Appunti sulla teoria degli automi, dei linguaggi e della calcolabilità

Marco Liverani*

Ottobre 2005

1 Introduzione

Alla base dell'Informatica, a fondamento della teoria su cui viene costruita la cosiddetta "scienza dell'informazione", viene posta la teoria della *calcolabilità*, ossia lo studio di ciò che è calcolabile sulla base di un procedimento automatico, di un algoritmo diremmo utilizzando un termine appena più preciso. Di conseguenza la teoria della calcolabilità studia anche ciò che calcolabile non è, ciò che con un procedimento automatico non può essere calcolato in un tempo finito, ossia ciò che può essere calcolato solo astrattamente impiegando un tempo enorme, che nessuno sarebbe mai disposto ad aspettare, ma anche ciò di cui non si conosce il limite temporale entro cui potremmo conoscere la soluzione esatta.

Ci si scontra così immediatamente con i limiti estremi dell'Informatica, la scienza che si occupa di definire metodi e tecniche per il calcolo e il trattamento automatico delle informazioni. Studiando ciò che è calcolabile e ciò che non lo è viene naturale analizzare più astrattamente l'universo dei problemi, per verificare se esiste una gerarchia nel grado di difficoltà dei problemi che possono essere affrontati e risolti (ossia la cui soluzione possa essere calcolata) mediante un algoritmo. In altre parole viene naturale chiedersi se esiste la possibilità di stabilire se un determinato problema è oggettivamente più arduo di un altro, al di là dell'opinione personale di chi tali problemi sta cercando di risolverli. Emerge così una teoria della *complessità* dei problemi e degli algoritmi mediante cui siamo in grado di risolverli.

Questa breve nota presenta degli appunti estremamente sintetici e schematici sulla teoria degli automi e della calcolabilità. Sono solo alcuni spunti che possono essere utili a dare un'idea di alcuni dei temi trattati da questa parte fondante dell'Informatica. Per il necessario completamento ed approfondimento degli argomenti che in queste pagine sono soltanto sfiorati, si rimanda ai testi citati nelle note bibliografiche. In particolare è bene precisare che l'ordine con cui sono trattati ed introdotti i diversi argomenti, riflette in modo sostanziale l'impostazione dell'ottimo testo di Hopcroft, Motwani e Ullman [3], ricco di numerosissimi esempi ed esercizi.

*E-mail: liverani@mat.uniroma3.it – http://www.mat.uniroma3.it/users/liverani/

2 Automi a stati finiti

Un **alfabeto** è un insieme finito e non vuoto di simboli; una **stringa** è una sequenza finita di simboli di un alfabeto. Un **linguaggio** è un insieme anche infinito di stringhe costruite su uno stesso alfabeto. Spesso si può confondere il concetto di linguaggio con quello di problema, visto che in teoria degli automi un problema è costituito dalla questione di decidere se una certa stringa appartiene o meno ad un determinato linguaggio.

Un **automa a stati finiti deterministico** (*Deterministic Finite Automata*, DFA) è un oggetto costituito dai seguenti elementi:

- un insieme finito di *stati* Q;
- un insieme finito di *simboli di input* Σ;
- una *funzione di transizione* che applicata ad un simbolo e ad uno stato restituisce uno stato: $\delta : \Sigma \times Q \to Q$;
- uno *stato iniziale* q_0 scelto fra gli stati di Q;
- un insieme di *stati finali* $F \subseteq Q$.

Dunque un DFA è una quintupla: $A = (Q, \Sigma, \delta, q_0, F)$. Un DFA può essere espresso elencando le sue componenti e descrivendo la funzione di transizione, oppure mediante un grafo detto *diagramma delle transizioni di stato* o mediante una *matrice di transizione*.

Dato un automa a stati finiti deterministico questo definisce un linguaggio sull'alfabeto Σ, costituito da tutte le stringhe ottenute concatenando le etichette presenti sul grafo delle transizioni sugli spigoli dei cammini che vanno dallo stato iniziale ad uno degli stati accettanti. In altre parole il linguaggio L su Σ definito da $A = (Q, \Sigma, \delta, q_0, F)$ è dato da tutte le stringhe che producono una sequenza di transizioni dallo stato iniziale q_0 ad uno degli stati accettanti di F.

Con $\hat{\delta}$ indichiamo la *funzione di transizione estesa,* che descrive cosa succede (in quale stato si arriva) quando, a partire da un certo stato, l'automa elabora non un solo simbolo, ma una stringa costituita da una sequenza di simboli. Possiamo definire ricorsivamente la funzione di transizione estesa (con ε indicheremo d'ora in avanti la stringa vuota):

$$\begin{cases} \hat{\delta}(q, \varepsilon) = q \\ \hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a), \text{ con } wa \neq \varepsilon \end{cases}$$

Allora il linguaggio di un automa a stati finiti deterministico $A = (\Sigma, Q, \delta, q_0, F)$ è definito come $L(A) = \{w : \hat{\delta}(q_0, w) \in F\}$. I linguaggio $L = L(A)$ per qualche automa a stati finiti deterministico A si chiamano **linguaggi regolari**.

Un **automa a stati finiti non deterministico** (NFA, *Nondeterministic Finite Automata*) è un automa che può trovarsi contemporaneamente in diversi stati. Anche un NFA si definisce come una quintupla $A = (\Sigma, Q, \delta, q_0, F)$, dove:

- Σ è l'alfabeto finito di simboli;

- Q è l'insieme finito di stati;
- $q_0 \in Q$ è lo stato iniziale;
- $F \subseteq Q$ è l'insieme degli stati finali accettanti;
- δ è una funzione $\delta : Q \times \Sigma \to P(Q)$,[1] che a fronte di un simbolo e di uno stato restituisce un sottoinsieme degli stati (dunque anche più di uno stato).

