

DFA, NFA e ϵ -NFA

DFA: è una quintupla formato da: $A=(Q, \Sigma, \delta, q_0, F)$ dove Q sono l'insieme finito di stati, Σ un alfabeto finito di simboli in input, δ è una funzione di transizione $(q,a) \rightarrow q'$ e q_0 in Q è lo stato iniziale mentre F l'insieme di stati finali sottoinsieme di Q .

Formalmente il linguaggio accettato da A è: $L(A) = \{w : \hat{\delta}(q_0, w) \in F\}$
 $L(A)$ è detto linguaggio regolare.

NFA:(non deterministico) come NFA ma qui la δ transizione prende in input (q,a) ma restituisce un sottoinsieme di Q non una singola destinazione.

Formalmente il linguaggio accettato da A è: $L(A) = \{w : \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$

Equivalenza tra DFA e NFA: per ogni NFA N c'è un DFA D tale che $L(D)=L(N)$ e viceversa.

Conversione da NFA a DFA: si rimuoveranno destinazioni multiple non deterministiche

1. creo tabella di tutti nodi con le relative destinazioni per i vari valori delle transizioni
2. se presenti destinazioni composte da più stati realizzo un nodo ulteriore nella tabella formato dall'unione di quei nodi le cui destinazioni sono l'unione delle destinazioni singole di ogni nodo padre per la data variabile
3. realizzo lo schema risultante con nuovi nodi formati dagli ibridi

ϵ -NFA: Automa a Stati Finiti Non Deterministico con ϵ -transizioni

(ϵ -NFA) è una quintupla: $A=(Q, \Sigma, \delta, q_0, F)$ dove i parametri sono come quelli precedenti mentre δ è una funzione di transizione che prende in input:

- uno stato in Q
- un simbolo dell'alfabeto Σ unito a $\{\epsilon\}$

Chiusura degli stati con ϵ -transazioni:

1. scrivo il nodo di partenza
2. scrivo tutti i nodi in cui finisce sommando ad esso pure le chiusure di questi nodi

Conversione da ϵ -NFA a NFA: per rimuovere le epsilon transizioni:

1. faccio chiusura di tutti nodi con epsilon transizioni
2. faccio tabella con stati e epsilon
3. **faccio una tabella delle transazioni Revisionata senza la colonna della epsilon** e trovo tutte le possibili transazioni per quelle che hanno gli stati marcati usando le loro chiusure

Espressione regolare: NFA o DFA sono metodi per costruire macchine che riconoscono linguaggi regolari, un modo dichiarativo per descriverlo sono le EXPREG. Sono formate da:

- ϵ per stringa vuota, linguaggio vuoto e simboli
- operatori con precedenza esecuzione
 1. $*$ per la chiusura di Kleene(loop da 0 a n)
 2. $.$ (concatenazione)

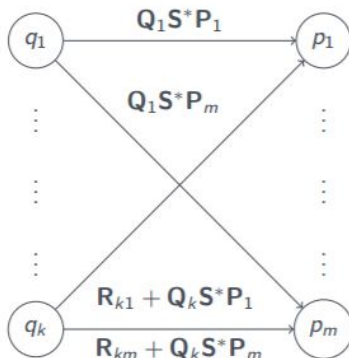
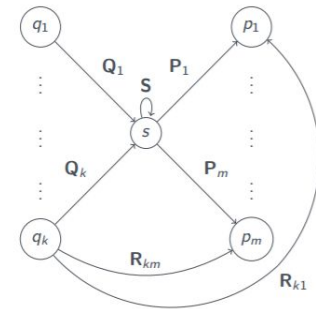
3. +(unione)

- raggruppati con parentesi tonde ()

Per ogni espressione regolare R esiste un ε -NFA tale che $L(A) = L(R)$ lo stesso al contrario.

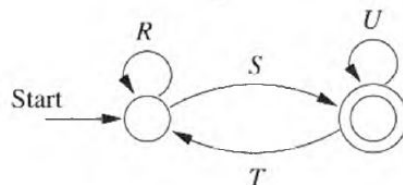
Conversione da Automa a Espressione Regolare:

- per ogni nodo che voglio togliere faccio:
 - lista predecessori e successori
 - per ognuno di questo seguo ogni possibile combinazione di associazioni tra P e S aggiungendo P^*S frecce per ogni nodo che tolgo fino ad ottenere due nodi



- Dobbiamo **ricreare la transizione** per ogni **coppia predecessore-successore** q_i, p_j
- Se non c'è la transizione diretta, l'etichetta è $Q_iS^*P_j$
- Se c'è la transizione diretta, l'etichetta è $R_{ij} + Q_iS^*P_j$

- quando raggiungo automa solo con un nodo iniziale e uno finale allora rimuovo tramite la ricetta descritta a destra: $(R + SU^*T)^*SU^*$

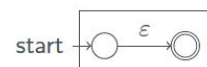


Conversione da Espressione Regolare a Automa ε -NFA:

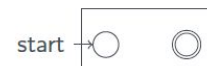
procedura iterativa con caso:

- **Base**

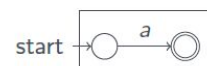
automa per ε



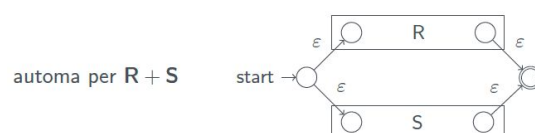
automa per \emptyset



automa per a



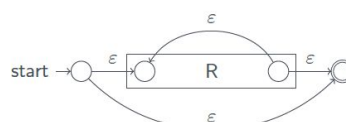
- **Induttivo**



automa per RS



automa per R^*



Studio per determinare se Linguaggio è/non è Regolare con Pumping Lemma:

sia L un linguaggio regolare allora:

- esiste una lunghezza $n \geq 0$ tale che ogni parola $w \in L$ di lunghezza $|w| \geq n$
- può essere spezzata in $w = xyz$ tale che
 - $y \neq \varepsilon$ (secondo pezzo non vuoto)
 - $|xy| \leq n$ (i primi due pezzi sono lunghi al massimo n) devo ricordare questa condizione quando divido nei tre pezzi la mia espressione
 - per ogni $k \geq 0$, $xy^kz \in L$ (possiamo pompare y rimanendo in L)

La dimostrazione quindi che L sia regolare viene fatto supponendo che lo sia e che quindi queste ipotesi siano tutte valide. Nel caso in cui non sia possibile per tutte si può dire che quel linguaggio è non regolare.

Se voglio dire che L è non regolare mi basta mostrare che falsifica il Pumping Lemma ossia esiste un $k \geq 0$ per cui pompando y in xy^kz usciamo dal linguaggio tranne per $k=1$ che mi riporterebbe al caso di partenza non dimostrano nulla.

