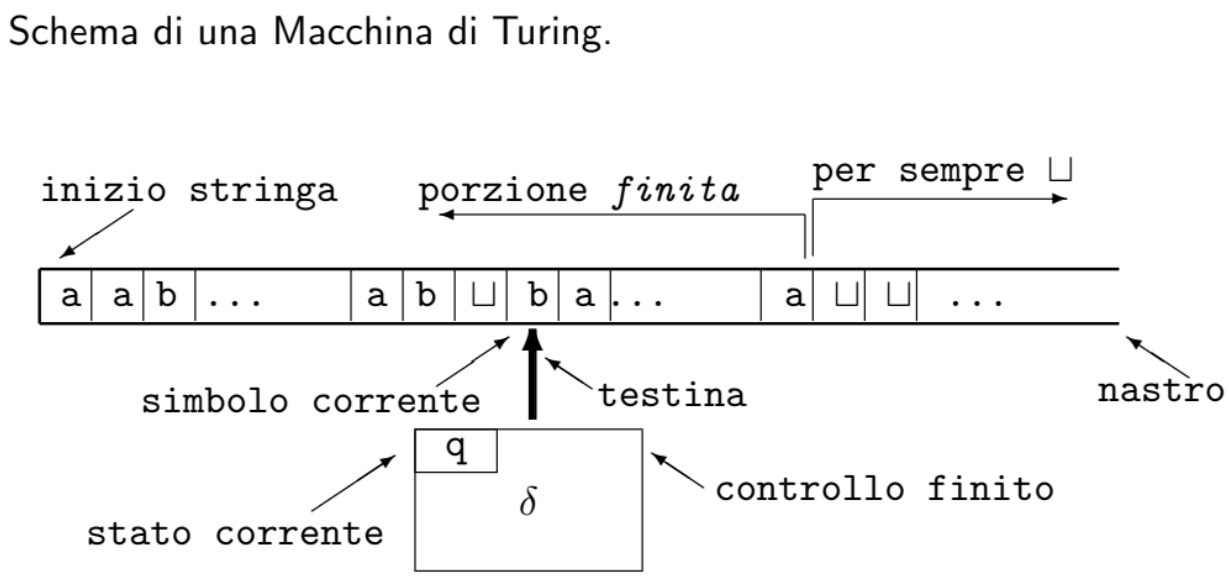
Dispensa ETC Completa (De Felice)

Elementi di Teoria della Computazione (Università degli Studi di Salerno)

Descrizione informale di una MdT



Una macchina di Turing utilizza un **nastro** semi-infinito. Inizialmente il nastro contiene solo la stringa di input. La macchina di Turing ha una **testina** di lettura e scrittura, che può muoversi lungo il nastro. In funzione dello stato in cui si trova e del simbolo input letto, la macchina cambia stato, scrive un simbolo nella cella su cui era posizionata, sposta la testina di una posizione a destra o a sinistra. La macchina continua a computare finché non **accetta** o **rifiuta**. Se non raggiunge uno stato corrispondente a uno delle due situazioni precedenti, la computazione andrà avanti **per sempre**.

Differenze tra una MdT e un automa a stati finiti

La definizione di Macchina di Turing (in seguito abbreviata con MdT o con TM) che daremo, metterà in evidenza le seguenti differenze con il modello automa finito:

- Una MdT ha un nastro semi-infinito (infinito a destra), diviso in celle, ciascuna contenente un carattere.
- Una MdT ha una testina che può muoversi a sinistra o a destra (ma non può oltrepassare il limite a sinistra del nastro).
- la testina può leggere o scrivere caratteri (testina di lettura-scrittura).
- Una MdT ha due stati speciali q_{accept}, q_{reject} rispettivamente corrispondenti all'accettazione e al rifiuto della stringa input. Se la MdT si trova in uno di tali stati non può effettuare ulteriori passi di computazione. (La frase sul testo di Sipser è: Gli stati speciali di accettazione e rifiuto hanno effetto immediato. Ossia la macchina si ferma immediatamente quando raggiunge uno dei due stati)
- Non c'è limite ai passi di computazione (in particolare il numero di tali passi non è limitato dalla lunghezza dell'input).

Esempio di Macchina di Turing

Una macchina di Turing che accetta $L = \{0^n 1^n \mid n > 0\}$

(Descrizione ad alto livello)
(passo 1) Se legge 0 lo sostituisce con *X*, se legge 1 rifiuta, se legge *Y* va al passo 4.
(passo 2) Scorre il nastro verso destra, se trova 1 lo sostituisce con *Y* , altrimenti rifiuta.
(passo 3) Scorre il nastro a sinistra fino a incontrare *X*, si sposta a destra e ripete dal passo 1.
(passo 4) Scorre il nastro a destra. Se legge solo delle *Y* e poi il simbolo di cella vuota, accetta. Altrimenti rifiuta.

000111→*X*00111…→*X*00*Y*11…→*XX*0*Y*11…→*XXXYYY*

Definizione formale di una MdT

Definizione

Una Macchina di Turing deterministica è una settupla

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$$

dove:

- Q: Insieme finito degli stati
- Σ: Alfabeto dei simboli input $(\sqcup \notin \Sigma)$
- Γ: Alfabeto (finito) dei simboli di nastro ($\sqcup \in \Gamma,\ \Sigma \subset \Gamma,\ L, R \notin \Gamma$)
- $\delta : (Q \setminus \{qaccept;\ qreject\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$ funzione di transizione
- $q_0 \in Q$: stato iniziale
- $q_{accept} \in Q$: stato di accettazione
- $q_{reject} \in Q$: stato di rifiuto, $q_{accept} \neq q_{reject}$

NOTA: Spesso (nel libro, sui lucidi) l'aggettivo deterministica è sottinteso.

NOTA: Q, Σ e Γ sono insiemi finiti.

- Σ è un sottoinsieme di Γ.
- Σ è un sottoinsieme proprio di Γ perché $\sqcup \in \Gamma\ ma\ \sqcup \notin \Sigma$.
- Q non è mai vuoto.
- Γ può contenere altri simboli oltre a quelli di Σ e a ⊔.

NOTA: Esistono molte varianti di questa definizione che hanno lo stesso potere computazionale (o espressivo), cioè riconoscono la stessa classe di linguaggi.

Vedremo due varianti: la macchina di Turing multi-nastro e la macchina di Turing **non deterministica**.

Funzione di transizione di una MdT

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$una MdT deterministica.

Il suo comportamento è descritto dalla funzione di transizione

$$\delta : (Q \setminus \{qaccept, qreject\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

Se $\delta(q, \gamma) = (q', \gamma', d)$, sappiamo che $q, q' \in Q,\ \gamma, \gamma' \in \Gamma, d \in \{L, R\}$

Se $\delta(q,\gamma) = (q',\gamma',d)$ e se M si trova nello stato q con la testina posizionata su una cella contenente γ, alla fine della transizione:

- M si troverà nello stato q',
- $\gamma' \in \Gamma$ sarà il simbolo scritto sulla cella del nastro su cui la testina si trovava **ALL'INIZIO** della transizione (contenente γ),
- la testina si sarà spostata sulla cella di sinistra se (tale cella esiste e se) d = L, sulla cella di destra se d = R.

NOTA: Data $\delta(q,\gamma) = (q',\gamma',L)$, se M si trova nello stato q con la testina posizionata su una cella contenente γ e se tale cella è quella più a sinistra del nastro, la testina non si sposta e resta sulla cella del nastro su cui si trovava all'inizio della transizione.

Spesso nel progetto di una Macchina di Turing si definisce una transizione che scrive un carattere speciale nella cella più a sinistra del nastro e che serve a individuarla.

Diagramma di stato di una MdT

Come nel caso degli automi, spesso preferiamo utilizzare il diagramma di stato di una specifica Macchina di Turing piuttosto che la descrizione formale della settupla.

Il diagramma di stato di una Macchina di Turing è un grafo i cui nodi hanno come nomi gli stati della macchina.

Le etichette degli archi hanno la forma:

$$"simbolo \rightarrow simbolo, d"$$

dove "simbolo" è un simbolo di Γ e d ∈ $\{L, R\}$. Il simbolo a sinistra della freccia rappresenta il simbolo letto, quello dopo la freccia il simbolo (eventualmente lo stesso) che sostituisce il simbolo letto per effetto della transizione, d lo spostamento per effetto della transizione.

Ad esempio, se sull'arco dal nodo q_1 al nodo q_2 compare l'etichetta:

$$0 \rightarrow \sqcup, R$$

questo corrisponde a:

$$\delta(q_1, 0) = (q_2, \sqcup, R)$$

Alle volte si usa l'abbreviazione di non indicare il carattere dopo la freccia se esso non è modificato dalla transizione.

Ad esempio, se sull'arco dal nodo q_3 al nodo q_4 troviamo l'etichetta:

$$0 \rightarrow R$$

questo vuol dire che $\delta(q_3, 0) = (q_4, 0, R)$.

Computazione di una MdT - descrizione informale

Una MdT M inizia la computazione:

- partendo dallo stato iniziale q_0,
- con l'input $w \in \Sigma^*$ posizionato sulla parte più a sinistra del nastro: le prime n celle a sinistra, se n =|w| è la lunghezza dell'input (il resto delle celle conterrà il carattere ⊔), con la testina posizionata sulla cella contenente il primo simbolo di input (quello più a sinistra)
- Poiché Σ non contiene il carattere ⊔, all'inizio della computazione il primo simbolo blank sul nastro individua la fine dell'input.

- La computazione di M procede secondo le regole descritte dalla funzione di transizione.
- La computazione di M procede fino a quando non viene raggiunto uno stato di accettazione o rifiuto. Se nessuno dei due stati viene raggiunto, M va avanti per sempre.

La computazione termina se M raggiunge

- lo stato di accettazione q_{accept}: Accetta input
- lo stato di rifiuto q_{reject}: Rifiuta input

NOTA: La computazione può non terminare. (Analogia: while C do *Something* dove C è una condizione sempre vera).

Esempio:

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, con Q = $\{q_0, q_{accept}, q_{reject}\}$, Σ = {a, b}, Γ = {a, b,␣},

δ(q0, a) = (q0, a, R),

δ(q0, b) = (q0, b, L),

δ(q0, ␣) = (qaccept,␣, L).

NOTA: ”M non accetta l’input w” **non è equivalente** a “M rifiuta l’input w”. M può accettare un input w o non accettare w. Se non accetta w, questo vuol dire che:

1. M rifiuta w
2. La computazione di M su w non termina.

È possibile simulare un automa finito deterministico attraverso una MdT.

La computazione di un automa finito deterministico su un input w può essere rappresentata da una sequenza di stati (gli stati che rappresentano i nodi del cammino di etichetta w nel diagramma di stato che rappresenta l’automa).

La posizione della testina è implicitamente rappresentata nella computazione poiché a ogni transizione si sposta di una cella a destra.

Il nastro è di sola lettura, il contenuto del nastro non viene modificato durante la computazione e quindi non è necessario considerarlo.

Computazione di una MdT – descrizione formale.

Configurazioni

Nella computazione di una macchina di Turing su un input w abbiamo bisogno di considerare tutti questi elementi:

- stato
- posizione della testina
- la parte iniziale del nastro contenente tutti i caratteri diversi da ␣ (eventualmente separati da ␣). Nota: è sempre costituita da un numero finito di celle.

Un’impostazione di questi tre elementi è chiamata una **configurazione** della macchina di Turing.

Intuitivamente una configurazione di una MdT è l’insieme delle informazioni costituito da una porzione “abbastanza lunga” del nastro, dalla posizione della testina e dallo stato della MdT a un dato istante.

Definizione

Una configurazione C di una MdT

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ è una stringa: $C = uqv \in \Gamma^* Q \Gamma^*$, dove:

- $q \in Q$ è lo stato corrente di M,
- $uv \in \Gamma^*$ è il contenuto del nastro (con la convenzione di aver eliminato tutti i simboli ⊔ dopo v, cioè dopo l'ultimo carattere di v il nastro contiene solo simboli blank),
- la testina è posizionata sul primo simbolo di v.

Ad esempio:

$$1011q_7 01111$$

rappresenta la configurazione in cui il nastro contiene

101101111⊔…⊔…

dove lo stato corrente è q_7 e la testina è posizionata sul secondo 0.

Nota: se C = uqv è una configurazione di una macchina di Turing, allora :

- u è la stringa di caratteri a sinistra del simbolo su cui è posizionata la testina.
- Sia u che v possono essere uguali a ϵ

Configurazioni: casi particolari

Se C = qv è una configurazione di una MdT allora la testina è posizionata sulla prima cella del nastro (posizione della testina a inizio nastro).

Se C = uq è una configurazione di una MdT allora la testina è posizionata sulla prima cella della porzione del nastro contenente solo ⊔ (posizione della testina sulla porzione del nastro contenente solo ⊔).

Passo di computazione

Intuitivamente un passo di computazione è la trasformazione della configurazione C1 nella configurazione C2 per effetto di una applicazione della funzione di transizione. È una relazione tra coppie di configurazioni definita dalla funzione di transizione.

Sia

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ una MdT deterministica.

Siano $q_i, q_j \in Q,\ a, b, c \in \Gamma\ e\ u, v \in \Gamma^*$.

Diremo che:

$$uaq_ibv\ produce\ uq_jacv \text{ se } \delta(q_i, b) = (q_j, c, L).$$

Diremo che

$$uaq_ibv\ produce\ uacq_jv \text{ se } \delta(q_i, b) = (q_j, c, R).$$

Nota: uaq_ibv, u$q_j$acv, uacq_jv sono configurazioni di M.

La definizione generale è più complessa perché bisogna considerare anche i casi particolari (C = qv, C = uq con u, v eventualmente uguali a ϵ).

Ad esempio:

$$q_ibv\ produce\ q_jcv \text{ se } \delta(q_i, b) = (q_j, c, L).$$

$$q_i bv \; produce \; cq_j v \;\; \text{se } \delta(q_i, b) = (q_j, c, R).$$

Occorre distinguere u = ϵ da u ≠ ϵ.

Siano $C_1 C_2$ due configurazioni di una MdT M.

Se C1 produce C2, scriveremo:

$$C1 \rightarrow C2$$

Il simbolo → ,rappresenta la trasformazione di C1 in C2 e prende il nome di **passo di computazione.**

→ corrisponde a un'applicazione della funzione di transizione di M.

Definizione

Una configurazione C si dice:

- **iniziale** (con input w) se $C = q_0 w$, con $w \in \Sigma^*$,
- di **arresto** se C = uqv, con $u, v \in \Gamma^*$ e $q \in \{q_{accept}, q_{reject}\}$ (non esiste nessuna configurazione C' tale che $C \rightarrow C'$),
- di **accettazione** se C = uqv, con $u, v \in \Gamma^*$ e $q = q_{accept}$,
- di **rifiuto** se C = uqv, con $u, v \in \Gamma^*$ e $q = q_{reject}$.

Nota: Le configurazioni di **arresto** sono configurazioni di accettazione o di rifiuto.

Definizione

Siano C, C' configurazioni.

$C \rightarrow^* C'$ se esistono configurazioni $C_1, \dots, C_k$ $k > 1$ tali che:

- $C_1 = C$
- $C_i \rightarrow C_{i+1,}$ $per\ i \in \{1, \dots, k-1\}$ (ogni C_i produce C_{i+1})
- $C_k = C'$.

Diremo che $C \rightarrow^* C'$ è una **computazione** (di lunghezza k – 1).

Sia M una MdT e sia C una configurazione di M. Ci sono tre possibili casi:

1. $C \rightarrow^* C'$ con C'= uq_{accept}v **configurazione di accettazione** (M si ferma in q_{accept}).
2. $C \rightarrow^* C'$ con C'= uq_{reject}v **configurazione di rifiuto** (M si ferma in q_{reject}).).
3. Per ogni configurazione C' tale che $C \rightarrow^* C'$ esiste una configurazione C'' tale che $C \rightarrow^* C' \rightarrow C''$ (M non si arresta).

Parola accettata da una MdT

Definizione

Una MdT M **accetta** una parola $w \in \Sigma^*$ se esiste una computazione $C \rightarrow^* C'$, dove C = $q_0 w$ è la configurazione iniziale di M con input w e $C' = uq_{accept}v$ è una configurazione di accettazione.

Quindi M **accetta** $w \in \Sigma^*$se e solo se esistono configurazioni $C_1, C_2, \dots, C_k$ $di\ M$, $k > 1$ tali che:

- $C_1 = q_0 w$ è la configurazione iniziale di M con input w
- $C_i \rightarrow C_{i+1,}$ $per\ i \in \{1, \dots, k-1\}$
- C_k è una configurazione di accettazione.

Allo stesso modo, M **rifiuta** una parola $w \in \Sigma^*$ se esiste una computazione $C \rightarrow^* C'$, dove C = $q_0 w$ è la configurazione iniziale di M con input w e $C' = uq_{reject}v$ è una configurazione di rifiuto.

Quindi M **si ferma** su $w \in \Sigma^*$se M **accetta** w *oppure* **rifiuta** w, cioè se e solo se esistono configurazioni $C_1, C_2, \dots, C_k \; di \; M \; , k > 1$ tali che:

- $C_1 = q_0 w$ è la configurazione iniziale di M con input w
- $C_i \rightarrow \; C_{i+1,} \; per \; i \in \{1, \dots, k-1\}$
- C_kè una **configurazione di arresto**.

Dunque, data una MdT M e una stringa $w \in \Sigma^*$, ci sono solo due possibili casi:

- M accetta w
- M non accetta w in quanto :
  - M rifiuta w
  - La computazione di M non si arresta mai sull'input w

Domanda:
Sia M una MdT. Sia w = $a_1 \dots a_n \; con \; a_j \in \Sigma$. Supponiamo che w è accettata da M.

Quindi esistono $u, v \in \Gamma^*$ tali che:

$$q_0 w \rightarrow^* u q_{accept} v$$

Possiamo concludere che in questa computazione M deve aver letto tutti i caratteri a_j di w?

E quindi che la computazione deve avere lunghezza almeno n = |w|?

Risposta: ciò non è necessario, in quanto la condizione da verificare può essere vera prima di aver letto tutti i caratteri della stringa w. Ad esempio, se la MdT accetta stringhe che iniziano per a, mi basta leggere solo il primo carattere presente sul nastro , ignorando tutti i restanti.

Linguaggio riconosciuto da una MdT

Definizione

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ una MdT. Il linguaggio L(M) riconosciuto da M, è l'insieme delle stringhe che M accetta:

$$L(M) = \{w \in \Sigma^* | \; \exists u, v \in \Gamma^* \; q_0 w \rightarrow^* u q_{accept} v\}$$

Quindi:

$$L(M) = \{w \in \Sigma^* | \; M \; accetta \; w\}$$

A differenza del caso degli automi finiti, per i quali una stringa input è accettata o rifiutata, nel caso delle macchine di Turing c'è una terza possibilità: che per un dato input la computazione non termini.

Data una configurazione di una MdT che non è di arresto, non possiamo sapere se, a partire da tale configurazione, la computazione terminerà in qualche istante futuro o meno.

Sarebbe utile avere un algoritmo che, dato in input una MdT M e una stringa w, determini in un tempo finito se la computazione eseguita da M su w termini o meno (**Problema della fermata**). Vedremo che un tale algoritmo non esiste.

Linguaggi Turing riconoscibili e linguaggi decidibili

Sia $R(M) = \{w \in \Sigma^* | M \; rifiuta \; w\}$. (R è inteso come il complemento di L)

In generale L(M) ∪ R(M) non coincide con Σ^*. (ciò è dovuto al fatto che possono esserci degli input in $\Sigma^* \in R(M)$ per i quali M non si ferma mai)

Se L(M) ∪ R(M) = Σ^*, allora M si arresta su ogni input. (Tutti i possibili input in $\Sigma^* \in L(M)\ o\ R(M)$ portano ad uno stato di arresto. In tal caso M è chiamata un **decisore** (o decider) ed L(M) è il linguaggio **deciso** da M.)

Occorre quindi definire **due** famiglie di linguaggi:

Definizione

Un linguaggio $L \subseteq \Sigma^*$ è **Turing riconoscibile** se esiste una macchina di Turing

M = (Q, Σ, Γ, δ, $q_0, q_{accept}, q_{reject}$) tale che:

- M **riconosce** L (cioè $L = L(M) = \{w \in \Sigma^* \mid \exists\, u, v \in \Gamma^*\ q_0 w \rightarrow^* u q_{accept} v\}$).

Definizione

Un linguaggio $L \subseteq \Sigma^*$ è **decidibile** se esiste una macchina di Turing

M = (Q, Σ, Γ, δ, $q_0, q_{accept}, q_{reject}$) tale che:

- M **riconosce** L (cioè $L = L(M) = \{w \in \Sigma^* \mid \exists\, u, v \in \Gamma^*\ q_0 w \rightarrow^* u q_{accept} v\}$).
- M si **arresta su ogni input** (cioè per ogni $w \in \Sigma^*, q_0 w \rightarrow^* uqv$) con $q \in \{q_{accept}, q_{reject}\}$

Vedremo che l'insieme dei linguaggi decidibili è un sottoinsieme proprio dell'insieme dei linguaggi Turing riconoscibili. Come conseguenza delle definizioni, un linguaggio L è Turing riconoscibile ma non decidibile se:

- esiste una MdT che riconosce L (quindi accetta tutte e sole le stringhe di L)
- non esiste nessuna MdT M tale che M accetta tutte le stringhe in L e rifiuta tutte le stringhe che appartengono al complemento $\bar{L}$ di L.

Macchine di Turing e Decider

Una MdT $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ **decide** un linguaggio L se per ogni $w \in \Sigma^*$:

$$q_0 w \rightarrow^* C$$

con C = uqv configurazione di **arresto** ed L = L(M).

In questo caso diciamo che L è il linguaggio deciso da M.

Nota: data una MdT M, esiste sempre il linguaggio riconosciuto da M ma non è detto che esista il linguaggio deciso da M, poiché ciò è possibile se e solo se M è un decider.

Se M è un decider allora esiste il linguaggio deciso da M e coincide con il linguaggio riconosciuto da M.

Se M non è un decider, esiste il linguaggio riconosciuto da M ma non il linguaggio deciso da M.

È importante non confondere la proprietà di un linguaggio (essere o non essere Turing riconoscibile, essere o non essere decidibile) con la proprietà di una MdT (essere o non essere un decider).

Esempio: $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ (dove L(M) = {a}*) con

$Q = \{q_0, q_{accept}, q_{reject}\}$,

$\Sigma = \{a, b\}$,

$\Gamma = \{a, b, \sqcup\}$,

$\delta(q_0, a) = (q_0, a, R)$, $\quad \delta(q_0, b) = (q_0, b, L)$, $\quad \delta(q_0, \sqcup) = (q_{accept}, \sqcup, L)$.

M non è un decider ma $L(M) = a^*$ è decidibile. (M per come è definita può non fermarsi su alcuni input e quindi non decide a*, mentre è possibile costruire una MdT che decide a^*)

Domande:

Ogni linguaggio regolare è Turing riconoscibile? Si.

Ogni linguaggio regolare è decidibile? No.

Ogni linguaggio decidibile è regolare? No. Per esempio c'è un decider per $a^n b^n$ ma non è regolare.

Ogni linguaggio Turing riconoscibile è regolare? Si.

Funzioni calcolabili

Le MdT possono essere utilizzate per il calcolo di funzioni.

Definizione

Una funzione $f : \Sigma^* \to \Sigma^*$ è calcolabile se esiste una macchina di Turing M = (Q, Σ, Γ, δ, $q_0, q_{accept}, q_{reject}$) tale che:

$$\forall\, w \in \Sigma^*, q_0 w \to^* q_{accept} f(w)$$

Quindi, una funzione $f : \Sigma^* \to \Sigma^*$ è calcolabile se esiste una macchina di Turing M che, su qualsiasi input w, si ferma avendo solo f(w) sul nastro.

Nota: in questo caso la MdT deve arrestarsi su ogni input.

Possibili Varianti*:*

1. nessun vincolo sulla posizione della testina nella configurazione di arresto.
2. nessuna distinzione tra q_{accept}, q_{reject}.

Varianti di MdT

Esistono diverse varianti della definizione di macchina di Turing deterministica.

Vedremo:

- la Macchina di Turing **multi-nastro**
- la Macchina di Turing **non deterministica**

Tali macchine di Turing hanno lo **stesso potere** computazionale (o espressivo) delle MdT deterministiche, cioè riconoscono la stessa classe di linguaggi.

Esistono anche altre varianti della macchina di Turing deterministica che hanno lo stesso potere espressivo.

Questa "**robustezza**" rispetto ai cambiamenti è una conferma che si tratta di un buon modello per la definizione di algoritmo.

Siano $\mathcal{T}_1$ e $\mathcal{T}_2$ due famiglie di modelli computazionali. Per dimostrare che i modelli in T1 hanno lo stesso potere computazionale dei modelli in T2 occorre far vedere che per ogni macchina M1 ∈ T1 esiste M2 ∈ T2 equivalente a M1 e **viceversa**.

Ricordiamo: Due macchine sono **equivalenti** se riconoscono lo stesso linguaggio. Abbiamo già dimostrato che la classe dei DFA ha lo stesso potere computazionale della classe degli NFA.

A differenza però degli automi finiti, alle macchine in T1 potrebbe corrispondere una rappresentazione delle configurazioni diversa dalla rappresentazione delle configurazioni delle macchine in T2.