Anche in questo caso si può definire la funzione di transizione estesa $\hat{\delta}$, che a fronte di una stringa restituisce l'insieme degli stati in cui l'automa può terminare la valutazione della stringa stessa.

Un NFA accetta una stringa w quando $\hat{\delta}(q_0, w) \cap F \neq \emptyset$, ossia quando la valutazione della stringa termina in almeno uno stato accettante. Quindi, se $A = (Q, \Sigma, \delta, q_0, F)$ è un NFA, allora $L(A) = \{w : \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ è il linguaggio definito da quel NFA.

Per ogni automa a stati finiti non deterministico $A_N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ è possibile costruire un automa a stati finiti deterministico $A_D = (Q_D, \Sigma, \delta_D, q_0, F_D)$, in grado di accettare lo stesso linguaggio di A_N.

A_D infatti può essere costruito scegliendo $Q_D = P(Q_N)$, ossia gli stati di A_D sono costituiti da tutti i sottoinsiemi degli stati di A_N. Quindi se $S \in Q_D$ è $S \subseteq Q_N$, definiamo $\delta_D(S, a) = \cup_{q \in S} \delta_N(q, a)$; inoltre F_D è l'insieme dei sottoinsiemi di Q_N che contengono almeno uno stato in F_N: $F_D = \{S \subseteq Q_N : S \cap F_N \neq \emptyset\}$. Osserviamo che se A_N ha n stati ($|Q_N| = n$) allora A_D avrà al più 2^n stati. Questo significa che l'elaborazione di una stessa stringa accettata da un NFA, richiede un numero di passi esponenzialmente maggiore per il DFA corrispondente.

Dato un automa a stati finiti deterministico $A_D = \{Q_D, \Sigma, \delta_D, q_0, F_D\}$ è possibile costruire un automa a stati finiti non deterministico $A_N = \{Q_N, \Sigma, \delta_N, q_0, F_N\}$ equivalente, cioè che accetta lo stesso linguaggio, ponendo $Q_N = Q_D$ e se $\delta_D(q, a) = p$ allora $\delta_N(q, a) = \{p\}$.

Teorema 1 *Un linguaggio L è accettato da un automa a stati finiti non deterministico se e solo se è accettato da un automa a stati finiti deterministico.*

Un **ε-NFA** è un automa a stati finiti non deterministico che esegue transizioni anche a fronte della stringa vuota ε. Gli ε-NFA non ampliano la classe dei linguaggi accettati da automi a stati finiti: se L è accettato da un ε-NFA, allora è possibile costruire un NFA e un DFA che accettano L e che quindi sono equivalenti all'ε-NFA.

Un ε-NFA si definisce esattamente come un NFA, salvo che per la funzione di transizione δ che accetta come argomenti uno stato di Q ed un simbolo in $\Sigma \cup \{\varepsilon\}$.

3 Espressioni regolari

È possibile definire tre operazioni fondamentali sui linguaggi:

Unione: dati due linguaggi L e M si definisce il linguaggio $L \cup M$ come l'insieme delle stringhe di L o di M o di entrambi.

[1] Con $P(Q)$ indichiamo l'insieme delle parti di Q, costituito da tutti i sottoinsiemi di Q.

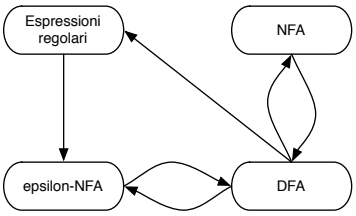

Figura 1: Schema delle dimostrazioni di equivalenza tra automi a stati finiti ed espressioni regolari

Concatenazione: dati due linguaggi L e M si definisce il linguaggio LM come l'insieme costituito da stringhe ottenute giustapponendo una stringa di L e una di M. La stringa nulla ε è l'identità per la concatenazione: $L\{\varepsilon\} = \{\varepsilon\}L = L$.

Chiusura: dato il linguaggio L, la chiusura L^* è l'insieme di stringhe ottenute concatenando un numero arbitrario di stringhe di L.

Le **espressioni regolari** sono delle espressioni algebriche costruite utilizzando i tre operatori su simboli e linguaggi per descrivere altri linguaggi. Se E è un'espressione regolare, indichiamo con $L(E)$ il linguaggio che è descritto e definito da E. Le espressioni regolari possono essere definite ricorsivamente:

1. Per ogni $x \in \Sigma$, x è un'espressione regolare che rappresenta il linguaggio $\{x\}$. Anche ε e $\emptyset$ sono espressioni regolari.

2. Se A e B sono espressioni regolari, allora $A + B$ è un'espressione regolare e rappresenta il linguaggio $L(A) \cup L(B)$.

3. Se A e B sono espressioni regolari, allora anche AB lo è; rappresenta il linguaggio $L(AB) = L(A)L(B)$.

4. Se A è un'espressione regolare, allora anche A^* lo è; rappresenta il linguaggio $L(A^*) = (L(A))^*$.

Le espressioni regolari costituiscono uno strumento per la definizione di linguaggi, equivalente agli automi a stati finiti: consentono di descrivere tutti e soli i linguaggi che è possibile definire con i tre tipi di automa a stati finiti (DFA, NFA, ε-NFA). Tali linguaggi sono chiamati **linguaggi regolari**.

Il diagramma in Figura 1 evidenzia le dimostrazioni di equivalenza tra espressioni regolari ed automi a stati finiti.

