

Linguaggi Formali

Linguaggi regolari e Automi a stati finiti

Un problema computazionale è una funzione che, dato un determinato input, restituisce un preciso output. Un problema decisionale è una funzione il cui output è nell'insieme $\{0, 1\}$ \leftarrow accetta o rifiuta, sì o no.

In questa parte del corso introduciamo il concetto di automa a stati finiti senza memoria, con il fine di trovare i limiti della computazione.

Un linguaggio è un insieme, finito o infinito, di parole finite, costruite su un alfabeto qualsiasi.

Per poter parlare di input e output dobbiamo decidere come descriverli: useremo un alfabeto Σ , finito, e cercheremo un modo di descrivere le parole che possiamo costruire.

Esistono, dato un alfabeto finito, infinite parole diverse che possiamo costruire. Il simbolo ϵ denota la parola vuota. Data una parola w , la parola w^{-1} rappresenta la sua inversa, e se w' è una sotto-parola iniziale (risp., finale) di w , scriviamo $w' \sqsubset w$ (risp. $w \sqsupset w'$).

L'input e l'output di un problema sono parole di un linguaggio.

Sui linguaggi possono essere definite delle operazioni, le operazioni regolari sono:

1. Composizione: dati L, L' il linguaggio composizione $L \circ L'$ è l'insieme $\{w | \exists s \in L, t \in L' (w = s \circ t)\}$
2. Unione: dati L, L' , il linguaggio unione L, L' è l'insieme $\{w | w \in L \text{ oppure } w \in L'\}$
3. Chiusura di Kleene: dato L , il linguaggio chiusura L^* è l'insieme $\{w = w_1 w_2 \dots w_n | \forall i (w_i \in L) \text{ e } n \in \mathbb{N}\} \cup \{\epsilon\}$

Esempio:

Se $\Sigma = \{a, b\}$, $L = \{aab\}$ e $L' = \{bbba\}$, allora:

- $L \circ L' = \{aabbba\}$
- $L \cup L' = \{aab, bbba\}$
- L^* contiene infinite parole, tra cui $\epsilon, aab, aabaabbba, bba, \dots$

Spesso la notazione insiemistica per i linguaggi regolari è sostituita da una notazione più leggera.

Diciamo che r è una espressione regolare su Σ se:

1. $r = a$ con $a \in \Sigma$, o $r = \epsilon$
2. $r = r_1 r_2 \leftarrow$ composizione di espressioni regolari
3. $r = r_1 + r_2 \leftarrow$ unione di espressioni regolari
4. $r = r_1^* \leftarrow$ chiusura di espressioni regolari
5. $r = r_1^+ \leftarrow r^+ = rr^* \text{ e } L(r)$ è il linguaggio generato da r

Un linguaggio regolare è un qualsiasi linguaggio ottenuto partendo da un alfabeto Σ attraverso unicamente operazioni regolari. Denotiamo con REG l'insieme di tutti i linguaggi regolari.

Se FIN è l'insieme di tutti i linguaggi finiti, e REG l'insieme di tutti i linguaggi regolari, allora $\text{FIN} \subset \text{REG} \subseteq P(\Sigma^*)$.

Dato un linguaggio regolare ci poniamo il problema di riconoscerlo \leftarrow Essenza della computabilità

Automi deterministici senza memoria a stati finiti:

Un automa a stati finiti deterministico (*DFA*) è una tupla $A = (Q, \Sigma, \delta, q_0, F)$
dove Q è l'insieme degli stati, Σ è l'alfabeto (dell'input), δ è una funzione di transizione tale che $\delta : Q \times \Sigma \rightarrow Q$, $q_0 \in Q$ è lo stato iniziale, e $F \subseteq Q$ è l'insieme degli stati finali.

In generale, abuseremo questa notazione scrivendo *DFA* anche per indicare l'insieme di tutti e soli i linguaggi riconoscibili da un automa a stati finiti deterministico.

Un automa A legge un nastro di sola lettura, e muove la testina di lettura ad ogni cambio di stato.

I caratteri letti non possono essere riletati in seguito.

La computazione di ogni parola di entrata termina precisamente quando l'ultimo carattere è stato letto.

Una parola è accettata se, e solo se, al momento in cui la computazione termina, l'automa si trova su uno degli stati finali F . Ogni cambio di stato viene guidato da δ , che va letta così: se mi trovo in un certo stato q , e viene letto il carattere a , allora cambio lo stato a $\delta(q, a)$. Se $\delta(q, a)$ non esiste, la parola si considera rifiutata e la computazione terminata, indipendentemente dallo stato in cui ci troviamo.

Una configurazione è un elemento di $Q \times \Sigma^*$, che rappresenta lo stato corrente e la parte non ancora letta della parola di entrata.

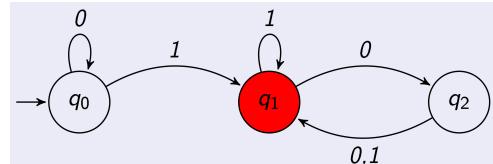
Una computazione è quindi una sequenza finita di configurazioni. Usiamo la notazione $(q, w) \rightsquigarrow (q', w')$ per indicare che la configurazione (q, w) porta in un passo alla configurazione (q', w') . Un automa A riconosce un preciso linguaggio, dato da tutte e sole le parole accettate da A .

Denotiamo con $L(A)$ il linguaggio di A .

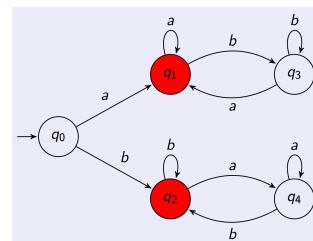
Un automa *DFA* è un algoritmo - tra l'altro un algoritmo particolarmente semplice.

Gli automi a stati finiti sono normalmente descritti attraverso grafi diretti, dove gli stati sono i vertici e gli archi rappresentano δ .

Costruiamo un *DFA* che riconosce il linguaggio di tutte e sole le parole di $\Sigma = 0, 1$ che contengono almeno un 1, e, dopo l'ultimo 1, nessuno o un numero pari di 0



Costruiamo un *DFA* che accetta tutte e sole le parole dell'alfabeto $\Sigma = a, b$ che iniziano e finiscono con lo stesso simbolo.



Gli automi a stati finiti deterministici risolvono una certa classe di problemi in modo imperativo.

Passiamo ora a un modello dichiarativo:

Automi non deterministici senza memoria a stati finiti:

Un automa a stati finiti non-deterministico (*NFA*) è una tupla $A = (Q, \Sigma, \delta, q_0, F)$
dove Q è l'insieme degli stati, Σ è l'alfabeto, δ è una relazione di transizione tale che: $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$,
 $q_0 \in Q$ è lo stato iniziale e $F \subseteq Q$ è l'insieme degli stati finali.

Un *NFA* è un *DFA* dove la funzione di transizione è stata sostituita da una relazione, permettendo cambi di stato diversi in presenza dello stesso carattere letto, e, a differenza del caso deterministico, un *NFA* accetta una stringa w se esiste una computazione di w che termina in uno stato finale.

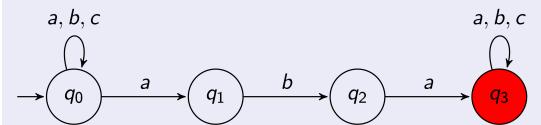
Si permettono cambi di stato senza necessariamente leggere nulla dal nastro (ϵ); queste transizioni si chiamano ϵ -transizioni.

Costruiamo un NFA che accetta tutte e sole le parole

dell'alfabeto

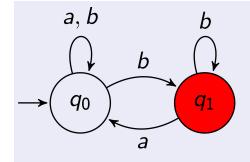
$\Sigma = \{a, b, c\}$ che contengono la parola aba .

Se fosse un DFA in q_1 e q_2 dovrei avere delle altre opzioni di lettura.



Costruiamo un NFA che accetta tutte e sole le parole dell'alfabeto $\Sigma = \{a, b\}$ corrispondenti all'espressione regolare $(a + b)^*(b)^+$.

Se fosse un DFA in q_0 scrivere b non mi potrebbe dare 2 diversi percorsi.



Un NFA risolve problemi simili a un DFA, ma in maniera più semplice.

L'insieme dei linguaggi riconoscibili da automi a stati finiti non-deterministici è lo stesso dei linguaggi riconoscibili da automi a stati finiti deterministici, cioè: $DFA = NFA$

Dimostrazione:

$DFA \subseteq NFA$: banale, perché ogni automa deterministico è un automa non deterministico, che non usa il determinismo.

$NFA \subseteq DFA$: Prendiamo un automa non deterministico A e proponiamoci di costruire un automa deterministico $A' = \{Q', \Sigma', \delta', q'_0, F'\}$ che accetta esattamente le stesse parole di A . Dove l'insieme di stati Q' è un insieme di insiemi di stati: $Q' = P(Q') = \{R_1, R_2, \dots, R_N\}$ quindi avrà 2^N sottoinsiemi di stati. Sia $E(R)$ (chiusura di R) l'insieme di tutti gli stati raggiungibili con ϵ -transizioni. Due stati uniti da una ϵ -transizione sono in realtà lo stesso stato!

Sia $\delta'(R, a) = \{q \in Q | q \in E(\delta(q, a)), \forall q \in R\}$ – partendo dallo stato iniziale e volendo l'insieme di tutti gli stati cui posso arrivare con a $\delta'(q_0, a)$, vedo tutti gli stati della sua decorazione, vedo tutte le transizioni non deterministiche che uscivano con a, e faccio un'unione di tutti gli stati che arrivano. Sono stati finali, tutti gli R che contengono un oggetto finale.

A parità di linguaggio riconosciuto, un automa deterministico può essere esponenzialmente più grande di uno non-deterministico. Questo è infatti un caso in cui lo sbalzo esponenziale nel numero di stati avviene in maniera necessaria.

L'insieme dei linguaggi riconoscibili da automi a stati finiti (non-)deterministici è precisamente REG , cioè $NFA = REG$

Espressioni in automi

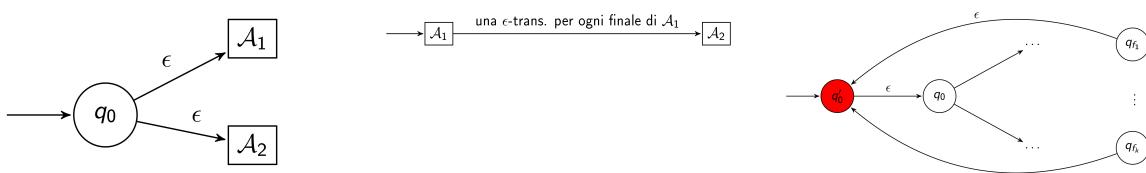
Le espressioni più semplici sono ϵ e a , dove $a \in \Sigma$. L'automa NFA che riconosce esattamente ϵ e quello che riconosce esattamente a sono immediati da costruire.

Siano r_1 e r_2 due espressioni regolari e A_1 e A_2 due automi tali che: $L(A_1) = r_1$ e $L(A_2) = r_2$

Unione: $r_1 + r_2$

Composizione $r_1 r_2$

Chiusura di Kleene:



Unione e composizione mantengono gli stati finali originali.

Automi in espressioni:

$NFA \rightarrow REG$: $R = \{i, j, k\}$ dove i : stato iniziale, j : stato finale e $k : 0, \dots, n$ e n : numero stati \leftarrow tabelle

$R(i, j, k) = R(i, j, k-1) \cup R(i, k, k-1) \circ R(k, k, k-1)^* \circ R(k, j, k-1)$

$R(i, j, k) = \{w | fanno passare da q_i a q_j usando al più stati numerati fino al $k - esimo\}$$

$REG = R(1, j, n)$ dove $q_j = q_f$

$DFA = NFA = REG$

REG è chiuso sotto unione, concatenazione, star di Kleene, intersezione, complemento e differenza insiemistica

Complemento: $L \in REG$. Esiste un automa A per cui $L(A) = L$. Se $A = (Q, \Sigma, \delta, q_0, F)$ allora esiste una automa $\bar{A} = (Q, \Sigma, \delta, q_0, Q \setminus F)$. Una parola è accettata da \bar{A} solo se è rifiutata da A . Esiste quindi $L(\bar{A}) = \bar{L}$, quindi \bar{L} è regolare.

Intersezione: $L_1 \cap L_2 = \overline{L_1 \cup \bar{L}_2}$

Differenza insiemistica: $L_1 \setminus L_2 = L_1 \cup \bar{L}_2$

Pumping Lemma

Se L è regolare, allora esiste $p \in \mathbb{N}$ tale che ogni parola $w \in L$, con $|w| \geq p$, può essere divisa in tre parti: $w = xyz$ tali che valgano le seguenti condizioni:

- a) $\forall i \geq 0, xy^i z \in L$
- b) $|y| > 0$
- c) $|xy| \leq p$

Siccome L è regolare, esiste un DFA $A = (Q, \Sigma, \delta, q_0, F)$ che lo riconosce. Prendiamo allora $p = |Q|$. Consideriamo una parola w tale che $w \in L, |w| \geq p$.

Durante il riconoscimento di w c'è sempre almeno uno stato che si ripete (dato che $|w| > Q$, sia q il primo stato a ripetersi).

Dividiamo quindi w così: x è la parte della parola riconosciuta prima di arrivare a q , y è la parte di parola che si riconosce tra due passaggi successivi per q , e z il resto. Poiché A è deterministico, non ci sono ϵ -transizioni, quindi $|y| > 0$. Siccome q è il primo stato che si ripete $|xy| \leq p$. Chiaramente, la parola xy^2z è riconosciuta, così come qualsiasi parola del tipo $xy^i z$.

Se w fosse regolare, allora esisterebbe una cardinalità p del suo automa a stati finiti e per ogni parola abbastanza lunga di p quella parola deve essere tale che il suo pezzo iniziale si deve poter aumentare, rimanendo all'interno del linguaggio.

Mostriamo che $\{a^n b^n\}$, con $\Sigma = \{a, b\}$, non è regolare:

Scegliamo un p qualsiasi, e una parola con due condizioni: dipende da p , ed è più lunga di p ; scegliamo $w = a^p b^p$.