La dimostrazione che per ogni macchina M1 ∈ T1 esiste M2 ∈ T2 equivalente a M1 (e viceversa) consiste spesso nella dimostrazione che per ogni M1 esiste M2 capace di simularla (e viceversa): sono in corrispondenza configurazioni iniziali, intermedie e di arresto.

Come nel caso dei DFA e degli NFA, in genere una delle direzioni della prova è evidente.

Esempio:

Consideriamo una variante della definizione data di macchina di Turing deterministica. Sia $\mathcal{T}_{(L,R)}$ l'insieme delle macchine di Turing che abbiamo definito e sia $\mathcal{T}_{(L,R,S)}$ l'insieme i cui elementi M sono definiti nel modo seguente:

$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ è una settupla dove Q, Σ, Γ, δ, $q_0, q_{accept}, q_{reject}$ sono definiti come in una MdT deterministica e la funzione di transizione δ è definita nel modo seguente:

$$\delta : (Q \setminus \{q_{accept}, q_{reject}\}) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, S\}$$

Se $\delta(q, \gamma) = (q', \gamma', d)$ e se M si trova nello stato q con la testina posizionata su una cella contenente γ, alla fine della transizione:

- M si troverà nello stato q',
- $\gamma' \in \Gamma$ sarà il simbolo scritto sulla cella del nastro su cui la testina si trovava all'inizio della transizione (contenente γ),
- la testina si troverà sulla stessa cella su cui si trovava all'inizio della computazione se d = S, si sarà spostata sulla cella di sinistra se (tale cella esiste e se) d = L, si sarà spostata sulla cella di destra se d= R.

Le nozioni di configurazione, di linguaggio deciso e di linguaggio riconosciuto da $M' \in \mathcal{T}_{(L,R)}$ sono estese in maniera ovvia alle macchine M in $\mathcal{T}_{(L,R,S)}$

I modelli in $\mathcal{T}_{(L,R)}$ hanno lo stesso potere computazionale dei modelli in $\mathcal{T}_{(L,R,S)}$.

Se $M' = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}) \in \mathcal{T}_{(L,R)}$, definiamo

$$M = (Q, \Sigma, \Gamma, \delta', q_0, q_{accept}, q_{reject}) \in \mathcal{T}_{(L,R,S)} \text{ con } \delta(q, \gamma) = \delta'(q, \gamma),$$

$$\text{per ogni } (q, \gamma) \in (Q \setminus \{q_{accept}, q_{reject}\}) \times \Gamma.$$

Ovviamente L(M) = L(M').

Viceversa, se

$M' = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}) \in \mathcal{T}_{(L,R,S)}$ definiamo

M = $(Q \cup Q^c, \Sigma, \Gamma, \delta', q_0, q_{accept}, q_{reject}) \in \mathcal{T}_{(L,R,S)}$dove:

- se $\delta(q_i, \gamma) = (q_j, \gamma', d)$ e $d \in \{L, R\}$ $allora$ $\delta'(q_i, \gamma) = \delta(q_j, \gamma', d)$
- se $\delta(q_i, \gamma) = (q_j, \gamma', S)$ $allora$ $\delta'(q_i, \gamma) = (q_j^c, \gamma, R)$ e $\delta'(q_j^c, \eta) = (q_j, \eta, L)$ $per\ ogni\ \eta \in \Gamma$.
- $Q^c = \{q^c \mid (q, \gamma, S) \in \delta((Q \setminus \{q_{accept}, q_{reject}\}) \times \Gamma)\}$.

Anche in questo caso L(M) = L(M').

Macchina di Turing multi-nastro

Una macchina di Turing multi-nastro (abbreviata MdTM) è una macchina di Turing in cui si hanno più nastri contemporaneamente accessibili in scrittura e lettura che vengono aggiornati tramite più testine (una per nastro).

Definizione (MdT a k nastri)

Dato un numero naturale k, una macchina di Turing con k nastri è una settupla:

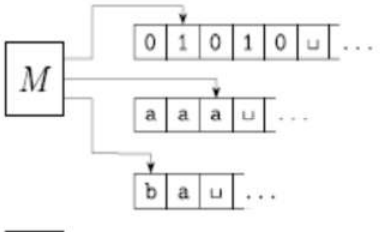
$(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ dove Q, Σ, Γ, $q_0, q_{accept}, q_{reject}$ sono definiti come in una MdT deterministica e la funzione di transizione δ è definita nel modo seguente:

$$\delta : (Q \setminus \{q_{accept}, q_{reject}\}) \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, R, S\}^k$$

Nota: Γ^k è il prodotto cartesiano di k copie di Γ e $\{L, R, S\}^k$ è il prodotto cartesiano di k copie di {L, R, S}.

Definizione (MdT multi-nastro)

Una macchina di Turing multi-nastro è una macchina di Turing a k nastri, con $k \in N$.



Funzione di transizione di una MdTM

Il codominio della funzione di transizione è un insieme di sequenze $(q_j, b_1, \dots, b_k, d_1, \dots, d_k)$ di lunghezza (2k + 1) dove:

- $q_j \in Q$
- $b_t \in \Gamma, per\ t \in \{1, \dots, k\}$
- $d_t \in \{L, R, S\},\ per\ t \in \{1, \dots, k\}$

Se $\delta\ (q_i, a_1, \dots, a_k) = (q_j, b_1, \dots, b_k, d_1, \dots, d_k)$ e se M si trova nello stato q_i con le k testine (una per nastro) posizionate su celle contenente $a_1, \dots, a_k$ rispettivamente, alla fine della transizione:

- M si troverà nello stato q_j,
- $b_t \in \Gamma$ sarà il simbolo scritto sulla cella del t-esimo nastro su cui la testina si trovava all'inizio della transizione (contenente a_t), $per\ t \in \{1, \dots, k\}$
- la testina sul t-esimo nastro si troverà sulla stessa cella cui si trovava all'inizio della computazione se $d_t = S$, si sarà spostata sulla cella di sinistra se (tale cella esiste e se) $d_t = L$, si sarà spostata sulla cella di destra se $d_t = R, per\ t \in \{1, \dots, k\}$.

Computazione di una MdTM

Una MdTM inizia la computazione:

- partendo dallo stato iniziale q_0,
- con l'input $w \in \Sigma^*$ posizionato sulla parte più a sinistra del primo nastro: le prime n celle a sinistra, se n = |w| è la lunghezza dell'input (il resto delle celle conterrà il carattere ⊔)
- Gli altri nastri conterranno solo il carattere ⊔

- la testina del primo nastro sarà posizionata sulla cella contenente il primo simbolo di input (quello più a sinistra), le altre sulla prima cella dei rispettivi nastri.

Le nozioni di configurazione, passo di computazione, di linguaggio deciso e di linguaggio riconosciuto da una MdT sono estese in maniera ovvia alle macchine MdTM.

Ad esempio, se M è una macchina di Turing con k nastri, una configurazione ha la forma

$$(u_1 q v_1, \dots, u_k q v_k)$$

dove $q \in Q$ è lo stato corrente di M e, $per\ t \in \{1, \dots, k\}$, $u_t v_t \in \Gamma^*$, la testina del t-esimo nastro è posizionata sul primo simbolo di v_t $se\ v_t \neq \epsilon$, su ⊔ altrimenti, $u_t v_t$ è il contenuto del t-esimo nastro, con la convenzione di aver eliminato tutti i ⊔ che seguono v_t.

$$L(M) = \{w \in \Sigma^* \mid \exists u_t, v_t \in \Gamma^*, t \in \{1, \dots, k\} : (q_0 w, \dots, q_0) \rightarrow^* (u_1 q_{accept} v_1, \dots, u_k q_{accept} v_k)\}$$

Esempio: MdT a 2 nastri per $\{0^n 1^n \mid n > 0\}$

1. Scorre primo nastro verso destra fino a primo 1: per ogni 0, scrive un 1 sul secondo nastro
2. Scorre primo nastro verso destra e secondo nastro verso sinistra: se simboli letti non uguali, termina in q_{reject}
3. Se legge ⊔ su entrambi i nastri, termina in q_{accept}

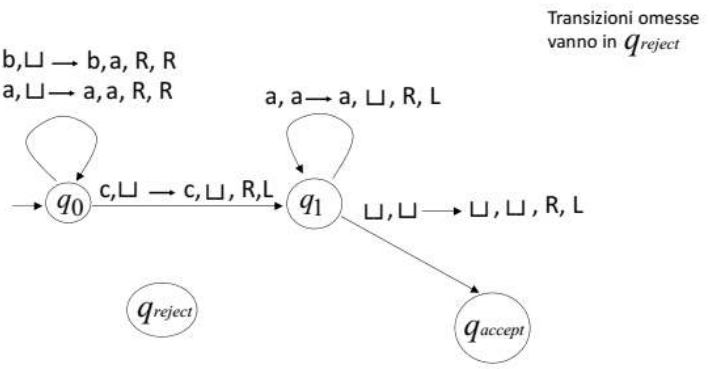
stato attuale	simbolo letto	Valore funzione δ
q_0	$(0, ⊔)$	$q_0, (⊔, 1), (R, R)$
q_0	$(1, ⊔)$	$q_1, (1, ⊔), (S, L)$
q_1	$(1, 1)$	$q_1, (⊔, ⊔), (R, L)$
q_1	$(⊔, ⊔)$	$q_{accept}, (⊔, ⊔), (S, S)$

Nota: Se $\delta(q, \gamma, \gamma')$ non è presente nella tabella, con $q \in Q \setminus \{q_{accept}, q_{reject}\}$ allora $\delta(q, \gamma, \gamma') = (q_{reject}, \gamma, \gamma', S, S)$.

Esempio: MdT a 2 nastri per $\{wca^{|w|} \mid w \in \{a.b\}^*\}$

Una macchina di Turing deterministica M a due nastri che decide il linguaggio

$$L = \{wca^{|w|} \mid w \in \{a, b\}^*\}$$



MdT multi-nastro e MdT

Teorema

Per ogni macchina di Turing multi-nastro M esiste una macchina di Turing (a nastro singolo) M' equivalente ad M, cioè tale che L(M) = L(M').

Dimostrazione

Faremo vedere che esiste

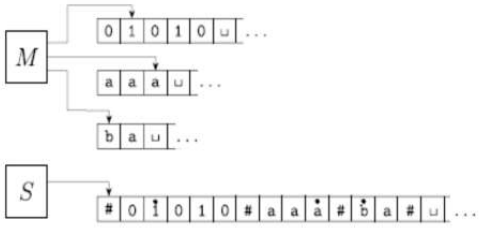
$M' = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ equivalente ad M, con

$$\delta : (Q \setminus \{q_{accept}, q_{reject}\}) \times \Gamma \to Q \times \Gamma \times \{L, R, \boldsymbol{S}\}.$$

Supponiamo che M abbia k nastri.

Per ogni configurazione di M $(u_1 q v_1, \dots, u_k q v_k)$ la macchina M' che simula M deve codificare su un solo nastro:

- il contenuto $u_1 v_1, \dots, u_k v_k$ dei k nastri,
- la posizione di ciascuna testina su ogni nastro.



- Il contenuto del nastro di M' sarà la concatenazione di k blocchi separati da # (seguita da ⊔).
- Ogni blocco corrisponderà a un nastro di M e avrà una lunghezza variabile che dipende dal contenuto del nastro corrispondente.
- Un elemento $\dot{\gamma}$, con $\gamma \in \Gamma$, nel blocco t-esimo indica la posizione della testina del nastro t-esimo, $t \in \{1, \dots, k\}$
- Quindi a una configurazione di M $(u_1 q a_1 v_1, \dots, u_k q a_k v_k)$ corrisponderà la configurazione di M' $q'\#u_1 \dot{a}_1 v_1 \# \dots \# u_k \dot{a}_k v_k \#$. *Nota*: Se Γ è l'alfabeto dei simboli di nastro di M, l'alfabeto dei simboli di nastro di M' sarà $\Gamma \cup \{\dot{\gamma} \mid \gamma \in \Gamma\} \cup \{\#, \dot{\#}\}$.

Sia $w = w_1 \dots w_n$ una stringa input, $w_t \in \Sigma$, $t \in \{1, \dots, k\}$

- (**generazione della configurazione iniziale di M**) M' passa dalla configurazione iniziale $q_0 w_1 \dots w_n$ alla configurazione $q'\#\dot{w}_1 \dots w_n \# \dot{\sqcup} \# \dots \# \dot{\sqcup} \#$ (**Nota**: Servono stati addizionali).
- Per simulare un passo di computazione di M, per effetto dell'applicazione di $\delta(q, a_1, \dots, a_k) = (s, b_1, \dots, b_k, d_1, \dots, d_k)$, la macchina M' scorre il dato sul nastro dal primo # al (k + 1)-esimo # da sinistra a destra e viceversa due volte:
  1) la prima volta M' determina quali sono i simboli correnti di M, $\dot{a}_t$, memorizzando nello stato i simboli marcati sui singoli nastri (**Nota**: Servono stati addizionali).
  2) la seconda volta M' esegue su ogni sezione (cioè un nastro di M) le azioni che simulano quelle delle testine di M: scrittura e spostamento (in particolare sposta il marcatore alla nuova posizione della testina corrispondente di M).

  Nota. Nel corso della simulazione M' potrebbe spostare una delle testine virtuali (puntino) su un delimitatore #. Questo significa che M ha spostato la testina del nastro (corrispondente al segmento del nastro di M') sulla "parte di nastro vuota" (sulla cella a destra del carattere diverso

da ␣ più a destra del suo nastro). In questo caso, M' cambia $\dot{\#}$ con $\dot{␣}$ e deve spostare di una posizione tutto il suffisso del nastro a partire da $\dot{\#}$ (cambiato con #) fino all'ultimo # a destra.

- **Poi la MdT entra nello stato che ricorda il nuovo stato di M e riposiziona la testina all'inizio del nastro.**
- Infine, se M si ferma, anche M' si ferma, eventualmente rimuovendo tutti i separatori # e sostituendo i caratteri $\dot{a}_t$ con a_t.

Teorema

Un linguaggio L è Turing riconoscibile se e solo se esiste una macchina di Turing multi-nastro M che lo riconosce, cioè tale che L(M) = L.

Dimostrazione.

Se L è Turing riconoscibile allora esiste una MdT M tale che L(M) = L. Poiché M è una MdT a k nastri con k = 1, esiste una macchina di Turing multi-nastro M tale che L(M) = L.

Viceversa, se esiste una macchina di Turing multi-nastro M tale che L(M) = L, per i teoremi precedenti esiste una macchina di Turing (a nastro singolo) M' tale che L(M') = L(M) = L. Quindi L è Turing riconoscibile.

Osservazione

Abbiamo dimostrato che le MdTM hanno lo stesso potere delle MdT. Introdurremo una misura del tempo necessario per decidere un problema. Più precisamente considereremo solo MdT che si fermano su ogni input. Assoceremo a ogni MdT di questo tipo una funzione che stimi il tempo che le è necessario per stabilire se w appartiene al linguaggio L(M) deciso da M. Vedremo che la macchina di Turing (a nastro singolo) M' equivalente alla MdTM M simula la computazione di M "perdendo" tempo in modo quadratico.

Ordine lessicografico

Definizione

Sia $\Sigma = \{a_0, \dots, a_k\}$ un alfabeto e sia $a_0 < a_1 < \cdots < a_k$ un ordinamento degli elementi di Σ.

Siano $x, y \in \Sigma^*$. Diremo che x < y rispetto all'ordine lessicografico se x e y verificano una delle condizioni seguenti:

- y = xz con $z \in \Sigma^+$, cioè x è un prefisso di y e x ≠ y.
- x = zax', y = zby', con z, x', y' ∈ $\Sigma^*, a, b \in \Sigma\ e\ a < b$.

Le parole in un dizionario sono ordinate in base all'ordine lessicografico.

Esempio. Supponiamo a < b < … < z. Allora latte < latteria, castagna < castello.

Ordine radix

Definizione

Sia $\Sigma = \{a_0, \dots, a_k\}$ un alfabeto e sia $a_0 < a_1 < \cdots < a_k$ un ordinamento degli elementi di Σ.

Siano $x, y \in \Sigma^*$. Diremo che x ≤ y rispetto all'ordine lessicografico se x e y verificano una delle condizioni seguenti:

- $|x| < |y|$.
- $|x| = |y|\ e\ x = zax', y = zby', con\ z, x', y' \in \Sigma^*, a, b \in \Sigma\ e\ a \leq b$.

In "Introduzione alla teoria della computazione", l'ordine radix è chiamato ordine per lunghezza. La lista dei numeri in base k in ordine crescente è una lista di stringhe su $\Sigma = \{0, \dots, k-1\}$ in ordine radix.

Esempio.

- Supponiamo a < b < … < z. Allora barca < aurora, castagna < castello.
- La lista delle stringhe su $\Sigma = \{0, 1\}$, con 0 < 1, in ordine radix:
  0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100,…
- La lista delle stringhe su $\Sigma = \{a, b, c\},\ con\ a < b < c$, in ordine radix:
  a, b, c, aa, ab, ac, ba, bb, bc, ca, cb, cc, …

Macchina di Turing non deterministica

Introdurremo una variante della macchina di Turing deterministica, detta macchina di Turing non deterministica. Essenzialmente, una macchina di Turing non deterministica differisce da una deterministica per il fatto che una configurazione può evolvere in più di una configurazione successiva. Vedremo che le macchine non deterministiche hanno lo stesso potere computazionale di quelle deterministiche. Tuttavia, vedremo che le macchine di Turing deterministiche possono simulare quelle non deterministiche con una perdita di efficienza **esponenziale**.

Definizione di Macchina di Turing non Deterministica

Una Macchina di Turing non deterministica è una settupla

$(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ dove:

- $Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}$ sono definiti come in una MdT deterministica
- la funzione di transizione δ è definita nel modo seguente:

$$\delta : (Q \setminus \{q_{accept}, q_{reject}\}) \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Quindi per ogni $q \in Q\ \{q_{accept}, q_{reject}\}, per\ ogni\ \gamma \in \Gamma$, risulta:

$$\delta(q, \gamma) = \{(q_1, \gamma_1, d_1), \dots, (q_k, \gamma_k, d_k)\}, con\ k \geq 0\ e\ (q_j, \gamma_j, d_j) \in Q \times \Gamma \times \{L, R\}, per\ j \in \{1, \dots, k\}.$$

Le configurazioni hanno la stessa forma uqv di quelle definite per una Macchina di Turing deterministica. Allo stesso modo non cambia il passo di computazione $C1 \to C2$ e una computazione $C \to C'$ continua a essere una successione finita (**non un albero**!) di configurazioni. Poiché la macchina di Turing è non deterministica, ci possono essere più configurazioni $u'q'v'$ che sono prodotte da uqv in un solo passo.

Quindi, come in un NFA, le computazioni (a partire da una configurazione) possono essere organizzate in un albero in cui la radice è la configurazione di partenza (tipicamente quella iniziale), i nodi sono configurazioni i cui figli rappresentano le possibili configurazioni raggiungibili da quel nodo. Una computazione è completamente determinata da una sequenza di scelte tra le varie configurazioni raggiungibili passo dopo passo. Quindi una computazione può essere rappresentata da un **cammino** nell'albero.

Linguaggio riconosciuto da una MdT non deterministica

Definizione

Sia M = $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ una MdT non deterministica. Il linguaggio L(M) riconosciuto da M, è l'insieme:

$$L(M) = \{w \in \Sigma^* | \exists\ una\ computazione\ q_0 w \to^*\ C\ con\ C = uq_{accept}v\}$$

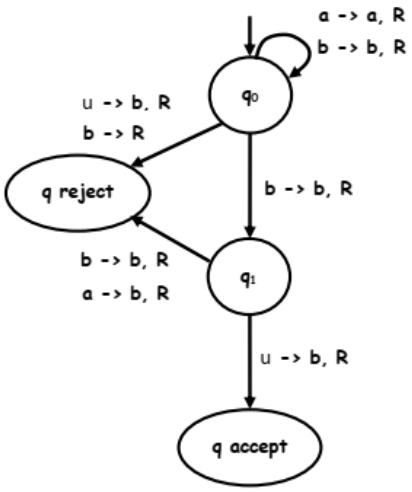
Come per le macchine di Turing deterministiche, partendo dalla configurazione $q_0 w$, la computazione può terminare (se si raggiunge una configurazione di arresto) o non terminare.

La stringa appartiene ad L(M) se esiste una successione di scelte tali che $q_0 w \to^* C, con\ C$ configurazione di accettazione. Non considereremo funzioni calcolate da una MdT non deterministica

Anche per le macchine di Turing non deterministiche sono definite due famiglie di linguaggi:

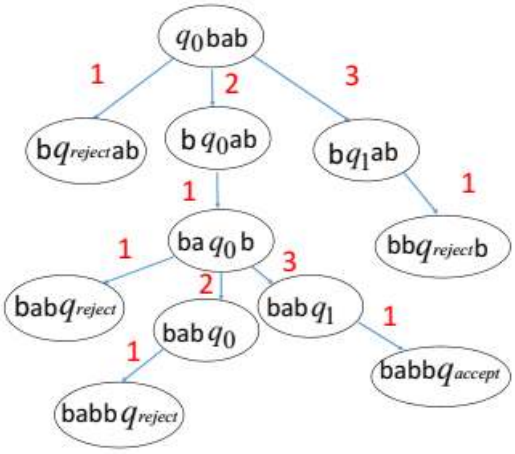
- I linguaggi L **riconosciuti** da macchine di Turing non deterministiche, cioè tali che esiste una MdT non deterministica M con L = L(M)
- I linguaggi L **decisi** da macchine di Turing non deterministiche. Un linguaggio $L \subseteq \Sigma^*$ `e deciso da una macchina di Turing non deterministica M = $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ se M è tale che:
  1) $per\ ogni\ w \in \Sigma^*,\ per\ ogni\ q_0 w \rightarrow^*\ C' esiste\ C = uqv\ tale\ che\ q \in \{q_{accept}, q_{reject}\}\ e\ q_o w \rightarrow^*\ C' \rightarrow^*\ C$ (tutte le computazioni a partire da $q_0 w$ terminano in una configurazione di arresto)
  2) M riconosce L cioè L = $L(M) = \{w \in \Sigma^* | \exists\ una\ computazione\ q_0 w \rightarrow^*\ C\ con\ C\ =\ uq_{accept}v\}$.

Macchina di Turing M1:



$$L(M_1) = \{wb\ | w \in \{a, b\}^*\}$$

Albero delle computazioni e codifica di M_1 su input bab



A ogni stringa sull'alfabeto {1, 2, 3} è associata una computazione.

$$1: q_0 bab \rightarrow\ bq_{reject} ab$$

$$2: q_0 bab \rightarrow bq_0 ab$$

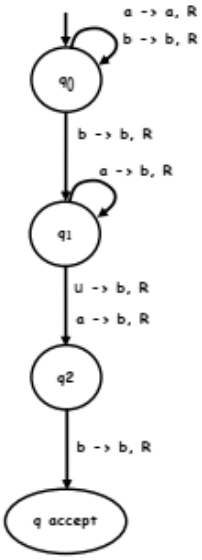
$$3: q_0 bab \rightarrow\ bq_1 ab$$

- 11: senza sbocco, muore. Lo stesso per 12 e 13.
- 21 : $q_0bab \rightarrow\ bq_0ab \rightarrow\ baq_0b$
  22, 23 senza sbocco, muore.
  …

La computazione che mostra che bab è accettata:

$$2131:\ q_0bab \rightarrow bq_0ab \rightarrow baq_0b \rightarrow babq_1 \rightarrow babbq_{accept}$$

Macchina di Turing M2:



$$L(M_2) = \{xba^nby\ |x, y \in \{a, b\}^*, n \in N, n \geq 1\}$$

Teorema

Per ogni macchina di Turing non deterministica N esiste una macchina di Turing deterministica D equivalente ad N, cioè tale che L(N) = L(D).

Idea della prova: Ogni computazione di N (a partire da una configurazione iniziale q_0w) è una sequenza di scelte che D deve riprodurre. Le possibili computazioni sono rappresentabili mediante cammini in un albero e quindi la computazione di D può essere rappresentata da una visita di tale albero.

Una strategia sbagliata: visitare l'albero in profondità (la visita rischierebbe di rimanere "incastrata" in un cammino corrispondente a una computazione che non termina escludendo un'eventuale accettazione).

Una strategia vincente: visitare l'albero per livelli (se esiste una computazione $q_0w \rightarrow^* uq_{accept}v$, D prima o poi effettuerà la sequenza di scelte corrispondenti).

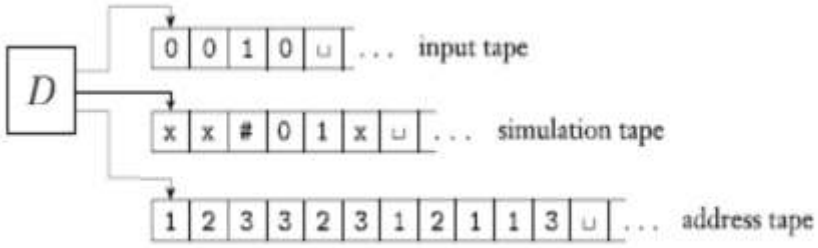
Dimostrazione

Data una macchina di Turing non deterministica N =$\left(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\right)$, la macchina di Turing D che simula N ha tre nastri.

Sul primo nastro è memorizzata la stringa input w (sulla parte più a sinistra) e il contenuto del primo nastro non verrà alterato dalle computazioni di D.