Due espressioni regolari sono *equivalenti* se generano lo stesso linguaggio. Dunque è opportuno conoscere alcune proprietà delle espressioni regolari che ci permettono di stabilire facilmente se due espressioni sono equivalenti e se un'espressione può quindi essere semplificata riconducendola ad un'altra, più semplice, ma equivalente.

Proprietà commutativa: se A e B sono espressioni regolari, allora $A + B = B + A$; infatti $L(A + B) = L(A) \cup L(B) = L(B) \cup L(A)$.

Proprietà associativa dell'unione: se A, B e C sono espressioni regolari, allora $(A + B) + C = A + (B + C)$.

Proprietà associativa della concatenazione: se A, B e C sono espressioni regolari allora $(AB)C = A(BC)$.

Identità per l'unione: $A + \emptyset = \emptyset + A = A$.

Identità per la concatenazione: $A\varepsilon = \varepsilon A = A$.

Annichilitore per la concatenazione: $A\emptyset = \emptyset A = \emptyset$.

Proprietà distributiva della concatenazione sull'unione: $A(B+C) = AB+AC$, $(B+C)A = BA+CA$.

Proprietà di idempotenza per l'unione: $A + A = A$.

Proprietà della chiusura: $(A^*)^* = A^*$, $\emptyset^* = \varepsilon$, $\varepsilon^* = \varepsilon$, $A^+ = AA^* = A^*A$, $A^* = A^+ + \varepsilon$, $A? := A + \varepsilon$ (quest'ultima è la definizione dell'operatore "?").

Osserviamo che la proprietà commutativa non vale per la concatenazione: in generale se A e B sono espressioni regolari $AB \neq BA$.

4 Proprietà dei linguaggi regolari

Per stabilire se un linguaggio L è regolare esiste uno strumento molto potente che definisce una condizione necessaria per la regolarità di un linguaggio: il *pumping lemma*.

Teorema 2 (Pumping Lemma) *Sia L un linguaggio regolare. Allora esiste una costante n dipendente da L tale che, per ogni $w \in L$ con $|w| \geq n$, esistono tre stringhe x, y e z tali che $w = xyz$ e:*

1. $y \neq \varepsilon$;
2. $|xy| \leq n$;
3. $\forall\ k \geq 0\ xy^kz \in L$.

Il *pumping lemma* afferma che se L è regolare, allora è sempre possibile trovare una stringa y, abbastanza vicina all'inizio di $w \in L$, tale da poter essere replicata più volte ($k > 1$) o da essere cancellata ($k = 0$), ottenendo comunque parole in L.

Il *pumping lemma* è quindi molto utile per provare (trovando un controesempio) che un linguaggio non è regolare.

Di fatto, rileggendo in modo informale il *pumping lemma*, si può dire che i linguaggi regolari sono quelli le cui stringhe non richiedono molta "memoria" (gli stati di un automa riconoscitore) per tenere traccia della struttura con cui i simboli già

analizzati si sono susseguiti. In questo senso, la presenza di un pattern y di lunghezza finita, minore di una certa costante n valida per ogni parola del linguaggio, che si ripete più volte, garantisce la finitezza del numero di stati necessari per riconoscere le parole del linguaggio e la possibilità di "riusare" gli stati utilizzati per riconoscere il pattern y.

È possibile provare numerose **proprietà di chiusura** per la classe dei linguaggi regolari. Tali proprietà sono ancora una volta uno strumento utile per riconoscere se un determinato linguaggio è o meno regolare.

Siano A e B due linguaggi regolari sull'alfabeto Σ; allora valgono le seguenti proprietà:

1. $A \cup B = \{w : w \in A \text{ o } w \in B\}$ è regolare.
2. $A \cap B = \{w : w \in A \text{ e } w \in B\}$ è regolare.
3. $\overline{A} = \{w \in \Sigma^* : w \notin A\}$ (complemento) è regolare.
4. $A - B = \{w : w \in A \text{ e } w \notin B\}$ è regolare.
5. $A^R = \{w = a_1 a_2 \dots a_n : w^R = a_n a_{n-1} \dots a_1 \in A\}$ (inverso) è regolare.
6. A^* (chiusura o *star di Kleene*) è regolare.
7. $AB = \{w : w = w_1 w_2, w_1 \in A, w_2 \in B\}$ (concatenazione) è regolare.
8. $H(A)$, l'omomorfismo H su A (sostituzione di simboli con stringhe), è regolare.
9. $H^{-1}(A)$, l'omomorfismo inverso (sostituzione di stringhe con simboli singoli), è regolare.

5 Grammatiche libere dal contesto

La classe dei **linguaggi liberi dal contesto** estende quella dei linguaggi regolari; tali linguaggi sono definiti dalle *grammatiche libere dal contesto* che descrivono le regole con cui possono essere derivate tutte le stringhe che appartengono al linguaggio.

Una **grammatica libera dal contesto** è una quaterna $G = (V, T, P, S)$, dove V è un insieme finito che rappresenta i *simboli non terminali* o le *variabili*, T è l'insieme finito dei *simboli terminali* o *alfabeto*, P è l'insieme delle *produzioni*, ossia delle regole che danno la definizione ricorsiva del linguaggio, ed S è il *simbolo iniziale* che rappresenta il linguaggio da definire.

Se esiste una regola di inferenza (produzione) che è in grado di trasformare la stringa A (composta di simboli terminali e non terminali) in B, allora si scrive $A \Rightarrow_G B$. Se esiste una successione di produzioni di G che trasformano A in B allora si scriverà $A \overset{*}{\Rightarrow}_G B$.