Noi in questo procedimento scegliamo: la forma della parola generica per n, il valore di p

CFG

Grammatica libera dal contesto CFG: $G=(V,T,P,S)$ dove V è l'insieme delle variabili, T i terminali, P l'insieme delle **produzioni** ed S il simbolo iniziale. Le produzioni ci permettono di capire se determinate stringhe appartengono al linguaggio di una certa variabile tramite una derivazione da corpo alla testa o viceversa.

Ci permette di esprimere definizioni ricorsive.


Produzioni formate da testa \rightarrow corpo.

Inoltre le grammatiche regolari non sono altro che un sottoinsieme delle grammatiche libere da contesto .

Conversione RegExp a CFG con: E per generare Espressioni e I per gli identificatori che porta alle nostre variabili se devo convertire un'espressione regolare numerandole tutte, poi devo formare tabella con:

Stringa	Tipo se I oppure E	N. produzione usata	Stringhe usate preced
---------	--------------------	---------------------	-----------------------

Derivazione: Partendo dal simbolo iniziale deriviamo le stringhe terminali sostituendo ripetutamente una variabile con il corpo di una produzione che ha quella variabile come testa. Ricordo quindi le due derivazioni leftmost o rightmost in base a quale variabile si

sceglie di sviluppare in precedenza se più a sinistra o a destra. Con  star sopra indica abbreviazione della derivazione.

Linguaggio di una grammatica: dato G una CFG il linguaggio di G, denotato con $L(G)$ è l'insieme delle stringhe terminali che hanno una derivazione dal simbolo iniziale(S) ossia



$$L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$$

quindi se L è linguaggio di una CFG anche detto Linguaggio libero dal Contesto o **CFL**.

Forme sentenziali: data $G=(V,T,P,S)$ con $S \Rightarrow^*$ è una forma sentenziale ossia una derivazione dal simbolo iniziale che produce stringhe con un ruolo speciale formate da variabili e terminali.

Albero sintattico: rappresentazione grafica di derivazioni di una grammatica G che è CFG. Se esiste un albero con radice etichettata da una variabile A e con prodotto w , dove w è in T^* . Allora esiste una derivazione a destra/sinistra $A \Rightarrow^* w$ nella grammatica G siccome ogni albero rappresenta un'unica derivazione leftmost e anche rightmost.



Eliminazione Ambiguità grammatica: G è ambiguo se esiste una stringa w in T^* per cui esistono due alberi di derivazione diversi come prodotto ma per risolvere anche se non funziona sempre si può:

1. cambiare grammatica implementando precedenza e associatività
2. variabile per ogni livello di precedenza

Se non si può cambiare grammatica si deve cambiare il Linguaggio!

Grammatica valida per utilizzo:

1. prima si dimostra che grammatica genera stringhe consone a utilizzo
2. dal linguaggio sopra dimostrato a derivazione

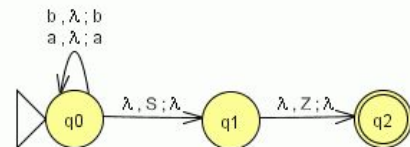
$L(G_{pal})$ insieme delle palindrome su $0,1$

PDA

Automa a pila: è un automa a stati finiti non deterministico con epsilon transizioni ma con uno stack in cui può memorizzare una stringa di simboli a cui può accedere con politica LIFO serve per riconoscere i CFL generalizzando DFA e gli NFA, dimostra che ogni linguaggio regolare è libero da contesto.

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

Dove Q gli stati, Σ è alfabeto, S simboli dello stack, Z_0 è il simbolo all'inizio dello stack mentre q_0 è lo stato iniziale e F quello finale.



$$\delta(q_i, a, X) = \{(q_j, Y)\}$$

Dove q_i è uno stato in Q , a un simbolo di input in Σ oppure epsilon e X è un simbolo di stack ossia membro di S . Il suo output è un insieme finito di coppie formate da q_j che è il nuovo stato e Y la stringa di simboli di stack che rimpiazza X alla sommità dello stack.

Se troviamo più di una coppia a destra vuol dire che l'automa potrebbe prendere più di una strada per ogni stato possibile di destinazione.

Inoltre in ogni istante un PDA ha: (q, w, y) ossia stato corrente q , w parte non consumata input, y stack corrente dall'inizio ad ora.

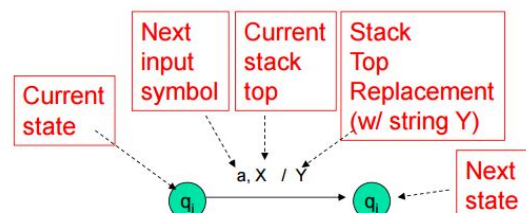
Le operazioni possibili nelle transizioni di $\delta(q_i, a, X) = \{(q_j, Y)\}$ sono:

- POP-PUSH con a in lettura sul nastro in input e passa nello stato q_j rimpiazza X con Y in cima alla pila e muove nastro input di una cella a destra.
- PUSH se $X = \epsilon$ con a in lettura sul nastro in input e indipendentemente dal simbolo in cima alla pila di passa allo stato q_j e muove il nastro di input di una cella a destra.
- POP se $Y = \epsilon$ con a in lettura sul nastro in input e X in cima alla pila, passa nello stato q_j ed elimina X dalla cima della pila e muove nastro input di una cella verso destra.
- DA RIVEDERE ($\delta(\text{stato}, \epsilon, \text{stack})$ permette di andare nella destinazione con un input vuoto ossia quando si vuole, se fosse presente invece destinazione pari a $\{(\epsilon, \text{stato})\}$ vorrebbe dire che staremo invece eseguendo una eliminazione dalla cima dello stack)

Attenzione se a è ϵ tutte e tre le mosse avvengono senza che la testina si sposti.

Notazione grafica PDA: I nodi corrispondono agli stati del PDA, sempre presente stato iniziale e archi corrispondono a tutte le transizioni del PDA.

Un arco etichettato $a, X/Y$ dallo stato q_i allo stato q_j significa che $\delta(q_i, a, X) = \{(q_j, Y)\}$ e come simbolo iniziale dello stack useremo sempre Z_0 .



Descrizione istantanea ID: configurazione formata da tripla (q, w, y) dove q è lo stato, w l'input residuo e y il contenuto dello stack, valido per entrambe le parti della notazione sottostante.