Sul terzo nastro vengono generate le codifiche delle possibili computazioni di N con input w.

Sul secondo nastro viene eseguita la simulazione di N ed inizialmente esso contiene solo ⊔. Precisamente, per ogni codifica C generata sul terzo nastro, D copia la stringa input w e simula la computazione C di N su w.



Come ottenere una codifica delle computazioni di N?

Sia b il massimo numero di alternative proposte dalla funzione di transizione δ di N.

Cioè δ(q, γ) è un insieme di cardinalità minore o uguale a b, per ogni $q \in Q, \gamma \in \Gamma$.

Posso ordinare l'insieme δ(q, γ). Cioè posso associare a ogni elemento di δ(q, γ) (e quindi ai figli del nodo che rappresenta la configurazione uqγv) un numero in $\{1, \dots, |\delta(q,\gamma)|\}$.

Poiché una computazione è individuata da una sequenza di scelte, un cammino nell'albero è codificato da una stringa sull'alfabeto $\Sigma_b = \{1, \dots, b\}$. Non tutte le stringhe rappresentano computazioni (per una coppia (q, γ) ci possono essere meno di b scelte). Se una stringa rappresenta una computazione, ogni simbolo nella stringa rappresenta una scelta tra le possibili alternative proposte dalla δ in un passo di computazione. La stringa vuota rappresenta la configurazione iniziale.

Una visita per livelli dell'albero corrisponde alla lista delle stringhe su Σ_b in ordine radix (in ordine di lunghezza crescente e, a parità di lunghezza, in ordine numerico).

Descrizione della MdT non deterministica D

1) Inizialmente il nastro 1 contiene l'input w e i nastri 2 e 3 contengono solo ⊔
2) D copia il contenuto del nastro 1 sul nastro 2.
3) Simula N con input w sul nastro 2 utilizzando la stringa $c_1 \dots c_t$ sul nastro 3, cioè riproducendo la successione di scelte del corrispondente cammino: al primo passo si utilizza l'alternativa c_1, al secondo c_2... Se si raggiunge una configurazione di accettazione, D accetta l'input. Altrimenti (se si raggiunge una configurazione di rifiuto, se la stringa non corrisponde a una computazione, o al termine della simulazione sulla stringa) D passa al passo 4.
4) D genera sul nastro 3 la stringa successiva a quella corrente in ordine radix e torna al passo 2.

$$D \text{ accetta } w \text{ se e solo se } N \text{ accetta } w, \text{quindi } L(D) = L(N).$$

Osservazione.

Sia N una MdT non deterministica, sia D la MdT deterministica che simula N. Per determinare se w appartiene al linguaggio L(N) riconosciuto da N, occorre:

1) cercare la sequenza $c_1 \dots c_t$ che guida D verso una configurazione di accettazione ($c_1 \dots c_t$ può essere vista come una "prova" dell'appartenenza di w ad L(N)),
2) verificare che $c_1 \dots c_t$ è una "prova" dell'appartenenza di w ad L(N)

Corollario

Un linguaggio L è Turing riconoscibile se e solo se esiste una macchina di Turing non deterministica N che lo riconosce, cioè tale che L(N) = L.

Dimostrazione

Se L è il linguaggio L(N) riconosciuto da una macchina di Turing N, per il teorema precedente esiste una MdT deterministica D tale che L(D) = L(N) = L.

Viceversa, se M =$\left(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\right)$, è una MdT deterministica, la macchina di Turing non deterministica

M' =$\left(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject}\right)$, dove δ'(q, γ) = {δ(q, γ)} , per ogni $q \in Q$ e $\gamma \in \Gamma$, è tale che

$$L(M') = L(M).$$

Abbiamo visto che una macchina di Turing non deterministica N può essere simulata da una MdT deterministica D. È possibile modificare la prova di questo risultato per provare che se N è una macchina di Turing non deterministica che si arresta su ogni input, allora esiste una MdT deterministica D che simula N e quindi che si arresta su ogni input (cioè D è un decisore).

Corollario

Un linguaggio L è decidibile se e solo se esiste una macchina di Turing non deterministica N che lo decide.

Altre Varianti Equivalenti (NON TRATTATE)

- Altri esempi di varianti equivalenti sono:
- Macchina di Turing con nastro infinito (in entrambe le direzioni).
- Macchina di Turing con |Σ| = 1.
- Enumeratori (varianti di macchine di Turing per generare le stringhe in un linguaggio)

Definizione di Algoritmo

Algoritmi per eseguire calcoli in svariati campi erano noti già nell'antichità (esempio: Algoritmo di Euclide per il calcolo del MCD, circa 300 A.C.).

Turing ha il merito di aver definito in modo formale il concetto di algoritmo. L'esigenza nacque a causa di un problema posto da Hilbert, al Congresso internazionale di Matematica del 1900 (Parigi), il decimo di una lista di 23 problemi:

(**Decimo problema di Hilbert**) progettare un algoritmo per decidere se un polinomio ha radici in Z.

Esempio: $6x^3yz^2 + 3xy^2 - x^3 - 10$ ammette la radice x = 5, y = 3, z = 0.

Vi era l'assunzione implicita che ogni enunciato potesse essere provato o confutato mediante un algoritmo.

Nota: per provare che un problema è risolubile mediante un algoritmo basta esibire l'algoritmo. Per provare che per un problema non esiste nessun algoritmo che lo risolve, è necessario definire in modo formale la nozione di algoritmo. Risolvendo negativamente un altro problema posto da Hilbert nel 1928 (il problema della decisione), Turing introdusse nel 1936 il modello formale che da lui prende il nome. Usando tale modello e basandosi sul lavoro di Davis, Putnam e Robinson, molti anni dopo (1970), Matijasevic provò l'indecidibilità del decimo problema: **non esiste nessun algoritmo per decidere se un polinomio ha radici intere.**

Molti altri modelli formali per la definizione di algoritmo furono dati negli stessi anni 30:

- i sistemi di Post (da Post)
- le funzioni μ-ricorsive (Godel, Herbrand)
- il λ-calcolo (Church, Kleene)
- la logica combinatoria (Schonfinkel, Curry)

Fu dimostrato che tutti questi modelli erano equivalenti, cioè conducevano alla stessa classe di problemi risolvibili mediante algoritmi.

Tesi di Church

L'equivalenza dei vari modelli proposti condusse alla **Tesi di Church (o Tesi di Church-Turing):**

La macchina di Turing è la definizione formale della nozione intuitiva di algoritmo.

La tesi di Church non è un teorema ed è indimostrabile. Tenendo conto che essa è antecedente alla costruzione dei primi computer, costituì un enorme salto intellettuale.

L'evoluzione dei sistemi di calcolo ha generato negli ultimi anni una corrente di studi rivolti al superamento (o alla integrazione) del modello di Turing.

Decidibilità e indecidibilità

Problemi di decisione

Un problema di decisione è un problema che ha come soluzione una risposta sì o no.

Esempi:

- **Problema PRIMO**: Dato un numero x intero e maggiore di uno, x è primo?
- **Problema CONNESSO**: Dato un grafo G, G è connesso?
- **Problema A_{DFA}**: Dato un DFA B e una stringa w, B accetta w?

Richiami di logica

Variabili Booleane: variabili che possono assumere valore 1 (o TRUE) o 0 (o FALSE)

- Operazioni Booleane: ∨ (o OR), ∧ (o AND), ¬ (o NOT)
- Denotiamo ¬x con il negato di x
- $0 \vee 0 = 0,\ 0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1$
- $0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0,\ 1 \wedge 1 = 1$
- $\bar{0} = 1,\ \bar{1} = 0$

Definizione

Dato un insieme di variabili booleane X, le formule booleane (o espressioni booleane) su X sono definite induttivamente come segue:

- le costanti 0, 1 e le variabili (in forma diretta o complementata) x, $\bar{x}$, con x ∈ X, sono formule booleane.
- Se ϕ, ϕ1, ϕ2 sono formule booleane allora (ϕ1 ∨ ϕ2), (ϕ1 ∧ ϕ2), $\bar{\phi}$ sono formule booleane.

Una formula booleana ϕ è soddisfacibile se esiste un insieme di valori 0 o 1 per le variabili di ϕ (o assegnamento) che renda la formula uguale a 1 (assegnamento di soddisfacibilità). Diremo che tale assegnamento soddisfa ϕ o anche che rende vera ϕ.

Esempi:

- $\phi_1 = (\bar{x} \vee y) \wedge (x \vee \bar{z})$ è soddisfacibile (assegnamento: x = 0, y = 1, z = 0),
- $\phi_2 = (\bar{x} \wedge y) \vee (x \vee \bar{z})$ è soddisfacibile,
- $\phi_3 = (\bar{x} \vee x) \wedge (y \vee \bar{y})$ è soddisfacibile (per qualunque assegnamento di valori delle variabili),
- $\phi_4 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ non è soddisfacibile.

Una formula booleana ϕ è **soddisfacibile** se esiste un assegnamento di valori di verità F o T alle variabili di ϕ che soddisfa ϕ, cioè che renda la formula uguale a T.

- **Problema SAT**: Data una formula booleana ϕ, ϕ è soddisfacibile?

Un **cammino Hamiltoniano** in un grafo orientato è un cammino (orientato) che passa per ogni vertice del grafo una e una sola volta.

- **Problema HAMPATH**: Dato un grafo orientato G e due vertici s e t, esiste un cammino hamiltoniano nel grafo da s a t?

Istanze dei problemi di decisione

Un problema di decisione considera **elementi** di un insieme. Tali elementi sono anche chiamati le **istanze** del problema.

Quindi un istanza di un problema è un particolare input per quel problema.

- Esempio: 3, 4, 6 sono istanze per il problema PRIMO.
- Esempio: $(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$è un istanza per il problema SAT.

L'insieme delle istanze (per un problema) è **unione** del sottoinsieme delle **istanze con risposta sì** e del sottoinsieme delle **istanze con risposta no**.

- Esempio: 3 è un'istanza sì per il problema PRIMO, 4 e 6 sono istanze no per il problema PRIMO.
- Esempio: $(x_1 \vee x_2) \wedge (\overline{x_1} \vee x_2)$ è un istanza sì per il problema SAT (prendere x1 = x2 = T).
  $(x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ è un istanza no per il problema SAT.

Esempi:

- Nel problema **PRIMO** le istanze sono i **numeri** x interi e maggiori di uno. L'insieme $\{n \in N \mid n > 1\}$ di tali numeri è unione dell'insieme dei **numeri primi (istanze sì)** e dell'**insieme dei numeri non primi o composti (istanze no)**.
- Nel problema **CONNESSO** le istanze sono **grafi G**. L'insieme dei grafi è **unione** dell'**insieme dei grafi connessi (istanze sì)** e dell'**insieme dei grafi non connessi (istanze no)**.
- Nel problema $\boldsymbol{A_{DFA}}$ le istanze sono **coppie (B, w)** costituite da un DFA B e da una stringa w. L'insieme di tali coppie (B, w) è unione dell'insieme delle (B, w) con **w** ∈ **L(B) (istanze sì)** e dell'insieme delle (B, w) con **w** ∉ **L(B) (istanze no)**.

Problemi di decisione e ricerca di una soluzione

In realtà in generale siamo interessati a trovare una soluzione piuttosto che a sapere se c'è una soluzione. Chiamiamo **problemi di ricerca** quelli per i quali cerchiamo una soluzione (se esiste).

Esempi:

- RSAT: Data una formula booleana ϕ, fornire, se esiste, un assegnamento di valori di verità che soddisfa ϕ.
- RHAMPATH: Dato un grafo orientato G e due vertici s e t, fornire, se esiste, un cammino hamiltoniano nel grafo da s a t.

Dato un problema di ricerca possiamo in genere usare come **sottoprogramma** un algoritmo per il corrispondente problema di decisione, se tale algoritmo esiste (altrimenti né l'uno né l'altro è algoritmicamente risolubile).

Questo giustifica la concentrazione sui problemi di decisione.

Esempio: Data una formula booleana ϕ, vogliamo fornire, se esiste, un assegnamento di valori di verità che soddisfa ϕ. Disponiamo di un algoritmo **AI** che risolve SAT.

Usiamo **AI** per stabilire se ϕ è soddisfacibile.

- Se ϕ non è soddisfacibile abbiamo una risposta al nostro problema di ricerca: l'assegnamento non esiste.
- Se ϕ è soddisfacibile
  1. Assegniamo a una delle variabili x il valore T.
  2. Eseguiamo AI sulla nuova formula ottenuta.
  3. Se tale formula non è soddisfacibile, assegniamo a x il valore F. La formula ottenuta sarà ora soddisfacibile perché se ϕ è soddisfacibile, uno dei due valori è quello giusto.
  4. Chiamiamo ϕ' la formula ottenuta da ϕ assegnando in essa a x il "giusto" valore. Se in ϕ' vi sono ancora variabili, riapplichiamo la procedura dal passo 1 a ϕ'.

Linguaggio associato a un problema di decisione

Il nostro obiettivo è analizzare i limiti della risoluzione dei problemi mediante algoritmi, cioè mostrare l'esistenza di problemi non algoritmicamente risolubili. Abbiamo detto che assumeremo che "algoritmo" sia sinonimo di "macchina di Turing".

L'input di una macchina di Turing è una stringa. Se vogliamo dare in input altri oggetti, questi devono essere codificati come stringhe. Quindi le istanze devono essere "**rappresentate**" come stringhe su un alfabeto.

Le **istanze** del nostro problema saranno **rappresentate con stringhe** su un alfabeto Σ.

- La corrispondenza che a una istanza associa una stringa deve essere una **codifica**, cioè deve rappresentare univocamente l'istanza (e solo quella istanza).
- Nel seguito non definiremo nei dettagli le codifiche degli oggetti, daremo solo qualche esempio.
- Useremo $\langle \boldsymbol{\mathcal{O}} \rangle$ per denotare una stringa che codifica l'oggetto $\mathcal{O}$, useremo $\langle \boldsymbol{\mathcal{O}_1}, \dots, \boldsymbol{\mathcal{O}_k} \rangle$ per denotare una stringa che codifica gli oggetti $\langle \mathcal{O}_1, \dots, \mathcal{O}_k \rangle$.

Esempio: Problema CONNESSO

Dato un grafo G, G è connesso?

Le istanze in questo problema sono i grafi. $\langle G \rangle$ rappresenta una codifica di G.

Come possiamo codificare un grafo G mediante una stringa su un alfabeto Σ?

Una possibile codifica è illustrata su un esempio.

Consideriamo il grafo

$$G = (\{1,2,3\}, \{(1,2),(2,3),(3,1)\})$$

Possiamo prendere Σ = {0,1, (,), #} e associare a G la stringa:

$$\langle G \rangle = (1\#10\#11)\big((1\#10)\#(10\#11)\#(11\#1)\big)$$

Esempio: codifica di una MdT

È possibile codificare una MT M con una stringa su un alfabeto Σ.

Esempio: Σ = {0, 1},

$\langle M \rangle = 111C_1 11C_2 11 \dots 11C_n$ (**parziale**),

$C_t = 0^i 10^j 10^h 10^k 0^m$ è la codifica di $\delta(q_i, a_j) = (q_h, a_k, D_m)$, con D1 = L, D2 = R.

È possibile anche codificare una MT M e una stringa w con una stringa su un alfabeto Σ.

Esempio: Σ = {0,1},

$\langle M \rangle = 111C_1 11C_2 11 \dots 11C_n 111B_w 111$, dove $111C_1 11C_2 11 \dots 11C_n$ è la codifica (**parziale**) di M e B_w è una codifica di w su Σ ={0,1}.

In generale per codificare una macchina di Turing occorre stabilire come codificare

- i simboli dell'alfabeto di nastro,
- gli stati,
- i possibili movimenti della testina,

- i valori della funzione di transizione.

Va bene una codifica qualsiasi delle istanze? In questo ambito sì, va bene una codifica qualsiasi delle istanze, a condizione che:

- la codifica sia definita mediante un algoritmo informale
- sia possibile discriminare le stringhe codifiche di istanze da quelle che non lo sono mediante un algoritmo informale
- sia possibile, mediante un algoritmo informale, risalire dalla codifica all'unica istanza che la codifica rappresenta.

Poi vedremo come nella teoria della complessità assuma rilevanza anche considerare codifiche **non "prolisse"** cioè tali che non vi siano istanze la cui rappresentazione sia artificiosamente lunga.

Esempio: considerare codifiche in base k ≥ 2 dei numeri (cioè **escludere la rappresentazione unaria** dei numeri), grafi come coppie di insiemi (di nodi e archi) o mediante la matrice di adiacenza, insiemi, relazioni, funzioni mediante enumerazione delle codifiche dei relativi elementi....

Mentre l'insieme delle istanze si divide in due sottoinsiemi (l'insieme delle istanze sì e quello delle istanze no), l'insieme delle stringhe su Σ che rappresentano codifiche si divide in **tre** sottoinsiemi:

1. L'insieme delle stringhe w che codificano istanze con **risposta sì**.
2. L'insieme delle stringhe w che codificano istanze con **risposta no**.
3. L'insieme delle stringhe w che **non sono codifiche di istanze**.

Il linguaggio L **associato** a un problema di decisione P è il linguaggio delle **codifiche** delle istanze che hanno **risposta sì.**

Se esiste una macchina di Turing che decide L il problema viene detto **decidibile**. Altrimenti il problema viene detto **indecidibile**.

Se esiste una macchina di Turing che riconosce L il problema viene detto **semi decidibile**.

In questo modo esprimiamo un problema computazionale come un problema di riconoscimento di un linguaggio. La macchina corrisponde a un algoritmo per il problema.

Esempio: il linguaggio associato al problema "CONNESSO" è:

$$A = \{\langle G \rangle \mid G \text{ è } un\ grafo\ connesso\}$$

Dove$\langle G \rangle$ denota una codifica di G mediante una stringa su un alfabeto Σ. Risolvere CONNESSO equivale a trovare una macchina di Turing che *decida* A.

Esempio:
Sia G = (V, E) un grafo non orientato, con insieme V di nodi e insieme E di archi. Un sottoinsieme V' di nodi di G è un *independent-set* in G se per ogni u, v in V', la coppia (u, v) non è un arco, cioè u e v non sono adiacenti.

Definire il linguaggio INDEPENDENT-SET associato al seguente problema di decisione:

Sia G un grafo non orientato e k un intero positivo. G ha un independent-set di cardinalità k?

INDEPENDENT-SET=
$\{\langle G, k \rangle \mid G \text{ è } un\ grafo\ non\ orientato, k \text{ è } un\ intero\ positivo\ e\ G\ ha\ un\ indipendent\ set\ di\ cardinalità\ k\}$

Esempio:

Definire il linguaggio E_{TM} associato al seguente problema di decisione:

Input: M macchina di Turing.

Domanda: Il linguaggio L(M) riconosciuto da M è vuoto?

$$E_{TM} = \{\langle M \rangle \mid M \text{ è } una\ macchina\ di\ Turing\ tale\ che\ L(M) = \emptyset\}$$

Studieremo i linguaggi associati ai problemi di decisione in relazione con la classe dei linguaggi decidibili e la classe dei linguaggi Turing riconoscibili.

La macchina di Turing **verifica**, come passo preliminare, che l'input corrisponda a una codifica dell'input del problema. Prosegue la computazione solo in questo caso. Spesso nella descrizione della macchina di Turing, il passo preliminare corrispondente a questa verifica è omesso.

Problemi di decisione

Diremo che un problema di decisione è:

- **decidibile** se il linguaggio associato è decidibile
- **semi decidibile** se il linguaggio associato è Turing riconoscibile
- **indecidibile** se il linguaggio associato non è decidibile.

Lo studio dei problemi di decisione è importante per essere consapevoli che non tutti i problemi possono essere risolti mediante algoritmi/programmi. I problemi indecidibili non sono esoterici o lontani dai problemi di interesse informatico.

Esempi di problemi indecidibili sono:

- Stabilire se un programma si arresta.
- Stabilire se due programmi forniscono lo stesso output.
- Stabilire se un programma è un virus.

Metodo della Diagonalizzazione

PREMESSE:

Il metodo della diagonalizzazione fu introdotto da Georg Cantor nel 1873 mentre cercava come confrontare gli insiemi infiniti, in particolare come stabilire se, dati due insiemi infiniti, uno sia "più grande" dell'altro.

Cantor osservò che due insiemi finiti hanno la stessa cardinalità se gli elementi dell'uno possono essere messi in corrispondenza uno a uno con quelli dell'altro. Estese questo concetto agli insiemi infiniti.

Introdusse il metodo della diagonalizzazione per provare che esistono insiemi infiniti di differente cardinalità. In particolare, mostrò che l'insieme N dei numeri naturali ha cardinalità "inferiore" a quella dell'insieme R dei numeri reali.

OBIETTIVI:

Il metodo della diagonalizzazione fu successivamente usato per dimostrare che esistono linguaggi non Turing riconoscibili. Il metodo della diagonalizzazione e l'autoreferenzialità sono usati per dimostrare il teorema seguente:

Teorema

Il linguaggio

$A_{TM} = \{\langle M, w \rangle \mid M \text{ è } una\ macchina\ di\ Turing\ che\ accetta\ la\ parola\ w\}$ non è decidibile.

Funzioni

Definizione

Dati due insiemi non vuoti X e Y , una funzione $f: X \to Y \; da \; X \; in \; Y$ è una relazione che associa a ogni elemento x in X uno e un solo y = f(x) in Y. X è il **dominio** della funzione, Y è il **codominio** della funzione.

Definizione

Una funzione $f : X \to Y$ è $iniettiva \; se \; \forall \; x, x' \in X \;\; x \neq x' \Rightarrow f(x) \neq f(x')$

Definizione

Una funzione $f : X \to Y$ è $suriettiva \Leftrightarrow \forall \; y \in Y, \; \exists \; x \in X : y = f(x)$

Definizione

Una funzione $f : X \to Y$ è una funzione biettiva di X su Y (o una biezione tra X e Y) se f è iniettiva e suriettiva.

- **Esempio** $f: \{1,2,5\} \to \{2,4,7\}$, dove f (1) = 2, f (2) = 2, f (5) = 4, è una funzione. Non è né iniettiva né suriettiva.
- **Esempio** $f: \{1,2,5\} \to \{2,4,7,9\}$, dove f (1) = 2, f (2) = 4, f (5) = 7, è una funzione iniettiva ma non suriettiva.
- **Esempio** $f: \{1,2,5\} \to \{2,4\}$, dove f (1) = 2, f (2) = 4, f (5) = 2, è una funzione suriettiva ma non iniettiva.
- **Esempio** $f: \{1,2,5\} \to \{2,4,7\}$, dove f (1) = 2, f (2) = 4, f (5) = 7, è una funzione biettiva.

Cardinalità

Definizione

Due insiemi X e Y hanno la stessa cardinalità se esiste una funzione biettiva $f: X \to Y$ di X su Y.

$$|X| = |Y| \Leftrightarrow \; esiste \; una \; funzione \; biettiva \; f: X \to Y$$

- **Esempio** $f: \{1,2,5\} \to \{2,4,7\}$, dove f (1) = 2, f (2) = 4, f (5) = 7, è biettiva
- **Esempio** $f: N \to \{2n \mid n \in N\}$, dove f (n) = 2n, per ogni n ∈ N, è biettiva
- **Esempio (funzione coppia di Cantor)**

  Sia $Q^+ = \left\{\frac{x}{y} \mid x, y \in N, y \neq 0\right\}$ definita come segue è biettiva:

  $$\forall (x, y) \in N^2, f\big((x, y)\big) = \frac{(x+y)(x+y+1)}{2} + x = \frac{1}{2}\left((x+y)^2 + 3x + y\right) \in N$$

Numerabilità

Definizione

Un insieme X è numerabile se esiste una funzione biettiva $f: N \to X$ di N su X.

Un insieme X è enumerabile se esiste una funzione biettiva **calcolabile** $f: N \to X$ di N su X.

Un insieme X è contabile se è finito o numerabile.

- **Esempio** L'insieme dei numeri pari è numerabile.
- **Esempio** $Q^+ = \left\{\frac{x}{y} \mid x, y \in N, y \neq 0\right\}, N^2$ sono numerabili.

R non è numerabile

R non è numerabile.

Non esiste nessuna funzione biettiva di N sull'insieme R dei numeri reali.

Nella dimostrazione che segue potremmo anche limitarci a considerare i numeri reali nell'intervallo]0; 1[perché questo insieme ha cardinalità non superiore a R.

Idea della prova (metodo della diagonalizzazione).

Supponiamo per assurdo che esista una funzione biettiva f di N su R.

Allora possiamo costruire la tabella dove:

n	$f(n)$				
1	$d_{1,1}$	$d_{1,2}$	...	...	...
2	$d_{2,1}$	$d_{2,2}$	...	...	...
.	...	...	...	...	...
.	...	...	...	...	...
i	$d_{i,1}$	$d_{i,2}$	...	...	...
.	...	...	...	...	...
.	...	...	...	...	...