Data una grammatica libera dal contesto $G = (V, T, P, S)$ questa definisce il *linguaggio libero dal contesto* $L(G) = \{w \in T^* : S \overset{*}{\Rightarrow}_G w\}$.

È possibile rappresentare le derivazioni di una grammatica libera dal contesto mediante **alberi sintattici**. Un albero sintattico è un albero definito come segue:

1. ogni vertice "interno" (un vertice che non sia una foglia dell'albero) è etichettato con una variabile di V;

2. le foglie sono etichettate con variabili, simboli terminali o ε. Se una foglia è etichettata con ε, allora deve essere l'unico figlio del vertice padre;

3. se un nodo interno è etichettato A e i suoi figli sono etichettati (da sinistra a destra) con $X_1, X_2, \ldots, X_n$, allora $A \Rightarrow X_1 X_2 \ldots X_n$ è una produzione in P.

Alcune grammatiche libere dal contesto possono essere **grammatiche ambigue**: in questi casi una stringa terminale ammette più di un albero sintattico che la produce. Per alcune grammatiche ambigue è possibile trovare delle grammatiche non ambigue equivalenti, che producono lo stesso linguaggio. Altre invece sono intrinsecamente ambigue: generano un linguaggio che può essere prodotto solo da grammatiche ambigue.

6 Automi a pila

Un **automa a pila** è un automa a stati finiti non deterministico con ε-transisioni (un ε-NFA) e con una caratterizzazione aggiuntiva: una pila (*stack*) in cui è possibile memorizzare stringhe. La pila è una struttura di tipo "LIFO" (*last in first out*) e dunque l'automa può leggere (estrarre) o inserire informazioni sempre solo dalla cima dello *stack*.

Un automa a pila (PDA, *Pushdown Automaton*) è definito formalmente come una settupla: $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, dove:

- Q è un insieme finito di stati;
- Σ è un insieme finito di simboli in input (alfabeto);
- Γ è l'alfabeto (finito) dei simboli che possono essere inseriti nello *stack*;
- δ è la funzione di transizione $\delta : Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$, con $\delta(q, a, W) \mapsto (p, \gamma)$, con $p, q \in Q$, $a \in \Sigma$, $\gamma = XY \ldots Z$, con $W, X, Y, \ldots, Z \in \Gamma$;
- q_0 è lo stato iniziale ($q_0 \in Q$);
- Z_0 è il simbolo che inizialmente si trova nello *stack* ($Z_0 \in \Gamma$);
- F è l'insieme finito degli stati finali accettanti ($F \subseteq Q$).

Un automa a pila pu'*øaccettare* una stringa $w \in \Sigma^*$ venendosi a trovare in uno stato finale accettante al termine dell'elaborazione di w (accettazione per "stato accettante"). Però si può anche definire il linguaggio di un automa a pila P, $L(P)$, come l'insieme di tutte le parole $w \in \Sigma^*$ tali che al termine dell'elaborazione di ognuna l'automa si ritrova con la pila vuota (accettazione per "pila vuota").

Si può dimostrare che un linguaggio L è accettato per "stato finale" da un automa a pila P se e e solo se esiste un automa a pila P' che accetta L per "pila vuota". In

Figura 2: Rapporti di inclusione tra linguaggi

generale $P \neq P'$: fissato un automa a pila P, il linguaggio $L(P)$ accettato per "stato finale" da P è diverso dal linguaggio $N(P)$ accettato dallo stesso P per "pila vuota".

I linguaggi accettati dagli automi a pila (per stato finale o, equivalentemente, per pila vuota) sono tutti e soli i linguaggi liberi dal contesto (quelli generati da grammatiche libere dal contesto).

È possibile definire dei particolari automi a pila, detti **automi a pila deterministici** (DPDA, *Directed Pushdown Automaton*), ossia automi a pila $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tali che:

1. $\delta(q, a, X)$ ha al massimo un elemento $\forall\ q \in Q$, $\forall\ a \in \Sigma$, o $a = \varepsilon$, $\forall\ X \in \Gamma$;
2. se $\delta(q, a, X)$ non è vuoto per un $a \in \Sigma$, allora $\delta(q, \varepsilon, X)$ deve essere vuoto.

I linguaggi accettati per "stato finale" dagli automi a pila deterministici, includono tutti i linguaggi regolari, ma sono inclusi nei linguaggi liberi dal contesto. I linguaggi accettati dagli automi a pila deterministici sono liberi dal contesto ed hanno tutti grammatiche non ambigue.

7 Proprietà dei linguaggi liberi dal contesto

Un simbolo $X \in \Sigma$ è *utile* per un certo linguaggio L se $S \overset{*}{\Rightarrow} aXb \overset{*}{\Rightarrow} w \in L$; se non è utile si dice che è *inutile*. Dato un linguaggio si può ottenere un linguaggio equivalente eliminando da Σ tutti i simboli inutili.

Una grammatica è in **forma normale di Chomsky** (CNF, *Chomsky Normal Form*) se non ha simboli inutili e se le sue produzioni sono solo di questi due tipi:

1. $A \Rightarrow a$, con A variabile e a terminale;
2. $A \Rightarrow BC$, con A, B e C variabili.

Teorema 3 *Se G è una grammatica libera dal contesto il cui linguaggio $L(G)$ contiene almeno una stringa $w \neq \varepsilon$, allora esiste una grammatica G_1 in forma normale di Chomsky tale che $L(G_1) = L(G) - \{\varepsilon\}$.*

Il teorema afferma che tutti i linguaggi liberi dal contesto privi di "ε-produzioni" possono essere generati da grammatiche in forma normale di Chomsky.

