LINGUAGGIO -> Un linguaggio è semplicemente un insieme di stringhe.

ALFABETO -> Un alfabeto è un insieme non vuoto e finito di simboli, quindi è una sequenza finita di simboli.

OSSERVAZIONE -> I linguaggi sono utili per descrivere problemi di calcolo.

AUTOMI A STATI FINITI DETERMINISTICI

Un automa a stati finiti deterministici è una quintupla:

M = (Σ, Q, δ, q0, F), dove:

- Σ è l'alfabeto di input dell'automa M;

- Q è l'insieme degli stati dell'automa M;

- δ è la funzione di transizione dell'automa M, ovvero è una funzione che mi dice a partire da uno stato e da un simbolo dell'alfabeto il nuovo stato che viene raggiunto una volta che si legge quel simbolo dell'alfabeto;

- q0 è lo stato iniziale dell'automa M;

- F è l'insieme degli stati finali o di accettazione dell'automa M.

OSSERVAZIONE -> Se una stringa w non giunge in uno stato di accettazione, allora w è rigettata dall'automa M, altrimenti è riconosciuta.

FUNZIONE ESTESA DI TRANSIZIONE -> $\delta^* : Q \times \Sigma^* \to Q$

$\delta^*(q, w) = q'$

Descrive lo stato risultante q' dopo aver esaminato la stringa w a partire dallo stato iniziale q.

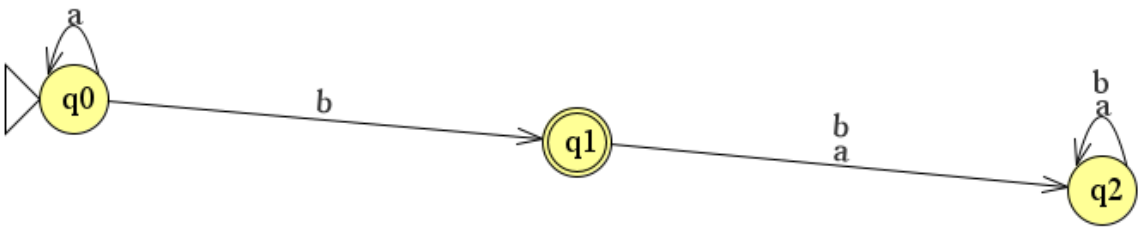
ESEMPIO -> $\delta^*(q0, abb) = q3$

$\delta^*(q0, abb) = \delta^*(\delta(q0,a), bb) = \delta^*(q1,bb) = \delta^*(\delta(q1,b), b) = \delta^*(q2, b) =$

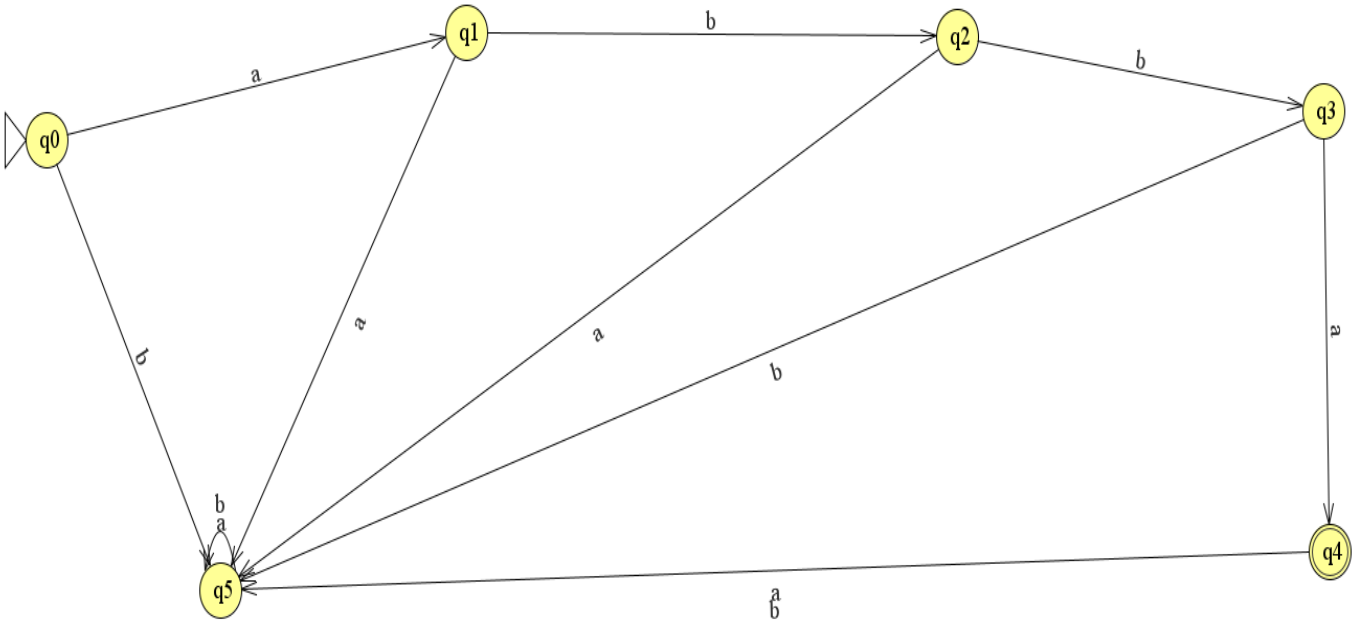
$= \delta^*(\delta(q2, b), \Lambda) = \delta^*(q3, \Lambda) = q3$

ESEMPI AUTOMI DETERMINISTICI

L = {W € x* | w = a^(n)b, n >= 0}



L = {w € X* | w contiene la parola abba}



LINGUAGGIO ACCETTATO DA UN DFA -> Un linguaggio accettato da un DFA è denotato con L(M) e contiene tutte e sole le stringhe che sono accettate dal DFA M.

Per un DFA M = (Σ, Q, δ, q0, F) il linguaggio accettato da M è:

L(M) = {w € Σ* | δ*(q0,w) € F}

LINGUAGGIO REGOLARE -> Un linguaggio L è regolare se esiste un DFA M che lo accetta, ovvero L(M) = L.

OSSERVAZIONE -> I linguaggi accettati da tutti i DFA formano la famiglia dei linguaggi regolari.

AUTOMI A STATI FINITI NON DETERMINISTICI

Un automa a stati finiti deterministici NFA è una quintupla:

M = (Σ, Q, δ, q0, F), dove:

- Σ è l'alfabeto di input;

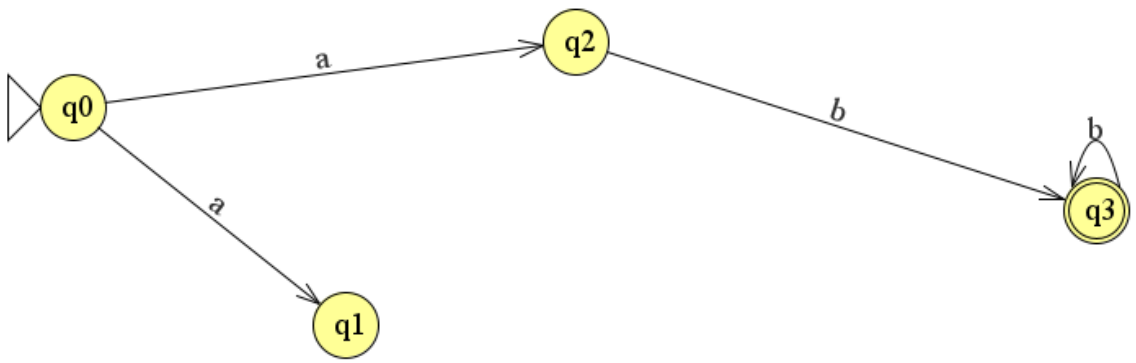
-Q è l'insieme degli stati;

- δ è la funzione di transizione, δ : Q x Σ -> 2^(Q), dove δ(q, x) = {q1, q2, q3, …, qk};

-q0 è lo stato iniziale;

-F è l'insieme degli stati finali ed è un sottoinsieme di Q.

ESEMPIO NFA ->



LINGUAGGIO ACCETTATO DA UN NFA -> Il linguaggio accettato da un NFA è denotato con L(M) e contiene tutte le stringhe riconosciute da M:

formalmente:

L(M) = {w € Σ* | δ*(q0, w) ∩ F ≠ Ø}.

OSSERVAZIONE -> Quindi quando δ* raggiunge almeno uno stato che è finale, allora il linguaggio è accettato da M.

TEOREMA -> NFA e DFA riconoscono gli stessi linguaggi.

Dato un DFA che riconosce un linguaggio L, allora esiste un NFA che riconosce lo stesso linguaggio L, e , viceversa, dato un NFA che riconosce il linguaggio L', allora esiste un DFA che riconosce il medesimo linguaggio L'.

DIMOSTRAZIONE -> Devo, quindi, dimostrare che gli NFA contengono i DFA e che i DFA contengono gli NFA.

Per quanto riguarda la prima implicazione, ovvero che gli NFA contengono i DFA, questa è sempre verificata, in quanto, ogni DFA può essere visto come un NFA e quindi accetta lo stesso linguaggio.

Per quanto riguarda la seconda implicazione, ovvero che i DFA contengono gli NFA, devo attuare un procedimento di conversione che a partire da un NFA giungo a un DFA.

Sia dato un automa NFA M = (Σ, Q, δ, q0, F), allora l'automa DFA M' = (Σ, Q', δ', q0', F') equivalente a M si costruisce nel seguente modo:

- Σ è l' alfabeto di input;

- Q' = 2^(Q);

-q0' = {q0};

-F' = {p contenuto in Q | p ∩ F ≠ Ø)

- δ' : Q' x Σ -> Q', dove δ'(q', x) = δ'({q1, q2, …, qi}, x) =$\bigcup_{i=1}^{j} \delta'(qj, x) = \bigcup_{q \in q,} \delta(q, x)$

Pertanto, una volta attuato questo algoritmo di conversione ho dimostrato che il DFA risultante accetta lo stesso linguaggio accettato dall'NFA iniziale.

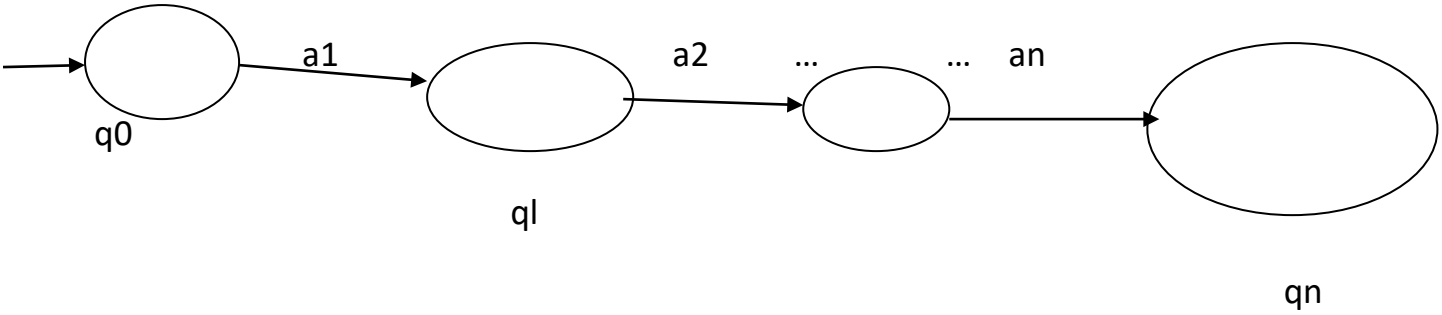
Quindi ho dimostrato anche la seconda implicazione, pertanto ne consegue che i DFA e gli NFA riconoscono gli stessi linguaggi.

TEOREMA -> Se un NDA M viene trasformato in un DFA M', allora i due automi sono equivalenti, ovvero L(M) = L(M')

DIMOSTRAZIONE -> Dimostro che L(M) è contenuto in L(M').

Considero una stringa generica w = a1, a2, … , an.

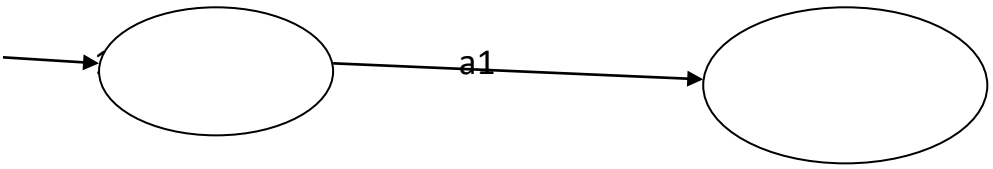
Quindi M è fatta nel modo seguente:



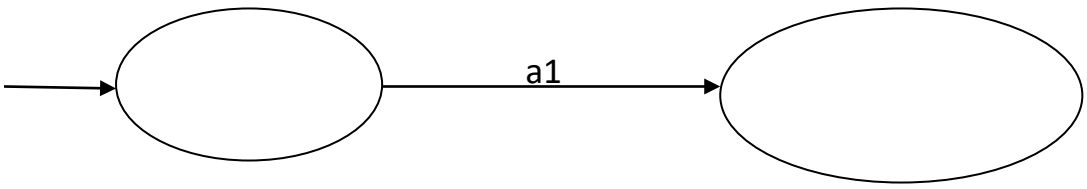
Ora applico il principio d'induzione sulla lunghezza di w:

PASSO BASE -> |w| = 1, quindi w = a1

Quindi in questo caso l'automa M sarà semplicemente il seguente:

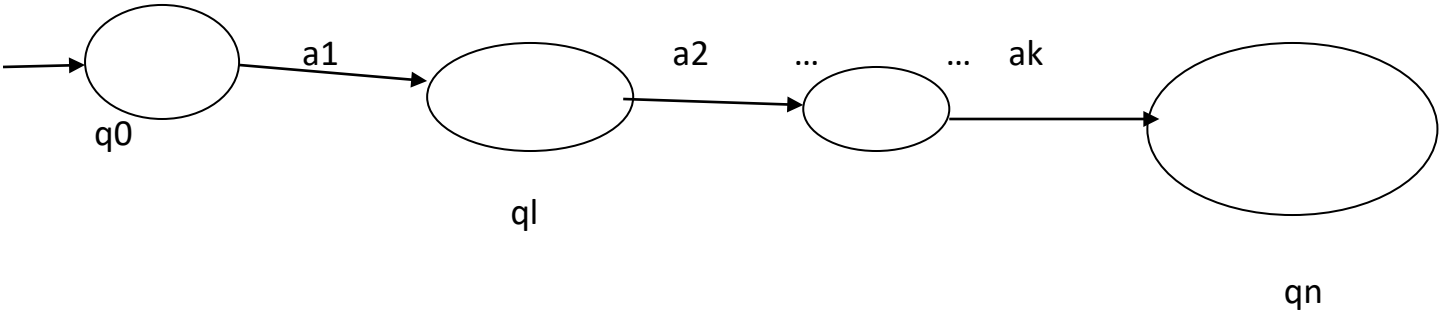


Mentre l'automa M' che riconosce la stringa w sarà il seguente:

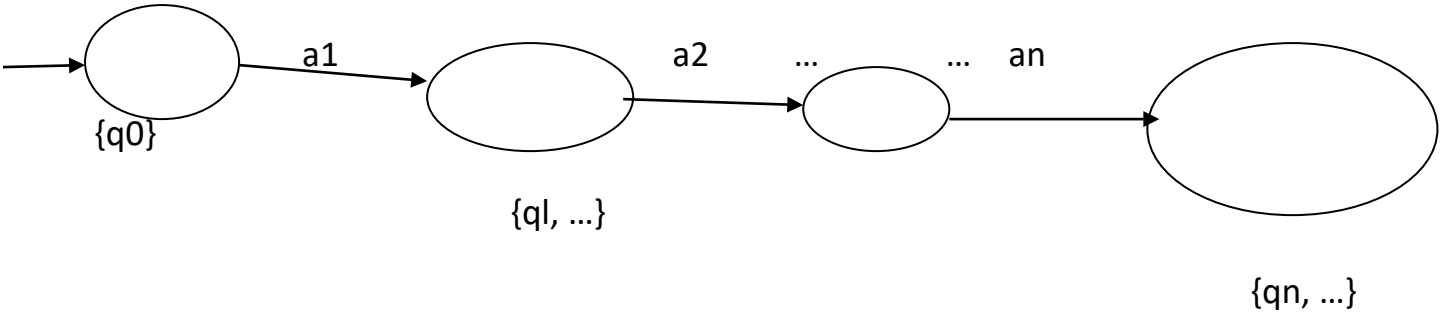


IPOTESI INDUTTIVA -> Considero la stringa w' = a1, a2, …, ak, 1 <= |w| <= k.