- $d_{1,1}, d_{1,2}$ … è la parte decimale del numero reale f(1),
- $d_{2,1}, d_{2,2}$ … è la parte decimale del numero reale f(2)
- e in generale $d_{i,1}, d_{i,2}$ … è la parte decimale del numero reale f(i).

e quindi:

- $f(1) = r_{1,}d_{1,1}d_{1,2}$ …,
- $f(2) = r_{2,}d_{2,1}d_{2,2}$ …,
- $in\ generale: f(i) = r_{i,}d_{i,1}d_{i,2}$ …

Le cifre sulla diagonale di questa matrice $d_{1,1}d_{2,2}$ …, definiscono un numero reale r = $0, d_{1,1}d_{2,2}$ …

Definisco ora il numero r' =$0, d'_{1,1}d'_{2,2}$ … che si ottiene scegliendo in ogni posizione (della parte decimale) una cifra diversa dalla corrispondente cifra in r. (quindi con $d'_{1,1}\ di\ r' \neq d_{1,1} di\ r, \dots$)

Da ciò ottengo che r' non è immagine di nessun intero positivo in quanto:

- non può essere f(1) perché $d'_{1,1} \neq d_{1,1}$,
- non può essere f(2) perché $d'_{2,2} \neq d_{2,2}$ e così via.

Cardinalità di insiemi conosciuti.

$$|X| \leq |Y| \Leftrightarrow esiste\ una\ funzione\ iniettiva\ f: X \to Y$$

$$se\ |X| \leq |Y|\ e\ |X| \neq |Y| \Rightarrow |X| < |Y|$$

$$se\ |X| \leq |Y|\ e\ |Y| \leq |X| \Rightarrow |X| = |Y|$$

$|N| < |R|$

- $n \in N \to n \in R$ è una funzione iniettiva di N in R. Quindi $|N| \leq |R|$.
- Non esiste nessuna funzione biettiva di N sull'insieme R dei numeri reali. Quindi $|N| \neq |R|\ e\ quindi\ N\ e\ R$ non hanno la stessa cardinalita.
- Quindi da $|N| \leq |R|\ e\ |N| \neq |R|$ deduciamo che $|N| < |R|$.

Nota. Lo stesso metodo può essere usato per mostrare che per ogni insieme S risulta $|S| < |\mathcal{P}(S)|$.

Il Metodo della diagonalizzazione per provare che esistono linguaggi non Turing riconoscibili

È possibile usare il metodo della diagonalizzazione per provare che esistono linguaggi che non sono Turing riconoscibili.

La prova consiste nel provare le affermazioni seguenti:

- Dato un alfabeto Σ, l'insieme Σ^* è numerabile.
- L'insieme delle codifiche delle macchine di Turing, e quindi l'insieme delle macchine di Turing è numerabile. Nota: la codifica di una Macchina di Turing è una stringa.
- L'insieme dei linguaggi Turing riconoscibili è numerabile. Nota: a ogni linguaggio Turing riconoscibile è associata (la codifica di) una macchina di Turing.
- L'insieme dei linguaggi sull'alfabeto Σ ha cardinalità maggiore del numerabile.

Numerabilità di Σ^*

Σ^* è numerabile.

Idea della dimostrazione:

Provare che $|N| \leq |\Sigma^*|$ e poi che $|\Sigma^*| \leq |N\ x\ N| = |N|$

Dimostrazione

Sia $\sigma \in \Sigma$, l'applicazione $f: n \in N \rightarrow \sigma^n \in \Sigma^*$ è iniettiva, quindi

$$|N| \leq |\Sigma^*|$$

Per ogni $n \in N$ sia $f_n: \Sigma^n \rightarrow N$ un'applicazione iniettiva. Siccome l'applicazione $g: w \in \Sigma^* \rightarrow \left(|w|, f_{|w|}(w)\right) \in N\ x\ N$ è iniettiva

$$|\Sigma^*| \leq |N\ x\ N| = |N|$$

Quindi $|\Sigma^*| = |N|$.

Esempio.

Sia Σ ={a, b}.

- Consideriamo l'applicazione f tale che $f(n) = a^n$. L'applicazione f è iniettiva, quindi $|N| \leq |\Sigma^*|$
- Per ogni $n \in N\ sia\ f_n: \Sigma^n \rightarrow N$ un'applicazione iniettiva.
  $$f_0(\epsilon) = 1, \quad f_1(a) = 1, \quad f_1(b) = 2.$$
  $$f_2(aa) = 1, \quad f_2(ab) = 2, \quad f_2(ba) = 3, \quad f_2(bb) = 4.$$
  $$f_n(w) = ?$$
- L'applicazione $g: w \in \Sigma^* \rightarrow \left(|w|, f_{|w|}(w)\right) \in N\ x\ N$ è iniettiva.
  Siano $x, y \in \Sigma^*$.
  Se $|x| \neq |y|$ allora $g(x) = \left(|x|, f_{|x|}(x)\right) \neq \left(|y|, f_{|y|}(y)\right) = g(y)$ perchè la prima coordinata è diversa.
  Se $|x| = |y|$ allora $f_{|x|}(x) \neq f_{|y|}(y)$ e $g(x) = \left(|x|, f_{|x|}(x)\right) \neq \left(|y|, f_{|y|}(y)\right) = g(y)$ perchè la seconda coordinata è diversa.

Linguaggi Turing riconoscibili e numerabilità

L'insieme

$$\{\langle M\rangle \mid M \text{ è } una\ \boldsymbol{TM\ sull'alfabeto\ \Sigma}\}$$

ha cardinalità minore o uguale a quella di N.

Dimostrazione

L'applicazione $f: \langle M \rangle \to \langle M \rangle \in \Sigma^*$ è iniettiva. Quindi

$$|\{\langle M \rangle \mid M \text{ è } una\ TM\ sull'alfabeto\ \Sigma\}| \leq |\Sigma^*| = |N|.$$

L'insieme dei linguaggi Turing-riconoscibili

$$\{L \subseteq \Sigma^* \mid L \text{ è } Turing\ riconoscibile\}$$

ha cardinalità minore o uguale a quella di N.

Dimostrazione

Possiamo associare a ogni linguaggio Turing riconoscibile una TM che lo riconosce e questa corrispondenza è iniettiva.

Quindi: $|\{L \subseteq \Sigma^* \mid L \text{ è } Turing\ riconoscibile\}| \leq |\{\langle M \rangle \mid M \text{ è } una\ TM\ sull'alfabeto\ \Sigma\}| \leq |N|$

Enumerazione di Stringhe

L'ordinamento radix produce una corrispondenza biunivoca tra N e Σ^* che permette di enumerare le stringhe:

$$w_0 = \epsilon, \qquad w_1 = a_1, \qquad w_2 = a_2, \qquad \dots,$$

Ad esempio, se Σ ={a, b}, la sequenza è $\epsilon, a, b, aa, ab, ba, bb, \dots$, e $w_1 = a,\ w_5 = ba,\ w_8 =$?

Non numerabilità dell'insieme dei Linguaggi

Sia $\Sigma^* = \{w_0, w_1, \dots, \}$

Sia B l'insieme delle sequenze binarie infinite cioè delle sequenze infinite di 0 e 1.

È possibile associare a ogni linguaggio L una sequenza infinita s_L (la sequenza caratteristica di L) così definita:

- il bit i-esimo di s_L è 1 se l'i-esima stringa w_i è in L, il bit i-esimo di s_L è 0 se $w_i \notin L$.

L'applicazione $f: \mathcal{P}(\Sigma^*) \to \mathcal{B}$ definita da f(L) = s_L è biettiva.

Non numerabilità dell'insieme delle sequenze binarie

L'insieme B non è numerabile.

Idea della prova (metodo della diagonalizzazione).

Mostriamo che non esiste nessuna funzione biettiva di N sull'insieme B delle sequenze binarie infinite. Supponiamo per assurdo che esista una funzione biettiva f di N su B.

Allora possiamo costruire

n	$f(n)$				
1	$d_{1,1}$	$d_{1,2}$	...	...	...
2	$d_{2,1}$	$d_{2,2}$	...	...	...
.	...	...	...	...	...
.	...	...	...	...	...
i	$d_{i,1}$	$d_{i,2}$	...	...	...
.	...	...	...	...	...

- $d_{1,1}, d_{1,2}$ … è la sequenza binaria infinita associata a f(1),
- $d_{2,1}, d_{2,2}$ … è la sequenza binaria infinita associata a f(2)
- e in generale $d_{i,1}, d_{i,2}$ … è la sequenza binaria infinita associata a f(i).

Le cifre sulla diagonale di questa matrice $d_{1,1}d_{2,2}$ …, definiscono una stringa binaria infinita s=$d_{1,1}d_{2,2}$ …,

Definisco ora la stringa binaria infinita $\bar{s}$=$\bar{d}_{1,1}\bar{d}_{2,2}$ … che si ottiene scegliendo in ogni il complemento della corrispondente cifra in s (quindi con $d_{1,1}\ di\ s \neq \bar{d}_{1,1} di\ s'$, …)

Da ciò ottengo che $\bar{s}$ non è immagine di nessun intero positivo in quanto:

- non può essere f(1) perché $\bar{d}_{1,1} \neq d_{1,1}$,
- non può essere f(2) perché $\bar{d}_{2,2} \neq d_{2,2}$ e così via.

Teorema.

L'insieme $\mathcal{P}(\Sigma^*)$ dei linguaggi su Σ non è numerabile.

Dimostrazione 1

Esiste un'applicazione biettiva f di $\mathcal{P}(\Sigma^*)$ in B, quindi

$$|\mathcal{B}| = |\mathcal{P}(\Sigma^*)|$$

Poiché, come abbiamo provato, B non è numerabile, concludiamo che $\mathcal{P}(\Sigma^*)$ non è numerabile.

Dimostrazione 2

Supponiamo per assurdo che $\mathcal{P}(\Sigma^*)$ sia numerabile.

Quindi esiste un'applicazione biettiva h di N in $\mathcal{P}(\Sigma^*)$.

Sia L_0, L_1 …, la lista dei linguaggi, cioè degli elementi di $\mathcal{P}(\Sigma^*)$, dove L_0= h(0), L_1= h(1) e in generale L_i=h(i).

Anche Σ^* è enumerabile e sia g una biezione di N in Σ^* e siano w_0, w_1 …, gli elementi di Σ^*, con w_i= g(i).

Costruiamo la tabella usando il metodo della diagonalizzazione:

	w_0	w_1	…	w_j	
L_0	0	1	…	0	…
L_1	1	1	…	1	…
L_2	0	0	…	1	…
.	…	…	…	…	…
L_i	1	0	…	0	…
.	…	…	…	…	…
.	…	…	…	…	…

La riga corrispondente ad L_i è la sequenza caratteristica di L_i. Scriviamo 1 all'incrocio tra la riga L_i e la colonna w_j se $w_j \in L_i$, altrimenti scriviamo 0.

Definiamo L $\forall i \geq 0,\ w_i \in L \Leftrightarrow w_i \notin L_i\ e\ L \in \mathcal{P}(\Sigma^*)$ ma per ogni i ≥ 0, $L \neq L_i$. Infatti, sia h tale che $L = L_h$.

La domanda "$\boldsymbol{w_h \in L}$?" conduce a una **contraddizione**.

- Assumiamo $w_h \in L$. Per la definizione di L deve essere $w_h \notin L_h$. Ma per ipotesi $L = L_h$, assurdo.

- Assumiamo $w_h \notin L$. Per la definizione di L deve essere $w_h \in L_h$.Ma per ipotesi $L = L_h$, assurdo.

Corollario

Esistono linguaggi che non sono Turing riconoscibili.

La funzione $h : \Sigma^* \to \mathcal{P}(\Sigma^*)$, dove h(w) = {w}, è **iniettiva**.

Quindi

$$|\{ L \subseteq \Sigma^* | \, L \text{ è } Turing\ riconoscibile\}| \leq |\Sigma^*| \leq |\mathcal{P}(\Sigma^*)|$$

Possiamo concludere che l'insieme dei linguaggi Turing riconoscibili è (al più) numerabile ma l'insieme di tutti i linguaggi ha cardinalità maggiore del numerabile.

Linguaggi Indecidibili

Il metodo della diagonalizzazione ci ha permesso di dimostrare che esistono linguaggi non Turing riconoscibili. La prova non era costruttiva per cui vorremmo poter esibire un **particolare linguaggio** che non sia Turing riconoscibile.

Esibiremo prima un **particolare linguaggio,** associato ad un problema di decisione, che è **indecidibile.** Nella prova viene utilizzato il metodo della diagonalizzazione e **l'autoreferenzialità.**

Autoreferenzialità e paradosso di Russel

Consideriamo i seguenti insiemi:

- A = l'insieme di tutti gli insiemi finiti
- B = l'insieme di tutti gli insiemi infiniti
- C =l'insieme di tutti gli insiemi che non sono elementi di sé stessi.

Domande :

$$A \in A? \, NO$$

$$B \in B? \, SI$$

$$A \in C? \, SI$$

$$B \in C \, ? \, NO$$

$$C \in C? \, ?$$

Esempio:

In un paese vive un solo barbiere, un uomo ben sbarbato che rade tutti e soli gli uomini del villaggio che non si radono da soli. Chi sbarba il barbiere?

Se il barbiere si sbarba da solo, vuole dire che non è lui il barbiere in quanto lui sbarba solo chi non si rade da solo(un paradosso). Se invece il barbiere non si sbarba da solo, allora dovrebbe essere sbarbato dal barbiere (ma il barbiere è lui) quindi si ha un paradosso.

Un linguaggio indecibile

Teorema

Il linguaggio

$$A_{TM} = \{\langle M, w\rangle | \, M \text{ è } una\ macchina\ di\ Turing\ e\ M\ accetta\ la\ parola\ w \}$$

non è decidibile.

Macchina di Turing Universale

Una TM universale U simula la computazione di una qualsiasi TM M. U riceve in input una **codifica** $\langle M, w\rangle$ di M e di un possibile input w di M.

La TM universale anticipò alcuni sviluppi fondamentali in informatica:

- Compilatore Java(o C,C++) scritto in Java(o scritto in C,C++)
- Sviluppo di un computer a programma memorizzato

Teorema

Esiste una TM Universale.

$$\langle M, w\rangle \rightarrow \text{MT Universale U} \rightarrow \begin{cases} accetta \rightarrow se\ M\ accetta\ w \\ rifiuta \rightarrow se\ M\ rifiuta\ w \\ non\ termina \rightarrow\ se\ M\ non\ termina \end{cases}$$

Teorema

Il linguaggio

$$A_{TM} = \{\langle M, w\rangle |\ M \text{ è } una\ macchina\ di\ Turing\ e\ M\ accetta\ la\ parola\ w\ \}$$

È Turing riconoscibile.

Dimostrazione

Definiamo una TM U una **TM Universale** che accetta A_{TM}:

sull'input $\langle M, w\rangle$ dove M è una TM e w è una stringa:

- Simula M sull'input w.
- Se M accetta, accetta; se M rifiuta, rifiuta.

Nota. U non termina su $\langle M, w\rangle$ se (e solo se) M non termina su w. Quindi U **non decide** A_{TM}

Teorema

Il linguaggio (**linguaggio associato al problema dell'accettazione di una macchina di Turing)**

$$A_{TM} = \{\langle M, w\rangle |\ M \text{ è } una\ macchina\ di\ Turing\ e\ M\ accetta\ la\ parola\ w\ \}$$

non è **decidibile**.

Dimostrazione

In questa dimostrazione, dato un decider S e una stringa w, S(w) denota il risultato della computazione sull'input w e cioè lo stato (q_{accept}, q_{reject}) nella configurazione di arresto raggiunta da S a partire dalla configurazione iniziale $q_0 w$.

Supponiamo per assurdo che A_{TM} sia decidibile. Quindi supponiamo che esista un **decisore H** che riconosca A_{TM}. In particolare, H accetta $\langle M, w\rangle$ se $\langle M, w\rangle \in A_{TM}$ e rifiuta $\langle M, w\rangle$ se $\langle M, w\rangle \notin A_{TM}$.(H accetta le stringhe di A_{TM} e rifiuta le stringhe che non sono in A_{TM})

Quindi:

$$H(\langle M, w\rangle) = \begin{cases} \boldsymbol{accetta}\ se\ M\ accetta\ w \\ \boldsymbol{rifiuta}\ se\ M\ non\ accetta\ w \end{cases}$$

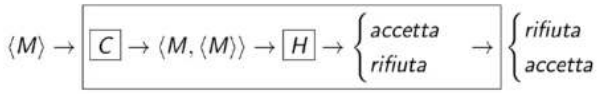
Costruiamo adesso una nuova **TM D** che usa **H** come sottoprogramma. Questa nuova macchina di Turing D chiama H su $\langle M, \langle M \rangle \rangle$. H accetta se M accetta $\langle M \rangle$ e rifiuta se M non accetta $\langle M \rangle$. Una volta che D ha determinato questa informazione, fa il contrario: rifiuta se M accetta ed accetta se M non accetta. Quindi:

D= "Sull'input $\langle M \rangle$, dove M è un TM:

- Simula H sull'input $\langle M, \langle M \rangle \rangle$
- Fornisce come output l'opposto di H, cioè se H accetta allora D rifiuta, se H rifiuta allora D accetta."

$$D(\langle M \rangle) = \begin{cases} \boldsymbol{rifiuta} \ se\ M\ accetta\ \langle M \rangle \\ \boldsymbol{accetta}\ se\ M\ non\ accetta\ \langle M \rangle \end{cases}$$

Graficamente, la macchina D è rappresentata dall'immagine sottostante:



D prende in input la codifica di M, crea la codifica per H copiando due volte M e valuta il risultato di H, dando in output il valore opposto.

Ora se diamo in input a D la sua stessa codifica $\langle D \rangle$ abbiamo che :

$$D(\langle D \rangle) = \begin{cases} \boldsymbol{rifiuta} \ se\ D\ accetta\ \langle D \rangle \\ \boldsymbol{accetta}\ se\ D\ non\ accetta\ \langle D \rangle \end{cases}$$

Cioè D accetta $\langle D \rangle$ se e solo se D non accetta $\langle D \rangle$, il che è ovviamente una **contraddizione.** Quindi le macchine D ed H **non possono esistere.**

NB(Facoltativo). Nella prova precedente è stato usato il metodo della diagonalizzazione(PAG 219 del libro).

In conclusione, abbiamo dimostrato che A_{TM} è Turing riconoscibile ma è indecidibile.

Che differenza c'è tra le due dimostrazioni?

Utilizzando il metodo della diagonalizzazione, abbiamo provato che esistono linguaggi che non sono Turing riconoscibili. Ma ancora non abbiamo visto un esempio di tale linguaggio.

Definizione

Diciamo che un linguaggio è **co-Turing riconoscibile** se $\bar{L}$ è Turing riconoscibile. Da ciò otteniamo che un linguaggio co-Turing riconoscibile è il complemento di un linguaggio Turing-riconoscibile.

Esercizi sulle proprietà di chiusura.

- **Esercizio 1**. La classe dei linguaggi Turing-riconoscibili è chiusa rispetto al complemento?
- **Esercizio 2**. La classe dei linguaggi decidibili è chiusa rispetto al complemento?
- **Esercizio 3**. La classe dei linguaggi Turing riconoscibili è chiusa rispetto all'unione? E rispetto all'intersezione?
- **Esercizio 4**. La classe dei linguaggi decidibili è chiusa rispetto all'unione? E rispetto all'intersezione?

Soluzione Esercizio 2 (cenni)

La classe dei linguaggi decidibili è chiusa rispetto al complemento. Sia A un linguaggio decidibile, sia M_A una macchina di Turing che decide A. Definiamo la macchina di Turing $M_{\bar{A}}$: sull'input w, $M_{\bar{A}}$ simula M_A e accetta w se e solo se $M_A$rifiuta w.

Poiché M_A si arresta su ogni input allora anche $M_{\bar{A}}$ si arresta su ogni input. Inoltre, il linguaggio di $M_{\bar{A}}$ è $\bar{A}$ perché $M_{\bar{A}}$ accetta w se e solo se $w \notin A$. Quindi $M_{\bar{A}}$ è una macchina di Turing che decide $\bar{A}$ ed $\bar{A}$ è decidibile.

Proprietà dei linguaggi decidibili

Teorema

Un linguaggio L è decidibile se e solo se L è Turing riconoscibile e co-Turing riconoscibile.

Dimostrazione.

Dobbiamo provare che L è decidibile ⇔ L e il suo complemento sono entrambi Turing riconoscibili.

(⇒) Se L è decidibile allora esiste un decider, cioè una macchina di Turing M con due possibili risultati di computazione(accettazione, rifiuto), tale che M accetta w se e solo se w ∈ L. In particolare, M riconosce L e quindi L è Turing riconoscibile. Inoltre, abbiamo provato che anche il complemento $\bar{L}$ di L è decidibile. Con lo stesso ragionamento concludiamo che $\bar{L}$ è Turing riconoscibile.

(⇐) Supponiamo che L e il suo complemento siano entrambi Turing riconoscibili. Sia M_1 una TM che riconosce L e M_2 una TM che riconosce $\bar{L}$. Definiamo una TM M tale che :

M= "Su input w:

- Esegue sia M_1 che M_2 su input w in parallelo.
- Se M_1accetta , accetta. Se M_2 accetta, rifiuta."

Quindi M è una macchina di Turing a due nastri. M, dopo aver copiato w sul secondo nastro, simula M_1 sul primo nastro e M_2 sul secondo nastro e alterna la simulazione di un passo di M_1con un passo di M_2 e continua finchè una delle due accetta.

Vogliamo provare che M decide L. Per farlo dobbiamo provare che:

- **(1)** M è un decisore
- **(2)** M riconosce L, cioè L=L(M)
  - **(1)** M è un decisore poiché, per ogni stringa w abbiamo due casi: w è in L, oppure w è in $\bar{L}$. Pertanto, una tra M_1 *e* M_2 deve accettare w. Poiché M si ferma ogni volta che M_1 accetta oppure M_2 accetta, allora M si ferma sempre. Quindi M è un decisore.
  - **(2)** Inoltre:
    1. $w \in L$. Ma $w \in L$ se e solo se M_1 accetta w. Quindi M accetta w.
    2. $w \notin L$. Ma $w \notin L$ se e solo se M_2 accetta w. Quindi M rifiuta w (conseguenza : se M accetta w allora $w \in L$)

Siccome M accetta w se e solo se $w \in L$ possiamo concludere che L(M) = L.

Un linguaggio non Turing riconoscibile

Teorema

$\overline{A_{TM}}$ non è Turing riconoscibile.

Dimostrazione.

Supponiamo per assurdo che $\overline{A_{TM}}$ sia Turing riconoscibile. Sappiamo che A_{TM} è Turing riconoscibile. Quindi A_{TM} sarebbe Turing riconoscibile e co-Turing riconoscibile. Per il precedente teorema, A_{TM} sarebbe decidibile, ma ciò è assurdo poiché abbiamo dimostrato che A_{TM} è indecidibile.

Riducibilità mediante funzione

Descrizione informale

È importante riuscire a riconoscere che un problema P è decidibile. Abbiamo due possibilità per farlo:

- Supporre l'esistenza di una TM che decide il linguaggio associato a P e provare che questo conduce ad una contraddizione.
- Considerare un problema P' di cui sia nota l'indecidibilità del linguaggio associato e dimostrare che P' "non è più difficile" del problema in questione P.
- Teorema di Rice.

Esempio:

$$\Sigma = \{0,1\}$$

$$EVEN = \{w \in \Sigma^* | \; w \text{ è } la\ rappresentazione\ binaria\ di\ n \in N\ pari\}$$

$$ODD = \{w \in \Sigma^* | \; w \text{ è } la\ rappresentazione\ binaria\ di\ n \in N\ dispari\}$$

Sia w ∈ Σ^* e sia n il corrispondente decimale di w. È facile costruire la TM *INCR:*

$$w \rightarrow INCR \rightarrow w' \; (= rappresentazione\ binaria\ di\ n+1)$$

Questa macchina l'abbiamo già vista e calcola il successore di un numero in binario.

MdT successor S_r

S_r calcola la funzione *Successor(n)=n+1* con input e output rappresentati in base 2.

Per costruire tale macchina INCR useremo:

- La macchina M_1 che calcola la funzione f, dove f(x) = $x.
- La macchina M_1^R che calcola la funzione g=f^{-1} ossia g($x) = x

È possibile generalizzare l'algoritmo per calcolare il successore di n con n rappresentato in una base qualsiasi b, con b intero positivo, b>2.

Definiamo INCR:

"Su input w:

1) Usa la macchina M_1 e trasforma w in e trasforma w in $w
2) Nello stato q_r scorre l'input da sinistra a destra fino a incontrare il simbolo ⊔
3) Passa nello stato q_l, si sposta a sinistra cambiando ogni carattere 1 che vede in 0 fino a leggere un carattere diverso da 1 (Nota: questo può accadere anche senza incontrare alcun 1)
4) Se questo carattere è 0 lo cambia in 1, si sposta a sinistra fino a leggere $. A questo punto usa la macchina M_1^R che elimina $ e sposta a sinistra di una casella la stringa di caratteri 0 e 1 sul nastro e si ferma.
5) Se questo carattere diverso da 1 è $, l'input era una stringa di caratteri uguali a 1. Allora cambia $ in 1 e si ferma su questo carattere."

Abbiamo da ciò quindi definito una trasformazione (funzione calcolabile)

$$f: w \in \Sigma^* \rightarrow w' \in \Sigma^*$$

Tale che:

$$w = \langle n \rangle \in EVEN \Leftrightarrow f(w) = w' = (n+1) \in ODD$$

Tale funzione f è una **riduzione** di EVEN a ODD.

Nota: f è definita su tutto Σ^*, non solo su EVEN E ODD.