$\delta(q, a, X) = \{(p, A)\}$ allora sarà vero che:

- $(q, a, X) \vdash (p, \varepsilon, A)$
- $(q, aw, XB) \vdash (p, w, AB)$

Nota. \vdash è una notazione turnstile e rappresenta una mossa mentre \vdash^* rappresenta una sequenza di mosse. Vediamo ora che i PDA accettano una stringa consumando in due modalità equivalenti:

1. per Accettazione per stato finale
2. per Accettazione per stack vuoto

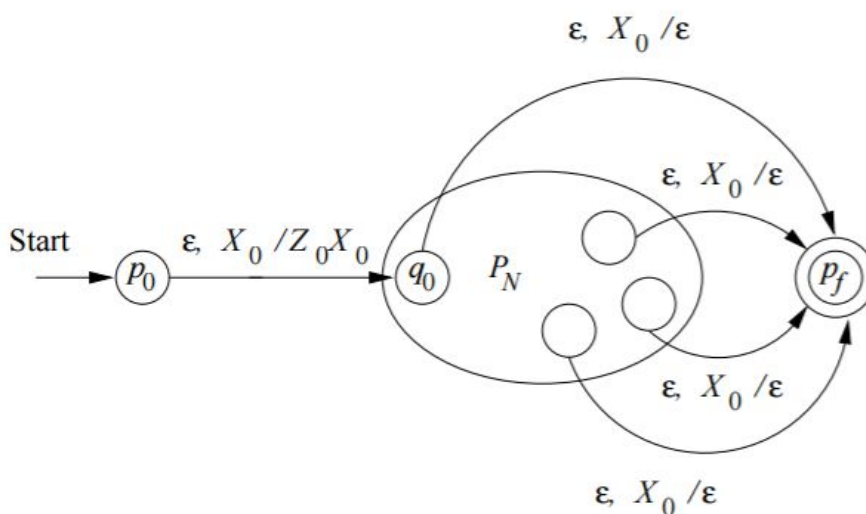
Accettazione per stato finale: ossia stringa "x" accettata da un PDA se e solo se al termine della computazione su "x" ci si trova sullo stato finale quindi sia $P = (Q, \Sigma, S, \delta, q_0, Z_0, F)$ il PDA, il linguaggio accettato da P per stato finale è:

- $L(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, a) \}$ per uno stato q in F e una stringa qualsiasi a dopo aver consumato w .

Accettazione per stack vuoto: sia $P = (Q, \Sigma, S, \delta, q_0, Z_0, F)$ il PDA, il linguaggio accettato da P per stack vuoto è:

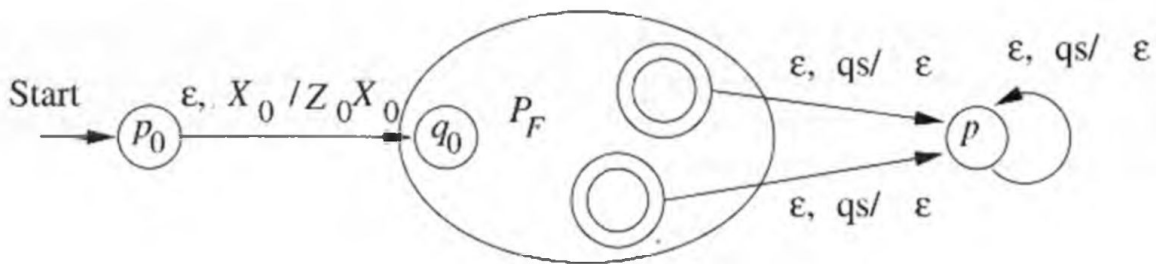
- $N(P) = \{w \mid (q_0, w, Z_0) \vdash^* (q, \varepsilon, \varepsilon) \}$ dove q può essere uno stato qualunque e notiamo che alla fine lo stack dovrà essere vuoto.

Da Stack vuoto a Stato Finale: se $L = N(P_N)$ per un PDA $P_N = (Q, \Sigma, S, \delta_N, q_0, Z_0, \text{"no } P_F")$ allora esiste un PDA P_F tale che $L = L(P_F)$

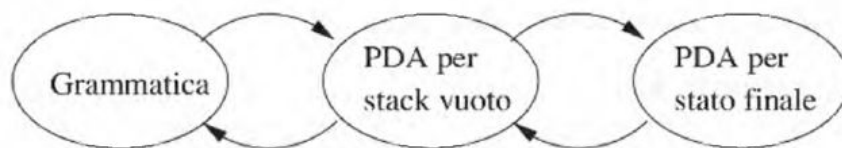


Dove $P_F = (Q \cup \{p_0, p_f\}, \Sigma, S \cup \{X_0\}, \delta_F, p_0, X_0, \{p_f\})$ $\delta_F = \delta_N \cup \{(p_0, X_0, Z_0, q_0), (q_0, X_0, X_0, q_0), (p_f, X_0, X_0, p_f), (p_f, X_0, q_0)\}$

Da Stato finale a Stack Vuoto:



Equivalenza di PDA e CFG



Costruzione di un PDA da un CFG con spiegazione: si costruisce un PDA stack vuoto che simulerà le derivazioni leftmost, infatti ogni forma sentenziale lm non terminale è: $x A \alpha$

- x è stringa di terminali
- A variabile più a sinistra
- α è stringa di variabili e di terminali

$$\underbrace{(a+}_{x} \underbrace{\overbrace{E}^A}_{\alpha})_{tail}$$

$A\alpha$ è la coda della forma sentenziale.

L'intuizione è che $S \Rightarrow^* lm \ x A \alpha \Rightarrow^* lm \ w$ dove tale che

$$(p, w, Z_0) \vdash^* (p, w', A\alpha) \vdash (p, \epsilon, \epsilon)$$

e se $A \rightarrow \beta$, allora esiste una transizione $\delta(p, \epsilon, A) = \{(p, \beta)\}$ e se β contiene terminali si effettua match con l'input.

Quindi in realtà il PDA simula le derivazione da S e usa i terminali che produce per consumare l'input e se l'input finisce e lo stack si vuota allora ha indovinato la derivazione giusta, sempre con derivazioni leftmost. Nei casi negativi la diramazione del PDA muore.

Th.6.13 se P è PDA costruito da G allora $N(G) = L(G)$ ossia vediamo come arrivare a un PDA con stato finale.

Conversione di CFG in un PDA:

Input: $CFG \Rightarrow G = (V, T, P, S)$ con **Output:** $PDA \Rightarrow P = (Q, \Sigma, S, \delta, q_0, I, F)$

1. converto le produzioni della CFG in una GNF.
2. Il PDA avrà un solo stato $\{q\}$.
3. Il simbolo iniziale di CFG sarà il simbolo iniziale nel PDA.
4. Tutti i NON-Terminali del CFG saranno i simboli dello stack del PDA e tutti i terminali saranno i simboli in input nel PDA.
5. Per ogni produzione nella forma $A \rightarrow aX|a|...$ dove " a " è il terminale e A, X sono le combinazioni di terminali e non terminali, formando una transizione $\delta(q, \epsilon, A)$ (che dovrò sviluppare a destra) $= \{(q, aX), (q, a), \dots\}$.