Teorema 4 (Pumping lemma per linguaggi liberi dal contesto) *Sia L un linguaggio libero dal contesto. Allora esiste $n > 0$ costante tale che se $z \in L$ e $|z| \geq n$ allora $z = uvwxy$ con le seguenti condizioni:*

1. $|vwx| \leq n$;
2. $vx \neq \varepsilon$ *(almeno una delle stringhe v e x non deve essere vuota);*
3. $\forall\ i \geq 0$ *risulta* $uv^i wx^i y \in L$.

Valgono le seguenti proprietà di chiusura per i linguaggi liberi dal contesto:

Unione: se $L_1, L_2 \in$ CFL allora $L_1 \cup L_2 \in$ CFL.

Concatenazione: se $L_1, L_2 \in$ CFL allora $L_1 L_2 \in$ CFL.

Chiusura: se $L_1 \in$ CFL allora $L_1^* \in$ CFL e $L_1^+ \in$ CFL.

Omomorfismo: se $L_1 \in$ CFL allora $H(L_1) \in$ CFL (H omomorfismo).

Inversione: se $L_1 \in$ CFL allora $L_1^R \in$ CFL.

Intersezione con altri linguaggi regolari: se $L \in$ CFL e R è un linguaggio regolare, allora $L \cap R \in$ CFL.

Data una grammatica libera dal contesto esiste un algoritmo di complessità lineare per verificare se genera almeno una stringa o se il linguaggio generato dalla grammatica è vuoto.

Utilizzando un algoritmo basato sulla tecnica della programmazione dinamica è possibile stabilire con una complessità di $O(n^3)$ se una certa stringa di lunghezza n appartiene ad un linguaggio libero dal contesto.

8 Macchine di Turing

Una **macchina di Turing** è un modello di calcolo astratto, definito negli anni '30 dal matematico inglese Alan Turing. Si può dimostrare che, sebbene la macchina di Turing sia un modello assai elementare, è dotato di una potenza di calcolo equivalente a quella di un computer. L'*ipotesi di Church* (o *tesi di Church-Turing*) afferma che le macchine di Turing sono in grado di calcolare tutte e sole le funzioni che sono effettivamente calcolabili (le *funzioni ricorsive parziali*), essendo in questo del tutto equivalenti ai moderni computer.

Una macchina di Turing è costituita da una unità di controllo, che può memorizzare un numero finito di stati, una testina mobile che può leggere e scrivere (e spostarsi a destra e a sinistra) su un nastro suddiviso in infinite celle. Ogni cella del nastro può contenere il simbolo nullo (*blank*) o un carattere preso dall'alfabeto finito dei simboli ammissibili Σ.

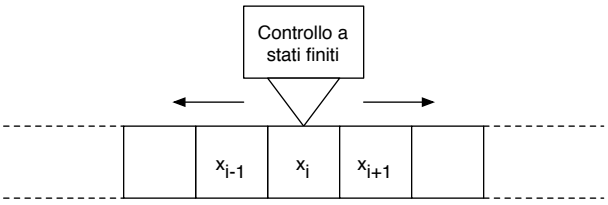

Figura 3: Schematizzazione di una macchina di Turing

Inizialmente il nastro contiene l'input fornito alla macchina per definire una specifica istanza del problema che si deve risolvere. La stringa (finita) di input è delimitata da simboli vuoti (*blank*). La macchina di Turing reagisce al simbolo letto nella cella del nastro infinito su cui è collocata la testina di lettura e scrittura, scrivendo al suo posto un altro simbolo (o lasciando invariato il simbolo presente), cambiando di stato (o rimanendo nello stato attuale) e spostandosi di una posizione verso sinistra o verso destra sul nastro (o rimanendo immobile).

Più formalmente una macchina di Turing può essere definita come una settupla $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, dove:

- Q è l'insieme finito degli *stati* del controllo;
- Σ è l'insieme finito di *simboli di input*;
- Γ è l'*alfabeto* dei simboli che possono essere scritti sul nastro; dunque $\Sigma \subseteq \Gamma$;
- δ è la *funzione di transizione* $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times D$ che associa ad una coppia (q, x) costituita da uno stato e da un simbolo di Γ, una tripla costituita dal nuovo stato assunto dalla macchina, dal simbolo scritto sul nastro al posto di x e da una direzione (D = {destra, sinistra, fermo}) con cui la testina si è mossa sul nastro dopo aver effettuato l'operazione di scrittura;
- q_0 è lo *stato iniziale*;
- B è il simbolo con cui si rappresenta il *carattere vuoto* (*blank*; $B \in \Gamma$);
- F è l'insieme degli *stati finali accettanti* ($F \subseteq Q$).

I linguaggi accettati da una macchina di Turing sono denominati **linguaggi ricorsivamente enumerabili**.

È possibile estendere la macchina di Turing introducendo le *macchine di Turing multinastro*, dotate di più di un nastro infinito, tra loro indipendenti, e le *macchine di Turing non deterministiche*, ossia con la capacità di passare da una certa configurazione (stato/simbolo) a più configurazioni (stato/simbolo/movimento) contemporaneamente. Si può dimostrare che queste macchine estese non sono più potenti della macchina di Turing deterministica con un solo nastro: accettano tutte lo stesso tipo di linguaggio, i linguaggi ricorsivamente enumerabili.

È possibile dimostrare la possibilità (teorica) di simulare il comportamento di una macchina di Turing con un computer; il limite sta nel fatto che la capacità di memoria di un computer è finita, mentre il nastro della macchina di Turing è di lunghezza infinita. Tuttavia si può sempre immaginare di aggiungere unità di memorizzazione (dischi) a dismisura, man mano che aumenta la necessità di memorizzare simboli per simulare il comportamento di una macchina di Turing.