EVEN "non è più difficile" di ODD: se esiste una TM R che decide ODD, la TM S decide anche EVEN.

$$S: w \to INCR \to w' \to R$$

Viceversa, se EVEN è indecidibile proviamo che anche ODD lo è: se per assurdo esistesse una TM R che decide ODD, la TM S deciderebbe EVEN.

IDEA: convertire le **istanze** di un problema P nelle istanze di un problema P' in modo che un algoritmo per P', **se esiste,** possa essere utilizzato per progettare un algoritmo per P: P non è più difficile di P'.

Sia A il linguaggio associato a P, sia B il linguaggio associato a P'. Allora proveremo che:

- B decidibile ⇒ A decidibile,
- A indecidibile ⇒ B indecidibile.(usando la definizione di contronominale)

Nota. Nulla è detto sulla decidibilità di A o B ma solo sulla decidibilità di A assumendo di disporre di una algoritmo per decidere B.

Definizione

Una funzione $f: \Sigma^* \to \Sigma^*$ è **calcolabile** se esiste una TM tale che su ogni input w, M si arresta con f(w), e solo con f(w), sul suo nastro.

Definizione

Un linguaggio $A \subseteq \Sigma^*$ è **riducibile mediante funzione** a un linguaggio $B \subseteq \Sigma^*$ $(\boldsymbol{A \leq_M B})$ se esiste una funzione calcolabile $f: \Sigma^* \to \Sigma^*$ tale che :

$$\forall w \in \Sigma^* \; w \in A \Leftrightarrow f(w) \in B$$

La funzione f è chiamata una **riduzione** da A a B.

Esempio: EVEN $\leq_m$ODD. Una riduzione è la funzione f da $\{0,1\}^*$ in $\{0,1\}^*$ tale che f($\langle n \rangle$)=$\langle n+1 \rangle$

Teorema.

$$A \leq_m B \; se \; e \; solo \; se \; \bar{A} \leq_m \bar{B}$$

Dimostrazione

Per ipotesi $A \leq_m B$, quindi esiste una riduzione di A a B. poiché f è una riduzione, f è calcolabile e inoltre

$$\forall w \in \Sigma^* \; w \in A \Leftrightarrow f(w) \in B$$

Proviamo che f è anche una riduzione da $\bar{A}$ a $\bar{B}$.

Infatti, poiché f è una riduzione, f è calcolabile e inoltre

$$\forall w \in \Sigma^* \; w \in A \Leftrightarrow f(w) \in B$$

Quindi:

$$\forall w \in \Sigma^* \; w \notin A \Leftrightarrow f(w) \notin B$$

(ottenuta per equivalenza negando la prima)

Cioè:

$$\forall w \in \Sigma^* \; w \in \bar{A} \Leftrightarrow f(w) \in \bar{B}$$

Teorema.

Se A$\leq_m$B e B è decidibile allora A è decidibile.

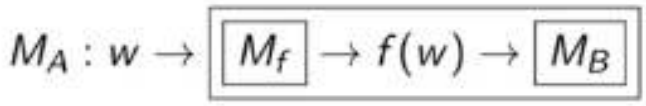
Dimostrazione.

Sia M_B una macchina di Turing che decide B, sia f una riduzione da A a B e sia M_f una macchina di Turing che calcola f.

Consideriamo la macchina di Turing M_A:

M_A="Sull'input w:

- simula M_f e calcola f(w)
- simula M_B su f(w)
- se M_B accetta, accetta; se M_B rifiuta, rifiuta."



M_A decide A. Infatti, si ferma su w se si fermano M_f e M_B. Ora, per ogni w, M_f si ferma con f(w) sul nastro e per ogni w, M_B si ferma su f(w) perché M_B è un decider.

Inoltre, M_A riconosce A. Infatti:

$$w \in L(M_A) \Leftrightarrow f(w) \in L(M_B)(per\ la\ definizione\ di\ M_A)$$

$$w \in L(M_A) \Leftrightarrow f(w) \in B(per\ la\ definizione\ di\ M_B)$$

$$w \in L(M_A) \Leftrightarrow w \in A(per\ la\ definizione\ di\ riduzione)$$

Teorema.

Se A$\leq_m$B e B è Turing riconoscibile allora A è Turing riconoscibile.

Dimostrazione.

Sia M_B una macchina di Turing che riconosce B, sia f una riduzione da A a B e sia M_f una macchina di Turing che calcola f.

Consideriamo la macchina di Turing M_A:

M_A="Sull'input w:

- simula M_f e calcola f(w)
- simula M_B su f(w)
- se M_B accetta, accetta; se M_B rifiuta, rifiuta."

M_A riconosce A. Infatti:

$$w \in L(M_A) \Leftrightarrow f(w) \in L(M_B)(per\ la\ definizione\ di\ M_A)$$

$$w \in L(M_A) \Leftrightarrow f(w) \in B(per\ la\ definizione\ di\ M_B)$$

$$w \in L(M_A) \Leftrightarrow w \in A(per\ la\ definizione\ di\ riduzione)$$

Corollario

Se A$\leq_m$B e A non è Turing riconoscibile allora B non è Turing riconoscibile.

Dimostrazione
Se B fosse Turing riconoscibile allora lo sarebbe anche A in virtù del teorema precedente (abbiamo usato il contronominale)

Corollario
Se A$\leq_m$B e A è indecidibile allora B è indecidibile.

Dimostrazione
Se B fosse decidibile allora lo sarebbe anche A in virtù del teorema precedente (abbiamo usato il contronominale).

Torniamo alla definizione di funzione calcolabile.

Funzione calcolabile
Una funzione $f: \Sigma^* \rightarrow \Sigma^*$ è **calcolabile** se esiste una TM tale che su ogni input w, M si arresta con f(w), e solo con f(w), sul suo nastro.

È importante sottolineare la differenza tra il definire una funzione f, cioè definire i valori di f e calcolare tali valori.

Le seguenti funzioni aritmetiche sono calcolabili (dove n, m$\in N$).

- $incr(n) = n + 1$
- $dec(n) = \begin{cases} n-1 \; se \; n > 0 \\ 0 \; se \; n = 0 \end{cases}$
- $(m,n) \rightarrow m + n$
- $(m,n) \rightarrow m - n$
- $(m,n) \rightarrow m \cdot n$

Le funzioni possono essere anche trasformazioni di descrizioni di macchine. Stabilire se funzioni di questo tipo sono calcolabili può non essere facile, soprattutto se le macchine di Turing sono descritte ad alto livello.

Esempio:
Data una macchina di Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$, in questo esempio denotiamo con M' la macchina di Turing che accetta le stringhe rifiutate da M e rifiuta quelle accettate (in generale M' **NON** riconosce il complemento di L(M)).

Consideriamo la funzione $f: \Sigma^* \rightarrow \Sigma^*$ così definita.

$$f(y) = \begin{cases} \epsilon, & se \; y \neq \langle M \rangle, M \; MdT \\ \langle M' \rangle, & se \; y = \langle M \rangle \end{cases}$$

Una tale funzione è calcolabile?

Consideriamo la MT F che sull'input y:

- Se $y \neq \langle M \rangle$, restituisce ϵ
- Se $y = \langle M \rangle$, "costruisce" la MT M' così definita:
  M' = Sull'input x:
  1. "simula" M su x
  2. Se M accetta, rifiuta
  3. Se M rifiuta, accetta
- Fornisce in output $\langle M' \rangle$

Ma F come "costruisce"? ed M' come simula?

F scorre l'input per verificare se è "legale". In caso affermativo , F copia la codifica di M su un altro nastro. Chiamiamo δ la funzione di transizione di M e δ' quella della macchina M' che sarà costruita.

F cerca nella copia della codifica di M la codifica delle transizioni della forma

$$\delta(q,a) = (q_{accept}, a', D), \qquad D \in \{L,R\}, \qquad a,a' \in \Gamma, \qquad q \in Q$$

e cambia ognuna di esse con la codifica di

$$\delta'^{(q,a)} = (q_{reject}, a', D), \qquad D \in \{L,R\}, \qquad a,a' \in \Gamma, \qquad q \in Q$$

Fa un'azione analoga sulla codifica delle transizioni del tipo

$$\delta(q,a) = (q_{reject}, a', D), \qquad D \in \{L,R\}, \qquad a,a' \in \Gamma, \qquad q \in Q$$

Che cambia con la codifica

$$\delta'^{(q,a)} = (q_{accept}, a', D), \qquad D \in \{L,R\}, \qquad a,a' \in \Gamma, \qquad q \in Q$$

Esiste una tale MT F?

Sì, perché F deve solo scorrere l'input, verificare se è legale e poi cambiare i caratteri in esso contenuti!.

Funzione non calcolabile.

Corollario

Se A$\leq_m$B e A è indecidibile allora B è indecidibile.

Esempio di esercizio.

Consideriamo A_{TM} e B= {ab}

Consideriamo la funzione $f: \Sigma^* \to \Sigma^*$, dove $a, b \in \Sigma^*$, così definita:

$$f(y) = \begin{cases} ab, & se\ y = \langle M,w\rangle \in A_{TM} \\ a, & altrimenti \end{cases}$$

Quindi f è una funzione tale che f(y) = a se non è della forma $\langle M,w\rangle$, oppure se y = $\langle M,w\rangle$ con $\langle M,w\rangle \notin A_{TM}$.

Invece f(y) = ab se y = $\langle M,w\rangle\ con\ \langle M,w\rangle \in A_{TM}$

Quindi per ogni $y \in \Sigma^*$,

$$y \in A_{TM} \Leftrightarrow f(y) \in \{a,b\}$$

Provare che f non è calcolabile è semplice in quanto se fosse calcolabile avrei una riduzione da $A_{TM}\ in\ B$. Ma ciò non è possibile per cui f non è calcolabile.

Nei teoremi seguenti proveremo l'esistenza di riduzioni da A_{TM} (o da un altro linguaggio indecidibile) ad alcuni linguaggi B associati a problemi di decisione sulle macchine di Turing. Una conseguenza importante di tali teoremi è che ognuno di questi linguaggi B è indecidibile.

In generale, quando descriviamo una macchina di Turing che calcola una riduzione da A a B, assumiamo che gli input che non sono "della forma corretta" sono mappati in stringhe al di fuori di B. Più precisamente, se A è il linguaggio associato a un problema di decisione, assumiamo che l'input sia la codifica di una istanza del problema.

Ad esempio, se A = A_{TM}, definiremo la riduzione(e la macchina di Turing che la calcola) sulle stringhe della forma $\langle M, w\rangle$, dove M è una TM e w è una stringa.

Dati:

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ è } una\ TM\ e\ w \in L(M)\}$$

$$HALT_{TM} = \{\langle M, w\rangle \mid M \text{ è } una\ TM\ e\ M\ si\ arresta\ su\ w\}$$

Teorema

$$A_{TM} \leq_m HALT_{TM}$$

Dimostrazione

Per provare che $A_{TM} \leq_m HALT_{TM}$ dobbiamo definire una funzione calcolabile $f: \Sigma^* \to \Sigma^*$ tale che, per ogni stringa $\langle M, w\rangle$, con M MT e w stringa,

$$\langle M, w\rangle \in A_{TM} \Leftrightarrow \langle M', w\rangle \in HALT_{TM}$$

Consideriamo la MT F che, sull'input $\langle M, w\rangle$:

- costruisce la macchina M':
  M'="Sull'input x:
  a) Simula M su x
  b) Se M accetta, accetta
  c) Se M rifiuta, cicla"
- Fornisce in output $\langle M', w\rangle$

Nota. La macchina M' si ferma su input x se e solo se M accetta x

La funzione f calcolata da F, che associa a $\langle M, w\rangle$ la stringa $\langle M', w\rangle$, è una riduzione da A_{TM} a $HALT_{TM}$.

Infatti, f è calcolabile (cioè è possibile definire una macchina di Turing F che ha il comportamento input/output descritto prima).

Inoltre:

$$\langle M, w\rangle \in A_{TM} \Leftrightarrow M\ accetta\ w \Leftrightarrow M'si\ arresta\ su\ w \Leftrightarrow \langle M', w\rangle \in HALT_{TM}$$

Teorema

Dati:

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ è } una\ TM\ e\ w \in L(M)\}$$

$$E_{TM} = \{\langle M\rangle \mid M \text{ è } una\ TM\ e\ L(M) = \emptyset\}$$

Allora:

$$A_{TM} \leq_m \overline{E_{TM}}$$

Dimostrazione

Data una MT M e una stringa w, sia M_1 la macchina di Turing tale che, su un input x:

Se $x \neq w$ allora M_1 si ferma e rifiuta

Se $x = w$ allora M_1simula M su w e accetta x se M accetta x = w

$$L(M_1) = \begin{cases} \{w\}, & se \langle M, w \rangle \in A_{TM} \\ \emptyset, & altrimenti \end{cases}$$

La funzione f che associa a $\langle M, w \rangle$ la stringa $\langle M_1 \rangle$, è una riduzione da A_{TM} *a* $\overline{E_{TM}}$.

Infatti, f è calcolabile: possiamo costruire una macchina di Turing F che sull'input $\langle M, w \rangle$, fornisce in output $\langle M_1 \rangle$. Inoltre:

$$\langle M, w \rangle \in A_{TM} \Leftrightarrow M \; accetta \; w \Leftrightarrow L(M_1) \neq \emptyset \Leftrightarrow \langle M_1 \rangle \in \overline{E_{TM}}$$

Teorema

$\overline{E_{TM}}$ è indecidibile.

Corollario

E_{TM} è indecidibile.

Prova. La classe dei linguaggi decidibili è chiusa rispetto al complemento. Se E_{TM} fosse decidibile lo sarebbe anche $\overline{E_{TM}}$ e questo è in contraddizione con il teorema.

NOTA. **Non esiste nessuna riduzione mediante funzione da A_{TM} a E_{TM}.**

Dato:

$$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ è } una \; TM \; e \; L(M) \text{ è } regolare\}$$

Teorema

$$A_{TM} \leq_m REGULAR_{TM}$$

Dimostrazione

Proviamo che $A_{TM} \leq_m REGULAR_{TM}$.

Data una MT M e una stringa w, sia R la macchina di Turing tale che, su un input x:

1. Se $x \in \{0^n 1^n \mid n \in N\}$, allora R si ferma e accetta x.
2. Se $x \notin \{0^n 1^n \mid n \in N\}$, allora R simula M su w e accetta x se M accetta w.

$$\mathrm{L(R)} = \begin{cases} \Sigma^*, & \text{se } \langle M, w \rangle \in A_{TM} \\ \{0^n 1^n \mid n \in N\}, & \text{altrimenti} \end{cases}$$

La funzione f , che associa a $\langle M, w \rangle$ la stringa $\langle R \rangle$, è una riduzione da A_{TM} *a* $REGULAR_{TM}$.

Infatti, f è calcolabile: possiamo costruire una macchina di Turing F che sull'input $\langle M, w \rangle$ fornisce in output $\langle R \rangle$.

Inoltre, abbiamo osservato che

• se M accetta w allora $L(R) = \Sigma^*$;

• se M non accetta w allora $L(R) = \{0^n 1^n \mid n \in N\}$,

Quindi

$$\langle M, w \rangle \in A_{TM} \Leftrightarrow M \; accetta \; w \Leftrightarrow L(R) \text{ è } regolare \Leftrightarrow \langle R \rangle \in REGULAR_{TM}$$

Teorema

$$REGULAR_{TM} = \{\langle M \rangle \mid M \text{ è } una \; TM \; e \; L(M) \text{ è } regolare\}$$

È indecidibile.

Dati:

$$E_{TM} = \{\langle M\rangle \mid M \text{ è } una\ TM\ e\ L = \emptyset\}$$

$$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2\ sono\ TM\ e\ L(M_1) = L(M_2)\}$$

Teorema

$$E_{TM} \leq_m EQ_{TM}$$

Dimostrazione.

Sia M_1 una macchina di Turing tale che $L(M_1) = \emptyset$. Quindi, data una macchina di Turing M, avremo che

$L(M) = L(M_1)$ se e solo se $L(M) = \emptyset$.

La funzione f , che associa a $\langle M\rangle$ la stringa $\langle M, M_1\rangle$, è una riduzione da $E_{TM}\ a\ EQ_{TM}$.

Infatti, f è calcolabile: possiamo costruire una macchina di Turing F che sull'input $\langle M\rangle$, fornisce in output $\langle M, M_1\rangle$. Inoltre:

$$\langle M\rangle \in E_{TM} \Leftrightarrow L(M) = \emptyset \Leftrightarrow L(M) = L(M_1) \Leftrightarrow \langle M, M_1\rangle \in EQ_{TM}.$$

Teorema

$$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2\ sono\ TM\ e\ L(M_1) = L(M_2)\}$$

È indecidibile.

Teorema di Rice

Sia $L = \{\langle M\rangle \mid M \text{ è } una\ TM\ che\ verifica\ la\ proprietà\ \mathcal{P}\}$ un linguaggio che soddisfa le seguenti condizioni:

1. L'appartenenza di $\langle M\rangle$ a L dipende solo da L(M), cioè
   $\forall M_1, M_2\ TM\ tali\ che\ L(M_1) = L(M_2), \langle M_1\rangle \in L \Leftrightarrow \langle M_2\rangle \in L$ ed L è un linguaggio "non banale", cioè L non è vuoto e che non contiene le codifiche di tutte le MdT :
2. $\exists M_1\ TM\ tale\ che\ \langle M_1\rangle \in L$;
3. $\exists M_2\ TM\ tale\ che\ \langle M_2\rangle \notin L$

Allora L è indecidibile.

Teoria della Complessità.

Se un linguaggio associato a un problema di decisione è decidibile, è sempre possibile in teoria scrivere un programma che, data in input una istanza del problema, dia in output la risposta al problema. E in pratica?

Anche quando un problema è decidibile, e quindi in linea di principio computazionalmente risolvibile, può non essere risolvibile in pratica se la soluzione richiede una quantità eccessiva di tempo o di memoria.

Iniziamo dal tempo. Come primo passo, introdurremo un metodo per misurare il tempo usato per risolvere un problema.

Poi mostreremo come classificare i problemi in base alla quantità di tempo che essi richiedono. Infine, consideriamo la possibilità che determinati problemi decidibili richiedano enormi quantità di tempo e di come sia possibile determinare quando ci troviamo di fronte a un problema di questo tipo. Ci riferiremo principalmente a problemi di decisione e quindi ai linguaggi associati.

Da questo momento in poi:

- Tutti i linguaggi considerati sono decidibili

- Tutte le MDT considerate sono decider.

Li studieremo dal punto di vista della "quantità di risorse di calcolo utilizzate" nella computazione.

Considereremo il "tempo" utilizzato nella computazione.

UNA MISURA DEL TEMPO

Il numero di passi che utilizza un algoritmo su un particolare input può dipendere da diversi parametri.

Ad esempio, se l'input è un grafo, il numero di passi può dipendere dal numero di nodi, dal numero di archi e dal grado massimo del grafo, o da una combinazione di questi e/o altri parametri.

Il tempo di esecuzione di un algoritmo sarà calcolato in funzione della lunghezza della stringa che rappresenta l'input senza considerare eventuali altri parametri.

Ad esempio, se l'input è un grafo G, il tempo di esecuzione di un algoritmo sui grafi sarà calcolato in funzione della lunghezza della codifica $\langle G \rangle$ di G.

Considereremo l'analisi del caso peggiore, quindi valuteremo il tempo di esecuzione massimo tra tutti gli input di una determinata lunghezza. (Non considereremo l'analisi del caso medio.)

Definizione.

Sia $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ una MdT deterministica, a nastro singolo, che si arresta su ogni input. **La complessità di tempo** di M è la funzione $f : N \rightarrow N$ dove f(n) è il massimo numero di passi di computazione eseguiti da M su un input di lunghezza n, n ∈ N.

Se M ha complessità di tempo f(n), diremo che **M decide L(M) in tempo (deterministico) f(n).**

Se $C_1, C_2, \ldots, C_{k+1}$, $k \geq 1$, sono configurazioni di M, tali che:

1. $C_1 = q_0 w$ è la configurazione iniziale di M con input w
2. $C_i \rightarrow C_{i+1}$ per ogni i ∈ {1, . . . , k}
3. C_{k+1} è una configurazione di arresto,

il numero di passi eseguiti da M su w è k.

Quindi, se f è la complessità di tempo di M, f(n) = massimo numero di passi in $q_0 w \rightarrow^* uqv$, $q \in \{q_{accept}, q_{reject}\}$, al variare di w in Σ^n.

Alcune astrazioni.

- Identifichiamo gli input che hanno la stessa lunghezza,
- Valuteremo la complessità nel **caso peggiore** (cioè relativo alla stringa di input di lunghezza n che richiede il maggior numero di passi),
- **Non** valuteremo **esattamente** la complessità di tempo f(n) di M ma piuttosto stabiliremo un limite asintotico superiore per f(n), usando la notazione O-grande (**analisi asintotica**).
- Quindi, dire che M ha complessità di tempo O(g(n)) vuol dire che M ha complessità di tempo f(n) ed f(n) è O(g(n)).

Analisi Asintotica

R^+= insieme dei numeri reali positivi.

Definizione

Siano f e g due funzioni $f: N \rightarrow R^+$, $g: N \rightarrow R^+$.

Diremo che f (n) è O(g(n)) oppure f (n) = O(g(n)) se esistono una costante **c** > 0 e una costante $\boldsymbol{n_0} \geq 0$ tali che, per ogni $\boldsymbol{n} \geq \boldsymbol{n_0}$,

$$f(n) \leq cg(n).$$

Diremo che g(n) è un limite superiore (asintotico) per f(n).

Esempio1. Qual è la complessità di tempo della macchina di Turing M che:

1. rifiuta se l'input è la parola vuota
2. cancella il primo carattere dell'input e accetta, se l'input è una stringa non vuota

Ovviamente in questo caso la complessità è O(1) in quanto in ogni caso la macchina esegue un numero constante di passi indipendente dall'input.

Esempio2. Qual è la complessità di tempo di una macchina di Turing M che copia il suo input?

In caso di una macchina a singolo nastro, la macchina ha complessità O(n^2) poiché deve scorre avanti e indietro il nastro per ogni carattere dell'input, quindi, fa 2n passi per ogni n caratteri ossia O(2n x n).

Esempio3.

Dato il linguaggio

$$L = \{0^k 1^k \mid k \geq 0\}$$

Abbiamo visto una MT M a nastro singolo che decide L.

Sull'input w, M:

1. Verifica che $w \in L(0^*1^*)$.
2. Partendo dallo 0 più a sinistra, sostituisce 0 con ⊔ poi va a destra, ignorando 0 e 1, fino a incontrare ⊔.
3. Verifica che immediatamente a sinistra di ⊔ ci sia 1: se così non è, rifiuta w. Altrimenti, sostituisce 1 con ⊔ e va a sinistra fino a incontrare ⊔.
4. Guarda il simbolo a destra di ⊔:
    a. Se è un altro ⊔, allora accetta w.
    b. Se è 1, allora rifiuta w.
    c. Se è 0 ripete a partire dal passo 2.

Analisi dell'algoritmo.

Sia |w| = n, cioè utilizziamo n per rappresentare la lunghezza dell'input.

Per analizzare M, consideriamo ciascuna delle sue quattro fasi separatamente.

Nella prima fase la macchina scansiona il nastro per verificare che l'input è in $L(0^*1^*)$, cioè del tipo 0^t1^q, $t, q \in N$. Tale operazione di scansione usa n passi, dove n = |w|. Per riposizionare la testina all'estremità sinistra del nastro utilizza ulteriori n passi. Per cui il totale di passi utilizzati in questa fase è 2n passi. Nella notazione O-grande diciamo che questa fase usa O(n) passi.

Si noti che non abbiamo fatto menzione del riposizionamento della testina del nastro nella descrizione della macchina. L'utilizzo della notazione asintotica ci permette di omettere quei dettagli della descrizione della macchina che influenzano il tempo di esecuzione al più di un fattore costante.

Le azioni 2 e 3 richiedono ciascuna O(n) passi e sono eseguite al più $\frac{n}{2}$ volte. Infatti, la macchina esegue ripetutamente la scansione del nastro e cancella uno 0 e un 1 ad ogni scansione. Ogni scansione utilizza O(n) passi. Poichè ogni scansione elimina due simboli, possono verificarsi al più $\frac{n}{2}$ scansioni. Poi la macchina accetta o rifiuta. Quindi M decide L in tempo $O(n) + \frac{n}{2}O(n) = O(n) + O(n^2) = O(n^2)$.

Classi di Complessità.

Classificheremo i linguaggi in base alla complessità di tempo di un algoritmo che li decide.

Raggrupperemo in una stessa classe linguaggi i cui corrispondenti decider hanno complessità di tempo che sia un O-grande della stessa funzione.

Definizione (Classe di complessità di tempo deterministico)

Sia $f: N \to R^+$ una funzione, sia M l'insieme delle MT deterministiche, a nastro singolo e che si arrestano su ogni input.

La classe di complessità di tempo deterministico TIME(f(n)) è

$$TIME(f(n)) = \{L \mid \exists M \in M \; che \; decide \; L \; in \; tempo \; O(f(n))\}$$

Una classe di complessità di tempo è una famiglia di linguaggi. La proprietà che determina l'appartenenza di un linguaggio L alla classe è la complessità di tempo di un algoritmo per decidere L.

Esempio.