Quindi l' NDA M sarà fatto in questo modo:

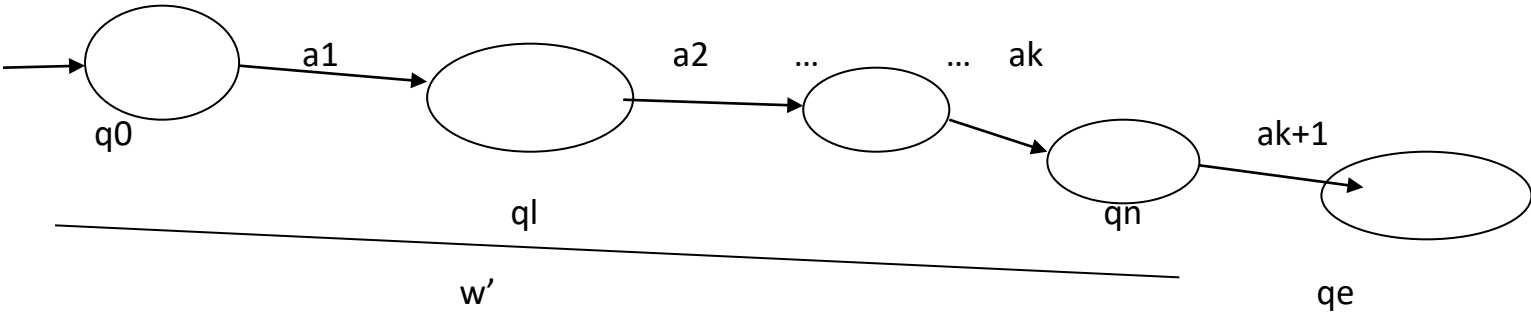


Mentre l' automa DFA M' sarà il seguente:

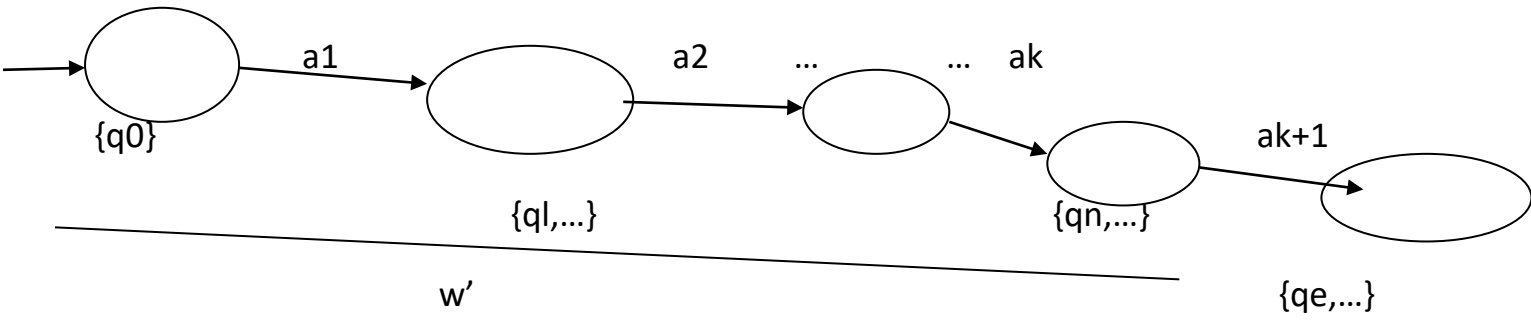


STEP INDUTTIVO -> Ora considero la stringa v = a1, a2, …, ak, ak+1, dove v = w'ak+1

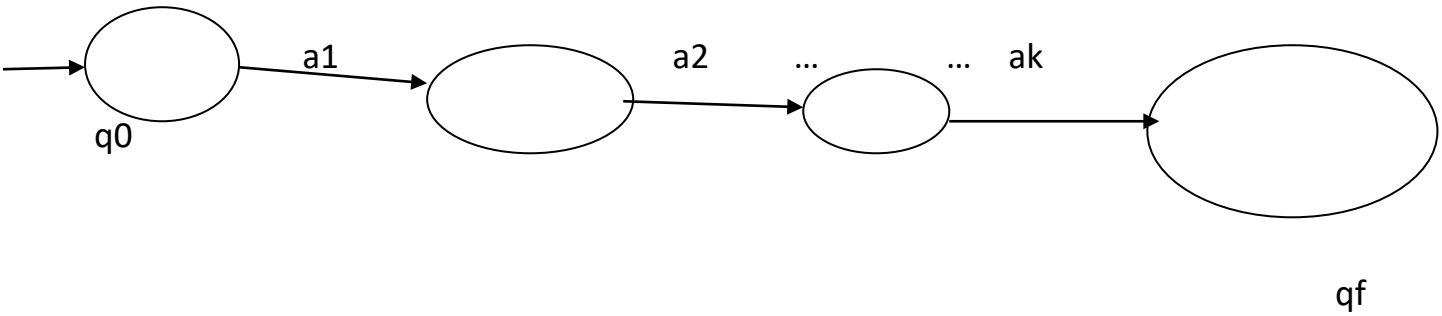
Pertanto, l'automa M che riconosce la stringa v sarà il seguente:



Mentre l'automa M'sarà il seguente:

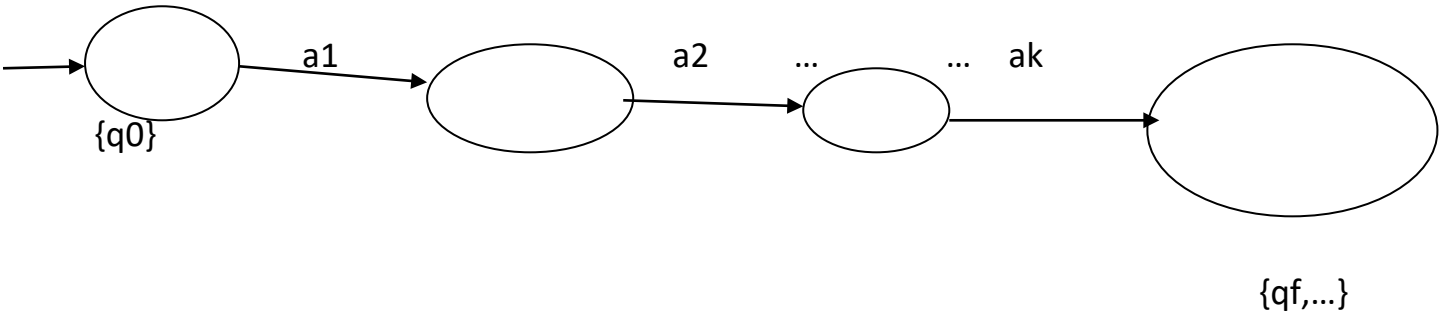


Pertanto, se w € L(M), w = a1, a2, …, ak, l'automa M che riconosce la stringa w sarà il seguente:



dove qf è stato finale.

Mentre l'automa M' che riconosce la stringa w sarà il seguente:



Quindi w € L(M').

Pertanto risulta dimostrato che dopo aver trasformato un automa M NFA in un automa M' DFA, i due automi sono equivalenti, ovvero L(M) = L(M').

FINE DIMOSTRAZIONE

LINGUAGGIO REGOLARE

Un linguaggio L è regolare se L è finito oppure L deriva da una delle seguenti operazioni:

- L = L1 U L2, dove L1 e L2 sono regolari;

- L = L1 . L2, se L1 e L2 sono regolari;

- L = L1*, se L1 è regolare.

Allo stesso modo si può affermare che un linguaggio è regolare se è riconosciuto da un DFA o da un NFA.

ESPRESSIONE REGOLARE -> Un espressione è regolare se e solo se vale una delle seguenti condizioni:

- L(r1 + r2) = L(r1) U L(r2);
- L(r1 . r2) = L(r1) . L(r2);
- L(r1)* = (L(r1))*
- L((r1)) = L(r1).

TEOREMA -> I linguaggi generati da espressioni regolari sono uguali ai linguaggi regolari.

DIMOSTRAZIONE -> Devo dimostrare la doppia inclusione, ovvero:

1) I linguaggi generati da espressioni regolari sono contenuti nei linguaggi regolari;

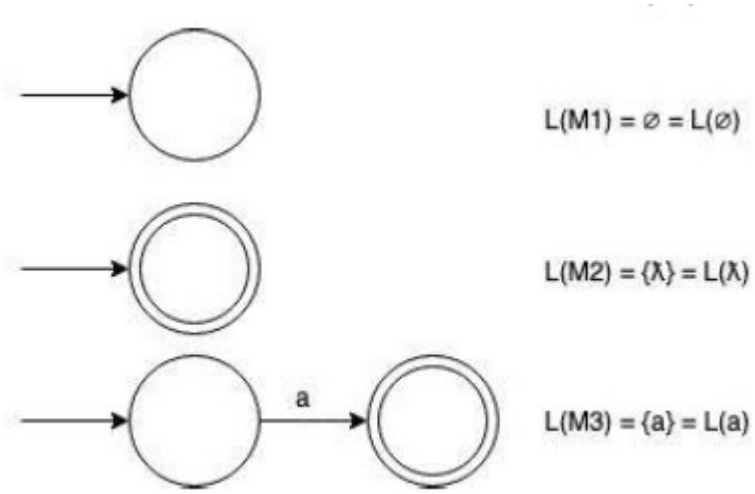
2) I linguaggi regolari sono contenuti nei linguaggi generati da espressioni regolari.

1) Quindi devo provare che per ogni espressione regolare r, L(r) è un linguaggio regolare.

Applico il principio d'induzione sulla lunghezza di r.

PASSO BASE -> Considero le espressioni regolari basilari: Ø, ʌ, a.

Per ognuna di queste espressioni vi è il corrispondente automa:



IPOTESI INDUTTIVA -> Suppongo che per due espressioni regolari r1 ed r2, L(r1) e L(r2) sono linguaggi regolari.

STEP INDUTTIVO -> Devo provare, quindi, che:

- L(r1 + r2);
- L(r1 . r2);
- L(r1)*.

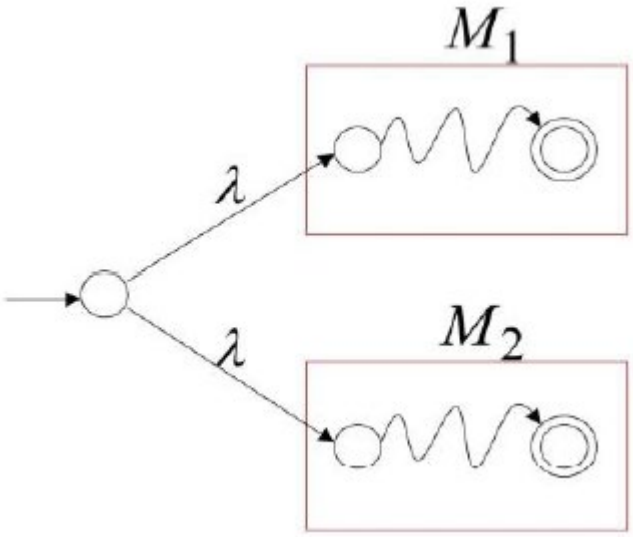
Pertanto, lo provo utilizzando la definizione di espressione regolare:

- L(r1 + r2) = L(r1) U L(r2);
- L(r1 . r2) = L(r1) . L(r2);
- L(r1)* = (L(r1))*.

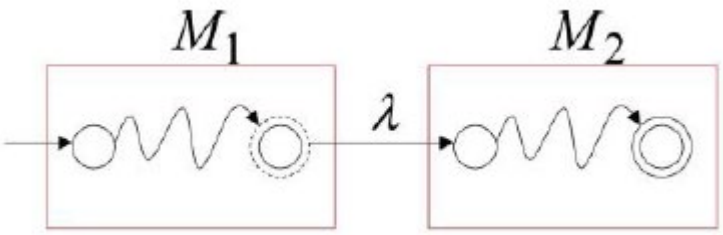
Inoltre, io so che i linguaggi regolari sono chiusi rispetto alle operazione di unione, concatenazione, star; sfruttando questo fatto posso costruire un NFA M tale che L(M) = L(r).

Pertanto assumo che L(M1) = L1 e L(M2) = L2.

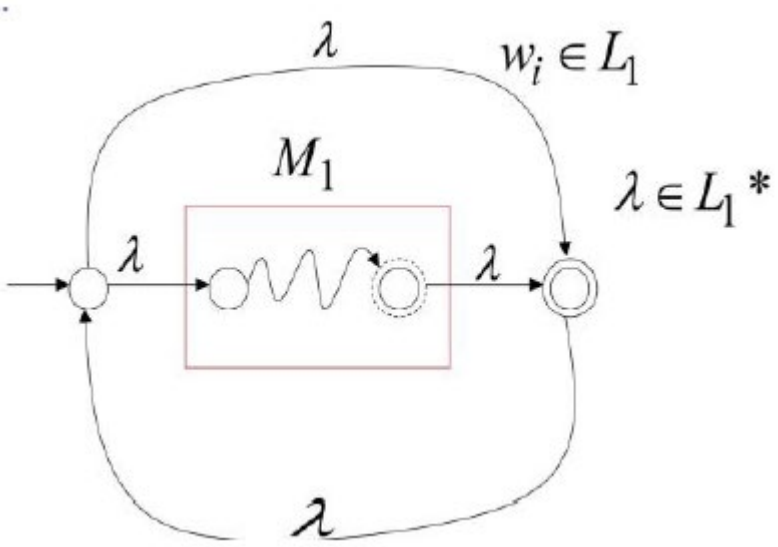
- UNIONE



- CONCATENAZIONE



- STAR



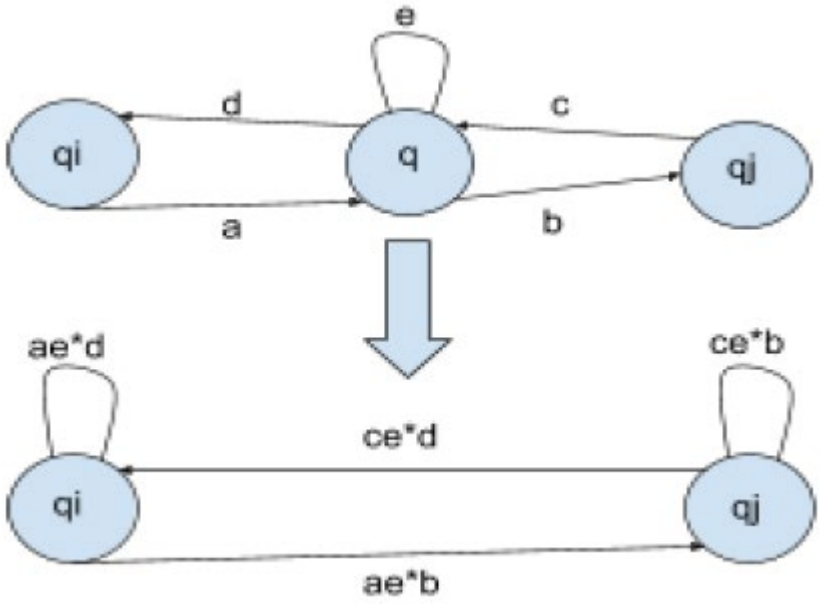
2) Ora devo provare che i linguaggi regolari sono contenuti nei linguaggi generati da espressioni regolari.

Per ogni linguaggio regolare L esiste un'espressione regolare L(r), tale che L(r) = L.

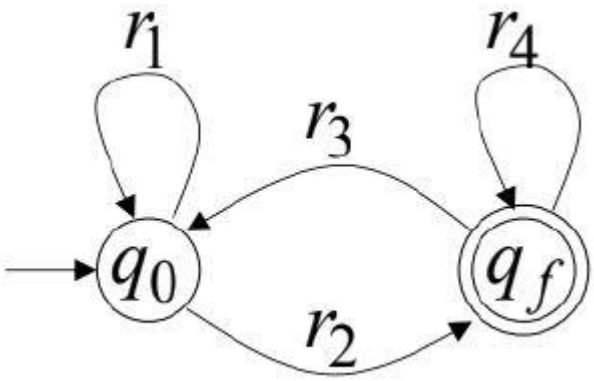
Pertanto, dato che L è regolare, posso costruire un NFA M che riconosce L e trasformarlo in un'espressione regolare.