6. Per tutti i terminali come "a" invece li svilupperà nella forma $\delta(q, a, a) = \{(q, \epsilon)\}$

Costruzione di una CFG da un PDA con spiegazione: ragioneremo pensando a come un PDA consuma x e vuota la pila quindi l'azione determinante è l'eliminazione dalla cima dello stack di un simbolo con conseguente lettura di una parte di input e così facendo il PDA può pure cambiare stato scendendo di un livello nello stack. Quindi la costruzione della grammatica equivalente è composta da due eventi con simbolo $[pXq]$:

1. eliminazione effettiva di un simbolo X dallo stack
2. il passaggio dallo stato p allo stato q, dopo la sostituzione di X con ϵ sullo stack

Th.6.14 sia $P = (Q, \Sigma, S, \delta, q_0, Z_0)$ un PDA. Allora esiste una grammatica libera dal contesto, G, tale che $L(G) = N(P)$

Conversione da PDA a CFG:

Input: un CFG, $G = (V, T, P, S)$ e **Output:** un PDA, $P = (Q, \Sigma, S, \delta, q, Z_0)$

	S		$[q Z_0 q][q Z_0 p]$	stato iniziale con simbolo iniziale più tutti possibili stati finali
$\delta(q, 1, Z_0) = \{(q, X Z_0)\}$	$[q Z_0 q]$		$1 [q X q] [q Z_0 q]$	tutti i possibili casi con tutti gli stati possibili quindi 2^n -stati
	$[q Z_0 q]$		$1 [q X p] [p Z_0 q]$	
	$[q Z_0 p]$	\rightarrow	$1 [q X q] [q Z_0 p]$	
	$[q Z_0 p]$		$1 [q X p] [p Z_0 p]$	
$\delta(q, 0, X) = \{(p, X)\}$	$[q X p]$		$0 [p X p]$	un caso per ogni singolo stato
	$[q X q]$		$0 [p X q]$	
$\delta(p, 1/\epsilon, X) = \{(p, \epsilon)\}$	$[p X p]$		1	simbolo unico

dopo aver realizzato queste forme sostituisco con delle variabili tutte le parti dentro le parentesi quadre e semplifico il tutto

Creazione di automa PDA partendo da un linguaggio regolare usando una CFG: ! DA FARE !

DPDA

DPDA - Automi a pila deterministici: un PDA è deterministico (sottoclasse di PDA) se in ogni situazione non c'è scelta fra più mosse alternative ossia se e solo se:

- $\delta(q,a,X)=\{(p,a)\}$ è sempre vuoto o con un solo elemento
- $\delta(q,a,X)$ non è vuoto allora $\delta(q,\epsilon,X)$ deve essere vuoto

Ossia non è possibile avere la scelta fra più mosse diverse nello stesso stato, con lo stesso input e lo stesso simbolo di stack. Inoltre non possibili neppure più transizioni con ϵ con stesso input in cima allo stack per diversi stati ma solo uno.

Th.6.17 Se L è un linguaggio regolare, allora $L = L(P)$ per un DPDA P . Ossia simula un automa a stati finiti deterministico ma ignorando stack dove tiene solo Z_0 e in altri termini realizzando un DFA.

Relazione tra dimensione classi di linguaggi: $LR \subset L(DPDA) \subset CFL$, inoltre i DPDA che accettano per stack vuoto sono meno espressivi e possono riconoscere solo CFL con la proprietà del prefisso.

Linguaggi accettati da DPDA: dato un linguaggio L regolare e DPDA P

- accetta per stato finale se $L = L(P)$ per P Th.6.17
- accetta per stack vuoto solo linguaggi che hanno proprietà prefisso $L = N(P)$ per P Th.6.19 ma quindi questi DPDA non riconoscono tutti i L_{reg} che non soddisfano proprietà prefisso.
- o anche accettati alcuni linguaggi non regolari ma alla fine tutti non ambigui

Differenza fra stato finale stack vuoto rilevante ma colmabile con conversione linguaggio in uno che soddisfa la proprietà del prefisso aggiungendo un simbolo nuovo al L alla fine di ogni stringa essendo quindi unico.

Proprietà del Prefisso: L ha questa proprietà se non esistono due stringhe distinte in L tali che una è prefisso dell'altra. Es. $L = w c w^r$ possiede questa proprietà mentre $\{0\}^*$ no.

Dobbiamo quindi capire che esistono linguaggi accettati da un PDA che non possono essere accettati da un DPDA come il linguaggio dei palindromi.

Th.6.19 $L = N(P)$ per qualche DPDA P se e solo se L ha la proprietà del prefisso e $L = L(P')$ per qualche DPDA P' .

Concludiamo che i linguaggi accettati dai DPDA per stato finale includono propriamente i linguaggi regolari, ma sono inclusi propriamente nei CFL.

Th.6.20 Se $L = N(P)$ per qualche DPDA P , allora L ha una CFG non ambigua. Se L è in $L(DPDA)$ allora non è ambiguo.

Dimostrabile perchè P accetta stringa w per stack vuoto tramite una sequenza di mosse unica ossia possiamo determinare l'unica scelta di derivazione a sinistra mediante la quale la grammatica deriva w .

Inoltre pure DPDA che accettano per stato finale hanno grammatiche non ambigue ossia partendo da PDA per stack vuoto cambiamo il linguaggio in questione per soddisfare la proprietà del prefisso aggiungendo un simbolo "segnale di fine".

Th.6.21 Se $L=L(P)$ per qualche DPDA P , allora L ha una CFG non ambigua. Dimostrazione con aggiunta simbolo $\$$ e ausilio Th.6.19 e Th.6.20.

Attenzione i due modi di accettazione, per stato finale e per stack vuoto, nei DPDA non hanno la stessa portata. I linguaggi accettati per stack vuoto coincidono con quelli accettati per stato finale che hanno la proprietà di prefisso: nessuna stringa nel linguaggio è il prefisso di un'altra parola nel linguaggio stesso.

In conclusione i DPDA accettano (per stato finale) tutti i linguaggi regolari e alcuni linguaggi non regolari. I linguaggi dei DPDA sono liberi dal contesto e hanno tutti CFG non ambigue. Perciò si collocano, in senso stretto, tra i linguaggi regolari e i linguaggi liberi dal contesto.

CNF

La forma Normale di Chomsky (CNF) per le CFG: non contiene produzioni ϵ a meno che non siano nell'assioma comparando a destra, non contiene produzioni unitarie come $A \rightarrow B$ e non contiene simboli inutili non fertili o non raggiungibili da assioma inoltre prevede che tutte le produzioni siano di tipo 2 forma ridotta con:

- $A \rightarrow a$ con a contenente solo 2 simboli non terminali
- $A \rightarrow B$ dove B contiene un solo simbolo terminale

Nessuna contaminazione di terminali con variabili, no produzioni unitarie e non c'è ϵ !!