Adesso, consideriamone ogni possibile divisione xyz per vedere che nessuna di queste rispetta la condizione xy^2z (esiste un i tale che le condizioni del lemma falliscono). Si noti che la scelta di y è limitata alla parte di soli a (condizione c del lemma); quindi y può essere solamente del tipo a^+ , mentre z può essere formato sia da a che da $b \rightarrow x = a^j \quad y = a^j \quad z = a^* b^p - xy^2z \notin L$ perché se ripeto la y due volte (anche prendendo $j = 1$) $|a| > |b|$

e $xy^2z = a^m b^p$ con $m > p$.

Linguaggi liberi dal contesto e automi con pila

I linguaggi liberi dal contesto estendono i linguaggi regolari.

Un linguaggio libero dal contesto (CFG) è una quadrupla $G = (V, \Sigma, R, S)$, dove V è un insieme di simboli non terminali, Σ è un alfabeto, R è un insieme di regole, e $S \in V$ è il simbolo iniziale.

Le grammatiche libere estendono in modo conservativo e proprio le espressioni regolari.

Scriviamo una grammatica G che genera il linguaggio $a^n b^n$. A questo fine, definiamo $G = (V = \{S\}, \Sigma = \{a, b\}, R, S)$, dove R è: $S ::= \epsilon | aSb$.

$S \rightarrow aSb \rightarrow aaSbb \rightarrow aaebb \rightarrow aabb$

Scriviamo una grammatica G che genera tutte e sole le espressioni aritmetiche con somma e moltiplicazione correttamente parentesizzate. A questo fine, definiamo $G = (V = \{S, E, T, F\}, \Sigma = \{id, *, +, (), (\)\}, R, S)$, dove R è:

$S ::= S + T \mid T \quad T ::= T * F \mid F \quad F ::= (S) \mid id$

CGF denota l'insieme di tutti i linguaggi liberi dal contesto. Gli automi progettati per riconoscere i linguaggi CGF sono automi a stati finiti a pila.

Un automa a pila (PDA) è una tupla $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$ dove Q è l'insieme degli stati, Σ l'alfabeto dell'input, Γ è l'alfabeto della pila, $q_0 \in Q$ è lo stato iniziale, $F \subseteq Q$ è l'insieme degli stati finale, e δ è una relazione di transizione tale che: $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma^* \times (Q \times \Gamma^*)$.

Come nel caso degli automi a stati finiti, abbiamo un input in un nastro di sola lettura, e omettiamo l'output, limitandoci ai problemi decisionali. Come si vede, questa macchina è naturalmente non deterministica. Se A è un PDA , allora $L(A)$ è il linguaggio riconosciuto da A . Con $\alpha, \beta, \gamma, \dots$ indichiamo sia elementi di Γ che di Γ^* . δ può essere letto come: se sulla pila c'è scritto questo, allora fai quello. Come sempre, PDA denota anche l'insieme di tutti e soli i linguaggi riconoscibili da un automa a pila.

Una transizione durante la computazione di una certa parola in un certo automa a pila A consiste nel leggere un simbolo della parola (o ϵ), leggere ed eliminare un certo numero (anche zero) di simboli della pila (POP), cambiare stato, e inserire un certo numero

(anche zero) di simboli sulla pila (PUSH). Pertanto una configurazione è un elemento di $Q \times \Sigma^* \times \Gamma^*$, che tiene conto dello stato attuale, della porzione di parola non ancora letta, e del contenuto della pila.

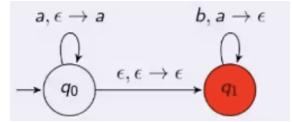
Usiamo la notazione $(q, w, \alpha) \rightsquigarrow (q', w', \beta)$ per indicare che la configurazione (q, w, α) porta in un passo alla configurazione (q', w', β) . A volte è conveniente assumere che $\Sigma \subseteq \Gamma$.

Costruiamo un *PDA* che riconosce il linguaggio $\{a^n b^n \mid n \geq 0\}$ con $\Sigma = \{a, b\}$ se e solo solo se a fine computazione la pila è vuota.

$a, \epsilon \rightarrow a$ significa: se leggi a , sostituisce ϵ con a sulla pila (ossia aggiungi a alla pila)

$\epsilon, \epsilon \rightarrow \epsilon$ significa: se leggi ϵ non fare cambi alla pila

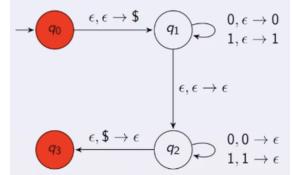
$b, a \rightarrow \epsilon$ significa: se leggi b , togli a dalla pila



$w = aaab \rightarrow 1 : \delta((q_0, a), q_0, [a]), 2 : \delta((q_0, a), q_0, [a, a]), 3 : \delta((q_0, a), q_0, [a, a, a]), 4 : \delta((q_0, \epsilon), q_1, [a, a, a]), 5 : \delta((q_1, b), q_1, [a, a])$ fine. Nonostante ci si trovi in uno stato finale, la pila non è vuota, quindi la parola non è accettata.

Un *PDA* è una generalizzazione di un *NFA*, in quanto semplicemente si aggiungono i comandi di POP e PUSH, e di conseguenza anche generalizzazione di *DFA*.

Costruiamo un *PDA* che riconosce $\{ww^{-1} \mid w \in \{0, 1\}^*\} \leftarrow$ palindromi, con $\Sigma = \{0, 1\}$.



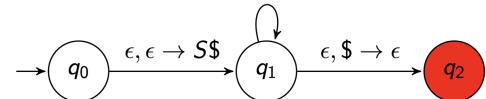
$\Gamma = \{0, 1, \$\}$ dove $\$$ denota la pila vuota, grazie a questo possiamo dire che si raggiunge lo stato finale q_3 solo se la pila è vuota.

L'insieme dei linguaggi riconoscibili da un *PDA* è esattamente *CFG*, cioè: $CFG = PDA$

Grammatiche in automi:

Sia $G = (V, \Sigma, R, S)$ una grammatica libera. Vogliamo costruire un *PDA* A tale che $L(G) = L(A)$. L'automa che costruiamo avrà precisamente tre stati, e, in effetti, rimarrà "quasi" sempre nello stesso stato durante una computazione. Inoltre, userà $V \cup \Sigma$ come alfabeto di pila Γ , ossia tutti i simboli di Σ e tutti simboli non terminali. Abbiamo quindi $A = (\{q_0, q_1, q_2\}, \Sigma, V \cup \Sigma, q_0, \{q_2\})$, definito come →

$\epsilon, A \rightarrow \alpha$ (per ogni $A \rightarrow \alpha \in R$)
 $a, a \rightarrow \epsilon$ (per ogni $a \in \Sigma$)



$S : S \in V$ è il simbolo non terminale iniziale, $A \rightarrow \alpha$: sostituisco il simbolo non terminale con la stringa terminale che descrive.

Automi in grammatiche:

Consideriamo un *PDA* $A = (Q, \Sigma, \Gamma, \delta, q_0, F)$. Per prima cosa, osserviamo che possiamo limitarci a un tipo particolare di automa, senza perdita di generalità, e, in concreto, possiamo assumere che A abbia un solo stato finale e che in ogni transizione si esegua precisamente un solo POP o un solo PUSH. Possiamo mostrare in maniera costruttiva che questi automi ristretti hanno precisamente lo stesso potere di quelli generali. Da un automa A ristretto, dobbiamo adesso mostrare come generare una grammatica libera G ; per prima cosa diciamo che la nostra grammatica avrà un simbolo non terminale $A_{p,q}$ per ogni coppia di stati p, q di A . L'idea è che $A_{p,q}$ generi tutte le parole che portano A dallo stato p (con stack vuoto) allo stato q (con stack vuoto). Quindi, le regole di G possono essere ricavate direttamente da δ :

- il simbolo iniziale S di partenza per G sarà A_{q_0, q_f}
- per ogni collezione p, q, r, s di stati in Q , simbolo $\gamma \in \Gamma$, $a, b \in \Sigma \cup \epsilon$, se $((p, a, \epsilon), (r, \gamma)) \in \delta$ e $((s, b, \gamma), (q, \epsilon)) \in \delta$, allora si aggiunge la regola $A_{p,q} ::= a A_{r,s} b$
- per ogni coppia di stati p, q in Q , se $((p, \epsilon, \epsilon), (q, \epsilon)) \in \delta$, si aggiunge la regola $A_{p,q} ::= \epsilon$
- per ogni tripla di stati p, q, r in Q , si aggiunge la regola $A_{p,q} ::= A_{p,r} A_{r,q}$
- per ogni stato p in Q , si aggiunge la regola $A_{p,p} ::= \epsilon$ in G

Per il riconoscimento di una parola, lo stack deve essere vuoto sia all'inizio che alla fine; per questo durante il riconoscimento, alla lettura di un simbolo, questo viene inserito a terminale, ma prima o poi dovrà essere tolto

L'insieme dei linguaggi liberi dal contesto include strettamente quello dei linguaggi regolari, cioè: $REG = DFA = NFA \subset CFG = PDA$.

Infatti l'unione corrisponde ad avere più opzioni nella stringa non terminale ($A|B$), la concatenazione (AB) e la star di Kleene ($A ::= 0A1$).

Per costruzione $NFA \subseteq PDA$, perchè un PDA è una generalizzazione di un NFA , quindi l'inclusione è stretta.

Ogni grammatica libera dal contesto G può essere trasformata in una grammatica libera dal contesto G' , con $L(G) = L(G')$, tale che ogni regola è della forma $S ::= \epsilon$, $A ::= BC$ oppure $A ::= a$, dove A, B, C sono non terminali e dove a è terminale. Questa è detta forma di Chomsky di una grammatica libera dal contesto.

Una volta riscritte le regole contenenti disgiunzioni (ad esempio $A ::= B|C$ riscritta come $A ::= B, A ::= C$) ogni regola generica presenta al membro destro una concatenazione di un certo numero di simboli terminali e non terminali. Se la regola non è in forma di Chomsky, è sempre riscrivibile come concatenazione di due nuovi simboli non terminali, a cui corrispondono due nuove regole da introdurre. ϵ può essere solo il risultato del simbolo iniziale.

Sia a un generico simbolo terminale, T un generico simbolo non terminale, e γ_i un generico simbolo (terminale o non terminale). Data una regola qualsiasi, eventualmente non in forma di Chomsky, possiamo produrre il suo corrispondente insieme di regole in forma di Chomsky ($ch(R)$) in questo modo:

Una regola può contenere o una lettera terminale o due lettere non terminali. Solo il simbolo iniziale può contenere una ϵ produzione.

$$\begin{cases} \{R\} & \text{se } A ::= \epsilon, A ::= a, A ::= T_1 T_2 \\ \{A ::= A_1 A_2, A_1 ::= a\} \cup Ch(A_2 ::= \gamma_1 \dots \gamma_n) & \text{se } R \in A ::= a \gamma_1 \dots \gamma_n, n > 0 \\ \{A ::= TA_2\} \cup Ch(A_2 ::= \gamma_1 \dots \gamma_n) & \text{se } R \in A ::= T \gamma_1 \dots \gamma_n, n \geq 0 \end{cases}$$

→ per qualsiasi grammatica G , $G' = \bigcup_{R \in G} Ch(R)$, dove G' è la grammatica equivalente in forma normale.

Grazie alla forma normale di Chomsky, il Pumping Lemma che conosciamo per i linguaggi regolari si può adesso estendere e generalizzare per il caso dei linguaggi libri dal contesto.

Se L è libero dal contesto, allora esiste $p \in N$ tale che ogni parola $w \in L$, con $|w| \geq p$, può essere divisa in cinque parti $w = uvxyz$ tali che le seguenti condizioni valgono:

- a) $\forall i \geq 0, uv^i xy^i z \in L$
- b) $|vy| > 0$
- c) $|vxy| \leq p$

Siccome L è un linguaggio libero dal contesto, allora esiste una grammatica libera che lo genera.

Possiamo assumere che G sia in forma normale di Chomsky, ossia ogni regola ha la forma $A ::= BC$ oppure $A ::= a$. Poniamo adesso $p = 2^{|V|+1}$, e consideriamo una parola w di lunghezza superiore o uguale a p .

Siccome G è in forma normale, la derivazione di w può essere vista come risultato dello sviluppo di un albero binario, la cui radice è S e le cui foglie sono elementi di Σ . La frontiera di questo albero è proprio w . Siccome $|w| \geq p$, allora l'altezza dell'albero è maggiore o uguale a $|V| + 1$. Almeno uno dei simboli di V , diciamo A , si ripete nei nodi.

Chiamiamo u la parte della frontiera da sinistra a destra che non si trova sotto la prima occorrenza di A , v la parte seguente che non si trova sotto la seconda occorrenza di A , x la parte centrale della frontiera, y la parte a destra corrispondente a v , e z l'ultima parte. Chiaramente il sotto-albero radicato nella prima occorrenza di A , escludendo il sotto-albero radicato nella seconda, può essere trapiantato e ripetuto un numero arbitrario, anche zero, di volte, ottenendo un nuovo albero di derivazione da S , e perciò una nuova parola di G . Questa è del tipo $uv^i xy^i z$, come volevamo, e, chiaramente $|vxy| \leq p$ e v e y non sono entrambe vuote.

CFG è chiuso sotto unione, concatenazione e star di Kleene. Inoltre, CFG non è chiuso per intersezione, complemento, e differenza insiemistica.

Siano G_1 e G_2 due grammatiche libere, tali che $G_1 = (V_1, \Sigma_1, R_1, S_1)$ e $G_2 = (V_2, \Sigma_2, R_2, S_2)$

Unione: $G = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S ::= S_1|S_2\}, S)$, chiaramente $L(G) = L(G_1) \cup L(G_2)$

Concatenazione: $G' = (V_1 \cup V_2 \cup \{S\}, \Sigma_1 \cup \Sigma_2, R_1 \cup R_2 \cup \{S ::= S_1 S_2\}, S)$ dove $L(G') = L(G_1) \cdot L(G_2)$

Kleene: $G'' = (V_1 \cup \{S\}, \Sigma_1, R_1 \cup \{S ::= \epsilon|SS_1\}, S)$ dove $L(G'') = G_1^*$

Intersezione: $L_1 = \{a^n b^n c^n | n \geq 0\}$ e $L_2 = \{a^* b^n c^n | n \geq 0\}$, $L_1, L_2 \in CFG$. Chiaramente $L_1 \cap L_2 = \{a^n b^n c^n | n \geq 0\} \notin CFG$.