Torniamo al linguaggio $L = \{0^k 1^k \mid k \geq 0\}$. L'analisi precedente mostra che

$$L = \{0^k 1^k \mid k \geq 0\} \in TIME(n^2)$$

Infatti, abbiamo fornito una macchina di Turing M che decide L in tempo $O(n^2)$ e $TIME(n^2)$ contiene tutti i linguaggi che possono essere decisi in tempo $O(n^2)$.

Esiste una macchina che decide L in modo asintoticamente più veloce?

Potremmo migliorare il tempo di esecuzione, cancellando due simboli 0 e due simboli 1 ad ogni scansione invece di uno solamente, perché questo ridurrebbe il numero di scansioni della metà.

Ma questo migliorerebbe il tempo di esecuzione solo per un fattore 2 e non influenzerebbe il tempo di esecuzione asintotico.

La seguente macchina M_2, utilizza un metodo differente per decidere L asintoticamente più velocemente.

Essa mostra che $L \in TIME(n \; log \; n)$.

Il seguente algoritmo M (ovvero la MdT equivalente ad M) decide $L = \{0^k 1^k \mid k \geq 0\}$ in tempo $O(n \; log \; n)$.

M_2 = "Sulla stringa input w:

1. Verifica che $w = 0^t 1^q$ (altrimenti rifiuta w)
2. Ripete le due operazioni seguenti finché almeno un carattere 0 e almeno un carattere 1 resta sul nastro:
    a. Verifica che la lunghezza della stringa sia pari (altrimenti rifiuta).
    b. Cancella ogni secondo zero, a partire dal primo zero, e ogni secondo 1, a partire dal primo 1
3. Se nessun carattere uguale a 0 e a 1 resta sul nastro, accetta. Altrimenti rifiuta."

(Correttezza dell'algoritmo.) L'algoritmo, in primo luogo, verifica se w è una stringa di caratteri uguali a 0 seguiti da una stringa di caratteri uguali a 1, cioè $w = 0^t 1^q$.

In ogni scansione eseguita nella fase 2b, il numero totale di 0 rimanenti è ridotto della metà e ogni eventuale resto viene scartato. Ad esempio, dopo la prima esecuzione del passo 2b, l'algoritmo è applicato alla stringa $0^{\frac{t}{2}} 1^{\frac{q}{2}}$ se t e q sono pari, alla stringa $0^{\frac{t-1}{2}+1} 1^{\frac{q-1}{2}+1}$ se t e q sono dispari.

Quando la fase 2a controlla che il numero totale di 0 e 1 rimanenti sia pari, in realtà sta controllando la coerenza della parità (pari/dispari) del numero di 0 con la parità del numero di 1.

Cioè controlla che siano entrambi pari o entrambi dispari. Se le parità corrispondono sempre, le rappresentazioni binarie dei numeri t di 0 e q di 1 coincidono, e quindi i due numeri sono uguali.

Questo perché la sequenza delle parità fornisce sempre l'inversa della rappresentazione binaria.

Per analizzare il tempo di esecuzione di M_2, per prima cosa osserviamo che le fasi 1 e 3 vengono eseguite una volta, impiegando un tempo totale O(n).

Poi osserviamo che si tratta di un ciclo, l'iterazione è evidenziata al passo 2.

Ogni fase del ciclo richiede tempo O(n) = O(|w|).

Determiniamo poi il numero di volte in cui ognuna viene eseguita.

La fase 2b scarta almeno metà dei simboli 0 e 1 ogni volta che viene eseguita, quindi si verificano al massimo O(log|w|) iterazioni del ciclo prima di averli cancellati tutti.

Il tempo totale delle fasi 2,2a, e 2b è O(n log n). Il tempo di esecuzione di M_2 è O(n) + O(n log n) = O(n log n).

Da ciò otteniamo che:

$$L = \{0^k 1^k | \ k \geq 0\} \in TIME(n^2) \ e \ L \in TIME(n \log n)$$

Differenze tra teoria della computazione e teoria della complessità.

Abbiamo parlato di complessità di tempo di una MdT deterministica, a nastro singolo, che si arresta su ogni input. Anche nella definizione di classe di complessità abbiamo fatto riferimento al modello a nastro singolo.

Potremmo definire più in generale e in maniera analoga la complessità di tempo di una MdT deterministica multinastro. Cambierebbe qualcosa?

La complessità di tempo dipende dal modello di calcolo? <u>SI.</u>

Esempio.

Dato $L = \{0^k 1^k | \ k \geq 0\}$

- Esiste una macchina di Turing (con un nastro) che decide L in tempo O(n log n).
- Si potrebbe dimostrare che **non** esiste una macchina di Turing (con un nastro) che decide L in tempo O(n).

Tuttavia, L può essere deciso da una macchina di Turing M_3 con due nastri in tempo O(n).

M_3 = "Su input w:

a. Scorre il primo nastro verso destra e rifiuta se trova uno 0 a destra di un 1.
b. Scorre il primo nastro verso destra fino al primo 1: per ogni 0, scrive un 1 sul secondo nastro.
c. Scorre primo nastro verso destra leggendo i simboli 1 e scorre il secondo nastro verso sinistra. Per ogni 1 letto sui due nastri li cancella. Se i simboli letti non sono uguali, rifiuta.
d. Se legge ⊔ su entrambi i nastri, accetta."

Questa macchina è semplice da analizzare. Ciascuna delle quattro fasi utilizza O(n) passi.

Quindi il tempo di esecuzione complessivo risulta O(n), quindi lineare. Si noti che questo tempo di esecuzione è il migliore possibile perché sono necessari n passi solo per leggere l'input.

Questa discussione evidenzia una differenza importante tra la teoria della complessità e la teoria della computabilità. Nella teoria della computabilità, la tesi di Church-Turing implica che tutti i modelli computazionali sono equivalenti, ossia che tutti decidono la stessa classe di linguaggi.

Nella teoria della complessità, la scelta del modello influisce sulla complessità di tempo dei linguaggi. Per esempio, linguaggi decidibili in tempo lineare in un modello, non sono necessariamente decidibili in tempo lineare in un altro modello.

La teoria della complessità classifica i linguaggi (o problemi) decidibili L in base alla complessità di tempo di un algoritmo che decide L.

Ma con quale modello misureremo la complessità di tempo? Uno stesso linguaggio può avere differenti complessità di tempo in modelli diversi.

Vedremo che la nostra classificazione non sarà molto sensibile a relativamente piccole differenze di complessità, come quelle che si verificano passando da un modello deterministico a un altro.

Quindi la scelta del modello deterministico non sarà cruciale. Ad eccezione della macchina di Turing non deterministica, le varianti di macchine di Turing deterministiche introdotte sono polinomialmente equivalenti, cioè possono simularsi vicendevolmente con un sovraccarico computazionale polinomiale: se una di tali macchine M ha complessità di tempo f(n), può essere simulata da un'altra M' che ha complessità di tempo O(p (f(n))) dove p è un polinomio.

Relazioni tra i modelli: MdT multinastro.

Teorema

Sia t(n) una funzione tale che t(n) ≥ n. Per ogni macchina di Turing deterministica multinastro M con complessità di tempo t(n) esiste una macchina di Turing deterministica a nastro singolo M' con complessità di tempo O(t^2(n)), equivalente a M.

La complessità di tempo dipende dalla codifica utilizzata? SI.

Esempio 1.

Consideriamo la funzione (calcolabile) f definita da

$$f(\langle n \rangle) = \langle n \rangle \# 1^n$$

dove:

- n ∈ N,
- $\langle n \rangle$ è la rappresentazione in base b ≥ 1 di n ossia la codifica che scegliamo per l'input
- 1^n è la rappresentazione in base 1 (unaria) di n.

Quindi il valore della funzione dipende dalla base che scegliamo per la rappresentazione, cioè dalla codifica dell'input. A seconda della codifica che scegliamo, il valore della funzione cambia.

Per n = 7:

- se scegliamo per l'input la rappresentazione in base 2: $f\ (111) = 111\#1111111$,
- se scegliamo per l'input la rappresentazione in base 10: $f\ (7) = 7\#1111111$,
- se scegliamo per l'input la rappresentazione unaria: $f\ (1111111) = 1111111\#1111111$.

Se scegliamo per l'input la rappresentazione unaria, $\langle n \rangle$ = rappresentazione unaria di n, la macchina M che calcola f è la macchina che copia:

$$\langle n \rangle \rightarrow \langle n \rangle \# \langle n \rangle$$

M ha complessità di tempo $O(|\langle n^2 \rangle|)$ (**polinomiale**).

Se scegliamo per l'input la rappresentazione in base 2, $|\langle n \rangle|$ è O(log n) e la macchina M che calcola f è la macchina che esegue la trasformazione:

$$a_k \dots a_0 \rightarrow a_k \dots a_0 \# 1^n$$

dove $n = a_0 2^0 + \dots + a_k 2^k$ e $|\langle n \rangle|$ = k + 1.

Quindi M deve scrivere (dopo l'ultimo carattere a destra dell'input) una stringa che ha lunghezza tra 2^k + 1 e 2^{k+1} , cioè una stringa di lunghezza t, con $2^{|\langle n \rangle|-1} + 1 \leq t \leq 2^{|\langle n \rangle|}$. Poichè M deve eseguire almeno t passi, la complessità di tempo di M è O($2^{|\langle n \rangle|}$) (e anche Ω($2^{|\langle n \rangle|}$)).

Esempio 2.

Consideriamo il problema di decisione: Dato un numero x, x è primo?

e il linguaggio associato PRIMO = {$\langle x \rangle$ | x ∈ N, x è primo}

Un algoritmo semplice che decide PRIMO:

a. Dividi x per tutti gli interi i, con 1 < i < x.
b. Se tutti i resti delle divisioni sono diversi da zero, x è primo.

Quante divisioni vengono effettuate? x − 2. (escludo la divisione per se stesso e per 1)

Quindi, se n = $|\langle x \rangle|$ la complessità sarà:

O(n), se $\langle x \rangle$ è la rappresentazione unaria,

O(2^n), se $\langle x \rangle$ è la rappresentazione binaria.

Osservazioni sulla complessità di tempo.

Quali codifiche usare?

Occorre considerare codifiche "ragionevoli": non "prolisse" cioè tali che non vi siano istanze la cui rappresentazione sia artificiosamente lunga. In particolare, scartare la rappresentazione unaria degli interi positivi.

Codifiche "ragionevoli" dei dati sono polinomialmente correlate: è possibile passare da una di esse a una qualunque altra codifica "ragionevole" delle istanze dello stesso problema in un tempo polinomiale rispetto alla rappresentazione originale.

Relazioni tra i modelli : MdT non deterministica

Un discorso a parte va fatto per il modello non deterministico di macchina di Turing. La macchina di Turing non deterministica non corrisponde a nessun meccanismo di computazione reale. Ma la definizione di tempo di esecuzione di una macchina di Turing non deterministica è utile per caratterizzare la complessità di un'importante classe di linguaggi.

Ricordiamo che una macchina di Turing non deterministica è un **decisore** se, per ogni stringa input w, tutte le computazioni a partire da $q_0 w$ terminano in una configurazione di arresto.

Definizione (Tempo di esecuzione di una MdT non deterministica).

Sia $N = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ una macchina di Turing non deterministica che sia un decisore (ovvero tutte le computazioni, per ogni input w, terminano in una configurazione di arresto).

Il **tempo di esecuzione** di N è la funzione $f : N \rightarrow N$ dove f(n) è il massimo numero di passi eseguiti da N **in ognuna delle computazioni** su ogni input di lunghezza n, n ∈ N.

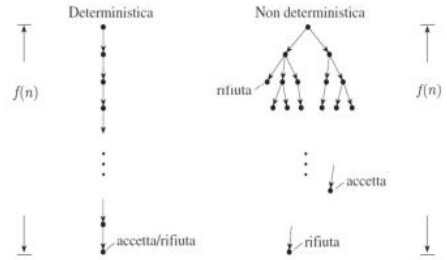


FIGURA 7.10
Misurazione del tempo nei casi deterministico e non deterministico

Figura tratta da M. Sipser, Introduzione alla teoria della computazione.

Il tempo di esecuzione di una macchina non deterministica su input w viene definito come il tempo usato dalla computazione corrispondente alla ramificazione più lunga (nell'albero delle computazioni su w).

Il tempo di esecuzione di una MdT non deterministica N è la funzione f : N → N dove f (n) = massimo delle altezze degli alberi, ognuno dei quali rappresenta le possibili computazioni su input w, al variare di w ∈ Σ^n .

Teorema.

Sia t(n) una funzione tale che t(n) ≥ n.

Per ogni macchina di Turing a nastro singolo, non deterministica N avente tempo di esecuzione t(n) esiste una macchina di Turing a nastro singolo, deterministica e di complessità di tempo $2^{O(t(n))}$equivalente ad N.

Tempo polinomiale.

Differenze polinomiali nel tempo di esecuzione sono considerate piccole, mentre differenze esponenziali sono considerate grandi. Perché?

Esempio.

Dato un elenco di n città, stabilire se esiste un giro turistico che visiti ogni città esattamente una sola volta.

- L'algoritmo basato sulla ricerca esaustiva (algoritmo di forza bruta) richiede tempo esponenziale. Nota: finora è l'unico algoritmo noto per risolvere il problema.
- Se eseguo tale algoritmo su un computer che ha m volte la potenza del migliore calcolatore attuale (m = numero stimato degli elettroni nell'universo), il tempo richiesto per ottenere la soluzione per n = 1000 è maggiore di $10^{79+13+9+12}$.

Algoritmi aventi tempo polinomiale sono abbastanza veloci per molti scopi mentre algoritmi aventi tempo esponenziale sono raramente utili. Algoritmi aventi tempo esponenziale si presentano in genere quando risolviamo problemi mediante una ricerca esaustiva nello spazio delle soluzioni. A volte, la ricerca mediante forza bruta può essere evitata attraverso una comprensione più approfondita del problema, che può suggerire un algoritmo polinomiale di maggiore utilità.

La decisione di non tener conto delle differenze polinomiali non significa che tali differenze non siano considerate importanti. Ma significa esaminare le soluzioni algoritmiche da una diversa prospettiva.

Tutti i modelli computazionali deterministici "ragionevoli" sono polinomialmente equivalenti. Cioè, uno di essi può simularne un altro con aumento solo polinomiale del tempo di esecuzione.

Classe P: tempo polinomiale.

Definizione.

La classe P è l'insieme dei linguaggi L per i quali esiste una macchina di Turing deterministica M con un solo nastro che decide L in tempo $O(n^k)$ per qualche k ≥ 1, cioè

$$P = \bigcup_{k \geq 1} TIME(n^k)$$

Esempio: $L = \{0^k 1^k \mid k \geq 0\} \in P$.

La classe P gioca un ruolo centrale nella teoria della complessità ed è importante perché:

a. P corrisponde, con una certa approssimazione, alla classe di problemi che sono realisticamente risolubili mediante programmi su computer reali. Quindi P è una classe rilevante dal punto di vista pratico
b. P è una classe "robusta dal punto di vista matematico".

Che significa che P è una classe "robusta dal punto di vista matematico"? P è invariante per tutti i modelli di computazione che sono polinomialmente equivalenti alla macchina di Turing deterministica a nastro singolo. La classe P è invariante rispetto alla scelta di una codifica "ragionevole" dell'input.

Nota: per mostrare che l'algoritmo può essere eseguito in tempo $O(n^k)$, su un input di lunghezza n, da M dobbiamo:

a. Fornire un limite superiore al numero dei passi eseguiti dall'algoritmo che sia polinomiale in n,
b. Mostrare che ogni passo può essere eseguito in tempo polinomiale in n da M (o da un qualsiasi "ragionevole" modello di computazione deterministico).

Allora esiste h ∈ N tale che ogni passo può essere eseguito in tempo $O(n^h)$. Se il numero dei passi eseguiti dall'algoritmo è $O(n^c)$, l'algoritmo potrà essere eseguito in tempo $O(n^{h+c})$, su un input di lunghezza n.

Precisazione.

Nel testo figura spesso la frase: "la composizione di (un numero finito) di polinomi è un polinomio."

In generale questa frase si riferisce alla seguente situazione: N è un algoritmo che, su un input di lunghezza n, simula una MdT M_1 di complessità in tempo polinomiale $O(n^k)$ e poi sull'output, simula una MdT M_2 di complessità in tempo polinomiale $O(m^t)$.

N è un algoritmo di complessità in tempo polinomiale.

$$N: w \rightarrow \boxed{M_1} \rightarrow \boxed{M_2}$$

Se M_1 ha complessità $O(n^k)$ ed M_2 ha complessità $O(m^t)$ allora N ha complessità $O(n^{kt})$.

Infatti, il numero di passi di N su un input di lunghezza n è uguale al numero di passi di M_1 su tale input più il numero di passi di M_2 sull'output di M_1. Se l'input di M_1 (e di N) ha lunghezza n, il corrispondente output ha lunghezza $O(n^k)$ (Perché? Perché la macchina non potrebbe scrivere più caratteri di quanti sono i passi che fa per spostare la testina e quindi la lunghezza dell'output sarà al più uguale al numero di passi effettuato).

Allora:

- Il numero di passi di M_1 su input di lunghezza n è $O(n^k)$.
- Il numero di passi di M_2 sull'output di M_1, di lunghezza $O(n^k)$, è $O((n^k)^t) = O(n^{kt})$.

- Quindi il numero di passi di N su un input di lunghezza n è $O(n^k) + O(n^{kt})$ **(polinomiale).**

Esempi di problemi in P.

$$PATH = \{\langle G, s, t\rangle \mid G \text{ è } un\ grafo\ orientato\ in\ cui\ c'\text{è } un\ cammino\ da\ s\ a\ t\}$$

Teorema

PATH ∈ P.

Una generazione esaustiva dei cammini di G (algoritmo di "forza bruta") condurrebbe a un algoritmo di complessità esponenziale. Se V è l'insieme dei nodi in G, bisogna generare tutti i sottoinsiemi di V di cardinalità maggiore o uguale di 2. Tale numero è $O(2^{|\langle G,s,t\rangle|})$.

Il seguente algoritmo M (ovvero la MdT equivalente ad M) decide PATH in tempo deterministico polinomiale.

M = "Sull'input $\langle G, s, t\rangle$, dove G è un grafo con nodi s e t:

1. Marca il nodo s.
2. Ripete questa operazione finché nessun nuovo vertice viene marcato:
    a. Scansiona tutti gli archi di G. Se trova un arco (a, b) che va da un nodo a marcato ad un nodo b non marcato, marca il nodo b.
3. Se t è marcato, accetta. Altrimenti rifiuta."

M decide PATH in tempo deterministico polinomiale.

Ovviamente, le fasi 1 e 3 sono eseguite una sola volta. Sia m il numero dei vertici di G. Il passo 2a viene eseguito al più m volte perché viene eseguito ogni volta che viene marcato un nuovo nodo in G e ne posso marcare al più m. Quindi il numero totale dei passi è al più 1 + 1 + m, che è polinomiale nella lunghezza dell'input. Infine, i passi 1, 2a e 3 possono essere implementati in tempo polinomiale nella lunghezza dell'input su una MdT deterministica.

Due numeri interi positivi x, y sono **relativamente primi** (o coprimi) se il loro massimo comun divisore è 1 (cioè 1 è il più grande intero che li divide entrambi).

$$RELPRIME = \{\langle x, y\rangle \mid x\ e\ y\ sono\ relativamente\ primi\}$$

Una ricerca esaustiva dei divisori non banali di x e y (metodo "forza bruta") condurrebbe a un algoritmo di complessità esponenziale.

Ricerca esaustiva: dividere x per tutti i naturali i, con 1 < i < x, e y per tutti i naturali j, con 1 < j < y. Ci sono x−2 naturali i, con 1 < i < x.

Se $\langle x\rangle = a_k \ldots a_0$ è la rappresentazione binaria di x allora x−2 è $O(2^k) = O(2^{|\langle x\rangle|})$.

L'algoritmo che permette di provare che RELPRIME ∈ P è basato sull'algoritmo di Euclide (circa 300 a.C.) per calcolare il massimo comune divisore MCD(x, y) di due numeri interi non negativi x, y.

Teorema (teorema di ricorsione del MCD)

Per qualsiasi numero intero a non negativo e qualunque intero b positivo, MCD(a, b) = MCD(b, a (mod b)).

Algoritmo di Euclide

MCD(a, b)

if b = 0 then MCD = a

else MCD = MCD(b, a (mod b))

Nota: si può provare che la procedura è corretta per induzione: se b = 0 la procedura restituisce un valore corretto, se $b \neq 0$ usiamo il teorema di ricorsione del MCD e l'ipotesi induttiva.

- Sono necessarie O(log b) chiamate ricorsive.
- Complessità di tempo di MCD logaritmica rispetto al valore dei due numeri. Quindi, *lineare* rispetto alla loro codifica (polinomiale, considerando anche il costo - logaritmico rispetto al valore dei due numeri - di ogni chiamata).

Teorema
RELPRIME ∈ P.

Dimostrazione
Consideriamo l'algoritmo R:

R = "Sull'input $\langle x, y \rangle$, dove x e y sono numeri naturali in binario:

1. Simula MCD su $\langle x, y \rangle$
2. Se il risultato è 1 accetta. Altrimenti rifiuta."

R (ovvero la MdT equivalente a R) decide RELPRIME in tempo deterministico polinomiale.

Classe EXPTIME
Nella teoria della complessità è centrale la classificazione dei linguaggi in linguaggi in P e in

$$EXPTIME = \bigcup_{k \geq 1} TIME(2^{n^k})$$

a cui corrisponde la classificazione dei problemi (di decisione algoritmicamente risolubili) in problemi **trattabili** e problemi **intrattabili,** ovvero problemi per i quali esistono algoritmi polinomiali per la loro soluzione e problemi per i quali esistono algoritmi esponenziali per la loro soluzione.

Ovviamente P ⊆ EXPTIME.

Inoltre, esistono linguaggi in EXPTIME \ P, a cui corrispondono problemi che richiedono, per essere risolti, tempo sicuramente esponenziale nella dimensione dei loro dati. Quindi $P \subsetneq EXPTIME$.

Un esempio di linguaggio in EXPTIME \ P si ottiene considerando una definizione più generale delle espressioni regolari.

Abbiamo dato delle espressioni regolari una definizione ricorsiva. La regola induttiva permette di costruire una nuova espressione regolare a partire dalle espressioni regolari R1 ed R2, usando le operazioni ∪, ∘ e $*$. Le espressioni regolari generalizzate (o ERG) aggiungono l'operazione $\uparrow$:

se R è un'espressione regolare e k ∈ N, $R \uparrow k$ è la concatenazione di R con sé stessa k volte.

Sia

$$EQ_{REX\uparrow} = \{\langle Q, R \rangle \mid Q \; ed \; R \; sono \; ERG \; equivalenti\}$$

risulta $EQ_{REX\uparrow}$ ∈ EXPTIME \ P.

La classe NP
I problemi corrispondenti ai linguaggi in EXPTIME \ P non sono in genere importanti nelle applicazioni pratiche mentre si incontrano comunemente problemi (ovvero linguaggi) decidibili ma tali che gli algoritmi per deciderli, attualmente noti, richiedono tempo esponenziale. Cioè per moltissimi problemi non è nota

una loro risoluzione efficiente. Come mai? Non si sa. Forse questi problemi ammettono algoritmi in tempo polinomiale che si basano su principi non ancora noti. O forse alcuni di questi problemi semplicemente non possono essere risolti in tempo polinomiale.

Esaminando questa questione si è però scoperto che le complessità di molti problemi sono legate tra di loro. Cioè molti di questi linguaggi hanno una caratteristica in comune che giustificherà l'introduzione di una nuova classe di linguaggi. Non è noto se tale classe sia o meno più estesa della classe P. Inoltre, un algoritmo polinomiale per alcuni di essi può essere utilizzato per risolvere un'intera classe di problemi. Per capire tale caratteristica, consideriamo i seguenti due esempi di problemi (computazionali).

HAMPATH

Un cammino Hamiltoniano in un grafo orientato è un cammino (orientato) che passa per ogni vertice del grafo una e una sola volta. Consideriamo il problema di stabilire se un grafo orientato contiene un cammino Hamiltoniano che collega due nodi specificati. Questo si può formulare come un problema di decisione, a cui corrisponde un linguaggio associato, il linguaggio HAMPATH.

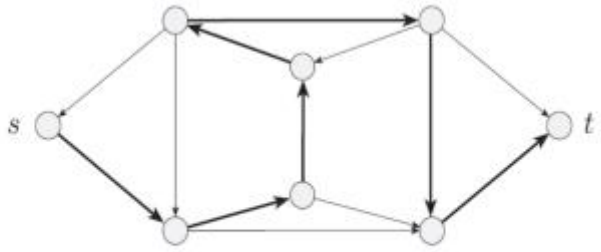


FIGURA 7.17
Un cammino Hamiltoniano attraversa ogni nodo esattamente una volta

Il linguaggio associato al problema di decisione del cammino Hamiltoniano:

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ è } un\ grafo\ orientato\ e\ ha\ un\ cammino\ Hamiltoniano\ da\ s\ a\ t\}$$

può essere deciso da un algoritmo di complessità esponenziale, utilizzando un metodo di forza bruta, che consiste nel costruire tutti i cammini in G (di lunghezza uguale al numero dei vertici in G - 1) e verificare se uno di essi è un cammino Hamiltoniano da s a t.