L'algoritmo generale di trasformazione è il seguente:

a) Si rimuove uno stato:



b) Si ripete il passo b fino a quando non si è in presenza di soli due stati come qui di seguito riportato:



Pertanto, l'espressione regolare associata all'automa risulta:

r = r1*r2(r4 + r3r1*r2)*.

Dunque, risulta cosi dimostrato che i linguaggi generati da espressioni regolari sono uguali ai linguaggi regolari.

FINE DIMOSTRAZIONE

GRAMMATICA -> Una grammatica è una quadrupla G = (V, T, S, P), dove:

- V è l'insieme delle variabili o non terminali per la grammatica;

- T è l'insieme dei simboli dell'alfabeto o terminali per la grammatica;

- S è il simbolo di partenza per la grammatica;

- P è l'insieme delle regole di produzione della grammatica.

GRAMMATICA LINEARE DESTRA -> Una grammatica lineare destra è una grammatica in cui dopo l'unico non terminale nella parte destra della produzione non compare nulla.

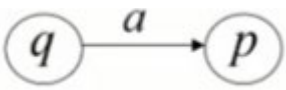
GRAMMATICA LINEARE SINISTRA -> Una grammatica lineare sinistra è una grammatica in cui, nella parte destra della produzione, i terminali compaiono a destra rispetto a l'unico non terminale.

CONVERSIONE DI UN AUTOMA M IN UNA GRAMMATICA LINEARE DESTRA

Sia dato un automa M = (Q, q_0, F, Σ, δ) e sia data una grammatica G = (X, V, S, P) lineare destra.

Allora per convertire l'automa M in una grammatica G lineare destra equivalente vi è la seguente procedura:

- per qualsiasi transizione del tipo:



si ottiene la seguente produzione:

q -> ap.

- Per qualsiasi stato finale q_f € F si aggiunge la seguente produzione:

q_f -> ʎ.

Pertanto, come è evidente, gli stati dell'automa M diventeranno variabili o non terminali e i simboli dell'alfabeto in input all'automa M saranno simboli terminali.

PUMPING LEMMA LINGUAGGI REGOLARI

In un linguaggio regolare ogni parola del linguaggio che è più lunga di un numero p, detto lunghezza del pumping, può essere replicata, ovvero essa conterrà una parte che se viene ripetuta infinite volte, allora la parola in questione apparterrà sempre al linguaggio regolare.

TEOREMA -> Se A è un linguaggio regolare, allora esiste un valore p tale che ogni stringa di A lunga almeno p può essere riscritta in tre sottostringhe.

Sia s una stringa del linguaggio A, allora s può essere riscritta nel seguente modo:

s = xyz e devono valere le tre seguenti condizioni:

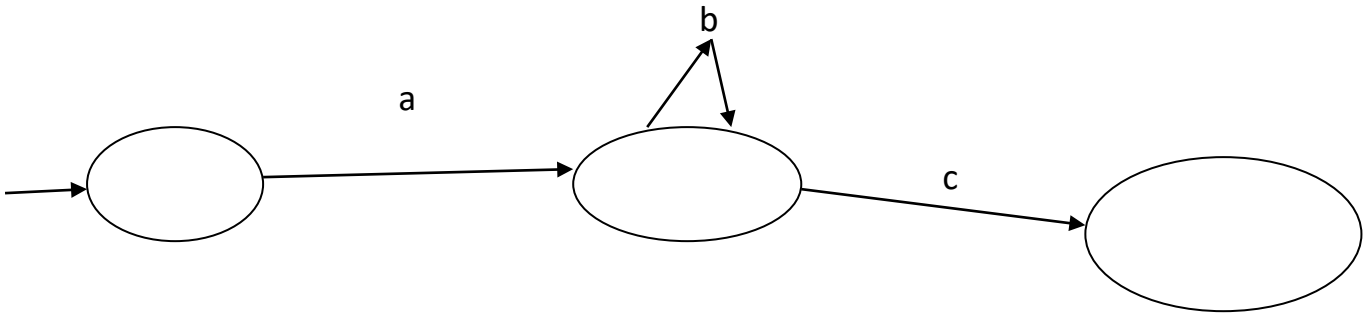
1) $x\ y^i\ z \in A$, per ogni i, $i >= 0$.
2) $|y| > 0$.
3) $|xy| <= p$.

IDEA -> Sia A un linguaggio ed M un DFA che lo accetta; assegno a p il numero degli stati dell'automa M.

Considero un stringa generica del linguaggio:

A = ab*c, s = abc.

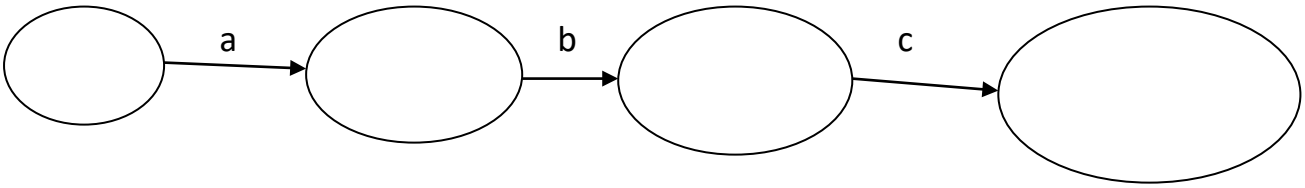
In questo caso, dunque, p = 3.



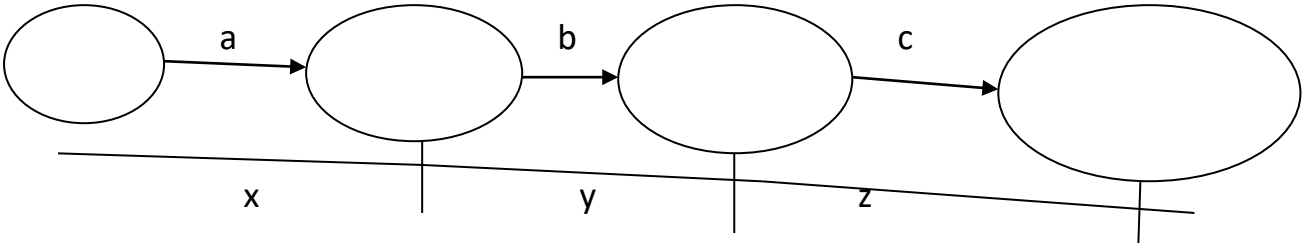
Prendo quindi una stringa di lunghezza n, n >= p.

Dunque l'automa M, per riconoscere la stringa s ha bisogno di n + 1 stati, perché servono n stati per riconoscere la parola e lo stato in più raffigura lo stato iniziale.

Evidentemente almeno uno tra i primi p + 1 stati deve essere lo stesso, pertanto risulterà:



Ora divido la stringa s in tre sottostringhe, s = xyz.



Pertanto risulta evidente che se replichiamo y, ovvero aumentiamo in modo indefinito le b, notiamo che la transizione sarà sempre su y fin quando non arriverà z, ovvero una c e, in quel caso si giungerà in uno stato finale.

PUMPING LEMMA LINGUAGGI CONTEXT-FREE

Sia L un linguaggio context free.

Allora dato che L è context free esiste un intero m tale che per ogni stringa z del linguaggio che è >= di m, z può essere riscritta come z = uvxyz, in cui due parti si possono ripetere in modo indefinito in parallelo non uscendo dal linguaggio L;

queste due parti sono v e y.

Quindi deve valere che:

-|vxy| <= m

- |vy| > 0

- uv^ixy^iz € L, per ogni i, i >= 0.

OSSERVAZIONE -> L'intero m è pari a:

$m = 2^{\text{numero delle variabili della grammatica}} - 1$, perché quando vi è un loop allora si va a ripetere una variabile chiamata in precedenza.

ESERCIZIO -> $L = \{vv : v \in \{a, b\}^*\}$.

Suppongo per assurdo che L è un linguaggio context free.

Poiché L è context free ed è infinito posso applicare il Pumping Lemma per i linguaggi context free, quindi esiste m, tale che posso ripetere due parti della strina in parallelo.

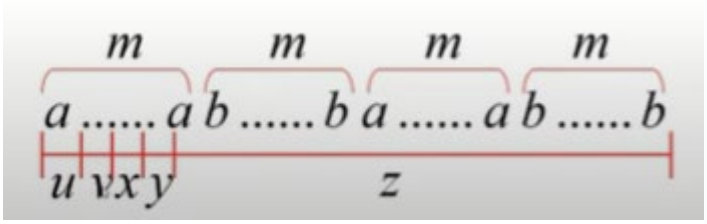
Prendo una stringa di L:

$z = a^m b^m a^m b^m$.

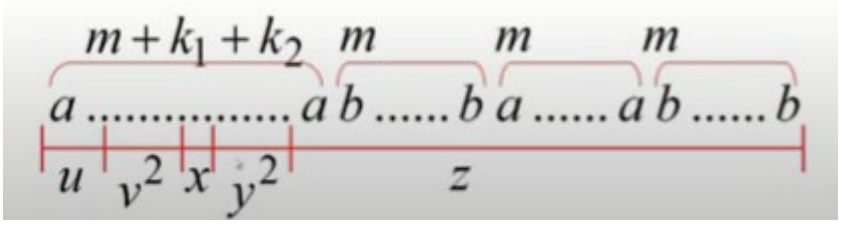
Allora posso scrivere $a^m b^m a^m b^m = uvxyz$.

Ora devo trovare vxy in modo tale che se ripeto la v e la y ottengo una parola del linguaggio.

CASO 1 -> vxy è nel primo a^m.

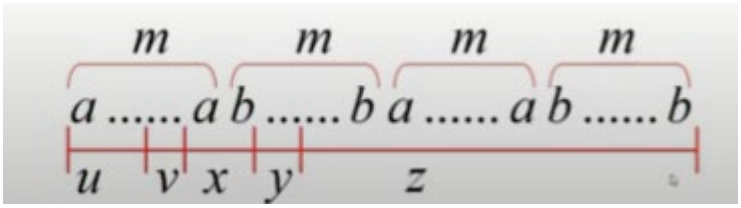


Però si nota che se vengono aumentate le v e le y ottengo una parola che non appartiene al linguaggio L:



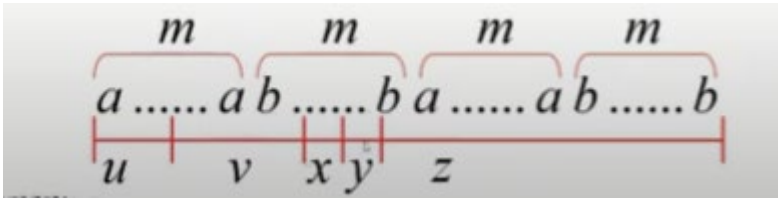
.

CASO 2 -> v è nel primo a^m, y nel primo b^m:

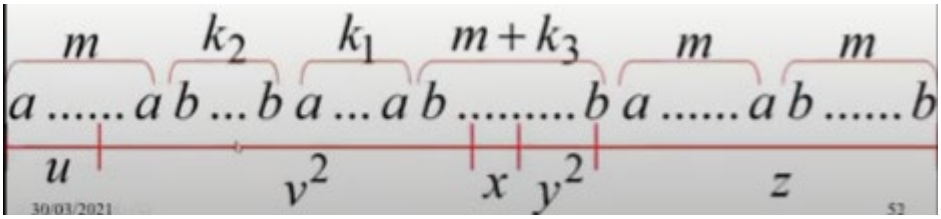


Anche in questo caso, come nel precedente, se aumento in parallelo la v e la y ottengo una parola che mi fa uscire dal linguaggio L.

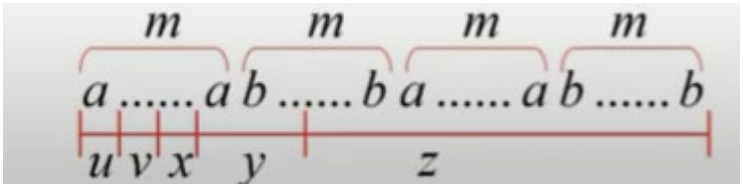
CASO 3 -> v si sovrappone sul primo $a^m b^m$, y è sul primo b^m:



Anche in questo caso se pompo esco da L:



CASO 4 -> v è nel primo a^m, y si sovrappone in $a^m b^m$:



Questo caso è simile al caso 3, quindi anche qui esco dal linguaggio.

Quindi risulta che uv^ixy^iz non € L. Assurdo.

L'assurdo deriva dall'aver supposto che il linguaggio L fosse context free.

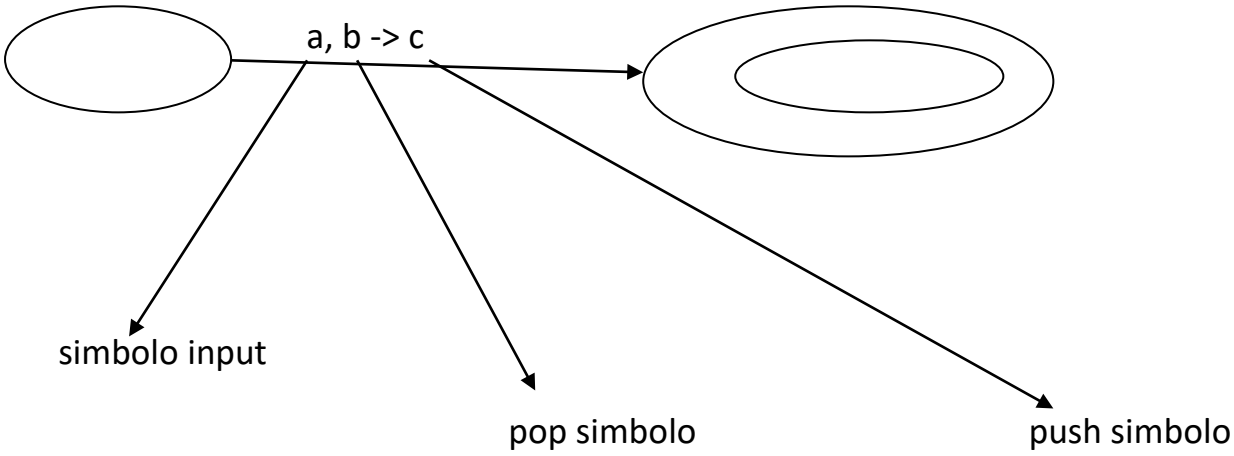
Pertanto risulta che il linguaggio L non è context free.

FINE ESERCIZIO

PUSHDOWN AUTOMATA

Il PDA è una settupla M = (Q, Σ, Γ, δ, q0, z, F), dove:

- Q è l'insieme degli stati;
- Σ è l'alfabeto;
- Γ è l'alfabeto dello stack;
- δ è la funzione di transizione;
- q0 è lo stato iniziale del PDA;
- z è il simbolo iniziale dello stack;
- F è l'insieme degli stati finali.



L' immagine si legge nel seguente modo:

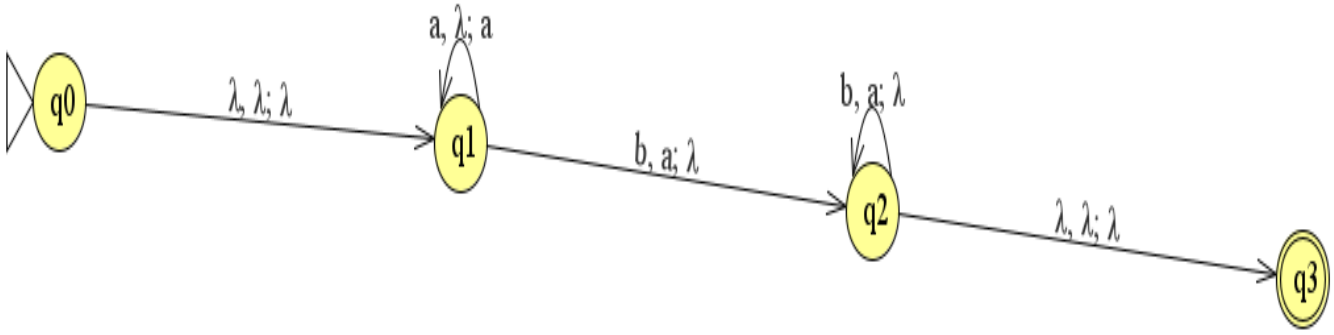
leggo il carattere a, tiro fuori dallo stack il carattere b e metto nello stack il simbolo c.

OSSERVAZIONE -> Se l'automa tenta di fare un pop dallo stack vuoto, allora l'input viene rigettato.

OSSERVAZIONE -> I PDA sono non deterministici perché permettono transizioni non deterministiche.