Complemento: $L_1 \cap L_2 = \overline{L_1 \cup \bar{L}_2}$. CFG non è chiuso per intersezione e quindi neppure per complemento.

Differenza insiemistica: $\bar{L} = \Sigma^* \setminus L$. CFG non è chiuso per differenza insiemistica perché non è chiuso per complemento.

Macchine di Turing

Le macchine di Turing sono un modello non ulteriormente estendibile, che, in qualche maniera, ci permetterà di trovare i limiti del computabile.

Una macchina di Turing è una tupla $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r\}$

Q è l'insieme degli stati, $\Sigma \subseteq \Gamma$, Γ è l'alfabeto del nastro, che è infinito a destra e finito a sinistra e contiene un carattere \square (blank), δ è una funzione di transizione tale che $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$, q_0 è lo stato iniziale, q_a lo stato accettante, e q_r lo stato di rifiuto.

Il nastro sostituisce la pila ed è in modalità scrittura e lettura. Le operazioni della testina riflettono il loro nome: R muove la testina di lettura a destra, mentre L la muove a sinistra.

Una TM è molto simile a un automa, però ci sono delle sottili differenze che la rendono un dispositivo molto diverso. L'input di una TM si considera già presente sul nastro.

Tutte le posizioni non scritte inizialmente (in particolare, tutto quello che viene dopo l'input) è \square (blank), e la prima occorrenza di \square indica la fine dell'input stesso.

Quando la testina di lettura si trova sulla posizione 0, indipendentemente da ciò che dice la funzione di transizione, la testina non si può muovere a sinistra.

La computazione si ferma immediatamente se il controllo arriva a q_a oppure q_r , e procede indefinitamente altrimenti.

In una TM , una configurazione è una tripla wqv , in cui la testina si trova sul primo simbolo di v e lo stato è q .

Nella configurazione $001q100$ il nastro contiene 001100 e la macchina si trova nello stato q e sta leggendo il secondo 1.

Diciamo che $wqv \rightsquigarrow w'q'v'$ (cioè che wqv porta a $w'q'v'$) se e solo se esiste una transizione in δ tale che, trovandosi la macchina in q e leggendo proprio il primo simbolo di v , si passa allo stato q' e si modifica lo stato del nastro e della testina in accordo con w' e v .

Se $\delta(q_1, a) = (q_2, b, R)$ succederebbe che $bbaq_1ab \rightsquigarrow bbabq_2b$.

Chiamiamo Turing riconoscibile (ricorsivamente enumerabile) un linguaggio per il quale esiste una TM tale che accetta esattamente tutte le sue parole, mentre le altre possono essere rifiutate o fare entrare la TM in un loop infinito.

Chiamiamo decidibile un linguaggio per il quale esiste una TM che accetta tutte le sue parole e rifiuta tutte le parole che non sono nel linguaggio, e che quindi termina sempre.

Chiamiamo $R.E.$ l'insieme di tutti e soli i linguaggi Turing riconoscibili e DEC l'insieme di tutti e soli i linguaggi decidibili.

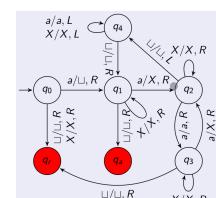
$DEC \subseteq R.E.$

Ogni linguaggio decidibile è anche Turing riconoscibile.

Descrivere una TM può essere fatto nella stessa maniera nella quale descriviamo gli automi a stati finiti o con pila. Questa tecnica porta però a descrizioni eccessivamente dettagliate. Alternativamente possiamo descrivere una TM con un semplice insieme di passi atomici.

Vediamo una TM che decide il linguaggio $\{a^{2^n} | n \in \mathbb{N}\}$ (\leftarrow quadrato perfetto) in forma algoritmica.

1. In un ciclo, scorri il nastro mettendo X su ogni seconda a.
2. Se il nastro contiene una sola a, accetta.
3. Se il nastro contiene più di una sola a, e in numero dispari, rifiuta.
4. Torna alla posizione iniziale del nastro;
5. Ripeti dal passo uno.



In forma di automa.

Costruiamo una TM per decidere il linguaggio $\{a^n b^n c^n | n \in \mathbb{N}\}$:

1. Controlla scorrendo il nastro di avere prima le a, poi le b, e poi le c, torna poi alla posizione iniziale del nastro;
2. Sostituisce al prima a, la prima b e la prima c con una X;
3. Se il nastro contiene solo X, accetta.
4. Se il nastro non contiene solo X, ma non ci sono a, b, o c, rifiuta.
5. Torna alla posizione iniziale del nastro (utilizzo left e blank);
6. Ricomincia da 2.

Costruiamo una TM per decidere il linguaggio $\{a^i b^j c^k \mid i < j < k, i, j, k \geq 1\}$

1. Scorri il nastro e controlla che l'input sia del tipo $a^+ b^+ c^+$ e torna all'inizio del nastro;
2. Sostituisce la prima a con X .
3. Sostituisce la prima b con B (o il contrario) e la prima c con X finché ci sono ancora b (o B). Se non ci sono c per una b (o B), rifiuta.
5. Se non ci sono più b (o B), ma ci sono ancora a e c , ritorna al punto 2.

Le estensioni della MT originale, le sono tutte equivalenti.

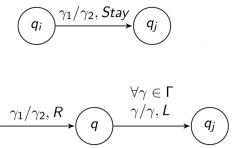
Una macchina di Turing con Stay è una tupla

$$M = \{Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r\}$$

Q è l'insieme degli stati, $\Sigma \subseteq \Gamma$, Γ è l'alfabeto del nastro, che è infinito a destra e finito a sinistra e contiene un carattere \sqcup (blank), δ è una funzione di transizione tale che $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L, Stay\}$, q_0 è lo stato iniziale, q_a lo stato accettante, e q_r lo stato di rifiuto.

In una Stay-TM la funzione di transizione è estesa con la possibilità di non muoversi invece di muovere il nastro a destra o a sinistra.

Per ogni Stay-TM esiste una TM standard che riconosce/decide precisamente lo stesso linguaggio.



La prima transizione è identica ma con lo spostamento a destra, la seconda dice: "qualunque cosa sia scritta sul nastro, ricopiala e spostati a sinistra".

Una macchina di Turing con k nastri ($k - TM$) è una tupla

$$M = \{Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r\}$$

Q è l'insieme degli stati, $\Sigma \subseteq \Gamma$, Γ è l'alfabeto dei nastri numerati da 1 a k , tutti infiniti a destra e finiti a sinistra, e contiene un carattere \sqcup , δ è una funzione di transizione tale che $\delta : Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{R, L\}^k$, q_0 è lo stato iniziale, q_a lo stato accettante, e q_r lo stato di rifiuto.

Una tipica transizione di una macchina con k nastri può essere descritta come:
 $\delta(q, a_1, \dots, a_k) = (q', b_1, \dots, b_k, M_1, \dots, M_k)$ dove $M_i \in \{R, L\}$ per ogni i .

Per ogni $k - TM$ M esiste una TM M' che riconosce/decide precisamente lo stesso linguaggio.

Posso simulare una $k - TM$ su una TM normale semplicemente aggiungendo all'alfabeto della seconda due simboli nuovi: $\$ \triangleleft$, e un nuovo simbolo (sottolineato) corrispondente ad ogni simbolo dell'alfabeto della $k - TM$, che indica che nella $k - TM$ la testina di lettura del k -esimo nastro si trova su quel carattere, rappresentando così k nastri in uno unico. La nuova macchina avrà $|Q| \cdot k$ nuovi stati.

Se la $k - TM$ opera in tempo $O(t)$ la TM corrispondente opererà in $O(t \cdot (|w| + (k \cdot t)))$.

Una macchina di Turing con nastro infinito a sinistra ($Left - TM$) è una tupla $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r\}$

Q è l'insieme degli stati, $\Sigma \subseteq \Gamma$, Γ è l'alfabeto del nastro, che è infinito sia a destra che a sinistra e contiene un carattere \sqcup , δ è una funzione di transizione tale che $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$, q_0 è lo stato iniziale, q_a lo stato accettante, e q_r lo stato di rifiuto.

Quindi, una Left-TM si definisce come una TM classica, ma tale che il nastro è isomorfo all'insieme Z invece che all'insieme N ; si assume che l'input si trovi sempre a partire dalla posizione 0.

Per ogni $Left - TM$ esiste una TM standard che riconosce/decide precisamente lo stesso linguaggio.

Questa macchina può essere facilmente simulata da una 2-TM, dove il primo nastro fa le veci della parte destra (rispetto allo 0) del nastro originale, e il secondo della parte sinistra.

Risulta naturale chiedersi se il modello della macchina di Turing può essere migliorato cambiando la gestione della memoria

Una Macchina a Accesso Casuale (*RAM*, Random Access Machine) è una coppia (p, Π) , dove p è un contatore di programma e Π un insieme finito di istruzioni.

Una *RAM* lavora su una memoria rappresentata come un array $[0, \dots, n]$, ogni posizione della quale contiene esattamente un numero intero non negativo. Inizialmente la memoria è tutta a zero, a meno dell'input. La macchina è dotata di un insieme finito di registri R_0, R_1, \dots sui quali si effettuano le operazioni. Una *RAM* esegue il programma dalla prima istruzione, cambiando il valore di p ad ogni passo, fino a quando un'istruzione halt viene eseguita.

Una *RAM* corrisponde in tutto e per tutto ad un programma in un linguaggio imperativo qualsiasi (es. assembler).

Somma di due numeri in $T[0]$ e $T[1]$

<i>loadc</i> 0	- $R_0 = 0$
<i>read</i> 0	- $R_0 = T[R_0]$
<i>store</i> 1	- $R_1 = R_0$
<i>loadc</i> 2	- $R_0 = 2$
<i>store</i> 2	- $R_2 = 2$

<i>loadc</i> 1	- $R_0 = 1$
<i>read</i> 0	- $R_0 = T[R_0]$
<i>add</i> 1	- $R_0 = R_0 + R_1$
<i>write</i> 2	- $T[R_2] = R_0$

Il risultato è ora in $T[2]$.

Le istruzioni della nostra RAM sono:

- *read j* ($R_0 = T[R_j]$)
- *write j* ($T[R_j] = R_0$)
- *store j* ($R_j = R_0$)
- *load j* ($R_0 = R_j$)
- *loadc c* ($R_0 = c$)
- *addc c* ($R_0 = R_0 + c$)
- *sub c* ($R_0 = max\{R_0 - R_j, 0\}$)
- *subc c* ($R_0 = max\{R_0 - c, 0\}$)
- *half* ($R_0 = \lfloor \frac{R_0}{2} \rfloor$)
- *jump s* ($p = s$)
- *jpos s* (if $R_0 > 0$ then $p = s$)
- *jzero s* (if $R_0 = 0$ then $p = s$)
- *halt*

Per ogni *RAM* esiste una *TM* che riconosce/decide precisamente lo stesso linguaggio, e viceversa.

Primo, cerchiamo una *RAM* M che sia equivalente a una *TM* M' data. Per ogni simbolo del nastro di M' possiamo rappresentare l'input sul nastro di M . Usiamo i registri di M per simulare il controllo: un registro mantiene la posizione della testina di lettura, e ogni stato di M' viene rappresentato con un intero; lo stato corrente è salvato in un altro registro. Quindi, ogni operazione di M' può essere immediatamente simulata.

In seconda istanza, cerchiamo una *TM* M'' che sia equivalente a una *RAM* M data. In questo caso usiamo una $k - TM$, con, almeno, un nastro per simulare la memoria di M , e un nastro per ogni registro di M . Nel nastro che simula la memoria si utilizza un simbolo (es. $\$$) per distinguere tra una posizione e la seguente. C'è una micro macchina di Turing per ogni operazione.

Una macchina di Turing non deterministica (*NTM*) è una tupla $M = \{Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r\}$

Q è l'insieme degli stati, $\Sigma \subseteq \Gamma$, Γ è l'alfabeto del nastro, che è in modalità lettura e scrittura, infinito a destra e finito a sinistra e contiene un carattere \sqcup , δ è una funzione di transizione tale che $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{R, L\}$, q_0 è lo stato iniziale, q_a lo stato accettante, e q_r lo stato di rifiuto.

Una *NTM* riconosce (o decide) un linguaggio se per ogni sua parola esiste una serie di passi non deterministici ottimi che portano all'accettazione.

Un altro modo di vedere una *NTM* è considerarla un verificatore: data una parola che, per ipotesi appartiene a un linguaggio, la macchina si limita a verificare che così è, in maniera deterministica.

$$L = \{n | n \in \mathbb{N}, n \text{ è non primo}\}$$

1. Scegli non deterministicamente $p, q \in \mathbb{N}$, $p, q \neq 1$
2. Controlla se $p \cdot q = n$, se si accetta
3. Rifiuta

Una *NTM* è una formalizzazione di un modello di computazione dichiarativo \rightarrow Non è naturale chiedersi come costruire una *NTM* per computare una funzione.

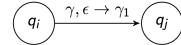
Per ogni *NTM* esiste una *TM* che riconosce/decide precisamente lo stesso linguaggio.

Una *NTM* è la sequenza di passi ottimi della *TM* corrispondente.

Per dimostrare questo, uso una $3 - TM$, con un nastro read-only per l'input, un nastro di lavoro e un terzo nastro.

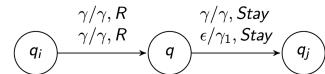
Per imitare una computazione non-deterministica, copio quello che c'è sull'input nel work tape. Nel terzo nastro scrivo, separate da un simbolo apposito, tutte le possibili computazioni dei passi (es. $[q_0 \$ q_0 q_1 \$ q_1 q_2 \$ \dots \$ q_0 q_1 q_3 \$ \dots]$) dopo di che seguo esattamente l'algoritmo, ma per ogni scelta non-deterministica, 'fai la prova' sul work tape.

$PDA \subset DEC$



Lo possiamo dimostrare utilizzando una $2 - TM$ non deterministica, con un nastro che simula la pila e un nastro per l'input.

L'inclusione è stretta in quanto $a^n b^n c^n \in DEC, a^n b^n c^n \notin PDA$



Tesi di Church-Turing: un linguaggio è calcolabile se e solo se è intuitivamente calcolabile, cioè se esiste una macchina di Turing che lo calcola → se qualcosa non può essere computato con la macchina di Turing, allora non può essere computato.