Si può anche dimostrare la possibilità di simulare il funzionamento di un computer con una macchina di Turing.

Possiamo quindi essere sicuri che un problema non risolubile con una macchina di Turing non è risolubile neanche con un computer.

9 Problemi decidibili e indecidibili

È possibile costruire una codifica per rappresentare come una stringa binaria ogni macchina di Turing con alfabeto di input $\Sigma = \{0,1\}$. Per far questo si stabilisce una numerazione degli r stati della macchina, $Q = \{q_1, q_2, \ldots, q_r\}$, in cui q_1 rappresenta sempre lo stato iniziale e q_2 è sempre lo stato accettante; si enumerano i simboli dell'alfabeto $\Gamma = \{X_1, X_2, \ldots, X_s\}$ dei simboli che possono trovarsi sul nastro, stabilendo che $X_1 = 0$, $X_2 = 1$ e $X_3 =$ *blank*; si stabilisce infine che le due direzioni in cui si può muovere la testina sono indicate con $D_1 =$ sinistra e $D_2 =$ destra.

Queste convenzioni non limitano la possibilità di definire macchine di Turing. È chiaro che modificando la numerazione degli stati di Q e dei simboli di Γ, ad una stessa macchina di Turing possono corrispondere codifiche diverse.

La codifica binaria di una macchina di Turing con alfabeto di input $\Sigma = \{0,1\}$ è completamente determinata dalla codifica delle sue regole di transizione: $\delta(q_i, X_j) = (q_k, X_l, D_m)$, con $q_i, q_k \in Q$, $X_j, X_l \in \Gamma$ e $D_m \in \{\text{sinistra, destra}\}$ è codificata dalla stringa $0^i 10^j 10^k 10^l 10^m$. Dunque nella codifica di ogni singola regola di transizione (visto che $i, j, k, l, m > 0$) non compaiono mai due 1 di seguito. Allora la codifica della macchina di Turing può essere ottenuta concatenando le codifiche delle regole di transizione, separate dalla sequenza 11.

In questo modo abbiamo ottenuto una enumerazione di tutte le macchine di Turing, creando un ordinamento delle stringhe binarie basato sulla lunghezza e, in seconda battuta, sull'ordine lessicografico delle stringhe con la stessa lunghezza: $1 < 01 < 10 < 001$. La macchina di Turing i-esima, M_i, è quella che corrisponde alla i-esima stringa binaria w_i. Non tutte le stringhe binarie rappresentano macchine di Turing: in tali casi associamo quelle stringhe, per convenzione, alla macchina di Turing con un solo stato e nessuna transizione.

Definiamo il **linguaggio di diagonalizzazione** L_d come l'insieme delle stringhe binarie w_i tali che M_i non accetta w_i (in altri termini $w_i \notin L(M_i)$).

Teorema 5 *L_d non è un linguaggio ricorsivamente enumerabile: non esiste alcuna macchina di Turing che accetta L_d.*

Definiamo i **linguaggi ricorsivi** come quei linguaggi L tali che se $L = L(M)$ per una certa macchina di Turing M, allora se $w \in L$ la macchina M la accetta in tem-

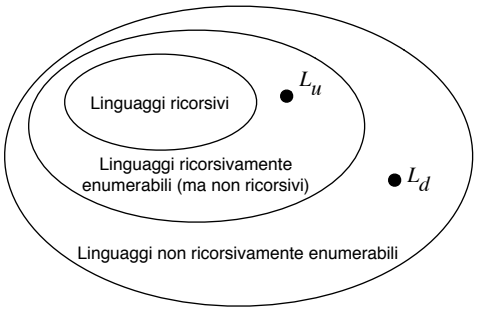

Figura 4: Linguaggi ricorsivi e non ricorsivi

po finito e se $w \notin L$ allora M si arresta comunque dopo un numero finito di passi, terminando l'elaborazione in uno stato non accettante.

Le macchine di Turing di questo tipo corrispondono a ciò che noi intendiamo per *algoritmo*: una procedura finita, che termina comunque dopo un numero finito di passi. I linguaggi ricorsivi, interpretati come problemi, sono detti **decidibili**, mentre i linguaggi non ricorsivi corrispondono ai problemi **indecidibili**.

Possiamo quindi classificare i linguaggi in tre insiemi contenuti l'uno nell'altro:

Linguaggi ricorsivi: esiste una macchina di Turing in grado di decidere in tempo finito (in un numero finito di passi) se una stringa appartiene o meno al linguaggio.

Linguaggi ricorsivamente enumerabili: esiste una macchina di Turing in grado di accettare in un numero finito di passi le stringhe che appartengono al linguaggio (non si può stabilire a priori se la macchina terminerà l'eleborazione di una stringa che non appartiene al linguaggio, dopo un numero finito di passi).

Linguaggi non ricorsivamente enumerabili: come per il linguaggio di diagonalizzazione L_d non esiste una macchina di Turing che accetta le stringhe del linguaggio.

Teorema 6 *Se L è ricorsivo allora anche $\overline{L}$ è ricorsivo.*

La dimostrazione è semplice: se $L = L(M)$ basta costruire la macchina di Turing complementare $\overline{M}$ che accetta tutto ciò che M rifiuta.

Teorema 7 *Se L è ricorsivamente enumerabile e anche $\overline{L}$ è ricorsivamente enumerabile, allora L e $\overline{L}$ sono ricorsivi.*

Anche in questo caso la dimostrazione è piuttosto semplice: basta costruire una macchina di Turing M^* ottenuta componendo M e $\overline{M}$, in modo che ogni stringa sia accettata o rifiutata in un numero finito di passi; così si prova che L è ricorsivo e quindi, per il Teorema precedente, lo è anche $\overline{L}$.