ESEMPIO PDA -> Devo costruire un PDA che riconosce il seguente linguaggio:

L = {w € Σ* | w = a^(n) b^(n)}



Il PDA seguente tramite q1 mette tante a nello stack, tramite q2 toglie le a solo se arrivano b, se alla fine lo stack è vuoto, allora il PDA ha riconosciuto un numero di a uguali a un numero di b con le b che seguono le a, ad esempio aaaaaabbbbbb.

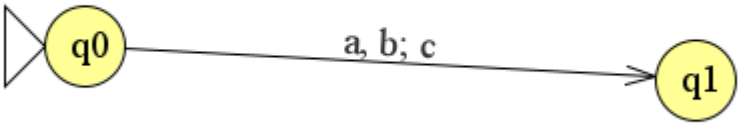
Mentre rigetta la stringa aabbb, perché non hanno lo stesso numero di a e di b.

LINGUAGGIO ACCETTATO DA UN PDA -> Un linguaggio è accettato da un PDA se, partendo dallo stato iniziale q0 e lo stack contiene solo il simbolo iniziale, allora, una volta che sono state eseguite tutte le computazioni si arriva nella situazione che nello stack vi è solo il simbolo iniziale dello stack e quindi la parola viene accettata.

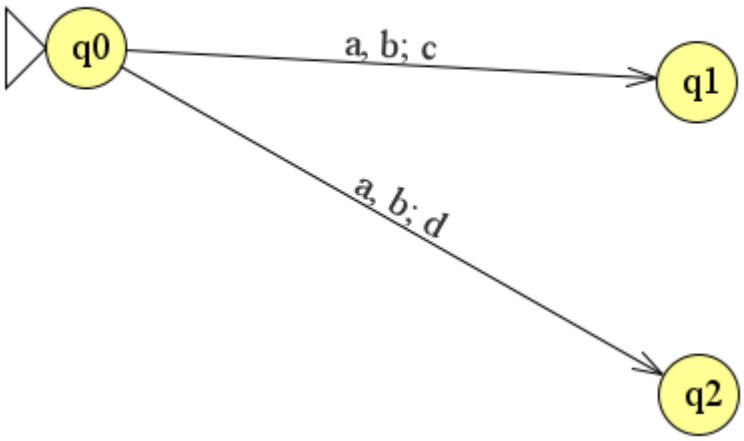
STRINGA ACCETTATA DA UN PDA -> Una stringa accettata da un PDA si ha quando l'input viene consumato interamente e l'ultimo stato è uno stato di accettazione.

FUNZIONE DI TRANSIZIONE -> La funzione di transizione per i PDA è fatta nel modo seguente:

δ(q0, a, b) = {(q1, c)}



δ(q0, a, b) = {(q1, c), (q2, c)}



FORMA NORMALE DI CHOMSKY -> Una grammatica context-free è in CNF se le regole di produzione sono del tipo:

A -> BC, A -> a, inoltre è permessa la produzione S -> ʌ.

CONVERSIONE IN CNF -> Si supponga di avere le seguenti regole di produzione:

S -> ABa, A -> aab, B -> Ac.

Introduco nuove variabili per i terminali:

Ta -> a, Tb -> b, Tc -> c, quindi risulta:

S -> ABTa, A -> TaTaTb, B -> ATc, noto che la terza produzione è già in CNF.

Introduco una nuova variabile V1 per spezzare la prima produzione:

S -> AV1BTa, S ->AV1, V1 -> BTa, ora sia la prima che la terza produzione sono in CNF.

Ora introduco una nuova variabile V2 per spezzare la seconda produzione:

A -> TaV2TaTb, A -> TaV2, V2 -> TaTb.

Ora ho tutte produzioni in CNF.

TEOREMA -> I linguaggi C.F. sono equivalenti ai PDA.

DIMOSTRAZIONE -> Devo dimostrare che i C.F. sono equivalenti ai PDA.

Per farlo devo dimostrare la doppia implicazione, ovvero:

1) I C.F. sono contenuti nei linguaggi accettati da PDA;

2) I C.F. contengono i linguaggi accettati da PDA.

1) Per dimostrare la seguente implicazione utilizzo questa procedura:

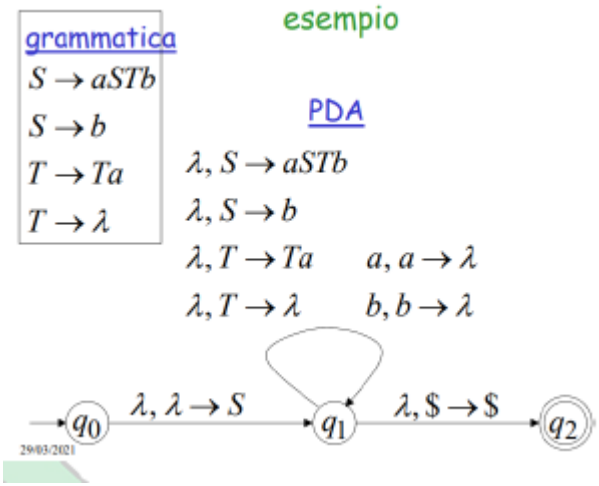
a) per ogni produzione del tipo A -> w, creo una transizione del tipo ʎ, A -> w, cosicchè, non appena nello stack ci sarà un non terminale, questo sarà sostituito con la stringa w prodotta.

b) Per ogni terminale, invece, genero la transizione a, a -> ʎ , cosi da eliminare ogni terminale presente nello stack.

Il PDA avrà tre stati; il primo è lo stato iniziale, il secondo è uno stato intermedio in cui cicleranno tutte queste transizioni e l'ultimo stato è quello finale o di accettazione.

La prima transizione del PDA sarà ʎ, ʎ -> S, cosi inserisco nello stack il simbolo iniziale della grammatica; l'ultima transizione sarà ʎ, $ -> $.

OSSERVAZIONE -> Il PDA è una macchina profondamente non deterministica, questo vuol dire che tutte le transizioni vengono applicate in parallelo e non una per volta.



Se alla fine lo stack sarà vuoto, allora il linguaggio è accettato dal PDA e L(G) = L(M).

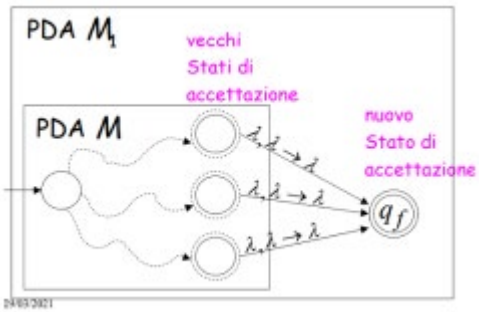
Ho dimostrato la prima implicazione.

2)Ora devo dimostrare la seconda implicazione, ovvero che i C.F. contengono i PDA.

Per fare questo devo dimostrare che per ogni PDA esiste una grammatica tale che L(PDA) = L(G).

Quindi mi serve un procedimento di conversione; attuo delle piccole modifiche al PDA:

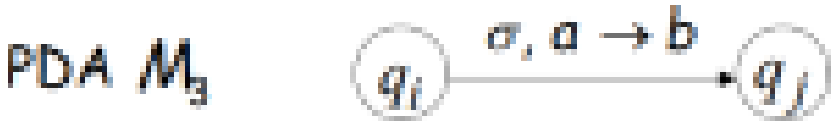
a) Il PDA deve avere un solo stato finale, il che vuol dire che collegherò tutti gli stati finali a questo nuovo e unico stato finale tramite delle transizioni nulle, ʎ, ʎ ->ʎ, cosi da avere un solo stato finale.



b)Prima di accettare si deve svuotare lo stack, quindi abbiamo bisogno di un nuovo simbolo iniziale dello stack, cosicchè quando svuotiamo lo stack ci fermiamo quando incontriamo questo simbolo .

c) Ogni transizione deve effettuare o una pop o una push.

Nel caso in cui una transizione effettui entrambe, allora creo uno stato intermedio cosi da effettuare prima una pop e poi la push.



Una volta che queste modifiche sono state effettuate, ottengo un nuovo PDA che è equivalente a quello precedente, ma che ora è adatto alla conversione in grammatica.

Costruisco la grammatica:

1)Per ogni coppia di stati q_i,q_j creo una nuova variabile $A_{qi,qj}$.

2)Per ogni stato q creo la transizione A_{qq} -> ʎ.

3)Per ogni tre stati p,r,q genero la produzione A_{pq} -> A_{pr} A_{rq} che nel PDA vuol dire che parto dallo stato p e arrivo nello stato q tramite lo stato r.

4)Per ogni coppia di transizioni in cui viene effettuata una pop o una push di uno stesso simbolo nello stack, genero una produzione in cui dalla variabile che comprende lo stato iniziale e quello finale produco il primo simbolo letto, la variabile degli stati intermedi e il secondo simbolo letto.

grammatica

$$A_{pq} \rightarrow aA_{rs}b$$

A questo punto ho convertito il PDA in grammatica C.F. equivalente.

Per verificare che la grammatica è effettivamente equivalente devo dimostrare che per ogni derivazione $A_{q0,qf}$ -> * w, esiste un calcolo analogo nel PDA, ovvero che :

$(q_0, w, @)$ -> $*(q_f, \Lambda, @)$.

Ora procedo con una dimostrazione per induzione.

PASSO BASE -> A_{pq} -> w con un solo step di derivazione.

Questa derivazione esiste ed è A_{pp} -> Λ.

Questo vuol dire che p e q devono essere uguali.

Questa derivazione esiste nel PDA, infatti si ha che (p, Λ, Λ) -> * (p, Λ, Λ).

IPOTESI INDUTTIVA -> Suppongo che per ogni computazione del tipo A_{pq} -> w con k step di derivazione, esista una computazione del tipo (p, w, Λ) -> (q, Λ, Λ).

STEP INDUTTIVO -> Prendo in considerazione la produzione A_{pq} -> w con k + 1 step.

Analizzando le produzioni vi sono due casi, il primo ho *A_{pq} -> A_{pr} A_{rq} e il secondo caso ho A_{pq} -> a A_{rs} b.*

1) Nel primo caso posso dividere la stringa w in due parti, x e y, in modo tale che

A_{pr} -> x e A_{rq} -> y.

La nostra produzione iniziale deve essere di k + 1 step; al primo step produce i due non terminali, cosi ora mancano k step.

Per ipotesi induttiva io so che se un non terminale produce una stringa in k step, allora esiste una computazione valida nel PDA.

Pertanto, dato che le variabili prodotte dal primo step hanno una computazione valida, allora si possono sommare queste computazione e ottenerne una equivalente a quella iniziale.

2) Nel secondo caso ottengo a A_{rs} b.

Dato che A_{pq} deve avere k + 1 step di produzione, dopo il primo passaggio A_{rs} deve avere k step di produzione.

Per ipotesi induttiva, come nel caso precedente, so che esiste una computazione equivalente alla produzione A_{rs} -> y, ottenendo cosi una stringa di terminali ayb.

So anche che per le regole di produzione della grammatica, a e b seguono la push e la pop dello stesso simbolo e A_{rs} termina con lo stack vuoto.

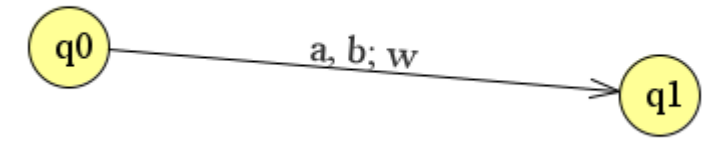
Quindi posso affermare che la produzione A_{pq} , partendo da uno stato iniziale p, mi porta a uno stato finale q con lo stack vuoto.

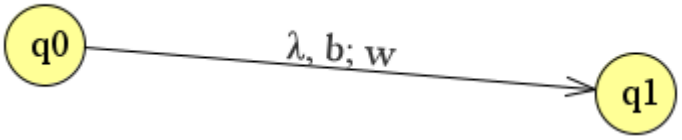
Risulta cosi dimostrato che i linguaggi C.F. sono equivalenti ai linguaggi accettati da PDA.

FINE DIMOSTRAZIONE

DPDA -> Il DPDA è un PDA che ha transizioni deterministiche.

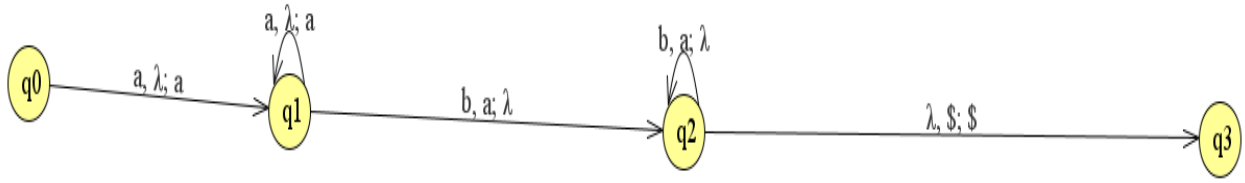
Le uniche transizioni permesse sono le seguenti:





Se il PDA non presenta queste due tipologie di transizioni, allora non è DPDA, ma solo PDA.

ESEMPIO DPDA -> Il DPDA per il linguaggio L = {w € Σ* | w = a^(n) b^(n), n >= 0} è il seguente:



TEOREMA -> I DPDA sono contenuti nei PDA e non il contrario.

OSSERVAZIONE -> Questo vuol dire che deve esistere un linguaggio che è riconosciuto da un PDA, ma non da un DPDA.

DIMOSTRAZIONE -> Dimostrerò che un linguaggio che è accettato da un PDA ma non da un DPDA.

Il linguaggio è:

L = {a^(n) b^(n)} U {a^(n) b^(2n)}, n >= 0.

Devo dimostrare che L è C.F. ma non è un deterministico C.F., ovvero che non posso costruire un DPDA che lo accetta.

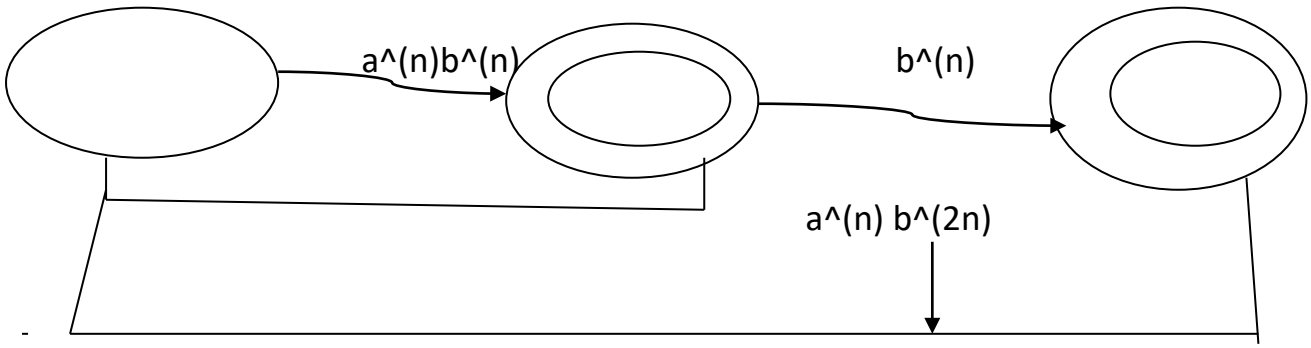
Il linguaggio è C.F., infatti:

S -> S1 | S2, S1 -> ʌ | aS1b, S2 -> ʌ | aS2bb.

Ora devo provare che non è deterministico C.F., ovvero che non esiste un DPDA che lo accetta.

Pertanto assumo per assurdo che il linguaggio sia deterministico C.F.

Quindi costruisco il DPDA che lo accetta:



Considero il linguaggio L’ = {a^(n) b^(n) c^(n)}.