La macchina di Turing diventa la definizione di algoritmo.

DEC è chiuso sotto unione, concatenazione, star di Kleene, intersezione, complemento e differenza insiemistica.

Complemento: Se $L \in DEC$ e M_L , se costruisco una TM come $M_{\bar{L}}$ in cui però inverti rifiuto e accettazione, questa riconosce precisamente \bar{L} .

Unione: Se $L_1, L_2 \in DEC$ e $M_1, M_2 \in TM$, posso costruire una TM M' che riconosce esattamente $L_1 L_2$, facendo sì che si comporti inizialmente come M_1 e, se questa rifiuta, come M_2 , accettandone il risultato.

Intersezione: $L_1 \cap L_2 = \overline{(L_1 \cup L_2)}$ Differenza insiemistica: $L_1 \setminus L_2 = L_1 \cup \bar{L}_2$

Composizione: Se $L_1, L_2 \in DEC$ e $M_1, M_2 \in TM$, posso costruire una $2TM$ M' che opera sul primo nastro come M_1 fino all'accettazione e poi come M_2 .

Star di Kleene: Dato L e M_L , per riconoscere L^* costruisco una $2TM$ che divide non deterministicamente l'input in n parti e usa il secondo nastro per simularne ciascuna su M_L , accettando se tutte le simulazioni sono accettate.

Gerarchia di Chomsky: dalla più alla meno espressiva

0. Grammatiche di tipo 0, o illimitate, che ammettono regole di qualsiasi tipo, anche con diminuzione della lunghezza della stringa, e generano precisamente l'insieme R.E.
1. Grammatiche di tipo 1, o dipendenti dal contesto, che sono simili a quelle libere, ma ammettono che certe regole di derivazione siano applicabili solo 'qualche volta', dipendendo, appunto, dal resto della parola (contesto);
2. Grammatiche di tipo 2, o libere dal contesto, che generano i linguaggi liberi dal contesto (CFL)
3. Grammatiche di tipo 3, o regolari, che generano esattamente i linguaggi regolari (REG);

Calcolabilità

Focalizziamo la nostra attenzione su problemi di accettazione da parte di macchine semplici come quelle che abbiamo visto.

Definiamo ad esempio il seguente problema: $A_{DFA} = \{\langle A, w \rangle | A \text{ è un } DFA \text{ che accetta } w\}$.

Mostrare che A_{DFA} è un problema decidibile significa mostrare che esiste una TM che lo riconosce e che termina sempre.

Ogni automa sarà rappresentato da una stringa che prima elenca tutti gli stati e poi, dopo un carattere speciale, tutte le transizioni.

Per una macchina con tre stati q_1, q_2, q_3 con q_2 come unico stato finale, con $\Sigma = \{a, b\}$, e transizioni $(q_0, a) = q_1, (q_0, b) = q_0, (q_1, a) = q_2, (q_1, b) = q_0$, la stringa rappresentativa può essere: $q_0 q_1 q_2 \$ q_0 a q_1 q_0 b q_0 q_1 a q_2 q_1 b q_0 \$ input \sqcup \dots$

Una macchina di Turing che prende, come input, una stringa che rappresenta una macchina di Turing è una formalizzazione del concetto di compilatore.

Data una macchina qualsiasi A , denoteremo con $\langle A \rangle$ la sua codifica, mentre $\langle A, w \rangle$ denota la codifica di A e della parola w .

Per costruire $\langle A \rangle$ abbiamo usato un alfabeto arbitrariamente complesso, in realtà, data una stringa qualsiasi appartenente a Σ^* , possiamo sempre trovare una stringa w' che rappresenta la stessa informazione e appartiene a $\{0, 1\}^*$

$$A_{DFA} \in DEC$$

Costruiamo la seguente macchina di Turing deterministica che prende in input una stringa v e che risolve il problema:

1. Controlla se v è del tipo $\langle A, w \rangle$, altrimenti rifiuta
2. Decodifica v e simula A con input w
3. Se la simulazione ha accettato, accetta, altrimenti rifiuta.

$$A_{NFA} \in DEC$$

Costruiamo la seguente macchina di Turing deterministica che prende in input una stringa v e che risolve il problema:

1. Controlla se v è del tipo $\langle A, w \rangle$, altrimenti rifiuta
2. Decodifica v , converti A nella sua versione deterministica A' e simula A' con input w
3. Se la simulazione ha accettato, accetta, altrimenti rifiuta.

$$\begin{aligned} E_{DFA} &= \{\langle A \rangle \mid A \text{ è un } DFA \text{ tale che } L(A) = 0\}, \\ E_{DFA} &\in DEC \end{aligned}$$

La seguente TM , con input la stringa w lo dimostra in quanto per avere $L(A) = 0$ i suoi stati finali non devono essere raggiungibili.

1. Controlla se w è del tipo $\langle A \rangle$, altrimenti rifiuta
2. Marca lo stato iniziale q_0 di A
3. Ripeti finché non si marcano nuovi stati: se esiste una transizione tra uno stato marcato q e uno non marcato q' , marca q'
4. Se uno stato in F è stato marcato rifiuta, altrimenti accetta.

$$A_{CFG} \in DEC$$

Poiché è possibile dimostrare che se G è una CFG in forma normale di Chomsky (e posso trasformare G in forma di Chomsky con una macchina di Turing), allora se una parola w appartiene a $L(G)$ esiste sicuramente una derivazione di w in G di al più $2 \cdot n - 1$ passi, dove $n = |w|$

Una parola generata da una grammatica libera può essere rappresentata come la frontiera dell'albero di derivazione, e tutte le grammatiche di Chomsky danno luogo ad un albero che è, nel caso peggiore, un albero che è binario fino al penultimo livello e all'ultimo livello unario. Infatti, da un simbolo non terminale la produzione in simbolo terminale non genera un branch, ma una singola foglia, e pertanto gli ultimi due livelli avranno lo stesso numero di foglie. Per una parola di lunghezza $|w|$, l'albero di derivazione avrà quindi $|w|$ foglie e altezza massima $\lceil \log_2 |w| \rceil + 1$, dove il $+1$ è dovuto all'‘anomalia’ che l’ultimo livello

presenta rispetto ad un normale albero binario. Il numero di passi di derivazione, pari al numero di nodi interni dell'albero, è pari al numero di nodi in un albero binario di altezza $\lceil \log_2 |w| \rceil$, ovvero $2^h - 1 = 2^{\log_2 |w| + 1} - 1 = 2 \cdot 2^{\log_2 |w|} - 1 = 2 \cdot |w| - 1$.

allora possiamo anche costruire la seguente TM con input v :

1. Controlla se v è del tipo $\langle G, w \rangle$, altrimenti rifiuta
2. Decodifica v in forma normale di Chomsky
3. Elenca tutte le derivazioni possibili di lunghezza $2 \cdot n - 1$ e se w appare accetta, altrimenti rifiuta.

$A_{TM} \notin DEC$

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

Assumiamo per assurdo che $A_{TM} \in DEC$, pertante dovrebbe esistere una macchina A che decide A_{TM} :

$$A(\langle M, w \rangle) = \begin{cases} ACCETTA & \text{se } M \text{ accetta } w \\ RIFIUTA & \text{se } M \text{ rifiuta } w \end{cases} \text{ e } A \text{ terminerebbe sempre.}$$

Poichè A è una TM , allora può essere usata dentro a una TM per la tesi di Church-Turing, in quanto diventa un unico passo:

$$D(\langle M, w \rangle) = \begin{cases} RIFIUTA & \text{se } M \text{ accetta } \langle M \rangle \\ ACCETTA & \text{se } M \text{ rifiuta } \langle M \rangle \end{cases} \text{ in cui } D \text{ simula } A \text{ con entrata } \langle M, \langle M \rangle \rangle \text{ e se ha accettato, rifiuta}$$

Si ha che $D(\langle D \rangle)$ accetta se e solo se $A(\langle D, \langle D \rangle \rangle)$ rifiuta, cioè se e solo se $D(\langle D \rangle)$ rifiuta \rightarrow una delle due non può esistere.

$A_{TM} \in R.E.$

Costruiamo una macchina di Turing che riconosca A_{TM} . Chiamiamo U questa macchina che, con entrata v :

1. Controlla che v sia del tipo $\langle M, w \rangle$, altrimenti rifiuta
2. Decodifica v
3. Simula M con entrata w e se ha accettato, accetta
4. Se la simulazione ha rifiutato, rifiuta.

Questa macchina U non dimostra che $A_{TM} \in DEC$ perchè la macchina M potrebbe non terminare mai, e quindi, anche U . Tutte le volte che M termina, lo farà anche U , ma negli altri casi non si può dire cosa succederà.

Questa macchina U si chiama macchina di Turing universale.

Se $L, \bar{L} \in R.E.$, allora $L, \bar{L} \in DEC$

Siccome entrambi $L, \bar{L} \in R.E.$, allora esistono due TM M_L e $M_{\bar{L}}$ che riconoscono, rispettivamente, L e \bar{L} .

Costruiamo adesso una nuova macchina M' , con entrata w :

1. Simula, alternativamente, un passo di computazione di M_L ed uno di $M_{\bar{L}}$ con input w
2. Se la simulazione di M_L ha accettato, accetta
3. Se la simulazione di $M_{\bar{L}}$ ha accettato, rifiuta
4. Torna al passo 1

Poichè certamente una delle due simulazioni terminerà accettando, la macchina che abbiamo costruito è un decisore (per L). Allo stesso modo, possiamo ottenere un decisore per \bar{L} .

$\bar{A}_{TM} \notin R.E.$

Poichè $A_{TM} \in R.E.$ e $A_{TM} \notin DEC$ e poichè per il teorema precedente se $L, \bar{L} \in R.E. \rightarrow L, \bar{L} \in DEC$
Allora $\bar{A}_{TM} \notin R.E.$ perchè altrimenti $A_{TM} \in DEC$.

$R.E.$ è chiuso sotto unione, concatenazione, star di Kleene, intersezione. $R.E.$ non è chiuso sotto l'operazione di complemento, né quella di differenza insiemistica.

Abbiamo mostrato l'esistenza di problemi che non appartengono a $R.E.$ (e di conseguenza neanche a DEC)

Definiamo il linguaggio: $H_{TM} = \{\langle M, w \rangle \mid M \text{ è una } TM \text{ che, computando } w, \text{ si ferma}\}$

$$H_{TM} \in R.E. \setminus DEC \rightarrow H_{TM} \in R.E., \quad H_{TM} \notin DEC$$

La dimostrazione di $H_{TM} \in R.E.$ è immediata, in quanto basta simulare e se si ferma, si accetta.

Per dimostrare $H_{TM} \notin DEC$ dobbiamo ragionare per assurdo \rightarrow tentiamo di dimostrare che $H_{TM} \in DEC$:

Dovrei poter costruire una macchina $A(H_{TM}) \in DEC$ ma $A(H_{TM})$ altro non è che una A_{TM} che abbiamo dimostrato essere non decidibile.

Per alfabeti Σ_1, Σ_2 dati e linguaggi $L_1 \subseteq \Sigma_1^*, L_2 \subseteq \Sigma_2^*$, diremo che L_2 si riduce a L_1 se esiste una funzione ricorsiva $f : L_1 \rightarrow L_2$ tale che per ogni parola $w, w \in L_1$ se e solo se $f(w) \in L_2$.

Questa funzione è detta **many-one** e si denota con $L_1 \leq_m L_2$.

L'idea alla base della riduzione è che la decidibilità (e la Turing-riconoscibilità) di un linguaggio si **eredita** attraverso una riduzione.

Siano L_1, L_2 due linguaggi tali che $L_1 \leq_m L_2$. Se $L_2 \in DEC$, allora $L_1 \in DEC$, e se $L_2 \in R.E.$, allora $L_1 \in R.E.$

Poichè la funzione è ricorsiva, esiste una macchina di Turing M_f che la calcola, allo stesso modo poichè L_2 è ricorsivo, esiste una macchina di Turing M_2 che lo riconosce \rightarrow posso calcolare $f(w)$ usando M_f , e posso calcolare $M_2(f(w)) \leftarrow$ questa macchina decide L_1

L_2 si Turing-riduce a L_1 , denotato con $L_1 \leq_T L_2$, se e solo se esiste una macchina di Turing M che termina sempre e che posso usare dentro la macchina M_1 per decidere (o semi-decidere, a seconda del caso) L_1 .

La Turing-riducibilità generalizza la riducibilità e ha precisamente le stesse proprietà.

$L_1 \leq_m L_2$ implica $L_1 \leq_T L_2$, ma non è vero il contrario.

Entrambe le riduzioni sono riflessive e transitive, e generano un pre-ordine tra tutti i linguaggi.

$$A_{TM} \leq_m H_{TM}$$

Poichè $A_{TM} \notin DEC$ allora la riduzione non può essere di Turing.

Ogni linguaggio non decidibile (non ricorsivamente enumerabile) può essere utilizzato alla sinistra di una riduzione.

$$\begin{aligned} REG_{TM} &\notin DEC \\ EQ_{TM} &\notin DEC \\ \bar{E}_{TM} &\notin R.E. \end{aligned}$$

Diremo che un linguaggio L appartiene a *CoR.E.* (è co-ricorsivamente enumerabile) se e solo se $\bar{L} \in R.E.$

Un problema non decidibile può essere ricorsivamente enumerabile (come A_{TM}), il che significa che possiamo, almeno, enumerarlo. Un problema indecidibile può anche essere non ricorsivamente enumerabile, ma co-ricorsivamente enumerabile, come A_{TM} , il che significa che possiamo, almeno, enumerare il suo complemento. Però, un problema può essere non decidibile, non ricorsivamente enumerabile, ed anche non co-ricorsivamente enumerabile.

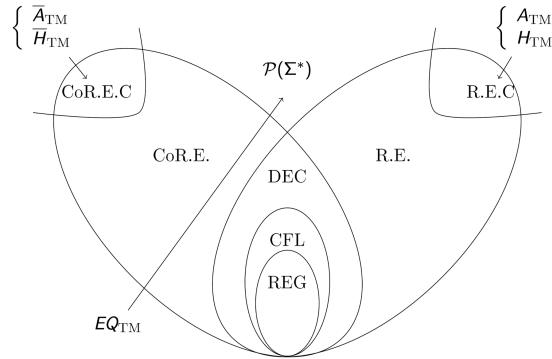
$$EQ_{TM} = \{\langle M, M' \rangle \mid M, M' \text{ sono } TM \text{ tali che } L(M) = L(M')\}$$

$$EQ_{TM} \notin R.E. \text{ e } EQ_{TM} \notin CoR.E.$$

Un problema si dice completo per una classe se tutti gli elementi di quella classe si riducono a lui. Quindi, un problema completo in una classe è il più difficile di quella classe. Per la classe *R.E.* abbiamo un candidato perfetto per la completezza: A_{TM} . Una nota terminologica: un problema L tale che tutti i problemi di una classe si riducono a lui è **hard** per quella classe, e se anche L vi appartiene, allora è anche completo per la classe.

