rightarrow$ a e y $\rightarrow$ b per a, b ϵ N, abbiamo che a ≤ b or b ≤ a.

Esempio 2:

- Data la logica Σ = {$\wedge$, V, e ¬, (,), $\forall$, $\exists$, x,$R_1(\cdot, \cdot)$}, col modello M_2 = (N,<)
- Possiamo dire che la sentenza $\forall$ x $\forall$ y[R_1(x, y) V R_1(y, x)] non è vera. Questo perché per l'assegnamento x $\rightarrow$ a e y $\rightarrow$ a con a ϵ N abbiamo a < a or a < a, che è chiaramente falso.

Esempio 3:

- Data la logica Σ = {$\wedge$, V, e ¬, (,), $\forall$, $\exists$, x,$R_1(\cdot\ ,\ \cdot\ ,\ \cdot)$}, col modello M_3 = (R,+) e R_1 relazione ternaria
- possiamo dire che la sentenza $\forall$ y $\exists$ x[R_1(x, x, y)] è vera. Infatti per ogni assegnamento x $\rightarrow$ a e y $\rightarrow$ b con a, b ϵ R abbiamo che +(a, a, b), o nella classica notazione matematica b = a + a, è vera. Falso se il dominio è N

Sia M un modello. Diremo che la collezione di tutte le sentenze vere sotto quel modello è la *teoria* del modello e scriveremo Th(M).

Teorema 6.12: la teoria Th(N,+) è decidibile.

Cosa significa che una teoria è decidibile? Significa che noi possiamo decidere se una particolare sentenza appartiene alla teoria o no. Quindi possiamo trattare la teoria Th(N,+) come un linguaggio e possiamo costruire un decisore per questo linguaggio.

Idea della dimostrazione: Consideriamo la sentenza $\forall$ x $\exists$ y[y = x + x]. Questa sentenza è vera ed è anche un elemento della teoria Th(N,+).

Consideriamo ora $\exists$ x $\forall$ y[y = x + x]. Questa sentenza è falsa è quindi non è un membro della teoria.

E' possibile mostrare la decidibilità della teoria Th(N,+) costruendo un automa finito che decide il linguaggio.

Sia Σ = { 00, 01, 10, 11} dove la coppia di nuperi ij rappresenta un elemento di una matrice trasposta 2 x 1 di binari.

Si noti che ogni parola costruita su Σ rappresenta due numeri binari. Per esempio 00 11 10 10 rappresenta i numeri 0111 e 0100

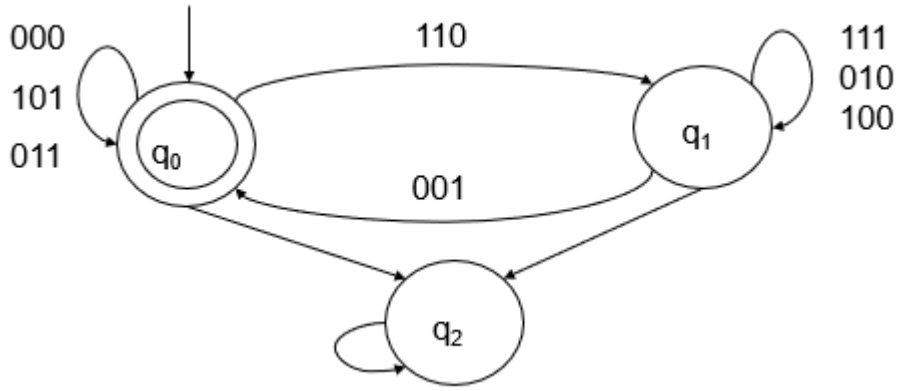
Sia A = { w ϵ Σ^* | la prima riga sia uguale alla seconda}.

Esempio: 00 11 00 11 11 11 ϵ A and ¬(11 00 10 00 11 01 00 ϵ A)

Sia Σ = {000, 001, 010, 011, 100, 101, 110, 111}.

Si consideri il seguente linguaggio: B = { w ϵ Σ* | la somma delle prime due righe è uguale alla terza}

Per esempio, 001 110 011 ϵ B, mentre ¬ (001 110 010 001 ϵ B)



Dimostrazione: Sia ϕ=Q_1x_1 … Q_nx_n(ψ) una sentenza di Th(N,+), dove ciascun Q_i è un quantificatore esistenziale (∃) o universale (∀) ψ non ha quantificatori.

Sia inoltre $\phi_i = Q_ix_i$ ... Q_nx_n (ψ). In particolare siano ϕ_0 = ϕ e ϕ_n = ψ. Sia Σ_i l'alfabeto di tutte le parole binarie di lunghezza i. Si costruisca A_n su Σ_n che accetti tutte le parole che rendano vera ϕ_n = ψ. Si noti che ψ non ha quantificatori e solo operazioni di somma. Si costruisca A_i a partire da A_{i+1}, nel seguente modo:

- Si assuma che Q_i sia esistenziale. Allora costruire A_i in modo da fare una scelta non deterministica sull'i+1-esimo elemento di Σ.
- Se Q_i è universale, allora a fronte della equivalenza $\forall x_{i+1}\ \phi_{i+i} = \neg\exists x_{i+1}\ \neg\phi_{i+i}$ si costruisce prima il complemento di A_{i+1} poi si applica il procedimento precedente per Q_i esistenziale e poi si complementa l'automa ottenuto

L'automa A_0 accetta qualche input se e solo se ϕ_0 = ϕ è vera. Dunque, l'ultimo passo dell'algoritmo è testare il vuoto A_0.

Una teoria non decidibile: Il seguente teorema ha delle conseguenze filosofiche molto profonde. Esso mostra che la matematica non può essere "meccanizzata".

Mostra inoltre che alcuni problemi nella teoria dei numeri non sono algoritmici, cosa che provocò una grande sorpresa nei matematici all'inizio del 1900. Allora infatti si credeva che tutti i problemi matematici potessero essere risolti algoritmicamente e che bisognasse solo trovare l'algoritmo per risolvere un dato problema.

Teorema 6.13: la teoria Th(N,+,×) è indecidibile.

Questo significa che non esiste un algoritmo che si ferma su tutte le sentenze ϕ sull'alfabeto appropriato. Quello che più sorprende è la semplicità della struttura di questa logica indecidibile.

Questo vuol dire che ci sono delle fondamentali limitazioni algoritmiche nella matematica. La dimostrazione si ottiene tramite una riduzione del problema A_{TM} alla teoria Th(N,+,×).

Teorema di incompletezza 1: la collezione di sentenze provabili in Th(N,+,×) è Turing-riconoscibile.

Dimostrazione:

- il seguente algoritmo P accetta il suo input ϕ se e solo se ϕ è dimostrabile.
- L'algoritmo P prova tutte le possibili stringhe come candidate per una prova π di ϕ usando un "proof checker"(verificatore della prova).
- Se viene trovata una stringa che è una prova, allora l'algoritmo accetta.

Teorema (di incompletezza di Kurt Gödels): alcune sentenze vere in Th(N,+,×) non sono dimostrabili.

Con qualche semplificazione, questo teorema afferma che: "In ogni formalizzazione coerente della matematica che sia sufficientemente potente da poter assiomatizzare la teoria elementare dei numeri naturali — vale a dire, sufficientemente potente da definire la struttura dei numeri naturali dotati delle operazioni di somma e prodotto — è possibile costruire una proposizione sintatticamente corretta che non può essere né dimostrata né confutata all'interno dello stesso sistema."