Th. per ogni CFG G esiste una CFG G' in CNF tale che $L(G') = L(G) - \{\epsilon\}$

Conversione di una grammatica CFG in FNC:

1. introduzione di nuovo simbolo di partenza S_0
2. eliminazione delle ϵ -produzioni $A \rightarrow \epsilon$ singole creando versioni alternative nella base dove compare A
3. eliminazione di tutte produzioni unitarie come $A \rightarrow B$ dove A e B sono variabili senza altri possibili valori come $A \rightarrow B \mid C$
 - Esempio di eliminazione di produzione unitarie
che partivano da E per arrivare a $\rightarrow I$ con merge
sostituzione verso l'alto da I ad E

$E \rightarrow E + T \mid T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid IO \mid I1$
 $T \rightarrow T * F \mid (E) \mid a \mid b \mid Ia \mid Ib \mid IO \mid I1$
 $F \rightarrow (E) \mid a \mid b \mid Ia \mid Ib \mid IO \mid I1$
 $I \rightarrow a \mid b \mid Ia \mid Ib \mid IO \mid I1$
4. sistemazione delle produzioni con più di due non-terminali come corpo non ancora in FNC ossia facendo in modo che:
 - corpi di lunghezza 2 o più siano di sole variabili introduco quindi nuovi variabili nelle produzioni con terminali e variabili per sostituire tutte le variabili
 - scomposizione corpi di lunghezza 3 o più in cascate di produzioni con corpi da due variabili per ciascuna

Alla fine di questa pulizia otteniamo solo simboli utili anche perché utilizzare ϵ produzioni pur essendo utilizzato in molte risoluzioni non è essenziale.

ATTENZIONE uno stato è non raggiungibile anche se per esempio in $S \rightarrow AB$ viene eliminata B allora lo stato è impossibile anche se A è generatore siccome la coppia dei due è inesistente, l'ordine delle AZIONI è IMPORTANTE non reversibile.

Proprietà su alberi delle CNF: una grammatica CNF produce solo alberi binari

Th.7.17 negli alberi di derivazione di una CFG in CNF è vero che se il cammino più lungo è di lunghezza n , allora il prodotto w è tale che $|w| \leq 2^{n-1}$.

Pumping Lemma per CFG: serve per determinare se una grammatica è libera dal contesto oppure no simile a quello delle ExpReg ma necessita di 5 sezioni dalla parola su cui ci basiamo, ossia se linguaggio è non CF. L è libero da contesto se ogni stringa sufficientemente lunga di un CFL posso trovare due sottostringhe vicine che è possibili eliminare o ripetere insieme, ottenendo sempre stringhe nel linguaggio. Sia L un CFL esiste un n tale che se z è in L e $|z| \geq n$ allora possiamo scrivere z uguale a $uvwxy$, con le seguenti condizioni:

- $|vwx| \leq n$ formato da parte centrale z senza bordi
- con esterni di questo $vx \neq \epsilon$
- per ogni $i \geq 0$ uv^iwx^iy appartiene a L

Per comprenderlo meglio immaginiamo di avere un albero A nel quale un sotto albero A' e nei suoi discendenti compare A'' al quale si arriva con un cammino con $|vx| > 0$.

Th.7.24 i CFL sono chiusi sotto **unione, concatenazione, chiusura di Kleene e chiusura positiva(+)**

Problemi indecidibili per CFL: non esiste alcun algoritmo in grado di decidere i seguenti problemi come verificare se CFG è ambigua o dire se due siano uguali o verificare intersezione di due CFL sia vuota.

Analisi lessicale e sintattica

1. **Analisi lessicale:** realizzata da automa a stati finiti che genera sequenza di Token lessicali(numero, variabile, operatore), il compilatore legge la stringa di caratteri sorgente e la raggruppa in sottosequenze chiamate lexemi per i quali viene assegnato un token $\langle \text{nome_token}, \text{valore} \rangle$ quindi valore di output
2. **Analisi sintattica(parsing):** si basa su una CFG che è fissata una volta per tutte al momento in cui viene definito il linguaggio, qui il compilatore trasforma mediante una grammatica la stringa di token in un albero sintattico anche detto albero di parsing, se grammatica ambigua deve essere possibile rimuoverla per non avere più alberi sintattici.

$LL(2)$ = possiamo guardare due token

$LL(k)$ = possiamo guardare k token

Noi vedremo quindi come fare una CFG $LL(1)$ con Parsing recursive descent con metodo top-down dove l'albero di derivazione è costruito dalla radice alla frontiera.
Esistono inoltre metodi di parsing bottom-up dalla frontiera alla radice utilizzabili da più CFG.

Tabella di parsing è in L1 se ogni entrata ha al massimo una produzione.

Dobbiamo generare 3 tabelle prima col first poi con simboli e terminali per capire dove arrivare e per usare pure \$ fare ultima tabella con Follow per riaggiornare 2 tabella con \$ potendo quindi sviluppare codice parser espressione scegliendo una riga 2 tabella come:

Simbolo $\rightarrow T E' \mid$ con $\text{Follow}(\text{Simbolo}) = \{ \$, + \}$

```
Token* parseSimbolo(Token* T){
    switch(T[0]){
        case '(',VAR,NUM:
            Token*k1=parseT(T+1);
            return parseE'(k1);
        case '+','(',')','$':
            return T; //genitore di Simbolo
        default:throw errore
    }
}
```

Macchine di Turing

Esegue in funzione del suo stato e del simbolo che si trova nella cella visitata dalla testina di lettura del nastro compiendo quindi al massimo 3 azioni:

- cambia stato
- scrivere un simbolo di nastro nella cella che visita
- muove la testina verso sinistra o verso destra

Macchina di Turing deterministica è una 7-upla pari a $M=(Q,\Sigma,\Gamma,\delta,q_0,B,F)$ dove:

- Q è un insieme finito di stati, almeno due uno accettante e uno di rifiuto
- Σ è un insieme finito di simboli di input non può contenere blank
- Γ è un insieme finito di simboli di nastro compreso blank
- δ è una funzione di transizione da Q a $Q \times \{L, R\}$
- q_0 è lo stato iniziale
- $B \in \Gamma$ è il simbolo blank
- $F \subseteq Q$ è l'insieme di stati finali

Una TM cambia configurazione dopo ogni mossa e le descrizioni istantanee sono fatte con una stringa ID pari a $X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n$ dove:

- q è lo stato della TM
- $X_1 X_2 \dots X_n$ è la porzione non blank/vuota del nastro
- la testina è sopra il simbolo i -esimo

Useremo \vdash_M per indicare una mossa di M da una configurazione ad un'altra.