Un problema completo per una classe è un problema rappresentativo della classe stessa.

Il linguaggio A_{TM} è *R.E.*-completo, ed il linguaggio \bar{A}_{TM} è *CoR.E.*-completo.



Logica proposizionale e frammenti causali

Il linguaggio della logica proposizionale è costituito da un insieme contabile di lettere proposizionali denotate da p, q, p_1, \dots , e dai simboli Booleani \wedge (and, congiunzione), \vee (or, disgiunzione), \neg (not, negazione), e \rightarrow (implicazione). La caratteristica principale di una proposizione è che può assumere unicamente i valori di vero o falso.

Le formule proposizionali ben formate sono tutte e sole quelle ottenibili attraverso la seguente grammatica:

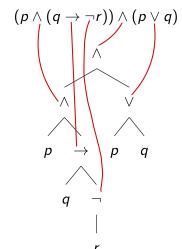
$$\phi ::= p \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg \phi \mid \phi \rightarrow \psi.$$

I simboli \perp, \top indicano rispettivamente il falso ed il vero sintattico. Definiamo $|\phi|$ la lunghezza di una formula proposizionale, cioè il numero totale di simboli utilizzati. Per esempio, $|p \rightarrow q| = 3$.

Le formule ben formate sono stringhe di simboli denotate da lettere greche, ma corrispondono anche ad alberi binari.

Questo albero permette di disambiguare la formula in assenza di parentesi, ed è anche la maniera più efficiente per rappresentare una formula nella memoria di una macchina. Poiché le formule ben formate sono prodotte da una grammatica libera dal contesto, chiaramente il problema di riconoscerle è decidibile (attraverso un automa a pila).

Le proposizioni corrispondono a frasi o fatti della lingua naturale che, come abbiamo detto in precedenza, hanno la caratteristica di poter assumere solo i valori vero o falso.



Per poter dare un significato ad una formula ben formata dobbiamo prima dare un significato ad ogni lettera proposizionale. Questa assegnazione di valori di verità si chiama interpretazione. Pertanto, se $AP = p, q, r$, ad esempio (AP significa atomic propositions e si usa per indicare le lettere proposizionali usate nel linguaggio), vi sono esattamente 8 possibili interpretazioni →

p	q	r
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

φ	ψ	$\varphi \wedge \psi$
0	0	0
0	1	0
1	0	0
1	1	1

φ	ψ	$\varphi \vee \psi$
0	0	0
0	1	1
1	0	1
1	1	1

Per poter assegnare un valore di verità ad una formula ben formata all'interno di una interpretazione, dobbiamo sapere come i valori di verità si trasferiscono da formule a formule più complesse attraverso gli operatori Booleani. Questo trasferimento segue le regole dettate dalle tavole di verità.

φ	ψ	$\varphi \rightarrow \psi$
0	0	1
0	1	1
1	0	0
1	1	1

φ	ψ	$\neg \varphi$
0		1
1		0

Tavole di verità

Un modo diverso di assegnare il valore di verità ad una formula proposizionale data una interpretazione è il seguente. Consideriamo una interpretazione I , e denotiamo con $I \models \phi$ ossia: ϕ è vera in I .

- $I \models p$ se e solo se p è vera in I
- $I \models \neg \phi$ se e solo se $I \not\models \phi$
- $I \models \phi \wedge \psi$ se e solo se $I \models \phi$ e $I \models \psi$
- $I \models \phi \vee \psi$ se e solo se $I \models \phi$ oppure $I \models \psi$
- $I \models \phi \rightarrow \psi$ se e solo se $I \not\models \phi$ oppure $I \models \psi$

Una formula ben formata per la quale esiste almeno una interpretazione che la soddisfa si chiama formula soddisfacibile.

Una formula ben formata tale che ogni interpretazione la soddisfa si chiama tautologia.

$$p \vee \neg p$$

Una formula ben formata tale che nessuna interpretazione la soddisfa si chiama contraddizione.

$$p \wedge \neg p$$

Una formula soddisfacibile che non è una tautologia si chiama anche contingenza.

Una interpretazione che soddisfa una formula si chiama anche modello della formula.

Chiamiamo: $SAT = \{\langle\phi\rangle \mid \phi \text{ è soddisfacibile}\}$ dove ϕ è una formula proposizionale ben formata.

Date due formule ϕ e ψ diremo che esse sono equivalenti se e solo se, per ogni interpretazione possibile, succede che $I \models \phi$ se e solo se $I \models \psi$. In questo caso possiamo scrivere che $\phi \leftrightarrow \psi$ (equivalenti in modo sintattico, ossia corrisponde a $(\phi \rightarrow \psi) \wedge (\psi \rightarrow \phi)$). Quando vogliamo affermare che ϕ e ψ sono equivalenti usiamo anche $\phi \equiv \psi$ (equivalenti in modo semantico).

Date due formule ϕ e ψ diremo che ψ segue logicamente da ϕ ($\phi \models \psi$) se solo se, per ogni interpretazione I , succede che $I \models \phi$ implica che $I \models \psi$.

Per ogni coppia di formule ϕ, ψ , succede che $\phi \models \psi$ se e solo se $\phi \rightarrow \psi$ è una tautologia

L'algoritmo delle tavole di verità per stabilire la soddisfacibilità di una formula $|\phi|$ ha complessità, in tempo, $O(2^{|\phi|})$.

Un metodo comunemente usato per stabilire la soddisfacibilità di una formula è chiamato tableau semantico, e consiste nel seguire la struttura della formula alla ricerca di una contraddizione oppure di un modello.

Usiamo un albero binario per rappresentare la formula ed un altro albero binario per rappresentare il suo tableau. Quest'ultimo ha come radice il nodo: $\phi.V$ dove con $.V$ indichiamo che cerchiamo di svolgere la formula alla ricerca di una interpretazione che la renda vera. La rappresentazione di ϕ come albero ci indica immediatamente l'operatore principale, e la sua corrispondente tavola di verità ci indica le condizioni essenziali perché sia vera.

Dalle regole semantiche possiamo ricavare delle regole generali per l'espansione di un giudizio:

1. La radice è il giudizio $\phi.V$
2. Finché esiste almeno un ramo ed almeno un giudizio su quel ramo non ancora espanso ripeti:

$\frac{\varphi \wedge \psi.V}{\varphi.V \quad \psi.V}$	$\frac{\varphi \wedge \psi.F}{\varphi.F \mid \psi.F}$	$\frac{\varphi \vee \psi.F}{\varphi.F \quad \psi.F}$	$\frac{\varphi \vee \psi.V}{\varphi.V \mid \psi.V}$
--	---	--	---

 - a. Espandi il giudizio non ancora espanso più vicino alla radice
 - b. Applica i risultati dell'espansione a tutti i nodi che passano per il nodo che conteneva il giudizio.
3. Se esiste almeno in ramo tale che non contiene alcuna contraddizione proposizionale, ϕ è soddisfacibile.

Non abbiamo però per nulla migliorato la complessità asintotica.

La forma clausale di una formula proposizionale è generata da una grammatica più stretta di quella generale.

Diciamo che un letterale è una lettera proposizionale o la sua negazione, ed una clausola è un oggetto della forma:

$$C ::= \neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_n \vee p_{n+1} \vee p_{n+2} \vee \dots \vee p_{n+m}$$

Una formula proposizionale si dice in forma clausale (*CNF*) se è una congiunzione di clausole:

$$\phi ::= C_1 \wedge C_2 \wedge \dots \wedge C_k$$

Per ogni formula proposizionale ϕ ne esiste una equisoddisfacibile in forma clausale, in cui ogni clausola ha al massimo tre letterali, e di dimensione polinomiale rispetto alla dimensione di ϕ .

Chiamiamo *CNF* una formula in forma normale congiuntiva, e *kCNF* una formula in forma normale congiuntiva in cui ogni clausola ha al massimo k letterali.

$$3SAT = \{\langle\phi\rangle \mid \phi \text{ è in } 3CNF \text{ ed è soddisfacibile}\}$$

Trasformiamo $p_1 \vee p_2 \vee p_3 \vee p_4$ in una forma clausale con 3 letterali: $(p_1 \vee p_2 \vee q) \wedge (\neg q \vee p_3 \vee p_4)$ dove q è una nuova lettera

Consideriamo nuovamente una clausola:

$$C ::= \neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_n \vee \dots \vee p_{n+m} \text{ se } m \leq 1 - \text{ossia se c'e al massimo un letterale non negato - si dice di Horn.}$$

Una formula è in forma di Horn se è in *CNF* ed ogni clausola ha al massimo un letterale positivo.

Mentre qualunque formula proposizionale può essere scritta come congiunzione di clausole, questo non è vero per clausole di Horn.

$$HornSAT = \{\langle\phi\rangle \mid \phi \text{ è in forma di Horn ed è soddisfacibile}\}$$

$$C ::= \neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_n \vee p_{n+1} \equiv (p_1 \wedge p_2 \wedge \dots \wedge p_n) \rightarrow p_{n+1}$$

L'essenza del frammento di Horn è che risponde all'intuizione algoritmica: stabilisco un insieme di condizioni iniziali (i fatti, cioè i letterali positivi singoli) e un insieme di regole (clausole di Horn).

Per stabilire se una certa formula di Horn è soddisfacibile, assumeremo che ogni clausola sia scritta in forma implicativa, e di dover stabilire quindi se una certa congiunzione di implicazioni ha o meno un modello.

Lo facciamo ricordando che:

$$C ::= \neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_n \vee p_{n+1} \equiv (p_1 \wedge p_2 \wedge \dots \wedge p_n) \rightarrow p_{n+1} \quad \text{e che:}$$

$$C ::= \neg p_1 \vee \neg p_2 \vee \dots \vee \neg p_n \equiv (p_1 \wedge p_2 \wedge \dots \wedge p_n) \rightarrow \perp$$

Algoritmo di forward checking:

- assumiamo che i fatti negativi siano scritti in forma $p \rightarrow \perp$

1. Sia A un insieme che contiene tutti e soli i fatti positivi di ϕ ;
2. Ripeti finché $\perp \notin A$ oppure finché A cambia:
 - a. Seleziona una clausola $\psi \rightarrow I$ tale che tutto ψ è in A , e tale che $I \notin A$ (I è un letterale positivo oppure \perp);
 - b. Metti I in A
3. Se $\perp \in A$ allora ϕ non è soddisfacibile. Altrimenti, lo è.

L'algoritmo del forward checking per stabilire la soddisficiabilità di una formula di Horn ϕ in forma clausale ha complessità, in tempo, $O(|\phi|^2)$.

La sintassi della logica proposizionale quantificata eredita interamente il linguaggio della logica proposizionale, e permette di quantificare, attraverso \forall (quantificatore universale) e \exists (quantificatore esistenziale) i valori di verità delle lettere proposizionali.

$\forall p \forall q (p \wedge q) \rightarrow \exists r$ per ogni valore di p e per ogni valore di q , $(p \wedge q)$ implica che esiste un valore di verità per cui vale r .

→ Passiamo da "è soddisfacibile?" a "per quali valori è soddisfacibile?"

Stabiliamo se una formula completamente quantificata è vera oppure no, usando la simbologia: $\models \phi$ e definendo il linguaggio: $QSAT = \{\langle \phi \rangle \mid \phi \text{ è proposizionale quantificata, ed è vera}\}$

Algoritmo di truth-checking:

Per stabilire se $Q_1 Q_2 \dots Q_n \phi$ è vera, usiamo il seguente algoritmo, chiamato con $i = 1$ e $I = 0$:

1. Se $i = n$, allora rispondi "Sì" se e solo se $I \models \phi$
2. Se $Q_i = \forall p$, richiamati su $Q_{i+1} Q_{i+2} \dots Q_n \phi$ con $I = I \cup \{p = 1\}$ e su $Q_{i+1} Q_{i+2} \dots Q_n \phi$ con $I = I \cup \{p = 0\}$, e rispondi 'Sì se e solo se entrambe le chiamate hanno risposto Sì';
3. Se $Q_i = \exists p$, richiamati su $Q_{i+1} Q_{i+2} \dots Q_n \phi$ con $I = I \cup \{p = 1\}$ e su $Q_{i+1} Q_{i+2} \dots Q_n \phi$ con $I = I \cup \{p = 0\}$, e rispondi 'Sì se e solo se almeno una delle chiamate ha risposto Sì'.

L'algoritmo del truth-checking per stabilire la verità di una formula proposizionale quantificata ha complessità $O(2^{|\phi|})$

Complessità

Complessità in tempo

Ci concentriamo sull'insieme DEC solamente, e ci interessiamo alla complessità dei problemi decidibili, attraverso gli algoritmi che li risolvono, in termini sia di tempo che di spazio.

Usiamo le stesse convenzioni utilizzate nel corso di Algoritmi: dimentichiamo i dettagli implementativi, assumiamo che ogni operazione elementare costi uno, e che la complessità in tempo di un algoritmo sia quindi funzione dei cicli, delle chiamate ricorsive, e delle chiamate a funzione. Supponiamo inoltre che tutti questi costrutti siano realizzabili, in linea di principio, attraverso macchine di Turing.

Diremo che una macchina di Turing M ha complessità in tempo $f(n)$, dove f è una funzione da \mathbb{N} a \mathbb{N} , se e solo se M esegue al massimo $f(n)$ passi semplici al computare qualunque parola w di lunghezza n . In questo caso diremo che M appartiene alla classe $TIME(f(n))$. In maniera simile, diremo che una macchina di Turing non deterministica M ha complessità in tempo $f(n)$, dove f è una funzione da \mathbb{N} a \mathbb{N} , se e solo se M esegue al massimo $f(n)$ passi semplici al computare qualunque parola w di lunghezza n in qualunque dei possibili rami di computazione. In questo caso diremo che M appartiene alla classe $NTIME(f(n))$. Come quando analizziamo algoritmi, ci interessiamo al comportamento asintotico, e pertanto, invece di parlare di complessità in tempo $f(n)$, parleremo di complessità in tempo $O(f(n))$.