Definiamo il **linguaggio universale** L_u come l'insieme delle stringhe binarie ottenute da (M, w), dove M è una macchina di Turing con alfabeto di input binario e w è una stringa accettata da M ($w \in L(M)$), concatenando la codifica di M con w.

L_u è ricorsivamente enumerabile, ossia esiste una macchina di Turing U tale che $L_u = L(U)$. Tuttavia L_u non è ricorsivo.

Un altro esempio famoso di linguaggio ricorsivamente enumerabile non ricorsivo è il linguaggio $H(M)$ definito dal **problema dell'arresto**: $H(M)$ è composto dalle parole w tali che M si arresta quando elabora w, indipendentemente dal fatto che w sia accettata oppure no. Il problema dell'arresto è costituito dalle coppie (M, w) tali che $w \in H(M)$.

Un problema P_1 si **riduce** ad un problema P_2 se esiste un algoritmo che traduce ogni istanza di P_1 in una istanza di P_2. Non è detto che ogni istanza di P_2 debba essere l'immagine della conversione di una istanza di P_1. Se il problema P_1 si riduce a P_2 allora P_2 è difficile almeno tanto quanto P_1: se so risolvere P_2 allora posso anche risolvere P_1. Al contrario se P_1 non è ricorsivo allora anche P_2 non può essere ricorsivo e se P_1 non è ricorsivamente enumerabile, allora non può esserlo neanche P_2.

Una proprietà per i linguaggi ricorsivamente enumerabili definisce un sottoinsieme di tali linguaggi (es.: il sottoinsieme dei linguaggi ricorsivamente enumerabili che godono della proprietà di essere liberi dal contesto). Una proprietà sui linguaggi ricorsivamente enumerabili è *banale* se definisce l'insieme vuoto o l'insieme di tutti i linguaggi ricorsivamente enumerabili. Se P è una proprietà, il linguaggio L_P è l'insieme dei codici delle macchine di turing M_i tali che $L(M_i) = P$.

Teorema 8 (Teorema di Rice) *Ogni proprietà P non banale dei linguaggi ricorsivamente enumerabili definisce un problema L_P indecidibile.*

Problema della corrispondenza di Post (CPC): date due liste con lo stesso numero di stringhe $A = (s_1, s_2, \ldots, s_k)$ e $B = (t_1, t_2, \ldots, t_k)$ è possibile scegliere la stessa sequenza di stringhe corrispondenti da A e da B e formare per concatenazione la stessa stringa? Ossia: esistono $i_1, i_2, \ldots, i_p$ tali che $s_{i_1} s_{i_2} \ldots s_{i_p} = t_{i_1} t_{i_2} \ldots t_{i_p}$?

Questo è un esempio di problema indecidibile ed è molto utile per dimostrare per riduzione l'indecidibilità di altri problemi, come ad esempio il problema di stabilire se una grammatica libera dal contesto sia ambigua oppure no.

10 Le classi di complessità $\mathscr{P}$ e $\mathscr{NP}$

Una macchina di Turing M ha complessità (in tempo) $T(n)$ se, a fronte di un input di lunghezza n, si arresta accettando o non accettando la strionga dopo non più di $T(n)$ mosse.

La **classe** $\mathscr{P}$ è costituita da tutti i linguaggi (o problemi) L per cui esiste un *polinomio* $T(n)$ tale che $L = L(M)$ per una macchina di Turing *deterministica* di complessità $T(n)$.

La **classe** $\mathscr{NP}$ è invece costituita da tutti i problemi (o linguaggi) L per i quali esiste una funzione polinomiale $T(n)$ tale che $L = L(M)$ per una macchina di Turing *non deterministica* di complessità $T(n)$.

Visto che le macchine di Turing deterministiche sono un caso speciale di quelle non deterministiche, allora risulta $\mathscr{P} \subseteq \mathscr{NP}$. Non si sa se $\mathscr{P} \neq \mathscr{NP}$: al momento è stato possibile individuare moltissimi problemi che sono in $\mathscr{NP}$, ma per i quali non si è riusciti a costruire una macchina di Turing deterministica in grado di risolverli in tempo polinomiale. Non si è riusciti neanche a provare che tale macchina di Turing deterministica non esiste. Dunque al momento si immagina che $\mathscr{P} \subset \mathscr{NP}$, ma senza poterlo provare.

Un problema P_1 può essere **ridotto in tempo polinomiale** al problema P_2 se esiste un algoritmo di complessità polinomiale in grado di trasformare ogni istanza di P_1 in una istanza di P_2. Se P_1 è riducibile in tempo polinomiale a P_2 allora:

- se $P_2 \in \mathscr{P}$ allora anche $P_1 \in \mathscr{P}$;
- se $P_1 \notin \mathscr{P}$ allora anche $P_2 \notin \mathscr{P}$.

La **classe dei problemi NP-completi** è costituita da quei linguaggi (problemi) L tali che:

1. $L \in \mathscr{NP}$;
2. $\forall\ L' \in \mathscr{NP}$ è possibile ridurre in tempo polinomiale L' a L.