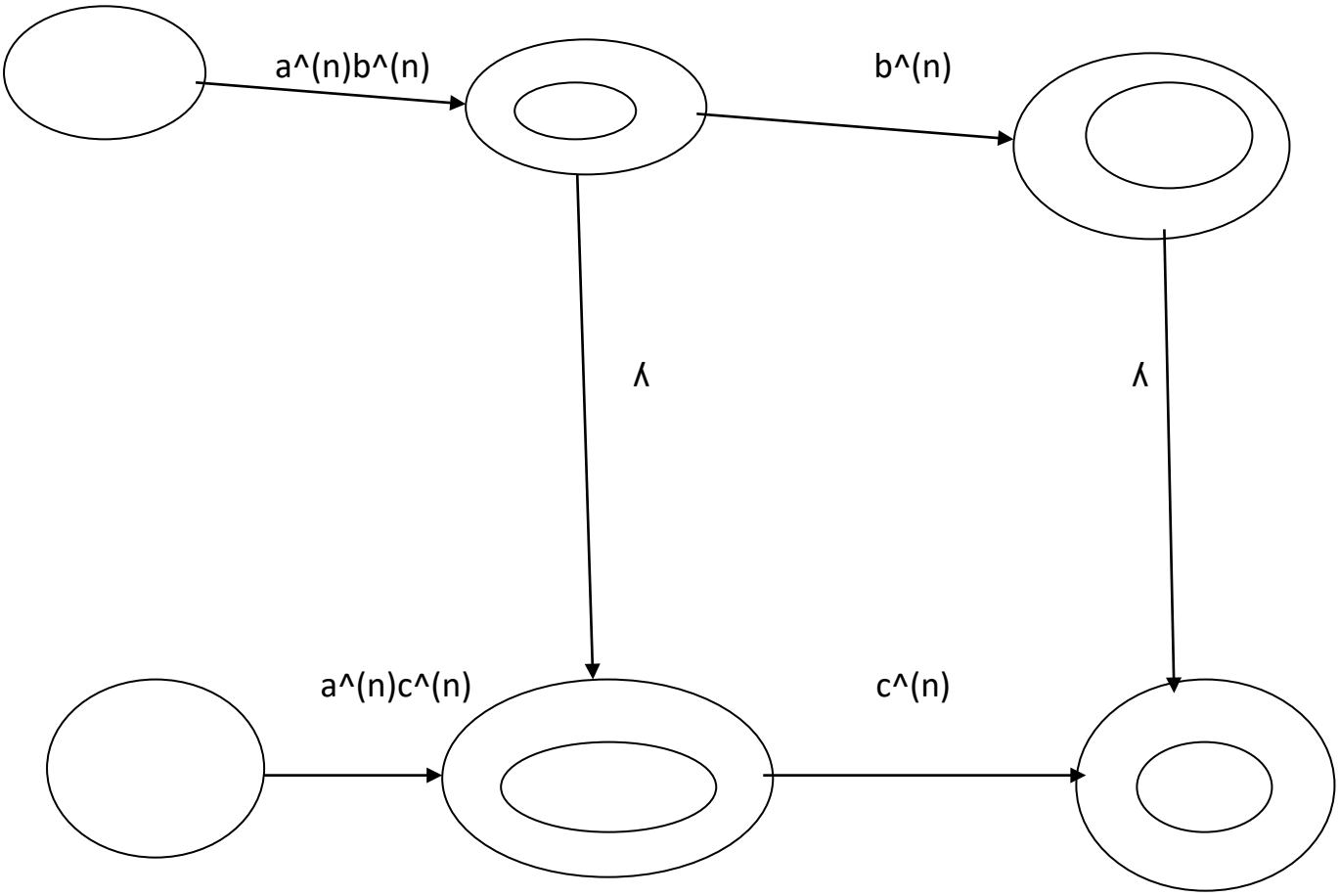
Questo linguaggio non è C.F..

Quindi il linguaggio L U L’ non deve essere C.F. perché L’ non lo è.

Quindi costruisco un PDA che accetta L U L’, il che è una contraddizione.

Prendendo in considerazione l’automa di sopra lo modifico in un automa M’ mettendo le c al posto delle b.

Pertanto ora sono in presenza di due DPDA, quindi connetto gli stati finali dei due automi in questo modo:



Quindi sono riuscito a costruire un PDA che accetta il linguaggio L’ che non è C.F.

Sono giunto a una contraddizione. La contraddizione deriva dall’aver supposto che

L U L’ fosse riconosciuta da PDA.

Pertanto ho dimostrato che i DPDA sono contenuti nei PDA e quindi che ci sono linguaggi accettati da PDA, ma non da DPDA. FINE DIMOSTRAZIONE

ALGORITMO CYK

L'algoritmo cyk è un algoritmo che stabilisce se una stringa appartiene a un linguaggio L C.F..

La grammatica sarà scritta in CNF.

L'algoritmo per arrivare alla decisione usa una matrice a tre dimensioni.

Ha tre dimensioni perché una è la riga, una la colonna e l'altra la variabile.

La matrice avrà lo stesso numero di righe e di colonne della lunghezza della stringa.

Ogni riga di questa matrice analizza una sottostringa lunga quanto il numero di riga, infatti la prima riga è formata dalle stringhe di lunghezza 1, ovvero tutte quelle stringhe che sono formate dalla regola: non terminale -> terminale.

Per la seconda riga, invece si analizzano le stringhe di lunghezza 2, ottenute concatenando le singole stringhe della prima riga.

Per quanto riguarda le altre righe si analizzano sempre stringhe di lunghezza pari al numero di riga, quindi se devo analizzare la terza riga, allora prendo una stringa di lunghezza 1 e una di lunghezza 2, poi una di lunghezza 2 e una di lunghezza 1, fin quando si arriva all'ultima riga che analizza la stringa intera.

Se nell'ultima riga compare il simbolo iniziale S, allora vuol dire che la stringa w € L.

PSEUDOCODICE CYK -> La matrice prende in input n caratteri e prende la grammatica del linguaggio che contiene r non terminali.

Prende in input anche un array di booleani P[n, n, r] che è presente in ogni posizione della matrice inizialmente i valori sono impostati tutti a FALSE, non appena è presente una variabile, in quella locazione nella posizione della variabile nel vettore di booleani viene messo il valore TRUE.

Per generare la prima riga ho il seguente pseudocodice:

Per ogni s = 1 … n

Per ogni produzione R_v -> a_s

Imposto

P[1, s, v], dove 1 è l'indice di riga, s è l'indice del carattere che si sta analizzando e v è la variabile che genera quel carattere.

Ora genero le altre righe:

Per ogni L = 2 … n -> (ho generato la seconda riga)

Per ogni s = 1 … n – L + 1 -> (questa formula mi rende la matrice a gradini)

Per ogni p = 1 … L – 1

Per ogni produzione R_a -> R_b R_c

SE P[p, s, b] AND P[L – s, s + p, c]

ALLORA P[L, s, a] = TRUE.

Pertanto, se in cima alla matrice compare il non terminale S, allora la stringa w è accettata.

MACCHINA DI TURING

La macchina di Turing è definita come una settupla M, dove:

M = (Q, Σ, Γ, δ, q_0, ◇, F)

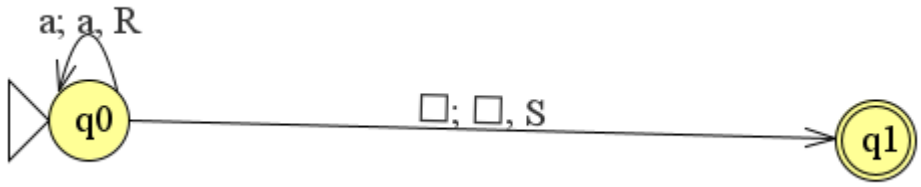
- Q è l'insieme degli stati della macchina;
- Σ è l'alfabeto in ingresso della macchina;
- Γ è l'alfabeto del nastro;
- δ è la funzione di transizione ed è cosi definita: δ(Q, a) -> (Q, b, operazione);
- q_0 è lo stato iniziale della macchina;
- ◇ è il simbolo del blank e indica lo spazio vuoto sul nastro;
- F è l'insieme degli stati finali della macchina.

La macchina di Turing è potenzialmente infinita, ciò vuol dire che non appena mi serve dello spazio sposto semplicemente la testina.

Quando una macchina di Turing giunge in uno stato di accettazione, ferma la computazione e decide se accettare o rigettare.

La mdT non permette transizioni non deterministiche.

ESEMPIO -> Σ = {a, b}, L = a*



ESEMPIO -> L = {a^(n) b^(n)}

Per costruire una macchina di Turing che mi riconosca questo linguaggio ho bisogno di un algoritmo, il quale può essere definito in questo modo:

la a più a sinistra la cambio in x, la b più a sinistra la cambio in y; continuo fino a esaurimento di a e b.

Quindi se non mi rimane nessuna a e nessuna b, allora accetto, se mi rimane o una b o una a, allora rigetto.

aaabbb

xaabbb

xaaybb

xxaybb

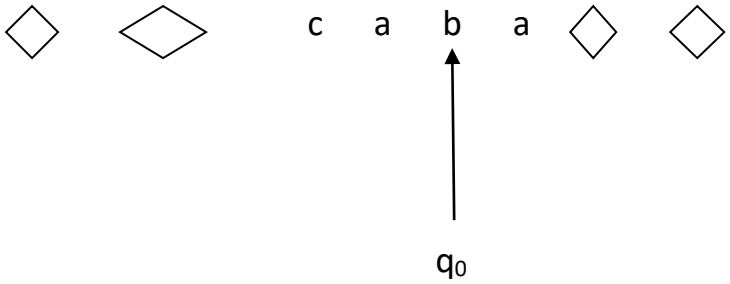
xxayyb

xxxyyb

xxxyyy

Non è rimasta nessuna a e nessuna b, quindi accetto.

CONFIGURAZIONE -> La configurazione serve per capire com'è fatta la memoria in quel momento e qual è il carattere osservato.



Quindi il carattere osservato è quello a destra di q_0, pertanto:

c a q_0 b a

OSSERVAZIONE -> La configurazione iniziale è sempre q_0w

LINGUAGGIO ACCETTATO DA UNA MACCHINA DI TURING -> Il linguaggio accettato da una mdT M è l'insieme di tutte le stringhe w che a partire dallo stato iniziale q_0 raggiungono lo stato finale:

$L(M) = \{w : q_0w \rightarrow^* x_1q_fx_2\}$

MDT MULTITRACCIA

Una macchina di Turing multi traccia M ad m tracce è definita come una sestupla M,

$M = (\Sigma, \diamond, K, q_0, q_f, \delta)$, la funzione delta è definita come segue:

$$\delta_m : (K - \{q_f\}) \times \Sigma_b^m \rightarrow K \times \Sigma_b^m \times \{d, s, i\}$$

SPIEGAZIONE DELTA -> La delta avrà uno stato che non sarà quello finale($K - \{q_f\}$), in quanto nello stato finale la computazione si ferma.

La delta avrà in input una parola di lunghezza m ($\sum_b^m$) e come risultato si otterrà un nuovo stato K, una nuova parola di lunghezza m e l'operazione di spostamento a destra, sinistra o identità({d, s, i}).

SPIEGAZIONE -> Il quadrato tratteggiato nella parentesi tonda non si legge è un errore di word.

La macchina multitraccia è formata da un nastro che a sua volta è suddiviso in molte tracce, in modo tale che, quando la testina effettua un movimento, essa si posizionerà su tutte quelle tracce che sono in corrispondenza.

Dato che la testina si sposta contemporaneamente su tutte le tracce, essa è in grado di leggere e scrivere caratteri vettoriali.

EQUIVALENZA MDT MULTITRACCIA E MDT SINGOLA TRACCIA

TEOREMA -> Una mdT singolo nastro multitraccia M^m con m tracce può essere simulata da una mdT singolo nastro singola traccia M.

DIMOSTRAZIONE -> Sia $M^m = (\Sigma, \text{b}, \delta, K, q_0, q_f)$ una mdT multitraccia con

$\Sigma = \Sigma_1 \times \dots \times \Sigma_m$.

Si supponga di avere una mdT singola traccia M definita in questo modo:

$M = (\Lambda, \text{b}, \delta, K', q_0', q_f')$, dove $|\Lambda| = |\Sigma_1| \times \dots \times |\Sigma_m|$.

ESEMPIO -> $\Sigma_1 = a, b, \quad \Sigma_2 = 1, 2$

Allora in questo caso si ha che $\Sigma_1 \times \Sigma_2 = 4$, ovvero a1, a2, b1, b2.

Nella singola traccia ogni singolo vettore sarà rappresentato da un simbolo:

a1 -> A, a2 -> B, b1 -> C, b2 -> D.

Si definisca, inoltre, una funzione iniettiva $\phi : \Sigma \to \Lambda$ che associa ad ogni simbolo di Σ un simbolo di Λ (a1 -> A, a2 -> B, b1 -> C, b2 -> D).

La funzione delta è definita in modo tale che in corrispondenza di una transizione $\delta_m = (q_i, \sigma) = (q_j, \sigma', v)$ della macchina M^m, la macchina M esegue la transizione

$\delta(q_i, \lambda) = (q_j, \lambda', v)$, con $\lambda = \phi(\sigma)$ e $\lambda' = \phi(\sigma')$.

L'alfabeto su cui opera la macchina singola traccia M è un alfabeto in cui ogni simbolo deriva dalla codifica del vettore di simbolo della macchina multi-traccia M^m.

$\lambda = \phi(\sigma)$ -> (a1 -> A).

In questo modo ho simulato la multitraccia tramite la singola traccia.

MDT MULTINASTRO

La mdT multi nastro ad m nastri è una sestupla:

$M = \{\Sigma, \text{b}, \delta_m, K, q_0, q_f\}$, dove la funzione di transizione è cosi definita:

$$\delta_m : (K - \{q_f\}) \times \Sigma_{\text{b}}^m \to K \times \Sigma_{\text{b}}^m \times \{d, s, i\}^m$$

L'unica differenza con la multitraccia riguarda lo spostamento, infatti qui ogni testina di ogni nastro ha un proprio spostamento.

EQUIVALENZA MDT MULTINASTRO E MDT SINGOLO NASTRO

TEOREMA -> Sia data una mdT M^k con k nastri, allora esiste una mdT a singolo nastro multi-traccia che la simula.

DIMOSTRAZIONE -> Sia M^k una mdT multinastro con k nastri cosi definita:

$M^k = (\Sigma, \delta, K, q_0, q_f, \text{b})$ dove si suppone che per ogni nastro i, 1 <= i <= k, l'alfabeto usato sia Σ_i.

Ora costruisco la mdT mononastro M', con 2k tracce cosi definita:

M' = (Σ', b, K', q_0', F', δ'), dove:

Σ' = {b, ↓} x Σ_1 … x {b, ↓} x Σ_k.

Il nastro di M' risulta composto in questo modo:

- Per ogni i, 1 <= i <= k, la traccia pari di indice 2i contiene la stringa presente sul nastro;
- Per ogni i, 1 <= i <= k, la traccia dispari di indice 2i – 1 contiene il solo simbolo ↓ che sta ad indicare la posizione della testina.

All'inizio della computazione suppongo che il nastro di M' sia fatto nel modo seguente:

- La traccia 1 contiene il solo simbolo ↓ che sta indicare la posizione della testina sul primo carattere a sinistra della traccia 2.
- La traccia due contiene la stringa di input della macchina M^k.
- Per ogni i, 2 <= i <= k, le tracce pari di indice 2i contengono il solo simbolo b.
- Per ogni 2 <= i <= k, le tracce dispari di indice 2i – 1 contengono il solo simbolo ↓ che sta ad indicare la posizione della testina sul carattere più a sinistra della traccia 2.

  Per simulare la funzione di transizione δ^k, la funzione di transizione δ' deve riscrivere 2k simboli, ovvero 1 per traccia.
  Quindi per simulare una funzione di transizione del tipo:
  $\delta^k = (q_i, a_{i1}, \ldots, a_{ik}) = (q_j, a_{i1}, \ldots, a_{ik}, d_1, \ldots, d_k)$
  deve eseguire i seguenti passi:

1) rintracciare i k simboli ↓ che rappresentano le posizioni delle k testine della macchina M^k nello stato q_i.
2) riscrivere i k simboli puntati dalle testine dei k nastri.
3) Posizionare i k simboli ↓ nella posizione delle testine della macchina M^k nello stato q_j.
4) Transire di stato.

Ad ogni passo i simboli ↓ si allontaneranno al massimo di due celle, quindi dopo t passi, nella peggiore delle ipotesi, si allontaneranno al massimo di 2t passi.
Quindi se la M^k compie t passi, allora la macchina mono-nastro M' compierà:

$$\Sigma_{i=1}^{t} 2i = 2\Sigma_{i=1}^{t} i = 2\frac{t(t+1)}{2} = t^2 + t = \mathcal{O}(t^2)$$

Per quanto riguarda la dimensione di Σ', osservo che l'alfabeto delle tracce dispari è 2, mentre quello delle tracce pari per ogni nastro i è Σ_i.
Quindi la dimensione dell'alfabeto Σ' usato dalla macchina M' sarà ottenuto dal prodotto delle cardinalità degli alfabeti dei singoli nastri:

$$\prod_{i=1}^{k} 2|\Sigma_i| = \mathcal{O}((\max|\Sigma_i|)^k)$$

Quindi una mdT multinastro può essere simulata da una mdT mono-nastro multitraccia in tempo quadratico usando un alfabeto di cardinalità esponenziale nel numero dei nastri.
FINE DIMOSTRAZIONE

MACCHINE DI TURING NON DETERMINISTICHE

Una mdT non deterministica è una sestupla:
M = (Σ, ~~b~~, K, q_0, q_f, δ), dove δ è definita nel seguente modo:

$$\delta : (K - \{q_f\}) \times \Sigma_{\not b} \to \mathcal{P}(K \times \Sigma_{\not b} \times \{d, s, i\})$$

Quindi, da una configurazione si può transire in una o più configurazioni simultaneamente e il grado di non determinismo è dato dal numero massimo di configurazioni generate dalla funzione δ.