$\delta(q, X_i) = (p, Y, L \text{ o } R) \rightarrow$ nel diagramma di transizione diventa:

- un nodo per ciascun stato q e p
- un arco che connette q e p con etichetta X_i / Y freccia sinistra \rightarrow o destra

Ogni volta ricordo quindi che è possibile spostarsi solo a sinistra o destra.

Linguaggi di una TM:

- Linguaggi ricorsivi: esiste una TM che si arresta su ogni stringa accettata sia se non accettata. TUTTI
- Linguaggi ricorsivi enumerabili: esiste una TM che si arresta se la stringa è accettata.
- Problema decidibile: esiste una TM che si arresta sempre. → Ricorsivi

Arresto di una TM:

La macchina muore in tutti i casi in cui trova un simbolo che non può gestire nello stato corrente ossia si arresta se nello stato q guardando simbolo di nastro X non è definita la transizione pari a $\delta(q, X)$.

Invece se una TM accetta una stringa assumiamo che si arresti mentre se non accetta non possiamo fare in modo che si arresti.

Macchina di Turing simulata sul web: <http://morphett.info/turing/turing.html> **PER USARLO:**
0 x1 x2 L 1 → $\delta(q_0, x_1) = (q_1, x_2, L)$ e con _ indico BLANK alla fine del nastro

Tecniche di programmazione per le TM:

- può essere utilizzata per calcolare come un computer tradizionale, ha però la capacità introspettiva di svolgere elaborazioni su altre macchine di Turing.

Introdurremo delle tracce multiple con memoria nello stato: ogni testina si muove indipendentemente.

$\delta([q, A, B, C], [X, Y, Z]) = (\dots, D)$ ossia stato q con memoria A, B, C e traccia 1 in X , tr.2 in Y e tr.3 in Z

Definendo quindi delle subroutine:

- una subroutine di una TM è un insieme di stati che eseguono una particolare procedura
- definiamo pure in questi stati lo stato iniziale come sempre e uno stampo temporaneamente senza mosse come "ritorno"

Una TM multinastro è solamente un caso particolare di quelle multinastro, inoltre ogni linguaggio accettato da una TM multinastro è ricorsivamente enumerabile, riassumendo i due tipi sono equivalenti tramite operazioni di trasformazione.

NTM macchina di Turing non deterministica: si distingue dalle TM base per la funzione di transizione δ associando a ogni stato q e simbolo X un insieme **finito** di triple.

$\delta(q, X) = \{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \dots, (q_k, Y_k, D_k)\}$

Il **linguaggio di una NTM** M è così definito:

- M accetta un input w se c'è una sequenza di scelte che parte dalla ID iniziale con input w a una ID con stato accettante inoltre

Restrizioni:

- TM con nastri semi-infiniti
- TM multi-stack (con due pile al posto del nastro)
- TM a contatori

Due DPDA porta ad accettare le stesse combinazioni di una TM.

TM multinastro: una TM $_k$ con k nastri è rappresentabile con una TM mononastro N a tracce multiple con un totale di $2 \cdot k$ tracce.

Computer == macchina di Turing siccome differenza di tempo tra essi è polinomiale, TM simula n passi di un computer in $O(n^3)$.

Linguaggio ricorsivamente enumerabile tale che M si arresta se accetta L è ricorsivamente enumerabile se $L = L(M)$ per una TM M , M si ferma se accetta una stringa.

Classi di linguaggi

- ricorsivi = decidibili = M si arresta sempre, se L è ricorsivo pure $!L$ lo è.
- ricorsivamente enumerabili = se L e $!L$ sono ricorsivamente enumerabili, allora L è ricorsivo provato dal fatto che se M_1 accetta L e M_2 accetta $!L$ le due testine si muovono insieme accettando o rifiutando negli stessi casi.
- non ricorsivamente enumerabili come L_d !

Enumerazione delle stringhe binarie: ogni stringa binaria rappresentata con $1w$ a partire da 1 con w contenente il resto della stringa da vuoto a infinito di 0 e 1 $\rightarrow 111=1w7 \quad w7=11$

Per rappresentare TM come numero binario avremo qr stati con q_1 iniziale e q_2 finale, separatori transizioni sono 11 e separatore da parola w è 111.

Data w_i i -esima stringa codifica data TM M , M è la i -esima TM denotata con M_i . Se w_i è codice non valido come 11001 allora diciamo che M_i è la TM che si arresta subito per qualunque input un solo stato nessuna transizione quindi $L(M_i) = \text{vuoto}$.

Linguaggio di Diagonalizzazione: L_d è l'insieme delle stringhe w_i tali che w_i non è accettato da $L(M_i)$ e per verificarlo abbiamo quindi due casi possibili.

Primo TH. parte TM: L_d non è un linguaggio ricorsivamente enumerabile.

- se w_i è in L_d allora la M_i macchina di turing accetta w_i ma non è possibile e lo stesso nel caso w_i non sia in L_d

Quindi entrambi i casi sono in contraddizione e per assurdo l'ipotesi iniziale non è verificata e L_d non è un problema decidibile dalla macchina di turing in termini generali.

Concetto di Indecidibilità **CAPITOLO 9 e 10**

Problemi per cui non esiste alcun programma che li possa risolvere, siccome inoltre esistono infinitamente più linguaggi che programmi allora devono esistere problemi indecidibili - Godel 1931.

Un linguaggio L è ricorsivamente enumerabile RE se $L = L(M)$ per una macchina turing M .

Due classi di linguaggi RE accettati da TM:

- linguaggi L in cui TM riconosce linguaggio e segnala pure se w non è in L
- linguaggi L che non sono accettati da alcuna TM che garantisca che si arrestino

Linguaggi ricorsivi: L è $_$ se $L=L(M)$ per una TM che appartiene a prima classe linguaggi RE quindi se che w è in L allora M accetta e se non lo è lo “segnala” entrando in uno stato accettante. Questo tipo corrisponde un funzionamento algoritmico ossia sequenza ben definita di passi che termina sempre dando una risposta.

Se L è considerato come problema è DECIDIBILE se è ricorsivo altrimenti è detto INDECIDIBILE se si tratta di un linguaggio non ricorsivo.

Th.9.3: se L è linguaggio ricorsivo allora lo è anche il suo complementare L^c

D: modifichiamo M in M_c rendendo stati accettanti M non accettanti e ha nuovo stato finale accettante nel quale arrivano tutte le transizioni possibili dagli stati non accettanti di M .

Th.9.4: se L e L^c sono linguaggi RE allora L è ricorsivo.