Ovviamente per dimostrare che un problema $L \in \mathscr{NP}$ è NP-completo basta provare che *un* problema L' NP-completo è riducibile in tempo polinomiale a L. Infatti la composizione di due algoritmi di complessità polinomiale dà luogo ad un algoritmo di complessità polinomiale; dunque visto che $L' \in$ NP-completo allora $\forall\ L'' \in \mathscr{NP}$ esiste un algoritmo polinomiale A in grado di ridurre tutte le istanze di L'' ad istanze di L'. Siccome esiste un algoritmo B che trasforma tutte le istanze di L' in istanze di L in tempo polinomiale, allora per ogni algoritmo A posso costruire un algoritmo polinomiale $B \circ A$ ("B dopo A") per convertire le istanze di L'' in istanze di L (per ogni $L'' \in \mathscr{NP}$).

Teorema 9 *Se un problema NP-completo è in $\mathscr{P}$ allora $\mathscr{P} = \mathscr{NP}$.*

Dimostrazione Se L è NP-completo e $L \in \mathscr{P}$, allora (per la NP-completezza di L) ogni problema $L' \in \mathscr{NP}$ si riduce in tempo polinomiale ad L. Dunque per ogni $L' \in \mathscr{NP}$ posso comporre l'algoritmo risolutivo polinomiale per L con l'algoritmo di riduzione polinomiale da L' ad L, ottenendo un algoritmo risolutivo polinomiale per L'. □

La **classe dei problemi NP-hard** (NP-ardui) è costituita da quei problemi L tali che per ogni $L' \in \mathscr{NP}$ è possibile ridurre in tempo polinomiale L' ad L. A differenza dei problemi NP-completi, i problemi NP-hard non si sa se sono $\mathscr{NP}$. I problemi NP-hard sono considerati problemi *intrattabili*.

Il **problema SAT** è così definito: un'espressione booleana assegnata è soddisfacibile? Ossia è possibile assegnare dei valori booleani alle variabili dell'espressione in modo tale che questa risulti vera?

Teorema 10 (Teorema di Cook) *SAT è NP-completo.*

In un'espressione booleana una *clausola* è una disgiunzione logica (*or*) di uno o più letterali. Una espressione booleana è in *forma normale congiuntiva* (CNF) se è la congiunzione logica (*and*) di una o più clausole. Una espressione logica è in *k-forma normale congiuntiva* (k-CNF) se le clausole sono tutte costituite dalla disgiunzione (somma) di k letterali distinti.

Vediamo di seguito una breve rassegna di alcuni famosi problemi NP-completi.

CSAT: data un'espressione booleana in forma normale congiuntiva è soddisfacibile?

Si dimostra che CSAT è NP-completo riducendo SAT a CSAT in tempo polinomiale.

k-SAT: data un'espressione booleana in k-forma normale congiuntiva, è soddisfacibile?

Si dimostra che 3-SAT ($k = 3$) è NP-completo riducendo in tempo polinomiale CSAT a 3-SAT.

Independent set (IS): dato un grafo $G = (V, E)$, un *insieme indipendente* è un sottoinsieme di V tale che ogni coppia di vertici non sia adiacente. Il problema IS, dato un grafo G e una costante k, chiede di stabilire se esiste un sottoinsieme indipendente di G con almeno k vertici.

Si dimostra che IS è NP-completo riducendo in tempo polinomiale 3-SAT a IS.

Vertex cover (VC): dato un grafo $G = (V, E)$ una *copertura di vertici* è un sottoinsieme di V tale che contenga almeno uno degli estremi di ogni spigolo del grafo. Il problema VC, dato un grafo G e una costante $k < |V|$ chiede di stabilire se esiste una copertura di vertici con al più k elementi.

Si dimostra che VC è NP-completo riducendo in tempo polinomiale IS a VC.

Un *ciclo hamiltoniano* su un grafo G è un ciclo che passa per ogni vertice di G esattamente una volta. Un ciclo hamiltoniano orientato è un ciclo hamiltoniano su un grafo G orientato.

Directed ham-cycle: dato un grafo orientato G esiste un ciclo hamiltoniano orientato su G?

Si dimostra che il problema è NP-completo per riduzione polinomiale da $3 - SAT$.

Ham-cycle: dato un grafo G esiste un ciclo hamiltoniano su G?

Si dimostra che il problema è NP-completo per riduzione polinomiale da DHC.

Traveling salesman problem (TSP): dato un grafo G con pesi interi sugli spigoli e una costante k, esiste un circuito hamiltoniano su G il cui peso complessivo sia minore o uguale a k?

Si dimostra che TSP è NP-completo per riduzione polinomiale da Ham-cycle.

Clique: dato un grafo G una *clique* è un sottoinsieme di vertici a due a due adiacenti.
Dato un grafo G e una costante k esiste una clique in G con k vertici?

Si dimostra che il problema Clique è NP-completo riducendo in tempo polinomiale Vertex-cover a Clique.

La **classe dei problemi CO-NP** è costituita da quei linguaggi L il cui complemento $\overline{L}$ è in $\mathscr{NP}$.

Siccome il complemento di un problema polinomiale è polinomiale, allora i problemi che sono in $\mathscr{P}$ sono sia in $\mathscr{NP}$ che in CO-NP. Ma non sono gli unici: pare che i problemi NP-completi oltre ad essere in $\mathscr{NP}$ sono anche in CO-NP.

Riferimenti bibliografici

[1] N. J. Cutland, *Computability – An introduction to recursive function theory*, Cambridge University Press, 1980.

[2] M. Frixione, D. Palladino, *Funzioni, macchine, algoritmi*, Carocci, 2004.

[3] J. E. Hopcroft, R. Motwani, J. D. Ullman, *Automi, linguaggi e calcolabilità*, Addison-Wesley, 2003.

[4] F. Luccio, *La struttura degli algoritmi*, Bollati Boringhieri, 1982.