EQUIVALENZA MDT NON DETERMINISTICHE E MDT DETERMINISTICHE

TEOREMA -> Per ogni mdT M non deterministica esiste una macchina deterministica $M^{(3)}$ a tre nastri equivalente.

DIMOSTRAZIONE -> Per simulare una mdT non deterministica utilizzo l'algoritmo di visita in ampiezza dell'albero di rappresentazione della mdT perché se usassi quello in profondità rischierei di rimanere in un loop se ci fosse un cammino infinito.

Sia d un valore che indica il grado di non determinismo, allora so che a ogni passo posso avere al massimo d nuove scelte.

Numerando queste scelte con numeri che vanno da 1 a d, dopo i passi, avrei al massimo d^i stringhe che rappresentano le possibili computazioni.

Ora costruisco la mdT $M^{(3)}$ a tre nastri nel seguente modo:

1) il primo nastro contiene la stringa di input.

2) Il secondo nastro contiene, per ogni passo di computazione i, stringhe che rappresentano il numero dei percorsi possibili scelti, il numero di stringhe è pari a d^i e la loro lunghezza è i.

3) Il terzo nastro contiene una copia dell'input e varrà utilizzata per le varie combinazioni e, ad ogni nuova combinazione scelta, il nastro verrà ripristinato al contenuto del nastro 1.

La simulazione avviene nel seguente modo:

1)Per ogni i, i >= 1, dove i indica il passo di computazione della mdT non deterministica, si copiano sul nastro 2 tutte le stringhe che rappresentano tutti i possibili percorsi.

2)Per ogni percorso effettuabile si fa una copia del contenuto del nastro 1 sul nastro di lavoro, ovvero il nastro 3, e per ogni indice j appartenente alla stringa del percorso, si applica la j-esima scelta di δ sul nastro 3(se la stringa è 13, allora si effettua prima il passo 1, poi l'1 e infine il 3).

Se esiste un cammino di lunghezza i che porta la mdT non deterministica in uno stato finale, allora esiste sicuramente una fasi di calcolo della macchina a 3 nastri che percorre tale cammino.

Se non esiste questo cammino allora la mdT a 3 nastri non giungerà mai in uno stato finale.

Ad ogni passo di computazione j di M, $M^{(3)}$ compie un numero di passi pari alla lunghezza del cammino (j) per il numero di cammini (d^j), $j * d^j$:

$$\Sigma_{j=1}^{k} j \cdot d^j \in \mathcal{O}(kd^k)$$

Quindi, una macchina non deterministica M può essere simulata tramite una macchina deterministica $M^{(3)}$ a tre nastri in un tempo esponenziale nel numero di passi della macchina non deterministica.
FINE DIMOSTRAZIONE

MACCHINA DI TURING UNIVERSALE

Una macchina di Turing universale è una macchina riprogrammabile che simula tutte le mdT.
La macchina di Turing universale prende in input la descrizione delle transizioni della macchina da simulare e una stringa e genera in output il risultato del calcolo della macchina sull'input.
Una mdT universale è fatta in questo modo:

- Sul primo nastro vi è la descrizione della macchina, quindi vi è il codice della macchina;
- Sul secondo nastro vi è la stringa di input;
- Sul terzo nastro ci sono le istruzioni della mdT da calcolare.

DESCRIZIONE DI M -> Descrivo la mdT M come una stringa di simboli, quindi M viene codificata con la premessa che uno 0 sta ad indicare semplicemente un separatore, mentre un doppio 0 sta ad indicare che l'istruzione è finita e ne segue un'altra:

CODIFICA ALFABETO

SIMBOLI	a	b	c	d
CODIFICA	1	11	111	1111

CODIFICA DEGLI STATI

STATI	q_1	q_2	q_3	q_4
CODIFICA	1	11	111	1111

CODIFICA DEI MOVIMENTI DELLA TESTA

MOSSA	L	R	I
CODIFICA	1	11	111

TRANSIZIONE -> $\delta(q_1, a) = (q_2, b, L)$ $\delta(q_2, b) = (q_3, c, R)$, ora la codifico:
10101101101001101101110111011
Quindi il nastro 1 della mdT universale contiene la codifica della macchina M da simulare.

INSIEMI ENUMERABILI -> Un insieme è enumerabile, se esiste una corrispondenza uno a uno tra gli elementi dell'insieme e i numeri naturali.

OSSERVAZIONE -> Ogni elemento dell'insieme è associato a un numero naturale in modo tale che non esistono due elementi che hanno lo stesso numero; quindi un insieme è enumerabile se esiste una procedura di enumerazione che definisce la corrispondenza con i numeri naturali.
DEFINIZIONE -> Sia S un insieme di stringhe. Un enumeratore per S è una mdT che genera tutte le stringhe di S una per una.
OSSERVAZIONE -> Se esiste per S un enumeratore, allora S è enumerabile; l'enumeratore descrive la corrispondenza di S con i numeri naturali.
TEOREMA -> L'insieme di tutte le mdT è enumerabile.
DIMOSTRAZIONE -> Ogni mdT può essere codificata con una stringa binaria di 0 e di 1.
Definisco un enumeratore per l'insieme delle stringhe che descrivono le mdT e l'enumeratore genera le stringhe binarie di 0 e di 1 in ordine proprio, se la stringa è presente sul nastro la stampo, altrimenti la ignoro.

STRINGHE BINARIE		MDT
0		
1		
01		
11		
.		
.		
.		
10101110		
10101111	----------→	S1 10101111
.		
.		
.		
10101111011101110101	-------→	S2 10101111011101110101

.

.

INSIEMI NON ENUMERABILI -> Un insieme è non numerabile se non esiste un enumeratore che lo enumera.

TEOREMA -> Se S è un infinito enumerabile, allora l'insieme delle parti di S, 2^S, non è enumerabile.

ESEMPIO -> A = {1, 2, 3}, l'insieme delle parti P(A) sarà:

P(A) = {Ø, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}}

DIMOSTRAZIONE -> Poiché S è numerabile, allora posso scrivere S = {S1, S2, …}

Suppongo per assurdo che l'insieme delle parti di S, ovvero 2^S, è numerabile.

Codifico ogni insieme con una stringa binaria di 0 e di 1:

ELEMENTI INSIEME DELLE PARTI	CODIFICA BINARIA				
	S1	S2	S3	S4	…
{S1}	1	0	0	0	…
{S2, S3}	0	1	1	0	…
{S1, S3, S4}	1	0	1	1	…

OSSERVAZIONE -> Ogni stringa binaria infinita corrisponde a un elemento dell'insieme delle parti.

ESEMPIO -> 1001110… corrisponde a:

{S1, S4, S5, S6, …} € 2^S.

Pertanto, assumo per assurdo che l'insieme delle parti è enumerabile, quindi posso enumerare gli elementi dell'insieme delle parti:

2^S = {t1, t2, t3, t4, …}

Suppongo, inoltre, la seguente codifica:

INSIEME DELLE PARTI	CODIFICA BINARIA
t1	10000…
t2	11000…
t3	11010…
t4	11001…

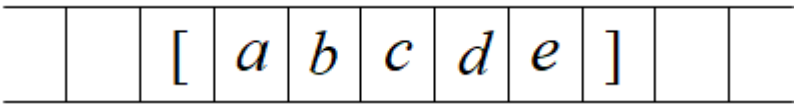
Prendo la diagonale della matrice e la complemento, ovvero complemento la sequenza in rosso:

t(complementato) = 0011

Allora t deve essere uguale a qualche t_i, dove t = t_i.

Ma l'i-esimo bit della codifica di t è il complemento dell'i-esimo bit di t_i, quindi ottengo che t è diverso da t_i, per ogni i.

Sono giunto a un assurdo. L'assunto deriva dall'aver supposto che l'insieme delle parti di S è numerabile.
Ho dimostrato dunque che l'insieme delle parti di S non è numerabile.
FINE DIMOSTRAZIONE
CONSIDERAZIONI -> Quanti sono i linguaggi se parto da un insieme enumerabile S?
I linguaggi sono 2^S.
Quante sono le mdT?
Le mdT sono enumerabili, quindi ci sono più linguaggi che mdT.
FINE CONSIDERAZIONI
AUTOMA LINEAR BOUNDED -> L' LBA è come una mdT solo che c'è una sottile differenza, ovvero che lo spazio in cui è memorizzato l'input è l'unico spazio che può essere utilizzato.



Tutta la computazione si svolge tra i due limiti sinistro e destro.
L' LBA è una macchina di Turing non deterministica.
OSSERVAZIONE -> Gli LBA hanno più potere dei PDA ma meno rispetto alle mdT.

GRAMMATICA CONTEXT-SENSITIVE

Una grammatica C.S. è una grammatica che ha una particolarità sulle regole di produzione, infatti le produzioni sono del tipo:
u -> v, dove |u| <= |v|, inoltre u e v sono stringhe di variabili e terminali.
ESEMPIO -> L = $\{a^nb^nc^n\}$ è C.S., infatti:
S -> abc | aAbc
Ab -> bA
Ac -> Bbcc
bB -> Bb
aB -> aa | aaA
TEOREMA -> Un linguaggio L è C.S. se è accettato da un LBA, ovvero, dato L, costruisco un LBA che riconosce tutte e sole le parole di L.
TEOREMA -> Un linguaggio L è Turing accettabile se e solo L è generato da una grammatica senza restrizioni.
OSSERVAZIONE -> I C.S. hanno come unica restrizione |u| <= |v|, se questa restrizione non ci fosse allora sarebbero Turing accettabili.

DIMOSTRAZIONE -> Data una grammatica senza restrizioni costruisco una mdT non deterministica a due nastri fatta in questo modo:

- Sul primo nastro ci sarà la parola in input;
- Il secondo nastro genera tutte le parole a partire da S o simbolo iniziale.

La mdT fa le seguenti mosse:

parte dalla sinistra del secondo nastro, sceglie una locazione di memoria e scrive S.

Quindi applico in maniera non deterministica le regole che possono semplificare S, quindi avrò terminali e non terminali:

A questo punto passo ai non terminali e applico in maniera non deterministica le regole che possono semplificare i non terminali e cosi via.

Se a un certo punto, sul secondo nastro produco una stringa di terminali che è uguale a quella presente sul primo nastro allora mi fermo e accetto, altrimenti riparto dall'inizio.

Quindi questo è il procedimento che a partire da una grammatica mi permette di costruire una mdT.

Se, invece, a partire da una mdT voglio costruire una grammatica, allora devo seguire il seguente procedimento:

suppongo che lo stato q = non terminale, il carattere c = terminale, ? = qualsiasi carattere, anche blank.

Allora avrò:

$\delta(q, c) = (q\_nuovo, c\_nuovo, L)$ dà q_c -> q_nuovo ? c_nuovo

$\delta(q, c) = (q\_nuovo, c\_nuovo, R)$ dà q_c -> q_nuovo c_nuovo ?

$\delta(q, c) = (q\_finale, c\_nuovo, L \mid R)$ dà q_c -> c_nuovo

FINE DIMOSTRAZIONE

LINGUAGGIO DECIDIBILE -> Un linguaggio è decidibile se è possibile decidere se una stringa € o ~~€~~ al linguaggio.

I linguaggi di tipo 3 e di tipo 2, in quanto posso costruire un automa o un PDA che mi dicono se una stringa € o € al linguaggio.

DEFINIZIONE -> Un problema computazionale è decidibile se il corrispondente linguaggio associato al problema è decidibile.

LINGUAGGI INDECIDIBILI -> Sono linguaggi per i quali non esiste un procedimento di decisione; non esiste una mdT che accetta il linguaggio e prende una decisione per ogni stringa di input.

PROBLEMA DELL'APPARTENENZA -> Sia M una mdT, sia w una stringa:

il problema è w € M?

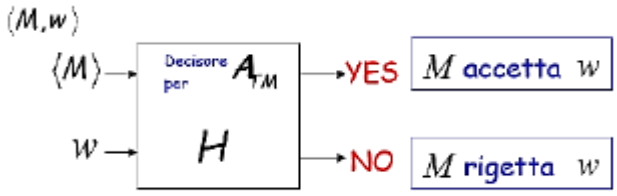
TEOREMA -> Il problema dell'appartenenza per le mdT è indecidibile.

DIMOSTRAZIONE ->Il linguaggio corrispondente al membership problem è A_{TM}, dove:

A_{TM} = {<M, w> : M è una mdT che accetta w}.

Quindi A_{TM} prende in input le coppie <M, w> dove M è una mdT che accetta w, inoltre M può essere scritta come codice.

Suppongo per assurdo che A_{TM} è decidibile, quindi dato che è decidibile esiste una macchina H che data la coppia M, w decide se M accetta w, quindi praticamente questa macchina prende il codice di M e la stringa w e decide se M accetta w :



Adesso definisco una macchina Diag che effettua l'esatto opposto di H, quindi se H accetta, Diag rigetta e se H rigetta, Diag accetta.

Diag, come H, ha due input, ovvero, il codice di M e la stringa w.

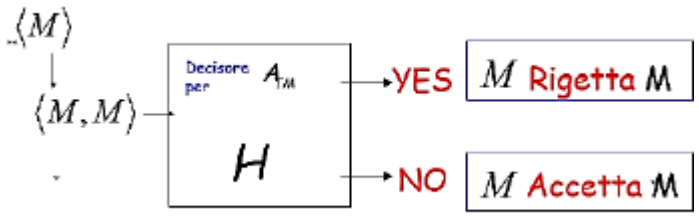
Ora semplifico Diag e al posto della stringa w gli passo il codice di M;

quindi Diag ha come input M, copia il codice di M e lo passa al decisore H, quindi possono accadere due cose:

M accetta M o M rigetta M.

Pertanto risulta:

- Diag accetta M se M rigetta M
- Diag rigetta M se M accetta M



Adesso al posto di M metto Diag perché è una mdT:

- Diag accetta Diag se Diag rigetta Diag
- Diag rigetta Diag se Diag accetta Diag
  Quindi sono davanti a un assurdo.
  L'assurdo deriva dall'aver supposto che A_{TM} fosse decidibile.
  Pertanto risulta cosi dimostrato che il problema dell'appartenenza per le mdT è indecidibile.

FINE DIMOSTRAZIONE

LINGUAGGIO TURING ACCETTABILE -> Sia M una mdT, allora per ogni input da una risposta se la stringa € al linguaggio e quindi la macchina si ferma, ma nulla può dire se la stringa € al linguaggio e dunque la macchina non si ferma.

Sia A_{TM} = {<M, w> : M è una mdT che accetta la stringa w}

A_{TM} è Turing accettabile?

Devo costruire una mdT che accetta A_{TM}:

1) calcolo M con input w;

2) se M accetta w, allora accetta anche <M, w>, se invece M non accetta w allora nulla potrò dire.

Pertanto risulta che A_{TM} non è decidibile ma è Turing accettabile.

DIMOSTRAZIONE -> I linguaggi C.F. sono chiusi rispetto all'intersezione?

Siano L1 = $\{a^n b^n c^k\}$ ed L2 = $\{a^h b^n c^n\}$.

L'intersezione tra i due linguaggi risulta:

L1 intersezione L2 = $\{a^n b^n c^n\}$, ma questo linguaggio per il teorema uvwxy o Pumping Lemma per i linguaggi liberi da contesto non è C.F.

Quindi risulta che i linguaggi C.F. non sono chiusi rispetto all'operazione di intersezione.

FINE DIMOSTRAZIONE

HALTING PROBLEM -> Ha in input una mdT M e una stringa w.

Il problema è: M si ferma nel processo di calcolo con stringa di input w?