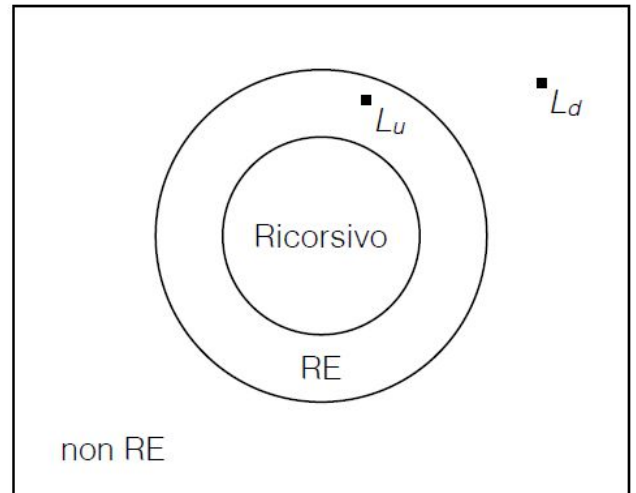
D: se M_1 accetta L e M_2 accetta L^c e M ottenuta unendo multinastro M_1 e M_2 avremo M che si arresta su tutti gli input e $L(M)=L$ e quindi possiamo concludere che L è ricorsivo. Se L è RE ma non ricorsivo allora L^c non è RE, vero pure il contrario.

Linguaggio universale: L_u è insieme delle stringhe binarie che codificano una coppia (M,w) dove w appartiene a $L(M)$ e esiste una TM universale tale che $L_u=L(U)$ dove U è una TM multinastro, 1 per codice di M e input w , 1 per nastro simulato di M , 1 per codifica stato di M quindi U simula M su w e accetta (M,w) se e solo se M accetta w .

Th. L_u è RE ma non ricorsivo, ricorriamo a Th.9.3

Problema dell'arresto: data una TM M definiamo con $H(M)$ insieme stringhe w tale che M si arresta con input w anche se M non accetta, linguaggio arresto con coppie (M,w) tali che w è in $H(M)$ ma non è ricorsivo come L_u quindi non esiste un algoritmo che possa dire se un dato programma termina o no ma esiste uno che dice che se programma in input termina si ferma tutto oppure non si arresta.

Riduzione: vorremmo ora dato un problema P_1 indecidibile vedere se un nuovo problema P_2 è indecidibile avendo un convertitore di input da w per P_1 la converta per P_2 quindi partendo da algoritmo P_2 abbiamo ottenuto con P_2 algoritmo che risolve P_1 potendo quindi dire che
Th.9.7: se esiste riduzione da P_1 a P_2 allora se P_1 è indecidibile lo sarà anche P_2 mentre se P_1 non lo è, non lo sarà neppure P_2 .



Macchine di Turing linguaggio vuoto? Iniziamo con Le linguaggio formato da codici delle TM che hanno linguaggio vuoto mentre Lne è linguaggio di tutte quelle TM che accettano almeno una stringa di input.

Th.9.8: Lne è ricorsivamente enumerabile, Turing ad ogni interazione genera congetture rispetto alle possibili vie d'uscita,

Th.9.9: Lne non è ricorsivo dimostrabile riducendo Lu a Lne e essendo Lu non ricorsivo e non potrà esserlo neanche Lne.

Th.9.10: Le non è ricorsivamente enumerabile e siccome Lne non è ricorsivo e Le è il complemento di Lne allora Le non può essere RE, se lo fosse sarebbero entrambi ricorsivi.

Proprietà dei linguaggi RE: tutte proprietà non banali TM sono indecidibili infatti è impossibile riconoscere con una TM stringhe che rappresentano codici di una TM di cui soddisfa le proprietà.

Formalmente una proprietà dei linguaggi RE è solamente un insieme di linguaggi RE come proprietà liberi da contesto è l'insieme di tutti i CFL.

Ricordiamo che una proprietà è **banale** se viene soddisfatto da tutti i linguaggi RE oppure da nessuno altrimenti è non banale.

Th. di Rice: ogni proprietà non banale nei linguaggi RE è indecidibile.

I problemi indecidibili sono:

- tutti quelli che riguardano il linguaggio accettato dalla TM

I problemi deducibili sono:

- tutti quelli riguardanti gli stati di una TM

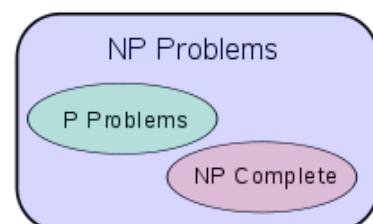
Esempio "Problema di corrispondenza di Post" indecidibile: liste di stringhe A e B voglio avere una sequenza i con indici per A e B tali che w_i di A = x_i di B

Problemi intrattabili e Classi P e NP

I problemi decidibili ossia ricorsivi sono detti trattabili se si può provare che sono risolvibili in tempo polinomiale in modo deterministico, gli altri sono detti invece intrattabili.

Tempo esponenziale intendiamo un qualunque tempo di esecuzione più grande di un qualsiasi polinomio.

I problemi che ora affronteremo sono tutti intrattabili che si fondano su ipotesi non dimostrata ovvero **P ≠ NP**.



Una TM M ha complessità in tempo $T(n)$ dato in input w di lunghezza n, M si ferma al massimo dopo $T(n)$ passi. Un linguaggio L è nella **Classe P** se esiste un polinomio $T(n)$ tale che $L=L(M)$ per una TM deterministica M con complessità di tempo $T(n)$

Esempio di problema P è trovare albero di copertura peso minimo in grafo, risolto da **Algoritmo di Kruskal**: soluzione greedy al problema albero copertura minima, simile a google maps dove ogni passo considero arco di peso minimo finchè tutti archi sono esaminati.

Un linguaggio L è nella **Classe NP**-Non Deterministica Polinomiale se esiste una TM non deterministica M tale che $L=L(M)$ e m ha input lungo n al massimo fa $T(n)$ mosse con $T(n)$ polinomiale. Inoltre $P \subseteq NP$ quindi una TM NP può esaminare numero esponenziale di strade in parallelo e quindi sembra che sia impossibile che $P=NP$ ma nessuno l'ha mai provato.

Problema di questo tipo ossia TSP è il Problema del commesso viaggiatore TSP: dato un grafo analogo a quello del problema di ricerca albero copertura minima dire se grafo ha circuito hamiltoniano di peso totale non superiore a W.

Problemi NP-completi: L è _ se L è in NP e per ogni altro linguaggio L' in NP esiste una riduzione polinomiale di L' a L come il TSP stesso. Nota che questi _ sono i più difficili problemi possibili di quelli in NP.

Th.10.4: se P1 è NP-completo e P2 è in NP posso ridurre polinomialmente P1 a P2 allora P2 è NP-completo.

Th.10.5: se un problema NP-completo è in P, allora $P=NP$

Problemi NP-ardui: un problema è _ se ogni problema in NP è riducibile polinomialmente a lui ma non è detto che sia in NP e quindi potrebbe anche non avere un algoritmo non-deterministico polinomiale per questa prima ipotesi.