Il linguaggio HAMPATH ha però una caratteristica chiamata **verificabilità polinomiale** che è importante per capire la sua complessità. Anche se non conosciamo un algoritmo polinomiale per determinare se un grafo contiene un cammino Hamiltoniano, se un tale cammino è stato scoperto in qualche modo, la verifica che si tratta di un cammino Hamiltoniano può essere fatta in tempo polinomiale. In altre parole, verificare l'esistenza di un cammino Hamiltoniano può essere molto più facile che determinare la sua esistenza.

Non si conoscono algoritmi polinomiali che decidono HAMPATH.

Ma se un grafo G = (V, E) ammette un cammino Hamiltoniano da s a t esiste una sequenza c di |V| nodi distinti $(u_1, \ldots, u_{|V|})$ tale che:

- $(u_{i-1}, u_i) \in E$, per ogni i con 2 ≤ i ≤ |V|,
- $u_1 = s$,
- $u_{|V|} = t$.

Esiste un algoritmo N, polinomiale in $|\langle G, s, t\rangle|$, che sull'input $\langle\langle G, s, t\rangle, c\rangle$, dove $c = (u_1, \ldots, u_{|V|})$, decide il linguaggio:

$$\{\langle\langle G, s, t, \rangle, c\rangle \mid G \text{ è } un\ grafo\ orientato\ e\ c' \text{è } un\ cammino\ Hamiltoniano\ da\ s\ a\ t\}$$

Basta verificare che i nodi della sequenza siano distinti, che $(u_{i-1}, u_i) \in E$, per ogni i con 2 ≤ i ≤ |V|, $u_1 = s$, $u_{|V|} = t$.

Nota: $|\langle c \rangle|$ è polinomiale in $|\langle G, s, t \rangle|$.

Quindi esiste un algoritmo polinomiale che verifica se una sequenza $c = (u_1, \dots, u_{|V|})$ di |V| nodi è un cammino Hamiltoniano da s a t.

COMPOSITES

Un numero è **composto** se è il prodotto di due interi maggiori di 1, cioè un numero composto è un intero che non è primo. Consideriamo il problema di stabilire se un numero non è primo. Questo si può formulare come un problema di decisione, a cui corrisponde un linguaggio associato, il linguaggio COMPOSITES.

$$COMPOSITES = \{\langle x \rangle \mid esistono\ interi\ p, q, con\ p > 1, q > 1\ tali\ che\ x = pq\}$$

Recentemente è stato dimostrato che COMPOSITES ∈ P.

Comunque, dati x, p, **verificare** che p è un divisore di x, con p > 1, p≠ x, può essere fatto in tempo polinomiale.

Per HAMPATH, e per molti altri linguaggi, **non è noto** se esiste un algoritmo polinomiale che decida l'appartenenza di una stringa w al linguaggio.

Ma un'informazione aggiuntiva c, detta **certificato** permette di **verificare** in tempo polinomiale se w appartiene al linguaggio.

Algoritmi di Verifica

La definizione della classe NP richiede l'introduzione di un nuovo concetto.

Abbiamo introdotto il concetto di linguaggio L decidibile:

- L è decidibile se esiste un algoritmo M che decide L (cioè tale che, sull'input w, esiste una computazione da $q_0 w\ a\ u q_{accept} v$ se e solo se w ∈ L).

Abbiamo introdotto la classe P e il concetto di linguaggio L decidibile in tempo polinomiale:

- L ∈ P, cioè L è decidibile in tempo polinomiale, se e solo se esiste un algoritmo M che decide L ed M ha complessità di tempo polinomiale.

Vogliamo formalizzare il concetto seguente:

- Un algoritmo V è un algoritmo di verifica (polinomiale) per un linguaggio A se V verifica le seguenti due proprietà:
  - V è un algoritmo (polinomiale in |w|) a "due argomenti" $\langle w, c \rangle$
  - per ogni stringa w, w ∈ A se e solo se esiste c tale che V accetta $\langle w, c \rangle$ (e |c| = O($|w|^t$))

Definizione.

Un **algoritmo di verifica (o verificatore)** V per un linguaggio A è un algoritmo tale che

$$A = \{w \mid \exists c\ tale\ che\ V\ accetta\ \langle w, c \rangle\}$$

La stringa c prende il nome di **certificato** o **prova**. A è il **linguaggio verificato** da V.

Complessità degli algoritmi di verifica.

La complessità di tempo di un algoritmo di verifica V sull'input $\langle w, c \rangle$ è misurata solo in termini della lunghezza |w| di w.

Definizione
Un algoritmo V è un verificatore per A in tempo **polinomiale** se:

- A è il linguaggio verificato da V, cioè $A = \{w \mid \exists c \; tale\ che\ V\ accetta\ \langle w, c\rangle\}$
- V ha complessità di tempo polinomiale in |w|.

In alcune definizioni del concetto di verifica in tempo polinomiale è richiesto che il certificato c abbia **lunghezza polinomiale** nella lunghezza di w, in altre no.

Nota: se V è un algoritmo di verifica e ha complessità polinomiale in |w|, allora il certificato ha lunghezza polinomiale nella lunghezza di w, cioè esiste t tale che per ogni w, |c| = O($|w|^t$).

Questo è imposto dal limite di tempo polinomiale per la computazione di V. Se V ha complessità di tempo O($|w|^k$), la computazione di V si arresta dopo O($|w|^k$) passi e se c non avesse lunghezza polinomiale in |w|, non sarebbe esaminabile da V.

Definizione
NP è **la classe dei linguaggi verificabili in tempo polinomiale.**

Esempi.

Per HAMPATH un certificato per una stringa $\langle G, s, t\rangle \in HAMPATH$ è un cammino Hamiltoniano da s a t.

Per COMPOSITES un certificato per una stringa $\langle x\rangle \in COMPOSITES$ è uno dei divisori di x.

NOTA: NP **NON** è abbreviazione di "tempo non polinomiale". NP è abbreviazione di **"tempo polinomiale non deterministico".** Deriva da una caratterizzazione equivalente di NP che usa le macchine di Turing non deterministiche di tempo polinomiale.

Teorema
Un linguaggio L è in NP se e solo se esiste una macchina di Turing non deterministica che decide L in tempo polinomiale.

Definizione(Classe di complessità di tempo non deterministico)
Sia $t: N \to R^+$ una funzione. La classe di complessità in tempo non deterministico NTIME(t(n)) è

$$NTIME(t(n)) = \{L \mid \exists\ una\ macchina\ di\ Turing\ non\ deterministica\ M\ che\ decide\ L\ in\ O(t(n))\}$$

Corollario

$$NP = \bigcup_{k \geq 1} NTIME(n^k)$$

Esempi di linguaggi in NP : CLIQUE

Definizione
Una **clique** (o cricca) in un grafo non orientato G è un sottografo di G in cui ogni coppia di vertici è connessa da un arco. Una **k-clique** è una clique che contiene k vertici.

Il problema di stabilire se un grafo non orientato G contiene una k-clique si può formulare come un problema di decisione, il cui linguaggio associato è CLIQUE.

$$CLIQUE = \{\langle G, k\rangle \mid G\ è\ un\ grafo\ non\ orientato\ in\ cui\ esiste\ una\ k-clique\}$$

Teorema

CLIQUE ∈ NP

Dimostrazione.

Un algoritmo V che verifica CLIQUE in tempo polinomiale:

V = "Sull'input $\langle\langle G, k\rangle, c\rangle$:

1. Verifica se c è un insieme di k nodi di G, altrimenti rifiuta.
2. Verifica se per ogni coppia di nodi in c, esiste un arco in G che li connette, accetta in caso affermativo; altrimenti rifiuta."

$$\exists c: \langle\langle G, k\rangle, c\rangle \in L(V) \Leftrightarrow \langle G, k\rangle \in CLIQUE$$

Prova alternativa: utilizzare le macchine di Turing non deterministiche.

Esempi di linguaggi in NP: SUBSET-SUM

SUBSET-SUM: Dato un insieme S di numeri interi e un numero intero t, esiste un sottoinsieme S' di S tale che la somma dei suoi numeri sia uguale a t?

$$SUBSET - SUM = \left\{\langle S, t\rangle \mid \exists S' \subset S \text{ tale che } \sum_{s \in S'} s = t\right\}$$

Esempio: $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ SUBSET-SUM perchè 4 + 21 = 25.

Nota: è possibile definire SUBSET-SUM in cui S, S' sono multinsiemi (cioè insiemi in cui alcuni elementi si ripetono).

Teorema

SUBSET-SUM ∈ NP

Dimostrazione

Un algoritmo V che verifica SUBSET-SUM in tempo polinomiale:

V = "Sull'input $\langle\langle S, t\rangle, c\rangle$:

1. Verifica se c è un insieme di numeri la cui somma è t, altrimenti rifiuta.
2. Verifica se S contiene tutti i numeri in c, accetta in caso affermativo; altrimenti rifiuta."

$$\exists c: \langle\langle S, t\rangle, c\rangle \in L(V) \Leftrightarrow \langle S, t\rangle \in SUBSET - SUM$$

Prova alternativa: utilizzare le macchine di Turing non deterministiche.

Esempi di linguaggi in NP: HAMPATH

Teorema

HAMPATH ∈ NP

Dimostrazione.

Un algoritmo N che verifica HAMPATH in tempo polinomiale:

N = "Sull'input $\langle\langle G, s, t\rangle, c\rangle$ dove G = (V, E) è un grafo orientato:

1. Verifica se $c = (u_1, \ldots, u_{|V|})$ è una sequenza di |V| vertici di G, altrimenti rifiuta.
2. Verifica se i nodi della sequenza sono distinti, che $(u_{i-1}, u_i) \in E$, per ogni i con 2 ≤ i ≤ |V|, $u_1 = s$, $u_{|V|} = t$, accetta in caso affermativo; altrimenti rifiuta."

$$\exists c: \langle\langle G, s, t\rangle, c\rangle \in L(N) \text{ se e solo se } \langle G, s, t\rangle \in HAMPATH.$$

Classe P e NP

- P = la classe dei linguaggi L per i quali l'appartenenza di una stringa w ad L può essere **decisa** da un algoritmo polinomiale in |w| (efficiente).
- NP = la classe dei linguaggi L per i quali l'appartenenza di una stringa w ad L può essere **verificata** da un algoritmo polinomiale in |w| (efficiente).

Teorema
$P \subseteq NP$.

Dimostrazione 1

Se L ∈ P, esiste un algoritmo M che decide L in tempo polinomiale.

Consideriamo l'algoritmo di verifica V che sull'input y:

- Se $y \neq \langle w, \epsilon \rangle$, w stringa, rifiuta y
- Se $y = \langle w, \epsilon \rangle$, w stringa, simula M su w
- Accetta $y = \langle w, \epsilon \rangle$ se e solo se M accetta w.

V verifica L in tempo polinomiale.

Dimostrazione 2
Se L ∈ P, esiste un macchina di Turing deterministica M che decide L in tempo polinomiale.

Esiste un macchina di Turing non deterministica M' equivalente ad M.

M' decide L in tempo polinomiale.

Una gerarchia di classi
Sappiamo che per ogni macchina di Turing a nastro singolo non deterministica N, che ha tempo di esecuzione t(n) ≥ n, esiste una macchina di Turing equivalente a nastro singolo deterministica e con complessità di tempo $2^{O(t(n))}$.

Quindi:

$$P \subseteq NP = \bigcup_{k \geq 1} NTIME(n^k) \subseteq EXPTIME = \bigcup_{k \geq 1} TIME(2^{n^k})$$

È noto che $P \subsetneq EXPTIME$.

Uno dei più grandi problemi aperti dell'informatica teorica: **P = NP**?

P, NP, coNP
Proposizione La classe P è chiusa rispetto al complemento.

La definizione della classe NP è meno semplice e intuitiva della definizione della classe P.

Ad esempio, se $\langle G, s, t \rangle \in HAMPATH$, si può costruire un certificato e verificare in tempo polinomiale che $\langle G, s, t \rangle \in HAMPATH$. Ma se $\langle G, s, t \rangle \notin HAMPATH$, non è richiesto (e non è sicuro) che questo sia certificabile.

Nota che se $\langle G, s, t \rangle \notin HAMPATH$ allora $\langle G, s, t \rangle \in \overline{HAMPATH}$.

Queste osservazioni si applicano a qualsiasi linguaggio in NP e pongono il problema del rapporto tra la classe **NP** e la classe **coNP = {L | $\bar{L} \in$ NP}.**

Esempi di linguaggi in $coNP$: $\overline{HAMPATH}, \overline{CLIQUE}, \overline{SUBSET - SUM}$.

Per ognuno di questi linguaggi non è noto se appartenga o meno a NP. Verificare che "qualcosa non c'è" sembra essere più difficile che verificare che "c'è".

NP = coNP?

Quattro possibili scenari:

1. $P = NP = coNP$
2. $P \subsetneq NP = coNP$
3. $se\ NP \neq coNP, allora\ \ P = NP \cap coNP \subsetneq NP\ (e\ quindi\ P = NP \cap coNP \subsetneq coNP)$
4. $se\ NP \neq coNP, allora\ P \subsetneq NP \cap coNP \subsetneq NP\ (e\ quindi\ P \subsetneq NP \cap coNP \subsetneq coNP)$

Questi 4 scenari sono mutuamente esclusivi ossia non possono mai essere veri contemporaneamente.

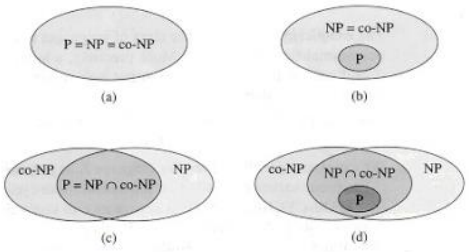


Figura tratta da T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms.

Il concetto di NP-Completezza

Un progresso importante sulla questione **"P = NP?"** ci fu all'inizio degli anni '70 con il lavoro di Stephen Cook e Leonid Levin.

Essi scoprirono una classe di linguaggi appartenenti a NP tale che la complessità individuale di ognuno di essi è correlata a quella dell'intera classe. Se esistesse un algoritmo di tempo polinomiale per uno qualsiasi di essi, tutti i linguaggi in NP diventerebbero decidibili in tempo polinomiale. Questi linguaggi vengono detti **NP-completi**. Il fenomeno della NP-completezza è importante sia per ragioni teoriche che pratiche. Il primo linguaggio NP-completo che fu scoperto è legato al **problema della soddisfacibilità** di un'espressione booleana.

Riduzione Polinomiale

Abbiamo definito il concetto di riduzione mediante funzione di un linguaggio a un altro. Quando un linguaggio A si riduce mediante funzione a un linguaggio B, un algoritmo per riconoscere B (se esiste) può essere usato per riconoscere A.

Ora definiamo una versione della riducibilità che tiene conto dell'efficienza della computazione. Quando un linguaggio A è riducibile efficientemente a un linguaggio B, un algoritmo efficiente per B può essere usato per decidere A efficientemente. Ricorda: in teoria della complessità consideriamo solo linguaggi decidibili.

Definizione

Una funzione $f: \Sigma^* \to \Sigma^*$ è **calcolabile in tempo polinomiale** se esiste una macchina di Turing deterministica M di complessità di tempo **polinomiale** tale che su ogni input w, M si arresta con f (w), e solo con f (w), sul suo nastro.

Da notare la differenza tra funzione **calcolabile** e funzione **calcolabile in tempo polinomiale**.

Definizione

Siano A, B linguaggi sull'alfabeto Σ.

Una ***riduzione di tempo polinomiale*** f di A a B:

- è una funzione $f: \Sigma^* \to \Sigma^*$
- **calcolabile** in **tempo polinomiale**
- tale che $\forall w \in \Sigma^* \; w \in A \Leftrightarrow f(w) \in B$

Definizione

Un linguaggio $A \subseteq \Sigma^*$ è **riducibile in tempo polinomiale** (o **riducibile mediante funzione in tempo polinomiale**) a un linguaggio $B \subseteq \Sigma^*$ $(A \leq_P B)$ se esiste una **riduzione di tempo polinomiale** di A a B.

La riducibilità in tempo polinomiale è l'analogo efficiente della riducibilità mediante funzione. Come con un'ordinaria riduzione mediante funzione, una riduzione in tempo polinomiale di A a B fornisce un modo per convertire questioni riguardanti l'appartenenza o meno ad A in questioni riguardanti l'appartenenza o meno a B ma ora la conversione viene realizzata efficientemente.

Note:

- Nella definizione di riduzione di tempo polinomiale non è richiesto che f sia una funzione iniettiva o suriettiva.
- Se A è un linguaggio su Σ e A è associato a un problema di decisione P, le stringhe w in Σ^* si dividono in tre gruppi:
    - w ∈ A cioè w è la codifica di un istanza di P per la quale P ammette risposta "sì"
    - w ∉ A e w è la codifica di un istanza di P per la quale P ammette risposta "no"
    - w ∉ A e w non è la codifica di un istanza di P
- In generale nelle prove di riduzione di tempo polinomiale di A a un altro linguaggio B, vengono considerate solo le stringhe dei primi due gruppi e si assume implicitamente che la riduzione f associa alle stringhe w del terzo gruppo (stringhe che non sono codifiche di istanze di P) una stringa f (w) che non è in B.

Se un linguaggio A è riducibile in tempo polinomiale a un linguaggio B e già si sa che B ha un decisore di tempo polinomiale, si ottiene un decisore di tempo polinomiale per il linguaggio originale A. Questo è dimostrato nel teorema seguente.

Teorema

Se $A \leq_P B$ e B ∈ P, allora A ∈ P.

Dimostrazione

Per ipotesi B ∈ P, quindi esiste un algoritmo M, di complessità $O(m^t)$, che decide B.

Inoltre, $A \leq_P B$: sia f la riduzione di tempo polinomiale di A a B e sia F l'algoritmo, di complessità $O(n^k)$, che calcola la funzione f .

Consideriamo l'algoritmo N che per un dato input w:

1. simula F su w e calcola f (w),
2. simula M sull'input f (w) per decidere se f (w) ∈ B.
3. N accetta w se M accetta f (w) , N rifiuta w se M rifiuta f (w).

N decide A (correttezza dell'algoritmo N):

$$N\ accetta\ w \Leftrightarrow M\ accetta\ f(w) \Leftrightarrow f(w) \in B \Leftrightarrow w \in A$$

essendo f una riduzione mediante funzione di A a B.

N è un algoritmo polinomiale in n = |w|. Infatti, F calcola f(w) in O(n^k) passi (primo passo dell'algoritmo: polinomiale). Inoltre, risulta |f(w)| = O(n^k) (cioè, per n sufficientemente grande, |f(w)| ≤ cn^k).

Al secondo passo M viene eseguito sull'input f (w) e si arresterà dopo c'$|f(w)|^t \leq c'\left(cn^k\right)^t$ passi, cioè dopo $O(n^{kt})$ passi (c', c, k, t costanti ,secondo passo dell'algoritmo: polinomiale, composizione dei due polinomi). In conclusione, N ha complessità $O(n^k) + O(n^{kt}) = O(n^{kt})$.

Quindi A ∈ P.

Teorema (Proprietà transitiva di $\leq_P$)

$Se\ A \leq_P B\ e\ B \leq_P C, allora\ A\ \leq_P C.$

Dimostrazione

Per ipotesi: esiste una riduzione di tempo polinomiale $f : \Sigma^* \to \Sigma^*$ di A a B ed esiste una riduzione di tempo polinomiale $g: \Sigma^* \to \Sigma^*$ di B a C. Consideriamo la composizione $g \circ f: \Sigma^* \to \Sigma^*$ delle funzioni f e g, definita da (g ◦ f)(w) = g(f (w)).

Risulta, per ogni $w \in \Sigma^*$: $w \in A \Leftrightarrow f(w) \in B \Leftrightarrow g(f(w)) \in C$

Inoltre, la funzione g ◦ f è una funzione calcolabile in tempo polinomiale.

Infatti, sia F l'algoritmo di complessità $O\left(n^k\right)$ che calcola la funzione f , sia G l'algoritmo di complessità $O(m^t)$ che calcola la funzione g.

Consideriamo l'algoritmo GF che sull'input w:

1. simula F su w e calcola f (w),
2. simula G sull'input f (w) e calcola g(f (w))
3. fornisce in output l'output di G.

L'algoritmo GF calcola g ◦ f perché prima esegue F su w calcolando f (w) (primo passo dell'algoritmo) e poi G su f (w) (secondo passo dell'algoritmo) fornendo quindi in output g(f (w)).

GF è un algoritmo polinomiale in n=|w|. Infatti, F calcola f (w) in $O(n^k)$ passi (primo passo dell'algoritmo: polinomiale). Inoltre, risulta |f (w)| = $O(n^k)$ (cioè, per n sufficientemente grande, |f (w)| ≤ cn^k). Al secondo passo G viene eseguito sull'input f (w) e si arresterà dopo c'$|f(w)|^t \leq c'\left(cn^k\right)^t$ passi, cioè dopo $O(n^{kt})$ passi (c', c, k, t costanti. Secondo passo dell'algoritmo: polinomiale, composizione dei due polinomi).

In conclusione, GF ha complessità $O(n^k) + O(n^{kt}) = O(n^{kt})$.

Quindi g ◦ f è una riduzione di tempo polinomiale di A a C.

Alcuni richiami di logica.

- Variabili Booleane: variabili che possono assumere valore 1 (o TRUE) o 0 (o FALSE)
- Operazioni Booleane: ∨ (o OR), ∧ (o AND), ¬ (o NOT)

- Denotiamo ¬x con x
- 0 ∨ 0 = 0, 0 ∨ 1 = 1 ∨ 0 = 1 ∨ 1 = 1
- 0 ∧ 0 = 0 ∧ 1 = 1 ∧ 0 = 0, 1 ∧ 1 = 1
- 0 = 1, 1 = 0

Definizione

Dato un insieme di variabili booleane X, le **formule booleane** (o **espressioni booleane**) su X sono definite induttivamente come segue:

- le costanti 0, 1 e le variabili (in forma diretta o complementata) x, $\bar{x}$, con x ∈ X, sono formule booleane.
- Se ϕ, ϕ_1, ϕ_2 sono formule booleane allora $(\phi_1 \vee \phi_2)$, $(\phi_1 \wedge \phi_2)$, $\bar{\phi}$ sono formule booleane.

Si definisce **letterale** ogni presenza in forma diretta o negata di una variabile in una espressione e **numero di letterali** il loro numero.

Esempio: $\varphi = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$ è un'espressione booleana con 4 letterali e 3 variabili x, y, z. Esiste una relazione tra formule booleane e circuiti combinatori booleani (o reti combinatorie).

Una formula booleana ϕ è **soddisfacibile** se esiste un insieme di valori 0 o 1 per le variabili di ϕ (o **assegnamento**) che renda la formula uguale a 1 (assegnamento di soddisfacibilità). Diremo che tale assegnamento soddisfa ϕ o anche che rende vera ϕ.

Esempi:

$\varphi_1 = (\bar{x} \vee y) \wedge (x \vee \bar{z})$ è soddisfacibile (assegnamento: x = 0, y = 1, z = 0),

$\varphi_2 = (\bar{x} \wedge y) \vee (x \wedge \bar{z})$ è soddisfacibile,

$\varphi_3 = (\bar{x} \vee x) \wedge (y \vee \bar{y})$ è soddisfacibile (per qualunque assegnamento di valori delle variabili),

$\varphi_4 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2})$ non è soddisfacibile.

Una clausola è un OR di letterali.

Esempio: $(\bar{x} \vee x \vee y \vee z)$

Una formula booleana ϕ è in **forma normale congiuntiva** (o forma normale POS) se è un AND di clausole, cioè è un AND di OR di letterali.

Esempio: $(\overline{x_1} \vee x_2 \vee x_3 \vee x_4) \wedge (x_3 \vee \overline{x_6}) \wedge (x_3 \vee \overline{x_5} \vee x_5)$

Esiste una nozione duale di forma normale (forma normale disgiuntiva o SOP).

Data un'espressione booleana ϕ se ne può costruire una equivalente ϕ' in forma normale congiuntiva (o disgiuntiva). Se ϕ ha n simboli, la sua forma normale può avere un numero di simboli esponenziale in n. Ricordiamo: due espressioni booleane ϕ, ϕ' sullo stesso insieme di variabili sono **equivalenti** se per ogni assegnamento alle variabili, ϕ e ϕ' assumono lo stesso valore.

Problema SAT

Il problema della soddisfacibilità di una formula booleana: Data una formula booleana ϕ, ϕ è soddisfacibile?

Il linguaggio associato è:

$$SAT = \{\langle\varphi\rangle \mid \varphi \text{ è } una\ formula\ booleana\ soddisfacibile\}$$

Teorema

SAT ∈ NP.

Dimostrazione

Un certificato per $\langle\phi\rangle$ sarà un assegnamento c di valori alle variabili di ϕ. Un algoritmo V che verifica SAT in tempo polinomiale nella lunghezza di $\langle\phi\rangle$:

V = "Sull'input y:

1. Verifica se $y = \langle\langle\phi\rangle, c\rangle$ dove ϕ è una formula booleana e c'è un assegnamento di valori alle variabili di ϕ, altrimenti rifiuta.
2. Sostituisce ogni variabile della formula con il suo corrispondente valore e quindi valuta l'espressione.
3. Accetta se ϕ assume valore 1; altrimenti rifiuta."

$$\exists c : \langle\langle\varphi\rangle, c\rangle \in L(V) \Leftrightarrow \langle\varphi\rangle \in SAT$$

Problema 3SAT

Definizione

Una formula booleana è in forma normale 3-congiuntiva se è un AND di clausole e tutte le clausole hanno tre letterali.