Il linguaggio corrispondente è:

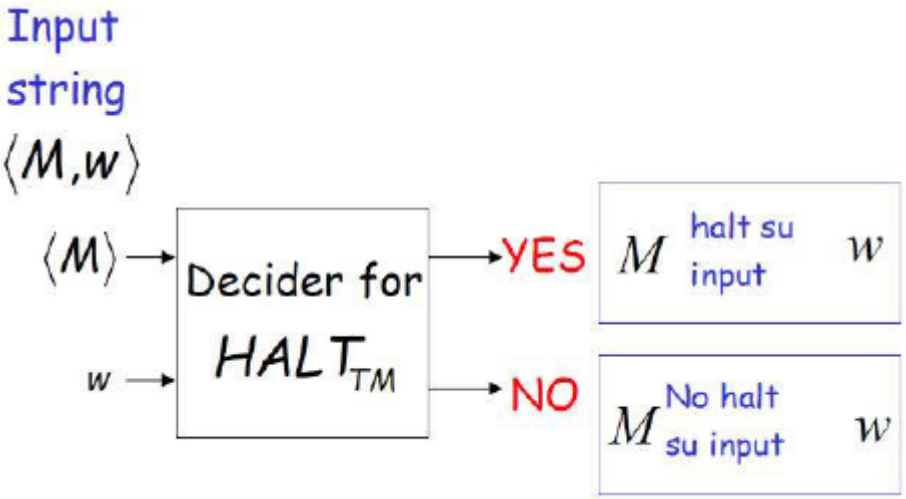
$HALT_{TM}$ = {<M, w> : M è una mdT che si ferma sulla stringa in input w}

TEOREMA -> Il problema dell'HALT è indecidibile.

DIMOSTRAZIONE 1 -> Suppongo per assurdo che $HALT_{TM}$ è decidibile.

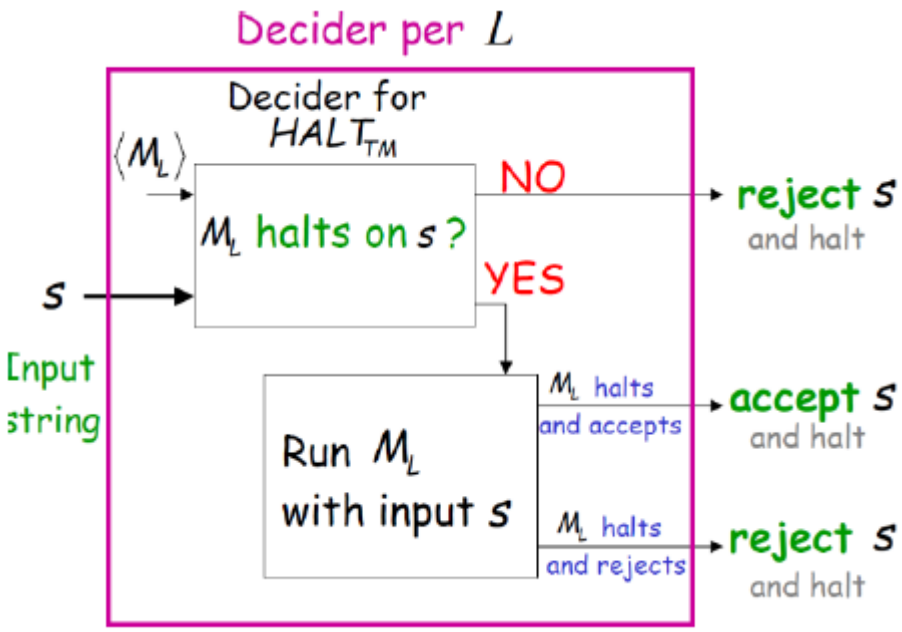
Proverò che ogni linguaggio Turing accettabile o semidecidibile è decidibile.

Se $HALT_{TM}$ è decidibile, allora esiste un decisore che mi da si se <M, w> converge, ovvero si ferma, o mi da no se <M, w> diverge.



Sia L un linguaggio Turing accettabile e sia M_L una mdT che accetta L.

Provo che L è decidibile, quindi costruisco un decisore per L:



Il decisore per L ha in input una stringa s e mi chiedo se M_L si ferma su s usando il problema dell'HALT.

Usando il problema dell'HALT, che ho supposto di esistere, se mi dice no allora rigetto la stringa s e mi fermo, se mi dice si calcolo M_L con input s, e se mi dice si accetto e mi fermo, se mi dice no rigetto ma mi fermo comunque.

Quindi, a questo punto, risulta che L è decidibile.

Poiché L è stato scelto arbitrariamente, allora risulta che ogni linguaggio semidecidibile è decidibile.

Ma per il problema dell'appartenenza so che esiste un linguaggio semidecidibile che è indecidibile.

Quindi la macchina non esiste, ovvero non esiste il decisore per L che avevo precedentemente costruito, quindi non esiste la macchina HALT.

Pertanto il problema dell'HALT è indecidibile.

FINE DIMOSTRAZIONE 1

DIMOSTRAZIONE 2 -> Suppongo per assurdo che $HALT_{TM}$ è decidibile.

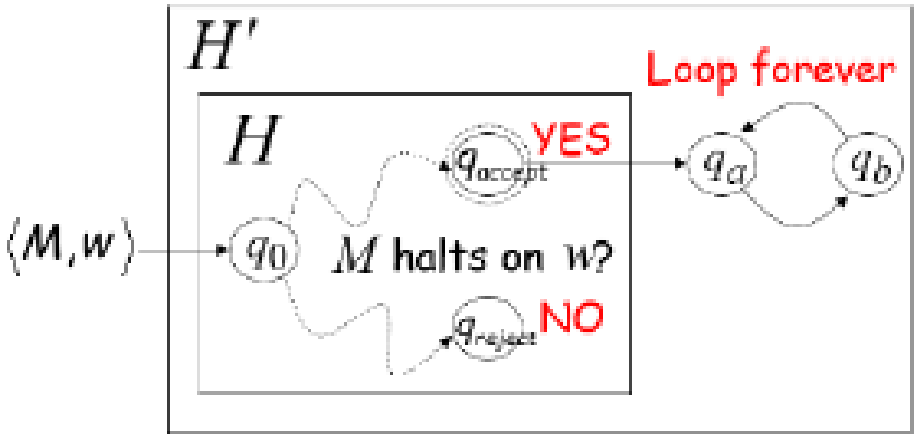
Se è decidibile allora esiste un decisore H con input la macchina M e la stringa w.

Se H mi dice si allora mi fermo e accetto, in tutti gli altri casi non mi fermo.

Ora costruisco una macchina H' diagonalizzando H, ovvero se H si ferma su w, allora H non si ferma su w.

Per fare ciò, nello stato di accettazione inserisco un loop forever, in modo tale che la macchina non si ferma nello stato finale.

Ora uso la stessa tecnica del membership problem, ovvero, ad H' passo in input M e una sua copia:



Quindi risulta che M si ferma su M se M va in loop e viceversa.

E' facilmente intuibile che questo è un assurdo, in quanto M non si può fermare se va in loop.

Quindi risulta che la macchina H non esiste e quindi non esiste neanche H'.

Pertanto risulta cosi dimostrato che il problema dell'HALT è indecidibile.

FINE DIMOSTRAZIONE 2

COMPLESSITA' TEMPORALE

La complessità temporale rappresenta il numero di passi che una mdT effettua per un calcolo.
Quindi, partendo da uno stato iniziale e giungendo in uno stato finale, l'mdT effettuerà un certo numero di transizioni le quali rappresentano la complessità temporale.
Adesso si considerino tutte le stringhe di lunghezza n.
Sia $T_M(n)$ il massimo tempo richiesto per decidere o calcolare una qualsiasi stringa di lunghezza n.
Siano, inoltre, f e g due funzioni, dove f, g : N -> R^+ .
f(n) = O(g(n)) se esistono due interi positivi c ed n_0 tale che per ogni n >= n_0 si ha che f(n) <= c * g(n).
Quindi, quando si scrive O(g(n)) si intende che g(n) è un upper bound per f(n) e, più precisamente, g(n) è un upper bound asintotico per f(n).
CLASSI DÌ COMPLESSITA' TEMPORALE -> TIME(T(n)) rappresenta la classe di complessità temporale, dove TIME(T(n)) sono tutti i linguaggi decidibili da una mdT deterministica in tempo O(t(n)), dove n è la lunghezza della stringa.
Pertanto P rappresenta la classe di complessità temporale:

$$P = \bigcup_{k>0} TIME(n^k)$$

TEOREMA -> Sia t(n) una funzione, dove t(n) >= n.
Ogni mdT non deterministica a singolo nastro con complessità O(t(n)) ha una mdT deterministica a singolo nastro con complessità $2^{O(t(n))}$.
CAMMINO HAMILTONIANO -> Un cammino in un grafo è detto hamiltoniano se questo cammino tocca tutti i vertici una e una sola volta.
Non vi sono soluzioni efficiente per questo problema, infatti l'unica soluzione sarebbe l'enumerazione totale, ovvero l'enumerazione di tutti i possibili cammini, ma questo comporterebbe una complessità esponenziale.

PROBLEMA DELLA CRICCA -> Dato un grafo e dato un intero k trovare un insieme di k nodi dove ciascun elemento è connesso a tutti gli altri.
Anche in questo caso si avrebbe complessità temporale esponenziale.
PROBLEMA DELLA SODDISFACIBILITA' -> Si devono avere espressioni booleane in forma normale congiuntiva, ovvero:
$t_1 \wedge t_2 \wedge \dots \wedge t_k$, dove ogni t rappresenta una clausola e ogni clausola è formata da gruppi di variabili collegati mediante OR:
$(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\dots)$.
Dove ogni gruppo di variabile deve generare valore 1 per rendere la formula valida.
Anche qui si dovrebbero fare tutte le combinazioni di variabile e trovare quella che rende la formula vera.

NON DETERMINISMO

La classe dei linguaggi NTIME(T(n)) rappresenta tutte le mdT non deterministiche i cui rami di computazione sono limitati dalla funzione T(n), quindi ogni cammino dell'albero di computazione è limitato dalla funzione T(n).
Pertanto, data una computazione non deterministica, se una stringa viene riconosciuta in tempo NTIME(T(n)), allora vuol dire che esiste una cammino che accetta la stringa in tempo T(n), dove n è la lunghezza della stringa.
VERIFICATORE -> Un verificatore per un linguaggio A è un algoritmo V, dove:
A = {w | V accetta <w, c> per qualche stringa c}
Quindi per ogni stringa w V accetta <w, c> dove c è una stringa di supporto la quale indica il cammino che si deve fare nell'albero di computazione affinché la stringa w venga riconosciuta.
OSSERVAZIONE -> Nel problema del cammino hamiltoniano, il verificatore coincide con la stringa giusta che da il cammino.
Ragionamento analogo si fa con il problema della cricca e quello della soddisfacibilità.

DIMOSTRAZIONE -> Devo dimostrare che il verificatore implica le mdT non deterministiche e che le mdT non deterministiche implicano il verificatore.
Per quanto riguarda la prima implicazione, ovvero che il verificatore implica le mdT non deterministiche, assumo che il verificatore V sia limitato da una funzione polinomiale n^k, dove n è la lunghezza della stringa in input al verificatore V.
Ora considero tutte le stringhe di lunghezza n^k, quindi la mdT non deterministica effettua le seguenti mosse:

- Su input w di lunghezza n in maniera non deterministica seleziono una stringa di lunghezza n^k;
- Calcolo V su (w, c), se V accetta, allora anche la mdT non deterministica accetta la computazione, se V rifiuta anche la mdT non deterministica rifiuta.

Ora devo dimostrare la seconda implicazione, ovvero che le mdT non deterministiche implicano il verificatore.

Quindi se si ha una mdT non deterministica e si vuole costruire il verificatore bisogna trovare la stringa giusta per il verificatore.

Quindi se la mdT non deterministica accetta la stringa allora evidentemente esiste un cammino, c, che porta da uno stato iniziale a quello finale.

Quindi, il verificatore V avrà in input (w, c), pertanto simulo la macchina sull'input w scegliendo il cammino consigliato da c.

Se la macchina accetta allora V di (w, c) accetta, se la macchina rifiuta anche V di (w, c) rifiuta.

FINE DIMOSTRAZIONE

ESERCIZIO -> L = {w : w è soddisfacibile}

Quindi se si deve valutare se una formula è soddisfacibile in maniera deterministica, allora si devono andare a fare tutte le combinazioni dei valori di verità e questo comporta un calcolo esponenziale.

In maniera non deterministica, invece, si suppone un' assegnazione di valori alle variabili e si verifica che questo assegnamento sia valido. Pertanto questa procedura comporta una complessità di O(n), quindi polinomiale.

FINE ESERCIZIO

OSSERVAZIONE -> La classe polinomiale non deterministica è:

$$NP = \cup NTIME(n^k)$$

OSSERVAZIONE -> Si deduce che il problema della soddisfacibilità è un problema NP.

LINGUAGGI NP-COMPLETI

Una funzione f : A -> B è calcolabile in tempo polinomiale se esiste una macchina M che calcola la funzione f in tempo polinomiale.

Un linguaggio A è riducibile in tempo polinomiale a un linguaggio B, A <= B, se esiste una funzione f tale che se x è elemento di A, allora f(x) € B.

NP-COMPLETEZZA -> Un problema A è NP-completo se:

- A è in NP.
- Ogni problema NP è riducibile ad A.

OSSERVAZIONE -> Se si può risolvere un problema NP-completo in tempo deterministico polinomiale, allora risulta che P = NP, ma questo è un problema aperto.

IMPORTANTE -> Ricordando che una formula SAT è una formula in cui ci sono variabili collegati in OR e questi gruppi di variabili sono in AND tra di loro.

Si vuole dimostrare che una formula SAT rappresenta una qualsiasi computazione non deterministica polinomiale.

TEOREMA DÌ COOK-LEVIN -> Il linguaggio SAT è NP-completo.

DIMOSTRAZIONE -> Quindi devo dimostrare che SAT è NP-completo, pertanto deve risultare che:

1) SAT è in NP;

2) devo ridurre tutti i linguaggi NP al problema SAT in tempo polinomiale.

1) SAT è in NP sia con il verificatore che in maniera non deterministica, infatti con il verificatore la stringa c rappresentava i valori di verità che rendevano la formula vera.

2) Sia dato un linguaggio L € NP.

Devo definire una riduzione polinomiale a SAT.

Dato che L € NP, allora esiste una mdT non deterministica che decide L in tempo polinomiale.

Per ogni stringa w costruisco in tempo polinomiale un'espressione booleana tale che se w € L, allora l'espressione è vera, quindi:

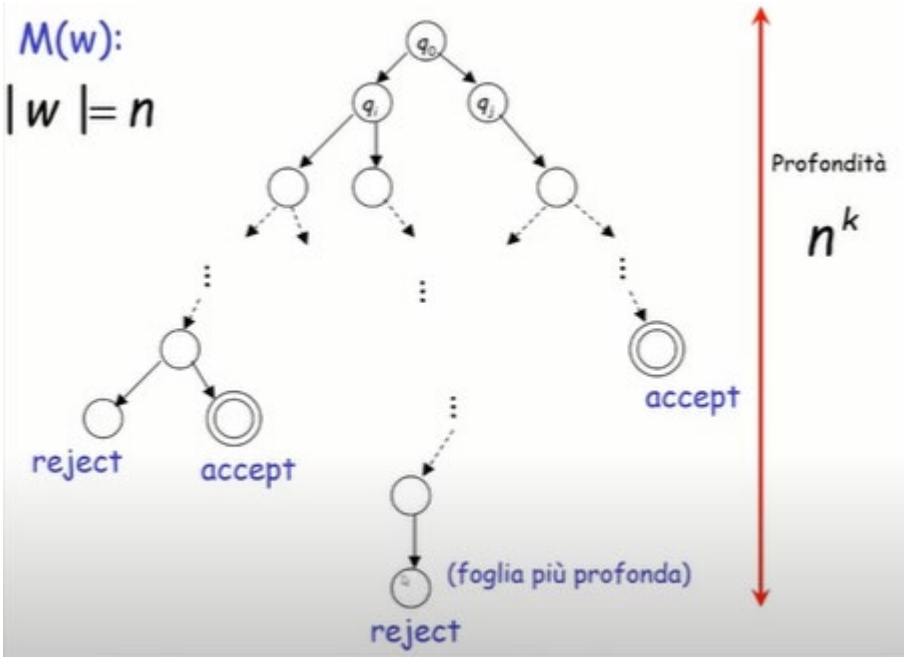
ϕ(M, w) è soddisfacibile se w € L e non soddisfacibile se w non € L.

Adesso vediamo come sono le computazioni della macchina non deterministica M.