Indipendentemente dal modello di computazione scelto, otteniamo lo stesso insieme di problemi computabili

Dato un linguaggio riconosciuto da una macchina di Turing M con k nastri in tempo $f(n)$, esiste una macchina di Turing M' con un solo nastro che riconosce lo stesso linguaggio in tempo $O(f(n)^2)$.

Abbiamo dimostrato che per ogni $k - TM M$ che riconosce un linguaggio L , esiste una TM con un solo nastro, che riconosce lo stesso linguaggio.

L'unico nastro di M' contiene l'informazione di tutti i k nastri di M ; l'osservazione fondamentale è che in tempo $O(f(n))$ si scrivono, nel peggior caso, $O(f(n))$ caselle di nastro: pertanto, $O(f(n))$ è un limite superiore alla lunghezza della parte attiva del nastro di M . Ogni movimento prevede un numero costante di scorrimenti interi del nastro di M , cioè ogni movimento che prima costava $O(1)$ adesso costa $O(f(n))$. Nel peggior caso, dunque, il tempo totale è $O(f(n)^2)$.

Il teorema precedente ci mostra che la differenza in complessità temporale tra varianti deterministiche della macchina di Turing è, al più, polinomiale.

Dato un linguaggio riconosciuto da una macchina di Turing M non deterministica in tempo $f(n)$, esiste una macchina di Turing M' deterministica che riconosce lo stesso linguaggio in tempo $O(2^{f(n)})$.

Una NTM è la sequenza di passi ottimi della TM corrispondente.

Uso una $3 - TM$, con un nastro read-only per l'input, un nastro di lavoro e un terzo nastro.

Per imitare una computazione non-deterministica, copio quello che c'è sull'input nel work tape. Nel terzo nastro scrivo, separate da un simbolo apposito, tutte le possibili computazioni dei passi (es. $[q_0 \$ q_0 q_1 \$ q_1 q_2 \$ \dots \$ q_0 q_1 q_3 \$ \dots]$) dopo di che seguo esattamente l'algoritmo, ma per ogni scelta non-deterministica, 'fai la prova' sul work tape. Ognuna delle possibili computazioni è lunga al massimo $O(f(n))$ e l'arietà dell'albero delle computazioni è c , quindi dobbiamo esplorare al massimo $O(c^{f(n)})$ elementi su questo nastro. Per ogni nuova simulazione, spendiamo $O(f(n))$ per leggere la computazione corretta, e la macchina a 3 nastri viene poi simulata a sua volta con una macchina ad un solo nastro con una spesa al più quadratica. Quindi, in totale, spendiamo $O(2^{f(n)})$.

Rinunciare al non determinismo costa al più un salto esponenziale, ma non abbiamo alcuna certezza di ottenere un risultato migliore.

P è la classe di tutti e soli i problemi che possono essere risolti con macchine di Turing deterministiche in tempo polinomiale:

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

Questa classe è indifferente a varianti deterministiche del modello di computazione. La classe P è interessante perché è matematicamente solida, e corrisponde ai problemi che comunemente consideriamo trattabili. Infatti, il fatto che un problema P si

possa risolvere in un tempo accettabile dipende quasi esclusivamente dall'hardware. Invece, problemi fuori da questa classe sono intrattabili a prescindere dall'hardware utilizzato.

$REG \in P$

Esempi di problemi appartenenti a P sono:

$$\begin{aligned} PATH &= \{\langle G, s, t \rangle \mid G \text{ è diretto e } s \rightsquigarrow t\}, \\ MST_k &= \{\langle G, k \rangle \mid G \text{ è indiretto pesato con un } MST \text{ di peso } \leq k\} \\ PATH_k &= \{\langle G, s, t, k \rangle \mid G \text{ è diretto pesato e } s \rightsquigarrow_k t\} \end{aligned}$$

$CFG \subset P$

Ogni problema che è in CFG è anche in $P \rightarrow$ per ogni problema in CFG deve esistere una MT che lo decide in tempo polinomiale deterministico, il che è implicato dall'affermazione $A_{CFG} \in P$. Per dimostrare questo, consideriamo $\langle G, w \rangle$ dove G è una grammatica libera in forma di Chomsky e w una stringa, usiamo una macchina di Turing che implementa l'algoritmo CYK :

Algoritmo CYK con input x :

1. Verifica che x sia del tipo $\langle G, w \rangle$, altrimenti rifiuta, estrai $w = w_1, \dots, w_n$ e G con simboli non terminali R_1, \dots, R_r
2. Costruisci la matrice Booleana ternaria $P[n, n, r]$, inizializzata tutta a falso, tranne gli elementi $P[1, i, j]$ quando esiste una regola $R_i ::= w_j$
3. Per i da 1 a n , j da 1 a $n - i + 1$, e k da 1 a $i - 1$, ripeti
 - a. Per ogni regola $R_a ::= R_b R_c$, ripeti
 - i. Se $P[k, j, b] \wedge P[i - k, j + k, c]$ allora metti $P[i, j, a]$ a vero
4. Se $P[n, 1, 1]$ accetta, altrimenti rifiuta.

Poiché devo riempire una matrice ternaria con indici non più lunghi di n , il tempo totale che ci metto è al massimo n^3 .

Poichè esiste almeno un linguaggio non libero dal contesto che è polinomiale (come $a^n b^n c^n$), l'inclusione è stretta.

P è chiusa sotto unione, concatenazione, star di Kleene, intersezione, complemento e differenza insiemistica.

In maniera analoga alla classe P , possiamo definire una nuova classe di problemi basata sulla computabilità per mezzo di macchine di Turing non deterministiche. Diremo che la classe di tutti e soli i problemi che possono essere risolti con macchine di Turing anche non deterministiche in tempo polinomiale costituisce la classe NP :

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

La classe NP è la classe dei problemi che si possono verificare in tempo polinomiale.

$P \subseteq NP$

Non sappiamo però se $P \subset NP$ oppure $P = NP$

$COMP \in NP$

$COMP = \{\langle n \rangle \mid n \in \mathbb{N} \text{ e } n \text{ non è primo}\}$

1. Verifica che n sia un numero naturale, altrimenti rifiuta
2. Scegli non deterministicamente due numeri naturali x, y
3. Se $n = xy$ allora accetta, altrimenti rifiuta.

Il caso di $COMP$ è emblematico. Infatti, ci affidiamo al non determinismo per indovinare il risultato, e poi operiamo una fase di verifica che deve essere polinomiale. Se la verifica non va a buon fine, allora la fase non deterministica non può avere successo. In altre parole, stiamo immaginando che il passo non deterministico esplori, in tempo $O(1)$, tutto lo spazio di soluzioni, e questo è coerente con la maniera in cui abbiamo trasformato una macchina non deterministica in una deterministica.

$HAM \in NP$

$HAM = \{\langle G \rangle \mid G \text{ è un grafo diretto con un ciclo Hamiltoniano}\}$

Un ciclo in un grafo è detto Hamiltoniano se e solo se tocca tutti i vertici esattamente una volta.

La seguente macchina con input w lo dimostra:

1. Verifica che w sia del tipo $\langle G \rangle$, altrimenti rifiuta
2. Scegli non deterministicamente una sequenza di vertici v_1, \dots, v_n
3. Se $n \neq |V|$, allora rifiuta; se $v_i = v_j$, con $j \neq n$ e $i \neq j$, allora rifiuta
4. Se per qualche i non esiste l'arco (v_i, v_{i+1}) , allora rifiuta
5. Accetta

NP è chiusa sotto unione, concatenazione, star di Kleene, e intersezione. Non è noto se NP sia chiusa sotto complemento e differenza insiemistica.

Se $L \in NP$, allora esiste una macchina M , probabilmente non deterministica, che in tempo polinomico termina e decide L . Sia M' la macchina che si ottiene da M invertendo il risultato. M' non decide necessariamente L . Infatti, se $w \in L$, allora $w \notin L'$; quindi, le scelte non deterministiche che opera M' non contribuiscono a stabilire lo status di w rispetto a L . Pertanto la risposta data da M' non è necessariamente corretta. Poiché esiste questo problema riguardo al complemento, ha senso denire la classe $CoNP$ di tutti quei problemi tali che il loro complemento è un problema NP .

Il problema \overline{HAM} è un esempio \leftarrow è chiaro che non possiamo costruire un verificatore per questo preblema, anche se questo non dimostra che $\overline{HAM} \notin NP$ in maniera definitiva.

$P \subseteq NP \cap CoNP$

Ovviamente $P \subseteq NP$. Poichè P è chiusa sotto complemento, sappiamo che $\overline{L} \in P \rightarrow \overline{L} \in NP \rightarrow L$ è tale che il suo complemento sta in NP , quindi $L \in CoNP$

Recap:

- P è la classe dei problemi facili da risolvere.
- NP è la classe dei problemi la cui soluzione è facile da verificare - è la classe dei giochi con vincoli per una sola persona.
- $CoNP$ è la classe di tutti i problemi tali che il loro complemento è un problema NP .

Tesi di Church-Turing: qualunque modello di computazione è equivalente alla macchina di Turing

L_1 si riduce polinomialmente a L_2 se esiste una funzione $f : L_1 \rightarrow L_2$ tale che per ogni parola w , $w \in L_1$ solo se $f(w) \in L_2$. Questa forma di riduzione si dice polinomica, si denota con $L_1 \leq_p L_2$, e specializza la riduzione many-one.

Siano L_1, L_2 due linguaggi tali che $L_1 \leq_p L_2$. Allora, se $L_2 \in P$, allora $L_1 \in P$

Assumiamo che $L_1 \leq_p L_2$ e $L_2 \in P$. Esiste una macchina di Turing deterministica M_f che la calcola con costo polinomico, dato che L_2 è in P , esiste una macchina di Turing deterministica M_{L_2} che riconosce il linguaggio in tempo polinomico.

Costruiamo quindi una TM che riconosce L_1 con input w :

1. Calcola $f(w)$ usando M_f
2. Esegui $M_{L_2}(f(w))$, se accetta, accetta, altrimenti rifiuta.

La macchina è corretta grazie alle ipotesi fatte su f ; pertanto decide L_1 in tempo polinomico e in maniera deterministica, mostrando che $L_1 \in P$.

La riduzione polinomica specializza la riduzione many-one. La riduzione di Turing generalizza quella many-one.

Esiste la riduzione polinomiale di Turing definita come la riduzione di Turing ma con il vincolo che la macchina sia deterministica

polinomiale.

Dato un problema L diremo infatti che è NP -hard se e solo se ogni problema in NP si può ridurre a L , e se $L \in NP$, allora diremo che L è anche NP -completo. In altre parole $L \in NP$ è completo per la classe NP se ogni altro problema $L' \in NP$ è tale che $L' \leq_p L$.

$$SAT = \{ \langle \phi \rangle \mid \phi \text{ è una formula proposizionale soddisfacibile} \}$$

SAT è NP -completo

1. Utilizza una grammatica libera dal contesto per stabilire se $w = \varphi$ è una formula ben formata, e se non lo è, rifiuta
2. Scegli non deterministicamente una assegnazione I per tutte le lettere proposizionali del linguaggio
3. Controlla in tempo polinomiale che $I \models \phi$ e se è così accetta, altrimenti rifiuta

Consideriamo un linguaggio qualsiasi $L \in NP$. Vogliamo mostrare che $L \leq_p SAT$, cioè che esiste una funzione f tale che $w \in L$ se e solo se $f(w) \in SAT$, cioè $w \in L$ se e solo se $f(w)$ è una formula soddisfacibile. In altre parole, vogliamo che $f(w)$ descriva la computazione di w in M_L . Quindi $f(w)$ è una congiunzione logica di formule che:

1. Descrivono che la computazione inizia con w
2. Dicono che la computazione rispetta δ
3. Assicurano che la computazione giunge a uno stato di accettazione

Se M_L termina in tempo polinomico, $O(n^k)$ allora possiamo dire che, durante tutta la computazione di w , sul nastro non si scrivono mai più di n^k simboli

$$kSAT = \{ \langle \phi \rangle \mid \phi \text{ è una formula proposizionale soddisfacibile in k-CNF} \}$$

$kSAT$ è NP -completo per ogni $k \geq 3$

Chiaramente $kSAT \in NP$, perchè si applica la stessa macchina usata per il caso generale. Per mostrare che è anche completo, mostriamo che $3SAT$ può essere ridotto polinomicamente a SAT .

Poichè possiamo sempre trasformare una formula ϕ qualsiasi in una formula equisoddisfacibile in forma clausale con al massimo 3 letterali in tempo polinomico, sappiamo che $SAT \leq_p 3SAT$.

$CLIQUE = \{ \langle G, k \rangle \mid G \text{ è indiretto esiste una clique } \geq k \text{ elementi} \}$ dove con clique si intende gruppo di vertici tale che ogni coppia è connessa ad un arco \leftarrow vogliamo trovare il massimo sottoinsieme di vertici $S \subseteq V$ tali che, per ogni coppia di vertici $u, v \in S$, succede che $(u, v) \in E$ (si noti che questo vale se e solo se $(v, u) \in E$)

$CLIQUE$ è NP -completo

È NP -completo in quanto L può essere deciso in tempo polinomico:

1. Controlla se w è del tipo $\langle G, k \rangle$, altrimenti rifiuta
2. Scegli non deterministicamente un sottoinsieme di vertici di cardinalità k
3. Se per ogni coppia di vertici nel sottoinsieme scelto esiste un arco che li connette, accetta, altrimenti rifiuta

È $NPNP3SAT \leq_p CLIQUE$:

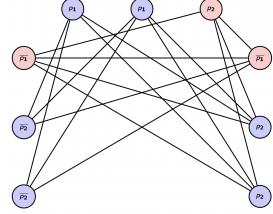
Prendiamo una ϕ in forma clausale con al massimo tre letterali, ad esempio $\phi = (p_1 \vee p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2 \vee \neg p_2) \wedge (\neg p_1 \vee p_2 \vee p_2)$, scopo di questo esercizio è costruire in maniera automatica un grafo G_ϕ (che dipende da ϕ), e un k (che dipende da ϕ) tali che G ha una k -clique (o maggiore) se e solo se ϕ è soddisfacibile, nel qual caso per ogni clausola troveremo una assegnazione di verità che rende vero almeno un letterale, quindi il nostro G avrà:

1. Un vertice per ogni letterale (anche quelli ripetuti).
2. Un arco che connette ogni coppia di vertici escluse quelle della stessa clausola o contraddittorie.