Un problema NP-completo è dedurre se una espressione booleana è soddisfacibile, le quali espressioni sono costruite a partire da valori 0 o 1 e operatori binari e dal NOT oltre che dalle parentesi.

Soddisfacibilità booleana ossia SAT: data un'espressione booleana questa è soddisfacibile se le variabili possono essere assegnate in modo che la formula assuma il valore di verità vero.

Rappresentazioni istanze SAT → rappresentazione con alfabeto finito per rappresentare espressioni booleane, codificato come segue:

- variabile xi rappresenta simbolo x seguito dalla rappresentazione binaria indice i

Th. di Cook 10.9: SAT è NP-completo ossia il problema di determinare se una qualunque formula proposizionale è soddisfacibile è NP-completo.

Si vorrebbe dimostrare l'NP-completezza di un'ampia gamma di problemi procedendo per riduzione polinomiale da SAT che risulta molto complicato venendoci in aiuto un importante problema intermedio ossia 3SAT.

3SAT è sempre un SAT ma richiede che le espressioni siano di una forma precisa formate da congiunzione logica di clausole ognuna delle quali è disgiunzione logica di tre variabili anche negate.

Un'espressione booleana si dice in forma normale congiuntiva **CNF** se è la congiunzione logica AND di una o più clausole che sono formate dalla disgiunzione logica OR di uno o più letterali che sono variabili o variabili negate.

es. $(x \vee y) \wedge (\neg x \vee z)$, e $x \wedge y$ sono in CNF

es. $(x \vee y \vee z) \wedge (\neg y \vee \neg z) \wedge (x \vee y \wedge z)$ non è in CNF

Espressione in forma normale k-congiuntiva (k-CNF) se è il prodotto di clausole che hanno k letterali distinti es. $(x \vee \neg y) \wedge (y \vee \neg z) \wedge (z \vee \neg x)$ è in 2-CNF.

Riduzioni di questi ultimi: Possiamo convertire espressioni booleane generiche in CNF in tempo polinomiale anche CSAT è NP-completo. Possiamo ridurre CSAT a 3SAT in tempo lineare quindi anche 3SAT è NP-completo.

Th 10.12: per ogni espressione booleana E esiste un'espressione equivalente F in cui le negazioni compaiono solo nei letterali.

Th 10.15: 3SAT è NP-completo

Conversione da booleana a booleana:

$$E1 = \neg((\neg(x \vee y)) \wedge (\neg x \vee y))$$

2CNF o 3CNF

$$E2 = (x \vee \neg y) \wedge (y \vee \neg z) \text{ è 2CNF}$$

$$E3 = (z \vee x \vee \neg y) \wedge (y \vee \neg z \vee \neg k) \text{ è 3CNF, ogni elemento ha 3 letterali}$$

Passi della conversione espressione:

1. portare negazioni vicino alle variabili booleane
 - a. usando le leggi di De Morgan $\neg(A \vee B) \equiv \neg A \wedge \neg B$ e $\neg(A \wedge B) \equiv \neg A \vee \neg B$
 - b. regola della doppia negazione $\neg\neg A \equiv A$
2. usare le regole di distribuzione per ottenere la formula in CNF
 - a. $A \vee (B1 \wedge \dots \wedge Bn) \equiv (A \vee B1) \wedge \dots \wedge (A \vee Bn)$
 - b. $(B1 \wedge \dots \wedge Bn) \vee A \equiv (B1 \vee A) \wedge \dots \wedge (Bn \vee A)$

Trasformazioni:

- $(A+B)$ OR
 - $(A+B+C)(A+B+\neg C)$

Trasformazioni Tseytin:

- $C = A \wedge B$
 - $(\neg A \vee \neg B \vee C) \wedge (A \vee \neg C) \wedge (B \vee \neg C)$
- $C = A \vee B$
 - $(A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg B \vee C)$
- $C = \neg A$
 - $(\neg A \vee \neg C) \wedge (A \vee C)$

CSAT ossia espressione booleana CNF pure NP-completo. k-SAT ossia espressione booleana k-CNF. **Attenzione** per $k \geq 3$ **kSAT** sono NP-completi mentre per **1SAT e 2SAT** esistono algoritmi in tempo lineare.

Dati due numeri m e n = $\max(m-n, 0)$

$m=4$ $n=2 \rightarrow$ output=2 ma la macchina leggerà sequenza di zero lungo m poi un uno rappresentante il meno e poi altri n zeri, in output stamperà il numero di zeri rimasti.

Quindi la TM ha 6 stati:

- q0 stato iniziale se trovo zero metto un BLANK se trovo 1 vado in q5
- q1 cicla gli zeri andando verso destra se trova uno andiamo nella seconda zona
- q2 se trova 0 lo trasforma in 1 e poi torna indietro su q1 se trova uno va avanti finché trova zero stando sempre su q2
- q3
- q4 torno
- q5
- q6 stato finale

Esercizio TM che genera L con stringhe con stesso numero di 0 e 1 sul quaderno.

Scrivere le ID della TM per i seguenti nastri in input: 0010, 010, 0100 e 000010

1. $q_0, 0010 \vdash Bq_1, 010 \vdash B0q_1, 101 \vdash B01q_2, 0 \vdash B0q_3, 11 \vdash Bq_3, 011 \vdash \mathbf{q_3, B011} \vdash Bq_0, 011$
(Attenzione che effettuo il passaggio in grassetto prima di tornare in q_0 dovendo spostare q_3 rispetto allo 0 che lo precede ma poi lo precede B) $\vdash BBq_1, 11 \vdash BB1q_2, 1 \vdash BB11q_2 B \vdash BB1q_4, 1$
2. $q_0, 10 \vdash Bq_5, 0 \vdash BBq_5 B$
3. $q_0, 0100 \vdash Bq_1, 100 \vdash B1q_2, 00 \vdash Bq_3, 110 \vdash q_3, B110 \vdash Bq_0, 110 \vdash BBq_5, 10 \vdash BBBq_5, 0 \vdash BBBBq_5, B \vdash BBBBBq_6 B$
4. 000010

Esercizi possibile compito:

- NO DIMOSTRAZIONI
- NUOVA MACCHINA DI TURING
- STATI COB
- PCP
- BOOLEANA A BOOLEANA

Note esercizi parte tre:

- su TM faccio esempi base con input e output partendo da casi limiti semplice come 0, 1 e da li rendo risolvibile con step determinati anche ripartendo da stato antecedente senza definire stati inutili.
- PCP trovo la sottosequenza minima che rende possibile dire se ho soluzione

<http://www.cse.msstate.edu/~luke/Courses/fl04/CS8813/hw5.pdf>