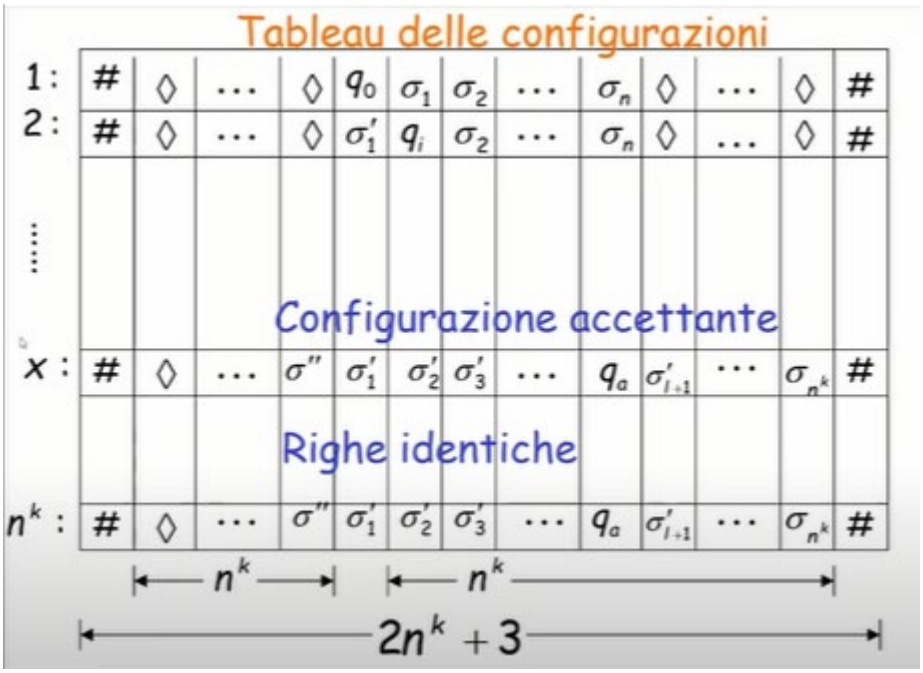
Dato che M è non deterministica ci saranno molte computazioni e dato che L € NP, allora l'albero delle computazioni avrà profondità al massimo n^k, dove n è la lunghezza della stringa in input.

OSSERVAZIONE -> L'albero ha profondità n^k perché n^k è il limite temporale del calcolo di M(w).

Ora considero una computazione accettante che ovviamente sarà al massimo n^k e l'input occuperà al massimo n^k celle di memoria a destra o n^k celle di memoria a sinistra, quindi la massima area di calcolo sul nastro sarà $2n^k$.



Adesso l'albero delle computazioni lo posso vedere sottoforma di una matrice che prende il nome di tableaux delle configurazioni:



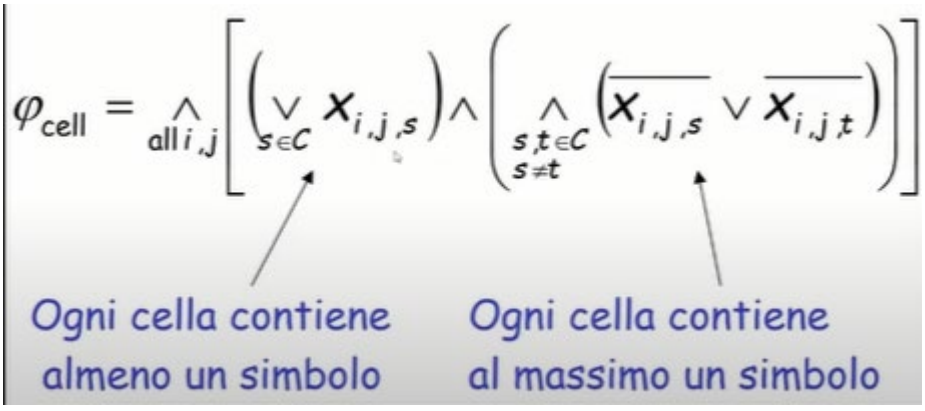
Quindi l'alfabeto del tableaux sarà : C = {{#} U {insieme degli stati} U {alfabeto del nastro}}

Per ogni cella con posizione i, j e per ogni simbolo dell'alfabeto s € C, creo la variabile $x_{i, j, s}$ in cui se nella posizione i e j vi è il simbolo s allora $x_{i, j, s} = 1$, altrimenti è uguale a 0.

Quindi ϕ(M, w) è costruita mediante la variabile $x_{i, j, s}$.

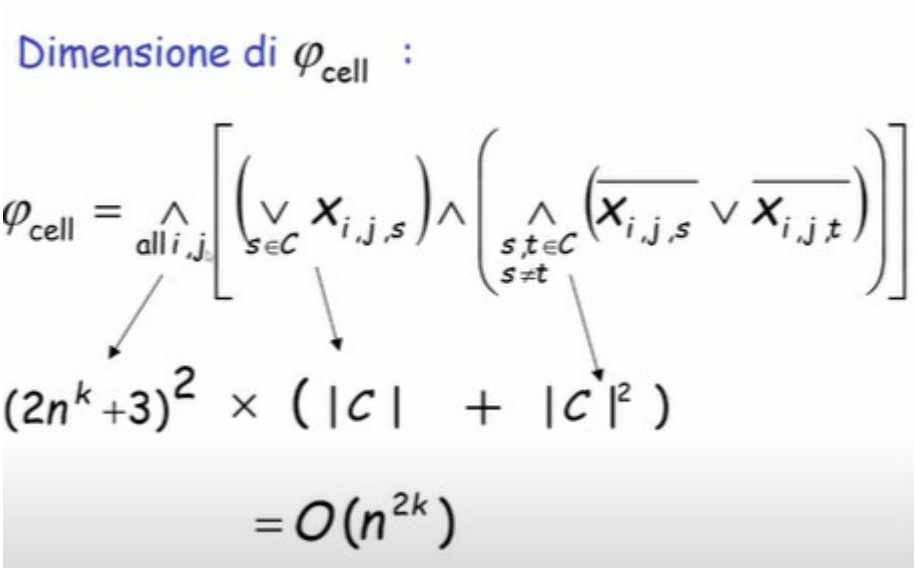
$\Phi(M, w) = \phi_{cell} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$

ϕ_{cell} -> Mi sta a indicare che ogni cella del tableaux delle configurazioni contiene uno e un solo simbolo:



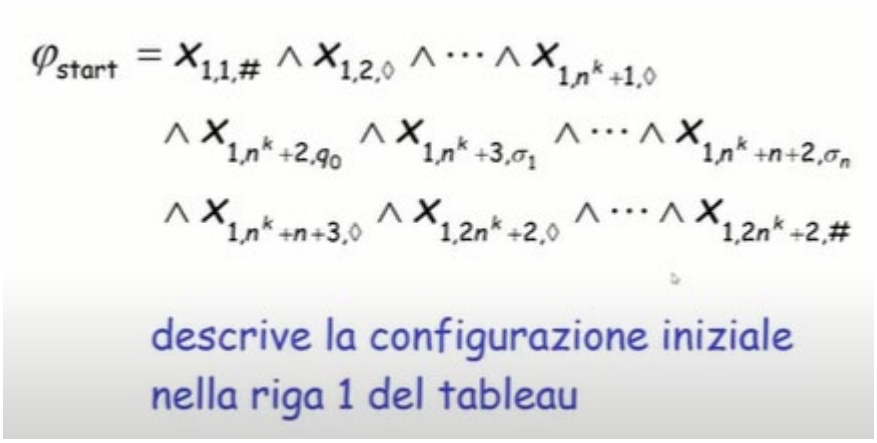
Pertanto per tutte le coppie di i e j esiste almeno un simbolo in ogni cella AND per tutti gli s e t, simboli dell'alfabeto, con s ≠ t deve succedere che o è presente s o è presente t, quindi in ogni cella vi è solo un simbolo e vi è la certezza che questo simboli c'è.

Per quanto riguarda la complessità si che:



Si ha complessità $2n^k + 3$ perché la massima area di calcolo è pari a $2n^k$, poi vi sono due #, uno all'inizio della riga e uno alla fine della riga e in più vi è lo stato, quindi $2n^k + 3$.

ϕ_{start} -> Mi indica come è formata la prima riga del tableaux, quindi come inizia la computazione :



La ϕ_{start} ha dimensione $2n^k + 3$, in quanto lo spazio di ogni singola riga è pari a $2n^k + 3$ e, quindi, ha complessità pari a $O(n^k)$.

ϕ_{accept} -> Mi da la certezza che la computazione raggiunge uno stato di accettazione, infatti risulta:

$$\varphi_{accept} = \bigvee_{\substack{all\ i,j \\ all\ q \in F}} X_{i,j,q}$$

Stati di accettazione

La ϕ_{accept} ha dimensione $(2n^k + 3)^2$, quindi ha complessità pari a $O(n^{2k})$.

ϕ_{move} -> Mi da la sicurezza che vi sono computazioni valide ed è espressa mediante windows legali.

i \ j			
	a	q_1	b
	q_2	a	c

$$x_{i,j,a} \wedge x_{i,j+1,q_1} \wedge x_{i,j+2,b}$$
$$\wedge x_{i+1,j,q_2} \wedge x_{i+1,j+1,a} \wedge x_{i+1,j+2,c}$$

Quindi la ϕ_{move} è pari a tutte le windows(i, j) legali.

La dimensione di una windows legale è pari a 6, il numero di possibili windows legali è pari a $|C|^6$ e il numero di possibili celle è dato dal numero di righe per il numero di colonne del tableaux delle configurazioni, quindi $(2n^k + 3) * n^k$, quindi risulta che la ϕ_{move} ha complessità pari a $O(n^{2k})$

Quindi risulta che :

Dimensione di $\varphi(M,w)$:

$$\varphi(M,w) = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{accept} \wedge \varphi_{move}$$

$$O(n^{2k}) + O(n^k) + O(n^{2k}) + O(n^{2k})$$

$$= O(n^{2k})$$

Quindi vi è complessità polinomiale in n e pertanto, data una mdT non deterministica, costruisco una formula ϕ(M, w) e ho che se w € L allora la formula risulta soddisfacibile.

FINE DIMOSTRAZIONE

3CNF FORMULA -> Una formula in 3CNF è una formula che in ogni gruppo di variabile ce ne sono al massimo 3.

ESEMPIO -> $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_3)$

OSSERVAZIONE -> La ϕ(M, w) può essere ridotta in una formula 3CNF in tempo polinomiale in questo modo:

$(a_1 \vee a_2 \vee \dots \vee a_i)$:

$(a_1 \vee a_2 \vee z_1) \wedge (\bar{z}_1 \vee a_3 \vee z_2) \wedge (\bar{z}_2 \vee a_4 \vee z_3) \dots$

Quindi il valore della formula non dipende dalla variabile introdotta z perché prima la si scrive e poi la si complementa, quindi il risultato non cambia.

Quindi 3CNF-SAT è equivalente a SAT e si deduce che 3CNF-SAT è NP-completo.

TEOREMA -> 3CNF-SAT è riducibile in tempo polinomiale al problema della cricca.

DIMOSTRAZIONE -> L'idea è che se si ha una formula in 3CNF-SAT allora si va a costruire un grafo e se nel grafo vi è una cricca di n nodi connessi tra di loro, dove n è il numero di variabili della 3CNF-SAT, allora la formula è vera:

Quindi vado a costruire il grafo con il seguente metodo:

Ogni variabile rappresenta un nodo e a questo punto da ogni nodo faccio partire un arco e lo collego a tutti i nodi tranne che al suo complemento, eseguo il procedimento per tutti i nodi, quindi per tutte le variabili.

Se alla fine vi è una cricca di grado pari al numero delle variabili allora vuol dire che la formula è vera.

FINE DIMOSTRAZIONE

TEOREMA -> Se il linguaggio A è NP-completo, se il linguaggio B è in NP e se A è riducibile in tempo polinomiale a B, allora B è NP-completo.

La cricca è NP-completo perché 3CNF-SAT è NP-completo, la cricca è in NP, e 3CNF-SAT è riducibile in tempo polinomiale alla cricca

COMPLESSITA' SPAZIALE

La complessità spaziale è il numero di celle di memoria che vengono occupate durante una computazione.

Quindi, a differenza di quella temporale, nella spaziale si parla in termini di vero e proprio spazio.

Infatti si nota la differenza marcata in cui, durante una computazione, si possono usare delle celle che precedentemente erano state usate per la computazione.

Questa è una fondamentale differenza, in quanto, per quanto riguarda la complessità temporale, ovviamente non si può tornare indietro nel tempo.

Infatti, ogni macchina non deterministica ha una macchina deterministica equivalente che calcola lo stesso problema in tempo esponenziale, mentre usa un piccolo spazio aggiuntivo, infatti si ha una mdT deterministica che compie il calcolo usando al massimo uno spazio quadratico rispetto alla mdT non deterministica.

Questo è dato dal fatto che posso andare a riutilizzare celle di memoria già utilizzate, mentre non posso tornare indietro nel tempo.

Come per la complessità temporale, anche in quella spaziale ci sono classi di complessità.

PSPACE rappresenta la classe di complessità spaziale in cui vi sono mdT deterministiche che effettuano il calcolo in spazio polinomiale deterministico.

NPSPACE rappresenta la classe di complessità spaziale in cui vi sono mdT non deterministiche che effettuano il calcolo in spazio polinomiale non deterministico.

TEOREMA DÌ SAVITCH -> Una mdT deterministica simula una mdT non deterministica utilizzando una spazio quadratico.

DIMOSTRAZIONE -> Una prima idea per dimostrare questo teorema sarebbe quella di simulare ogni ramo di computazione della macchina non deterministica, ma, facendo in questo modo, non poteri sovrascrivere aree di memoria siccome per passare da un ramo di computazione a un altro ho bisogno di memorizzare le scelte sul ramo di computazione.

Pertanto, utilizzando questo approccio, avrei bisogno di uno spazio esponenziale.

Per dimostrare questo teorema utilizzo lo Yeldability Problem, il quale verifica se una mdT non deterministica con input w può passare da una configurazione iniziale a una finale in un numero di passi che è minore o uguale a un dato t, dove t rappresenta il numero massimo di operazioni che la macchina non deterministica effettua per accettare la stringa w.

Quindi, siano c_1 ec$_2$ rispettivamente le configurazioni iniziale e finale che utilizzano al massimo uno spazio pari a f(n) e sia t una potenza del 2.

Allora vado a definire una funzione ricorsiva CANYELD(c_1, c_2, 2^t), allora possono accadere le seguenti casistiche:

1) t = 0, allora vuol dire che $c_1 = c_2$, oppure che si passa da c_1 a c_2 in un solo step e, quindi, accetto, altrimenti rigetto.

2) t > 0, allora, evidentemente, si passa dalla configurazione iniziale a quella finale mediante delle configurazioni intermedie c_m, quindi vado a effettuare due chiamate ricorsive della funzione CANYELD:

3) CANYELD(c_1, c_m, 2^{t-1});

4) CANYELD(c_m, c_2, 2^{t-1});

5)Se i passi 3) e 4) accettano allora la computazione viene accettata.

6) Se uno solo tra i passi 3) e 4) non accetta, allora la computazione viene rigettata.

OSSERVAZIONE -> 2^{t-1} perché devo passare prima da c_1 a c_m e poi da c_m a c_2.

Tutte le computazioni utilizzano uno spazio che al massimo è f(n).

Ora vado a definire la mdT deterministica che simula quella non deterministica.

La mdT non deterministica prima di accettare pulisce il nastro e si riporta all'inizio del nastro dove entra in una configurazione di accettazione c_{accept}.

Ora denoto con d una costante tale che la mdT non deterministica effettua al massimo d * f(n) configurazioni per accettare la stringa w.

Pertanto, con queste assunzioni risulta che la computazione viene accettata solo se si passa dalla configurazione iniziale a quella finale in al massimo $2^{d\ f(n)}$ passi.

L'output è raffigurato dal risultato della funzione ricorsiva

CANYELD(c_{start}, c_{accept}, $2^{d * f(n)}$)

Ogni computazione utilizza al massimo f(n) spazio.

Quindi la macchina deterministica utilizzerà per ogni chiamata ricorsiva O(f(n)) spazio per il numero delle chiamate ricorsive che è O(f(n)), quindi la mdT deterministica utilizzerà uno spazio pari a:

$O(f(n)) * O(f(n)) = O(f^2(n))$.

Pertanto una macchina non deterministica è simulata da una deterministica utilizzando uno spazio che è quadratico.

FINE DIMOSTRAZIONE