$\langle \phi \rangle \rightarrow \langle G, k \rangle$: Per ogni formula ϕ con k clausole, ϕ è soddisfacibile se e solo se G_ϕ ha una k -clique: Primo, si supponga che ϕ sia soddisfacibile. Allora deve esistere una maniera di assegnare true ad almeno un letterale di ogni clausola in maniera da soddisfare la formula.

Sceglieremo quel letterale per ogni clausola: avremo una scelta di k vertici su G_ϕ , e questi sono necessariamente non della stessa clausola e non contraddittori, quindi tutti a due a due

connessi. Quindi abbiamo una k -clique. Si supponga adesso che G_ϕ possieda una k -clique (dove k è il numero di clausole). Certamente questi vertici sono letterali non della stessa clausola, e certamente sono non contraddittori: quindi corrispondono a una assegnazione di verità che rende vera ogni clausola e pertanto la formula intera $\rightarrow \phi$ è soddisfacibile.



Dato un grafo $G = (V, E)$ diretto e non pesato, vogliamo stabilire se esiste un ciclo che passa esattamente una volta da tutti i vertici.

$$HAM = \{\langle G \rangle \mid G \text{ è un grafo diretto con un ciclo Hamiltoniano}\}$$

HAM è NP -completo

Abbiamo già dimostrato che $HAM \in NP$. Mostriamo che $3SAT \leq_p HAM$.

Usiamo lo stesso esempio visto prima: $\phi = (p_1 \vee p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2 \vee \neg p_2) \wedge (\neg p_1 \vee p_2 \vee p_2)$ per mostrare una riduzione polinomica con le seguenti caratteristiche: ogni formula ϕ del linguaggio proposizionale clausale con clausole al più ternarie produce un grafo G_ϕ tale che G_ϕ ha un cammino Hamiltoniano se e solo se ϕ è soddisfacibile.

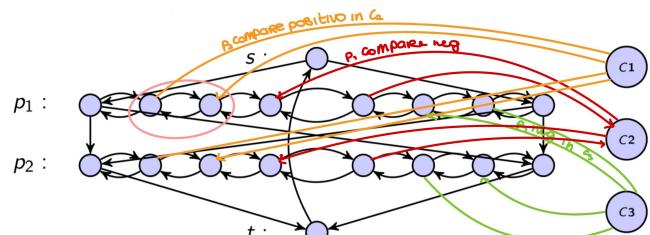
Nella nostra costruzione useremo due vertici s, t più altri $n \cdot (2c + 2)$ vertici, dove n è il numero di proposizioni e c il numero di clausole, aggiungendo poi un vertice per ogni clausola.

Il vertice s (da dove inizia il ragionamento) si connette alla prima fila tanto da un lato come dall'altro, per permettere l'interpretazione in maniera opportuna della prima variabile. Ogni fila è connessa alla seguente sia da destra che da sinistra.

Le lettere proposizionali che vengono interpretate come vere permettono la visita del ciclo da sinistra a destra, mentre quelle che vengono interpretate come false permettono la visita della loro fila da destra a sinistra.

Supponiamo che p_1 si valuti a vero: allora tutti i vertici della sua fila verranno visitati da sinistra a destra: le connessioni con le clausole nelle quali p_1 appare devono essere tali da poter andare e tornare dalla fila senza ripassare dallo stesso nodo. D'altra parte, se p_1 viene valutata a falso, usiamo lo stesso principio però da destra a sinistra.

Consideriamo, ad esempio, la relazione che sussiste tra p_1 e le clausole. Poiché appare positiva in C_1 , connetto il rappresentante sinistro di p_1 con C_1 , e C_1 con il rappresentante destro. In questo modo, posso proseguire sul mio ciclo Hamiltoniano (con p_1 vera). Inoltre poiché appare negativa in C_2 e C_3 , connetto il rappresentante destro di p_1 con C_2 e C_3 , e C_2 e C_3 con il rappresentante sinistro. Quindi, nell'esempio, assegnando $p_2 = true$ e $p_1 = false$ la formula è soddisfacibile.

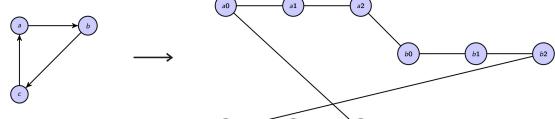


$$UHAM = \{\langle G \rangle \mid G \text{ è un grafo indiretto con un ciclo Hamiltoniano}\}.$$

$UHAM$ è NP -completo

$HAM \leq_p UHAM$: mostriamo che per ogni grafo diretto non pesato esiste un grafo indiretto non pesato tale che il primo ha un ciclo Hamiltoniano se e solo se lo ha il secondo.

Sia $G = (V, E)$ un grafo diretto qualsiasi, e sia $G' = (V, E')$ il grafo indiretto che vogliamo costruire. Per ogni vertice $v \in V$, mettiamo $v_0, v_1, v_2 \in V'$, e gli archi $(v_0, v_1), (v_1, v_2) \in E'$, in modo da dargli una direzionalità. Quindi per noi v_0 significa che si 'entra' in v , mentre v_2 che si 'esce'; mettiamo quindi, per ogni arco $(v, u) \in E$, un arco $(v_2, u_0) \in E'$.



$$TSP = \{\langle G, k \rangle \mid G \text{ è un grafo indiretto complesso con un ciclo Hamiltoniano di peso } \leq k\}.$$

TSP è *NP*-completo

UHAM \leq_p *TSP*:

Consideriamo un grafo indiretto non pesato $G = (V, E)$, vogliamo costruire un grafo indiretto, completo, pesato $G' = (V, E', w)$ tale che G ha un ciclo Hamiltoniano se e solo se per un certo k , computabile in maniera polinomica deterministica, G' ha un ciclo Hamiltoniano che pesa non più di k . A questo fine, definiamo $E' = E$ più tutti gli archi mancanti, e diamo peso 1 a ogni arco che era in E e peso 2 a ogni altro arco.

G' ha quindi un ciclo Hamiltoniano di peso inferiore o uguale a $|V|$ se e solo se G ha un ciclo Hamiltoniano: ovviamente questo è vero giacchè per avere un ciclo Hamiltoniano in G devo, come minimo, usare $|V|$ archi (che pesano tutti 1); se il peso è superiore, allora ho sicuramente usato un arco nuovo.

COVER = { $\langle G, k \rangle$ | G è un grafo indiretto con k copertura} \leftarrow è vero che esiste un sottoinsieme S di k vertici tali che ogni arco ha almeno uno dei suoi due estremi in S ?

COVER è *NP*-completo

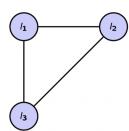
La seguente macchina di Turing decide *COVER* in tempo polinomico:

1. Controlla se w è del tipo $\langle G, k \rangle$, altrimenti rifiuta
2. Scegli, non deterministicamente, un sottoinsieme S di vertici di cardinalità k
3. Controlla se per ogni arco almeno uno dei suoi vertici è in S , se è così accetta, altrimenti rifiuta.

3SAT \leq_p *COVER*: in ogni formula ϕ in forma clausale con al massimo 3 letterali per clausola, e tale che presenta s clausole con t proposizioni diverse. Esiste un grafo indiretto non pesato G_ϕ tale che ϕ è soddisfacibile se e solo se G_ϕ ha una k -copertura con $k = t + 2s$.

Costruiamo questo sotto-grafo per ogni proposizione \rightarrow

È chiaro che, indipendentemente dalla struttura del grafo finale, almeno un vertice tra p_1 e \bar{p}_1 dovrà essere scelto nella copertura. Facciamo questa scelta per puntare a una scelta di valori di verità che soddisfa la formula.



Consideriamo adesso una clausola generica $(I_1 \vee I_2 \vee I_3)$ (ricorda: ogni letterale è una proposizione o la sua negazione), e costruiamo il sotto-grafo qua a sinistra.

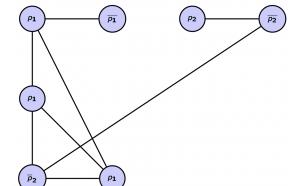
È necessario scegliere due vertici per coprire questo sotto-grafo. Se ϕ è soddisfacibile, deve essere possibile trovare un valore di verità per ogni proposizione in maniera non contraddittoria e identificare almeno un letterale soddisfatto per ogni clausola, o non più di due letterali non soddisfatti per ogni clausola.

A questo punto, per capire come combinare le parti che abbiamo disegnato, consideriamo un esempio:

$$\phi = (p_1 \vee \neg p_2 \vee p_1).$$

Immaginiamo un modello che soddisfa questa formula dato da $p_1 = \text{true}$ e $p_2 = \text{false}$.

Avremo \rightarrow



2SAT $\in P$

Dobbiamo trovare un algoritmo deterministico, che termina in tempo polinomico, che risolve il problema di stabilire se una certa congiunzione ϕ di clausole al più binare è o no soddisfacibile. A questo fine, costruiamo un grafo G_ϕ con tanti vertici quanti sono i letterali coinvolti e le loro negazioni. Gli archi di questo grafo sono tutti e soli quegli archi (u, v) tali che esiste una clausola equivalente a $\neg u \rightarrow v$.

Ad esempio, se $\phi = (p \vee q) \wedge (q \vee \neg r)$, allora avremo precisamente 6 vertici, etichettati con: $p, q, r, \neg p, \neg q, \neg r$, e i seguenti archi:

$$\neg p \rightarrow q, \quad \neg q \rightarrow p \quad e \quad \neg q \rightarrow \neg r, \quad r \rightarrow q.$$

Adesso usiamo un algoritmo di visita di un grafo diretto qualsiasi, e cerchiamo se è vero che, per qualche p :

1. Esiste un percorso da un vertice etichettato con p ad uno etichettato con $\neg p$, e

2. Esiste un percorso da un vertice etichettato con $\neg p$ ad uno etichettato con p .

Se è così, non vi è modo di assegnare un valore di verità a p in maniera coerente e che rispetti tutte le clausole. Quindi ϕ è insoddisfacibile. Inoltre, se per nessuna lettera proposizionale si verificano entrambe le condizioni, allora la formula è certamente soddisfacibile.

$2SAT$ può essere declinato in termini di problemi di ottimizzazione: quante clausole di una congiunzione posso soddisfare assieme, al massimo? Questo problema, chiamato anche $MAX-2SAT$, ha la seguente versione decisionale: posso soddisfare almeno k clausole tutte assieme in una congiunzione di clausole al più binarie?

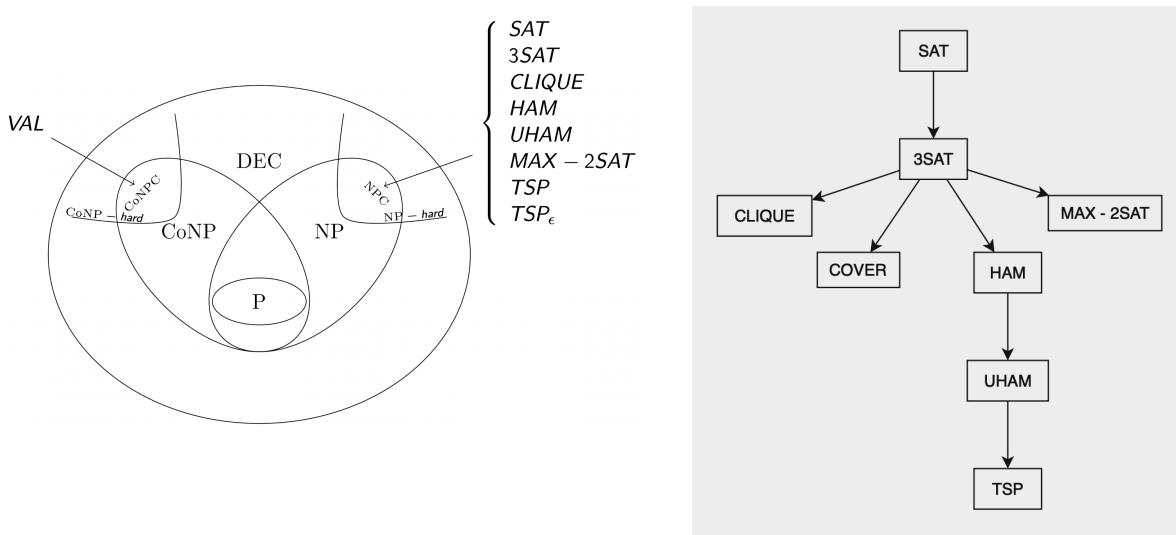
$MAX-2SAT = \{(\varphi, k) \mid \varphi \text{ è una congiunzione di clausole al più binarie delle quali almeno } k \text{ si soddisfano contemporaneamente}\}$

$MAX-2SAT$ è NP -completo

La seguente macchina di Turing non deterministica mostra che questo problema è in NP . Con entrata w :

1. Controlla se w è del tipo (ϕ, k) , altrimenti rifiuta
2. Scegli, non deterministicamente, un sottoinsieme di k di clausole
3. Controlla, usando l'algoritmo visto precedentemente, se il problema $2SAT$ composto da quelle k clausole è soddisfacibile, se è così accetta, altrimenti rifiuta

$3SAT \leq_p MAX-2SAT$



Complessità nello spazio

Ci siamo occupati della risorsa tempo in termini di numero di operazioni atomiche (cambi di stato di una macchina di Turing). Adesso ci occupiamo della risorsa spazio, tenendo a mente che queste due risorse non sono ortogonali, ma entrambe contribuiscono alla classificazione della difficoltà di un problema.

In generale, dobbiamo pensare ad una macchina di Turing come ad una macchina con almeno tre nastri, due dei quali con un uso predefinito: uno di sola lettura per l'input e il terzo di sola scrittura per l'output; il nastro centrale lo definiamo come nastro di lavoro, e definiamo lo spazio utilizzato come il numero di celle di questo nastro che si usano. Avere uno o più nastri di lavoro non cambia nulla.

Una macchina deterministica M ha complessità in spazio $f(n)$, dove $f : (\mathbb{N} \rightarrow \mathbb{N})$, se e solo se M utilizza al massimo $f(n)$ celle diverse di work tape per computare qualunque parola di lunghezza n . In questo caso diremo che M appartiene alla classe $SPACE(f(n))$. Diremo invece che una macchina di Turing anche non deterministica M che ha complessità in spazio $f(n)$ appartiene alla classe $NSPACE(f(n))$.

La classe di tutti e soli i problemi che possono essere risolti con macchine di Turing deterministiche usando spazio polinomiale è la classe $PSPACE = \bigcup_k SPACE(n^k)$.

Mentre sul modello non deterministico definiamo $NPSPACE = \bigcup_k NSPACE(n^k)$.

Lo spazio è chiaramente una risorsa più potente del tempo, perché, al contrario di quest'ultimo, è riutilizzabile. Il non determinismo non aggiunge potere computazionale quando misuriamo la complessità in termini di spazio:

$\text{Se } f(n) \geq \log(n) \text{ allora } \text{NSPACE}(f(n)) \subseteq \text{SPACE}(f^2(n))$. Pertanto $\text{NPSPACE} = \text{PSPACE}$

Da questo teorema capiamo che la classe NPSPACE non ha senso di esistere.

La classe PSPACE è chiusa rispetto a unione, concatenazione e star di Kleen, complemento, intersezione e differenza insiemistica. Non è chiusa per sottoinsieme.

La classe PSPACE può essere vista come la classe dei giochi con due giocatori.

$\text{NP}, \text{CoNP} \subseteq \text{PSPACE}$

Una macchina di Turing (anche non deterministica) che termina in tempo $O(f(n))$, scrive alla peggio su $O(f(n))$ celle di spazio. Quindi ogni classe di tempo, anche non deterministica, è inclusa nella classe di spazio corrispondente $\rightarrow \text{NP} \subseteq \text{PSPACE} \leftarrow$ vale anche per CoNP .

Poiché PSPACE è chiusa per complemento non ha senso definire una classe di problemi complemento.

Non conosciamo esattamente la relazione esatta tra PSPACE e NP , CoNP , come non conoscevamo quella tra P e NP .

Le riduzioni polinomiche in tempo sono ovviamente polinomiche in spazio e preservano quindi la classe PSPACE .

Un problema L è PSPACE -hard se e solo se, per ogni problema $L' \in \text{PSPACE}$ si ha che $L' \leq_p L$, e che è PSPACE -completo se succede anche che $L \in \text{PSPACE}$.

$\text{QSAT} = \{\langle \phi \rangle \mid \phi \text{ è proposizionale quantificata, ed è vera}\}$

QSAT è PSPACE -completo

Ci sono $O(n)$ diverse variabili proposizionali in una formula ϕ di lunghezza n , pertanto lo spazio che si richiede è lineare, quindi anche polinomiale. Quindi, $\text{QSAT} \in \text{PSPACE}$. Per mostrare che è anche completo per la classe PSPACE , consideriamo un problema $L \in \text{PSPACE}$ qualsiasi, e mostriamo che vale $L \leq_p \text{QSAT}$.

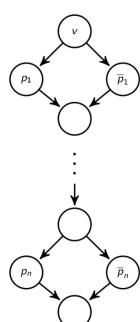
$\text{GG} = \{\langle G, v \rangle \mid G \text{ è un grafo vincente e } S \text{ vince}\} \leftarrow$ Il problema è stabilire se S ha una strategia vincente -generalized geography

GG è PSPACE -completo

$\text{QSAT} \leq_p \text{GG}$: Costruiamo un grafo diretto G_ϕ e scegliamo un vertice v tale che, per una data formula proporzionale quantificata ϕ , S vince il gioco (G_ϕ, v) se e solo se ϕ è vera. Possiamo assumere che ϕ si trovi nella forma: $Q_{p_1} Q_{p_2} \dots \phi'$

Senza perdere generalità possiamo anche assumere che si trovi in questa forma ristretta: $\exists p_1 \forall p_2 \exists p_3 \dots \exists q_n \phi''$.

Ipotizzando quindi che questa formula abbia n proposizioni tutte quantificate e k clausole, il grafo sarà costituito da una parte sinistra:



Per ogni lettera proposizionale creiamo un gadget che ha esattamente

– questa forma: quattro vertici collegati a diamante, con v come primo vertice

S (variabile globale turno) sceglie un elemento dopo v , il che corrisponde ad assegnare un valore di verità a p_1 ;

Per come è fatto G , la seconda variabile (o comunque le variabili pari) è assegnata da D , che 'sceglie' un vertice vuoto, in quanto lui non ha ruolo nelle sue scelte;

Per ipotesi, l'ultimo che sceglie è S (perché il primo e l'ultimo quantificatore sono \exists), quindi D sceglie un vertice vuoto e S proseguirà nella seconda parte del gioco.

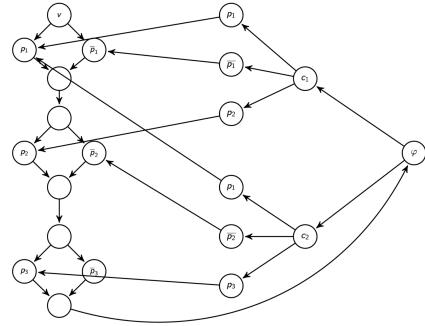
Quindi S sceglie le lettere proposizionale 1, 3, 5, 7, ..., n e D sceglie le altre.

Il resto della costruzione dipende da come è fatta la formula di partenza.

$$\phi \equiv \exists p_1 \forall p_2 \exists p_3 ((p_1 \vee \neg p_1 \vee p_2) \wedge (p_1 \vee \neg p_2 \vee p_3))$$

Due clausole → due gadget

Per ogni scelta di p_2 da parte di D , S vince.



Esistono molti problemi interessanti che possono essere risolti utilizzando solo una quantità di spazio logaritmica rispetto alla lunghezza dell'input, portando alle definizioni:

$LOG = \bigcup_k SPACE(k \cdot \log(n))$ è la classe di tutti e soli i problemi che possono essere risolti con macchine di Turing deterministiche usando spazio logaritmico

$NLOG = \bigcup_k NSPACE(k \cdot \log(n))$ è la classe di tutti e soli i problemi che possono essere risolti con macchine di Turing anche non-deterministiche usando spazio logaritmico.

Consideriamo un problema semplice come quello di decidere il linguaggio $L = \{a^n b^n\}$

Sappiamo già che $L \in P$. Ma possiamo dire qualcosa di più stretto?

Consideriamo la seguente macchina. Per una certa entrata w :

1. Se w non è del tipo $a^n b^n$, rifiuta
2. Azzera un contatore sul nastro di lavoro
3. Conta le a usando il contatore, e controlla se corrisponde al numero di b . In questo caso, accetta;
4. Rifiuta.

Lo spazio usato dal contatore è logaritmico in n , e quindi in $|w|$. Pertanto $L \in LOG$.

$$PATH = \{\langle G, s, t \rangle | G \text{ è diretto e } s \rightsquigarrow t\}$$

$$PATH \in NLOG$$

La macchina che normalmente usiamo per mostrare che $PATH \in P$ non è adatta a questa dimostrazione.

Consideriamo una entrata $w = \langle G, s, t \rangle$, e usiamo la seguente macchina:

1. Se w non è del tipo $\langle G, s, t \rangle$, rifiuta
2. Azzera un contatore sul nastro di lavoro, e considera il vertice s
3. Finché il contatore non è ancora arrivato a $|V|$, scegli non deterministicamente un vertice mai utilizzato e raggiungibile dal vertice corrente, e somma uno a k
4. Se t è stato raggiunto, allora accetta, altrimenti rifiuta.

L'essenza della dimostrazione è che lo spazio usato dal contatore più quello utilizzato per mantenere un vertice corrente è logaritmico in $|G|$.

$$LOG \subseteq NLOG$$

Ogni macchina di Turing deterministica che usa solamente spazio logaritmico è anche una macchina non-deterministica che non usa il non-determinismo.

Come nel caso delle classi P e NP , non è nota la relazione esatta tra LOG e $NLOG$; si crede che siano diverse.

Nella classificazione rispetto alla risorsa spazio usiamo concetti simili a quelli già visti, tra cui la riduzione tra problemi. A differenza degli altri casi, però, dobbiamo usare una riduzione che preservi lo spazio utilizzato, e la riduzione polinomica (in tempo) lo fa solo per spazi polinomici. Introduciamo allora il concetto di riduzione logaritmica.

Per alfabeti Σ_1 , Σ_2 dati, e per linguaggi $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$, diremo che L_1 si riduce logaritmicamente a L_2 se esiste una funzione $f : L_1 \rightarrow L_2$ tale che, per ogni parola w , $w \in L_1$ se e solo se $f(w) \in L_2$, questa funzione è ricorsiva, e calcolarla è un problema in LOG (quindi la macchina che la calcola usa solo spazio logaritmico).

Questa forma di riduzione si denota con $L_1 \leq_l L_2$, e specializza le riduzioni many-one e polinomica. La macchina che implementa f in questo caso si conosce in letteratura come log-space transducer.

Siano L_1, L_2 due linguaggi tali che $L_1 \leq_l L_2$. Allora, se $L_2 \in LOG$, allora $L_1 \in LOG$

Come per le altre riduzioni, assumiamo che $L_1 \leq_l L_2$ e $L_2 \in LOG$. Poiché la riduzione è di tipo \leq_l , esiste una macchina di Turing deterministica M_f che la calcola, e che usa solo spazio logaritmico. Dato che L_2 è in P , esiste una macchina di Turing deterministica M_{L_2} che lo riconosce in spazio logaritmico.

Costruiamo una macchina di Turing per riconoscere L_1 che usa solo spazio di lavoro logaritmico. Con input w :

1. Finchè non puoi accettare o riutare, ripeti: tenendo traccia dello stato corrente di M_{L_2} e della posizione sul nastro di entrata alla quale sarebbe M_{L_2} , ricalcola il giusto elemento di $f(w)$ a partire da w
2. Esegui il prossimo passo di M_{L_2} .

$PATH$ è $NLOG$ -completo.

Sappiamo già che $PATH$ è $NLOG$.

Per l'altro lato della dimostrazione dobbiamo costruire un analogo del Teorema di Cook-Levin: data una macchina di Turing M , anche non deterministica, che usa al massimo spazio logaritmico di lavoro, vogliamo formulare il problema di stabilire se accetta o meno, un certo input come una istanza di $PATH$.

Una configurazione di una macchina logspace è descrivibile con spazio logaritmico. Infatti, per descrivere una configurazione, ho bisogno di: posizione della testina di lettura sul nastro di entrata (un contatore, spazio logaritmico), carattere letto, stato del nastro di lavoro (che è limitato a spazio logaritmico).

Essenzialmente, il problema $PATH$ è un problema di raggiungibilità tra un vertice ed un altro. Diventa immediato convincersi che chiedersi se M termina consiste nel eseguire la seguente istanza di $PATH$:

1. Genera, usando spazio logaritmico, la configurazione C_0 iniziale di M con il suo input;
2. Finchè non si è raggiunta una configurazione accettante, genera non deterministicamente una nuova configurazione C_j e controlla che dalla configurazione attuale C_i si possa raggiungere in un passo la configurazione C_j ;
3. Se si è raggiunti una configurazione accettante, accetta, altrimenti rifiuta.

$PATH \in CoNLOG$, dunque $NLOG = CoNLOG$.

$NLOG \subseteq P$

Abbiamo appena dimostrato che $PATH$ è $NLOG$ -completo, pertanto ogni problema $L \in NLOG$ è tale che $L \leq_l PATH$. Ma questo significa anche che $L \leq_p PATH$.

Poiché, tra le altre cose, $PATH \in P$, questo, per il teorema sulla riduzione polinomica, implica che $L \in P$.

Sebbene abbiano già il nostro primo problema $NLOG$ -completo, per completezza di esposizione è corretto menzionare la sua controparte logica. Ricorderemo che $kSAT$ è NP -completo per ogni $k \geq 3$. Ma il seguente teorema mostra un aspetto sorprendente del ragionamento logico clausale.

$2SAT$ è $NLOG$ -completo.

Consideriamo la seguente macchina di Turing non-deterministica con input ϕ scritta in forma clausale implicativa, dove ogni clausola non ha più di due elementi.

1. Scegli non-deterministicamente, usando spazio logaritmico, una lettera p e la sua negazione;
2. Controlla, usando la tecnica del forward-checking, che da p si 'arrivi' a $\neg p$, e che da $\neg p$ si 'arrivi' a p , e in questo caso accetta, altrimenti rifiuta.

$PATH \leq_l 2SAT$: Consideriamo un grafo $G = (V, E)$ qualsiasi, e due vertici s, t su di esso. Usando spazio polinomiale possiamo trasformare ogni vertice v in una lettera proposizionale p_v , ed ogni arco (v, u) in una clausola $(p_v \rightarrow p_u)$. A questo punto, prendiamo la congiunzione di tutte le clausole ottenute, aggiungendo le clausole unitarie p_s e $\neg p_v$. Questa formula è soddisfacibile se e solo se in G da s si raggiunge t .

$$NLOG \subseteq P$$

Anche in questo caso non sappiamo se la relazione è, o meno, stretta.

$$HornSAT \text{ è } P\text{-completo}$$

Dobbiamo ricordare la dualità classica che conosciamo tra problemi di soddisfacibilità e problemi di validità.

Sappiamo che questi due concetti sono complementari e possiamo denire l'insieme $\overline{HornSAT}$ che è isomorfo a $HornVAL$. Nel caso di problemi in P (e sappiamo già che $HornSAT \in P$), dobbiamo ricordare che P è chiuso per complemento. Pertanto la dualità in questo caso è insostanziale e i due problemi sono, quindi, lo stesso problema.

Seguendo lo schema del Teorema di Cook-Levin, sappiamo che se $L \in P$ allora $L \leq_p \overline{HornSAT}$. Questo può essere fatto riprendendo la dimostrazione del problema originale, con due varianti: primo, quando codifichiamo δ , la disgiunzione è composta da un solo elemento perchè la macchina è deterministica (e quindi le formule relative diventano di Horn), e, secondo, la formula ϕ_{fin} diventa: $\phi_{fin} \equiv \bigwedge_{0 \leq g \leq n^k} \neg q_{j,a}$