Esempio: $(\overline{x_1} \vee x_2 \vee x_3) \wedge (x_3 \vee \overline{x_6} \vee x_6) \wedge (x_3 \vee \overline{x_5} \vee x_5)$

Abbreviazioni:

CNF = formula booleana in forma normale congiuntiva

3CNF = formula booleana in forma normale 3-congiuntiva

kCNF = formula booleana in forma normale k-congiuntiva = AND di clausole e tutte le clausole hanno k letterali

$$3SAT = \{\langle\varphi\rangle \mid \varphi \text{ è } una\ formula\ 3CNF\ soddisfacibile\}$$

Teorema

3SAT è riducibile in tempo polinomiale a CLIQUE.

Dimostrazione

Sia ϕ una formula 3CNF con k clausole:

$$(a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

Consideriamo la funzione f che associa a $\langle\phi\rangle$ la stringa $\langle G, k\rangle$ dove G = (V, E) è il grafo non orientato definito come segue:

- V ha 3 × k vertici. I vertici di G sono divisi in k gruppi di tre nodi (o triple) $t_1, \ldots, t_k$:

t_j corrisponde alla clausola $(a_j \vee b_j \vee c_j)$ e ogni vertice in t_j corrisponde a un letterale $(a_j \vee b_j \vee c_j)$. Quindi $V = \{a_1, b_1, c_1, \dots, a_k, b_k, c_k\}$.

- Non ci sono archi tra i vertici in una tripla t_j , non ci sono archi tra un vertice associato a un letterale x e i vertici associati al letterale $\bar{x}$.
- Ogni altra coppia di vertici è connessa da un arco.

Quindi la funzione f associa a $\langle\phi\rangle$, dove ϕ è una formula 3CNF con k clausole, la stringa $\langle G, k\rangle$ dove G = (V, E) è il grafo non orientato definito prima.

Nota: k è il numero di clausole in ϕ.

La funzione f è **calcolabile** e può essere calcolata in **tempo polinomiale**.

Per provare che f è una riduzione di tempo polinomiale di 3SAT a CLIQUE resta da dimostrare che $\langle\varphi\rangle \in 3SAT$ se e solo se $\langle G, k\rangle \in CLIQUE$ cioè ϕ è soddisfacibile se e solo se G ha una k-clique.

(⇒). Supponiamo che ϕ abbia un assegnamento di soddisfacibilità. Questo assegnamento di valori alle variabili rende vera ogni clausola $(a_j \vee b_j \vee c_j)$ e quindi esiste almeno un letterale vero in ogni clausola $(a_j \vee b_j \vee c_j)$. Scegliamo un letterale vero in ogni clausola $(a_j \vee b_j \vee c_j)$ e consideriamo il sottografo G' di G indotto dai nodi corrispondenti ai letterali scelti.

G' è una k-clique. Infatti, G' ha k vertici poiché abbiamo scelto un letterale in ognuna delle k clausole e poi i k vertici di G corrispondenti a tali letterali. Due qualsiasi vertici in G' non si trovano nella stessa tripla (corrispondono a letterali in clausole diverse) e non corrispondono a una coppia x, $\bar{x}$ perché corrispondono a letterali veri nell'assegnamento di soddisfacibilità. Quindi due qualsiasi vertici in G' sono connessi da un arco in G.

(⇐). Viceversa, supponiamo che G abbia una k-clique G'. Poiché due nodi in una tripla non sono connessi da un arco, ognuna delle k triple contiene esattamente uno dei nodi della k-clique.

Consideriamo l'assegnamento di valori alle variabili di ϕ che renda veri i letterali corrispondenti ai nodi di G'. Ciò è possibile perché in G' non ci sono vertici corrispondenti a una coppia x, $\bar{x}$. Ogni tripla contiene un nodo di G' e quindi ogni clausola contiene un letterale vero. Questo è un assegnamento di soddisfacibilità per ϕ cioè $\langle\phi\rangle \in$ 3SAT.

Linguaggio NP-Completo

I risultati precedenti ci dicono che se CLIQUE è decidibile in tempo polinomiale lo è anche 3SAT. Questa connessione tra i due linguaggi sembra veramente notevole perché i linguaggi sembrano piuttosto differenti. Ma la riducibilità in tempo polinomiale ci permette di collegare le rispettive complessità. A questo punto introduciamo una definizione che ci permette in modo simile di collegare le complessità di un'intera classe di linguaggi.

Definizione

Un linguaggio B è NP-completo se soddisfa le seguenti due condizioni:

1. B è in NP,
2. ogni A in NP è riducibile in tempo polinomiale a B.

Teorema

Se B è NP-completo e B è in P allora P = NP.

Dimostrazione

Siccome B è NP-completo, per ogni A ∈ NP, risulta $A \leq_P B$.

Ma abbiamo provato che se $A \leq_P B$ e B ∈ P allora A ∈ P

Quindi NP ⊆ P e siccome P ⊆ NP risulta P = NP.

Come abbiamo dimostrato che $HALT_{TM}, \overline{E_{TM}}, REGULAR_{TM}, EQ_{TM}, \overline{EQ_{TM}}$ sono indecidibili?

Abbiamo provato che A_{TM} è indecidibile e, per ognuno dei linguaggi precedenti A, abbiamo definito una riduzione di A_{TM} ad A.

Riducibilità polinomiale ed NP-Completezza

Teorema

Se B è NP-completo e $B \leq_P C$, con C ∈ NP, allora C è NP-completo.

Dimostrazione

Per ipotesi:

1. C ∈ NP,
2. Per ogni A ∈ NP, $A \leq_P B$ (B è NP-completo)
3. $B \leq_P C$

Allora, utilizzando la proprietà transitiva di $\leq_P$:

1. C ∈ NP,
2. Per ogni A ∈ NP, $A \leq_P C$

Cioè C è NP-completo.

Una possibile strategia per provare che un linguaggio B è NP-completo:

1. Mostrare che B ∈ NP
2. Scegliere un linguaggio A che sia NP-completo
3. Definire una riduzione di tempo polinomiale di A a B.

Teorema di Cook-Levine

Teorema (Cook-Levin) Senza Dim.

SAT è NP-completo.

Conseguenza: SAT ∈ P se e solo se P = NP.

Sappiamo che SAT ∈ NP. La prova del teorema di Cook-Levin consiste nel mostrare che ogni A ∈ NP è riducibile in tempo polinomiale a SAT. La riduzione di tempo polinomiale si ottiene definendo per ogni input w una formula booleana ϕ che simula la macchina di Turing non deterministica che decide A sull'input w.

$$B_{CNF} = \{\langle\varphi\rangle \mid \varphi \text{ è } una\ formula\ booleana\ in\ CNF\}$$

$$SAT_{CNF} = \{\langle\varphi\rangle \in B_{CNF} \mid \varphi \text{ è } soddisfacibile\}$$

$SAT_{CNF} \in NP$, allora SAT è riducibile in tempo polinomiale a SAT_{CNF}

$$SAT \leq_P SAT_{CNF}$$

Teorema

SAT_{CNF} è NP-completo.

Nota: la trasformazione classica di un'espressione booleana nella sua forma normale congiuntiva non definisce, in generale, una riduzione di tempo polinomiale.

$$3CNF = formula\ booleana\ in\ forma\ normale\ 3 - congiuntiva$$

$$3SAT = \{\langle\varphi\rangle \mid \varphi \text{ è } una\ formula\ 3CNF\ soddisfacibile\}$$

Teorema
3SAT è NP-completo.

Dimostrazione
3SAT è in NP.

Per provare che 3SAT è NP-completo basta dimostrare che $SAT_{CNF} \leq_P 3SAT$. La prova consiste nel costruire, a partire da ϕ in CNF, una formula booleana ψ in 3CNF tale che ϕ è soddisfacibile se e solo se ψ è soddisfacibile. Inoltre, ψ può essere costruita a partire da ϕ in tempo polinomiale.

Teorema
CLIQUE è NP-completo.

Dimostrazione
Sappiamo che CLIQUE ∈ NP.

Inoltre, 3SAT è NP-completo e $3SAT \leq_P CLIQUE$. Quindi CLIQUE è NP-completo.

Riduzioni di tempo polinomiale mediante gadgets
Quando costruiamo una riduzione di tempo polinomiale da 3SAT ad un linguaggio, cerchiamo strutture in quel linguaggio che possono simulare le variabili e le clausole nelle formule booleane.

Tali strutture sono a volte chiamate *gadgets* (tecnica di "riduzione mediante progettazione di componenti" o "gadgets").

Per ottenere una tale riduzione occorre:

- Definire per ogni variabile una componente (gadget, modella l'assegnazione di verità alla variabile)
- Definire per ogni clausola una componente(modella la soddisfacibilità della clausola)
- Collegare i due insiemi di componenti per garantire che l'assegnazione alle variabili soddisfi tutte le clausole.

Nota. Non è l'unica tecnica per ottenere una riduzione.

VERTEX-COVER è NP-COMPLETO
Sia G=(V,E) un grafo non orientato.

Sia $V' \subset V$, sia $(u, v) \in E$. Se V' contiene almeno uno dei due vertici dell'arco (u,v), diremo che V' **copre** l'arco (u,v).

Un **VERTEX-COVER** V' di G è un sottoinsieme di V' di V tale che per ogni (u,v) ∈ E risulta $\{u, v\} \cap V' \neq \emptyset$ (ossia V' copre ogni arco (u,v) in G).

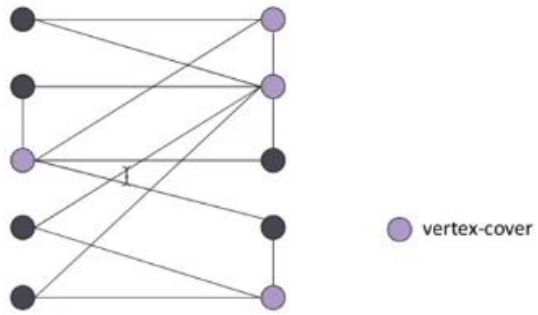
vertex-cover

Dato un grafo non orientato G = (V, E) , un vertex cover è un sottoinsieme V' di vertici, $V' \subseteq V$, tale che per ogni arco in E almeno uno dei suoi estremi è in V'

Ex. esiste un vertex-cover di cardinalità 4

Ex. Non esiste un vertex-cover di cardinalità ≤ 3

(autore slide: Kevin Wayne)



Il problema di stabilire se un grafo non orientato G=(V,E) ha un vertex cover di cardinalità k si può formulare come un problema di decisione, il cui linguaggio associato è VERTEX-COVER.

$$VERTEX - COVER = \{\langle G, k\rangle \mid G \text{ è } un\ grafo\ non\ orientato\ che\ ha\ un\ vertex\ cover\ di\ cardinalità\ k\}$$

Teorema.

VERTEX-COVER ∈ NP

Dimostrazione

Un algoritmo V che verifica VERTEX-COVER in tempo polinomiale:

V = "Sull'input $\langle\langle G, k\rangle, c\rangle$:

1. Verifica se c è un insieme V' di k nodi di G, altrimenti rifiuta.
2. Verifica se V' copre ogni arco in G, accetta in caso affermativo altrimenti rifiuta".

$$\exists c : \langle\langle G, k\rangle, c\rangle \in L(V) \Leftrightarrow \langle G, k\rangle \in VERTEX - COVER$$

Prova alternativa : utilizzare le macchine di Turing non deterministiche.

Teorema

VERTEX-COVER è NP-completo

Dimostrazione

Abbiamo provato che VERTEX-COVER ∈ NP. Per concludere la prova dimostriamo che :

$$3SAT \leq_P VERTEX - COVER$$

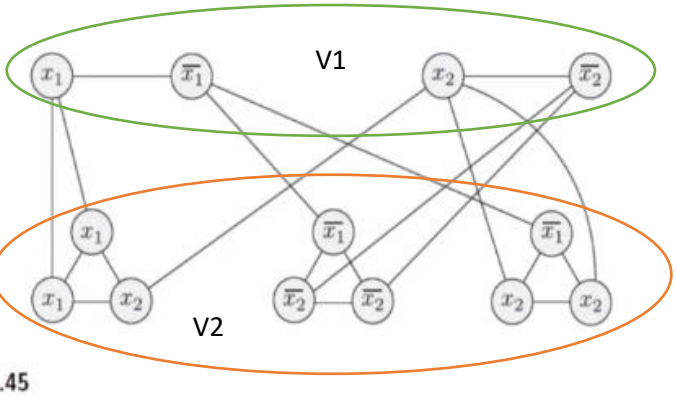


FIGURA 7.45
Il grafo che la riduzione produce a partire da
$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

Figura tratta da M. Sipser, Introduzione alla teoria della computazione.

Sia ϕ una formula 3CNF con *l* clausole e m variabili:

$$(a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_l \vee b_l \vee c_l)$$

Definiamo un grafo non orientato G e un intero k tale che ϕ è soddisfacibile se e solo se G ha un VERTEX-COVER di cardinalità k.

Inoltre, $\langle G, k \rangle$ può essere costruita in tempo polinomiale nella lunghezza di $\langle \phi \rangle$

Costruzione:

- G contiene due vertici per ogni variabile x etichettati con x e $\bar{x}$ (**gadget per le variabili**). Chiamiamo $\boldsymbol{V_1}$ questo insieme di vertici.
- G contiene tre vertici per ogni clausola, etichettati con i tre letterali della clausola (**gadget per le clausole**). Chiamiamo $\boldsymbol{V_2}$ questo insieme di vertici.
- Connettiamo i vertici associati a una variabile con un arco (arco tra x e $\bar{x}$ in V_1).
- Connettiamo i tre vertici associati a una clausola tra loro in un triangolo
- Connettiamo con un arco ogni vertice del triangolo (associato a una clausola) al vertice in V_1 (gadgets per le variabili) che ha la stessa etichetta.
- Se ϕ ha *l* clausole e *m* variabili allora G ha 2m+3l vertici e $V = V_1 \cup V_2$
- Prendiamo k=m+2l.

Proviamo che ϕ è soddisfacibile se e solo se G=(V,E) ha un VERTEX-COVER di cardinalità k.

($\Rightarrow$) Sia ϕ soddisfacibile e sia τ un assegnamento che soddisfa ϕ. Consideriamo il sottoinsieme V' di V che contiene:

- Tutti i vertici in V_1 (gadget per le variabili) che hanno come etichette i letterali veri in τ
- Due vertici per ogni triangolo (gadget per la clausola), escludendone uno che ha etichetta uguale a un vertice selezionato al passo precedente (ne esiste almeno uno).

Se ϕ ha *l* clausole e *m* variabili allora questo sottoinsieme V' di V ha taglia $k = m + 2l$

Inoltre V' è un VERTEX-COVER:

- Tutti gli archi in un triangolo sono coperti (dai due vertici selezionati)
- Tutti gli archi tra due vertici di V_1 o tra un vertice di V_1 e un vertice di V_2 sono coperti (dalla scelta dei vertici in V_1 o V_2).

(⇐) Supponiamo che G=(V,E) abbia un VERTEX-COVER V' di cardinalità $k = m + 2l$ e proviamo che ϕ è soddisfacibile.

V' deve contenere almeno 2 vertici di ogni triangolo e , per ogni arco tra due vertici di V_1 (gadget per le variabili), almeno 1 dei 2 vertici. Siccome il numero dei triangoli è l e il numero di archi tra i vertici in V_1 è m, l'insieme V' contiene **esattamente** due vertici di ogni triangolo e, per ogni arco tra due vertici in V_1, uno dei due vertici.

Assegniamo valore vero ai letterali che sono etichette di vertici $V' \cap V_1$. Proviamo che questo assegnamento τ soddisfa ϕ. Cioè che questo assegnamento rende vera ogni clausola.

Infatti, per ogni triangolo esiste un vertice u che non è in V'. Ma (u,v), con v ∈ V_1 deve essere coperto da V'. Quindi $v \in V'$.

Ma u e v hanno la stessa etichetta (per costruzione di G) che corrisponde a un letterale a cui è assegnato il valore 1 (per costruzione di τ). Dunque, per ogni clausola c, c'è un letterale a cui τ assegna valore 1 e quindi ϕ è soddisfacibile.

Problema HAMPATH

Consideriamo il problema di stabilire se un grafo orientato contiene un cammino Hamiltoniano che collega due nodi specificati.

Questo si può formulare come un problema di decisione a cui corrisponde il linguaggio HAMPATH:

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ è } un\ grafo\ orientato\ e\ ha\ un\ cammino\ Hamiltoniano\ da\ s\ a\ t\}$$

Teorema

HAMPATH ∈ NP

Dimostrazione

Un algoritmo N che verifica HAMPATH in tempo polinomiale:

N ="Sull'input $\langle\langle G, s, t\rangle, c\rangle$ dove G=(V,E) è un grafo orientato:

1. Verifica se $c = (u_1, \dots, u_{|V|})$ è una sequenza di |V| vertici di G, altrimenti rifiuta.
2. Verifica se i nodi della sequenza sono distinti, $u_1 = s, u_{|V|} = t$ e, per ogni i con $2 \le i \le n$, se $(u_{i-1}, u_i) \in E$, accetta in caso affermativo, altrimenti rifiuta."

$$\exists c: \langle\langle G, s, t\rangle, c\rangle \in L(N) \Leftrightarrow \langle G, s, t\rangle \in HAMPATH$$

HAMPATH è NP-completo

Mostriamo che 3SAT è riducibile in tempo polinomiale a HAMPATH.

La riduzione converte formule booleane in 3CNF in grafi, in cui i cammini Hamiltoniani corrispondono ad assegnamenti soddisfacenti la formula. I grafi contengono gadget che simulano variabili e clausole.

- Il gadget di una variabile è una struttura romboidale che può essere attraversata in due modi, corrispondenti ai due assegnamenti di verità.
- Il gadget della clausola è un nodo.

Assicurare che il cammino che attraversa ciascun gadget di clausola corrisponde ad assicurare che ciascuna clausola è soddisfatta nell'assegnamento che soddisfa la formula.

Sia ϕ una formula 3CNF con k clausole:

$$(a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \ldots \wedge (a_k \vee b_k \vee c_k)$$

Dove ciascun a, b, c è un letterale x_i *o* $\overline{x_i}$. Siano $x_1, \ldots, x_l$ le variabili di ϕ.

Definiamo un grafo orientato G, con due vertici s e t, tale che ϕ è soddisfacibile se e solo se G ha un cammino Hamiltoniano da s a t. Inoltre $\langle G, s, t \rangle$ può essere costruita in tempo polinomiale nella lunghezza di $\langle \phi \rangle$.

Rappresentiamo ciascuna variabile x_i con una struttura di forma **romboidale** che contiene una riga orizzontale di nodi, come mostrato nella figura seguente.

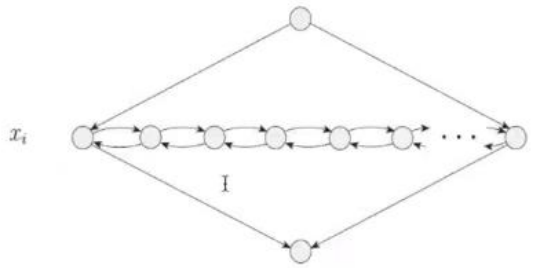


FIGURA 7.47
Rappresentazione della variabile x_i con una struttura romboidale

Figura tratta da M. Sipser, Introduzione alla teoria della computazione.

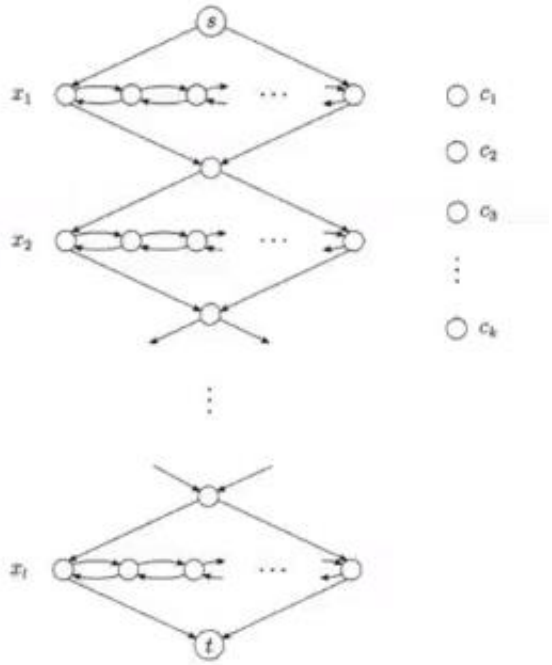
Specificheremo dopo il numero di nodi che compaiono nella riga orizzontale. Il numero di vertici sulla linea orizzontale è legato al numero di clausole nella formula.

Rappresentiamo ogni clausola di ϕ con un singolo nodo, come segue.

FIGURA 7.48
Rappresentazioni della clausola c_j con un nodo

Figura tratta da M. Sipser, Introduzione alla teoria della computazione.

La figura seguente riporta la struttura globale di G. Mostra quasi tutti gli elementi di G e le loro relazione. Mancano gli archi che rappresentano la relazione delle variabili con le clausole che le contengono.



Ciascuna struttura romboidale contiene una riga orizzontale di nodi collegati tramite archi orientati in entrambe le direzioni. La riga orizzontale contiene *3k+1 (k è il numero di clausole)* nodi in aggiunta ai due nodi all'estremità del rombo. Questi nodi sono raggruppati in coppie adiacenti, una per ciascuna clausola, con nodi separatori aggiuntivi tra le coppie, come mostrato nella figura seguente.

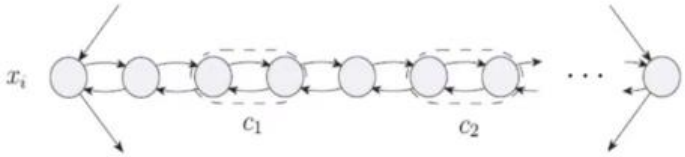


FIGURA 7.50
I nodi orizzontali in una struttura romboidale

Sulla linea orizzontale del grafo associato a una variabile ci sono 2 vertici per ogni clausola e ogni coppia è separata dalla successiva da un vertice "separatore" (*3k+1+2 vertici).*

Se la variabile x_i è presente nella clausola c_j, aggiungiamo i due archi seguenti della coppia j-esima nell'i-esimo rombo al j-esimo nodo della clausola.

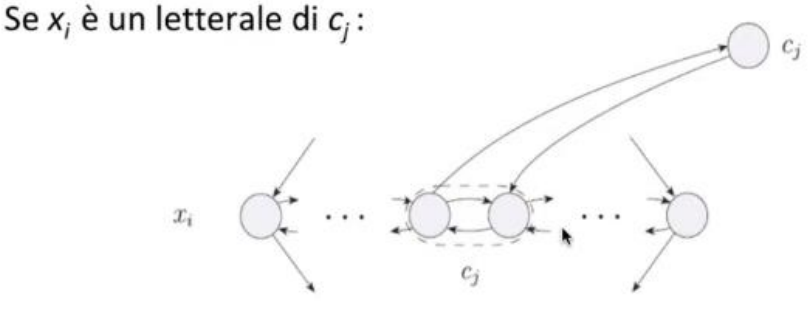


FIGURA 7.51
Gli archi aggiuntivi quando la clausola c_j contiene x_i

Se $\overline{x_i}$ è presente nella clausola c_j, aggiungiamo due archi dalla coppia j-esima nell'i-esimo rombo al j-esimo nodo della clausola.

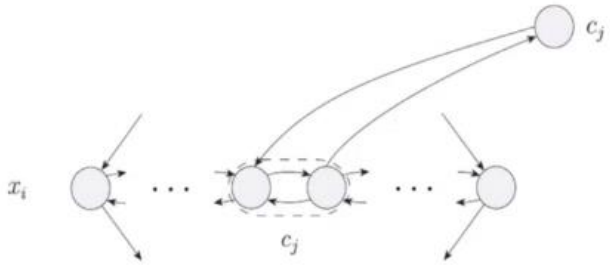


FIGURA 7.52
Gli archi aggiuntivi quando la clausola c_j contiene $\overline{x_i}$

Figura tratta da M. Sipser, Introduzione alla teoria della computazione.

Dopo aver aggiunto tutti gli archi corrispondenti a ciascuna occorrenza di x_i o di $\overline{x_i}$ in ciascuna clausola, la costruzione di G è completa. Per far vedere che questa costruzione funziona, dimostriamo che se ϕ è soddisfacibile, allora esiste un cammino Hamiltoniano da s a t e viceversa, se esiste un tale cammino in G, allora ϕ è soddisfacibile.

($\Rightarrow$)Supponiamo che ϕ sia soddisfacibile. Per mostrare che esiste un cammino Hamiltoniano da s a t, in un primo momento ignoriamo i nodi delle clausole. Il cammino che inizia da s, attraversa ciascun rombo in successione e termina in t. Per raggiungere i nodi orizzontali in un rombo, il cammino può procedere a **zig-zag** da sinistra a destra oppure a **zag-zig** da destra a sinistra.

L'assegnamento che soddisfa ϕ determina in quale senso la riga è attraversata.

- Se a $x_i$viene assegnato il valore 1, il cammino procede a zig-zag attraverso il rombo corrispondente. Dunque il cammino attraversa la riga i da sinistra a destra.
- Se a $x_i$viene assegnato il valore 0, il cammino procede a zag-zig attraverso il rombo corrispondente. Dunque il cammino attraversa la riga i da destra a sinistra.

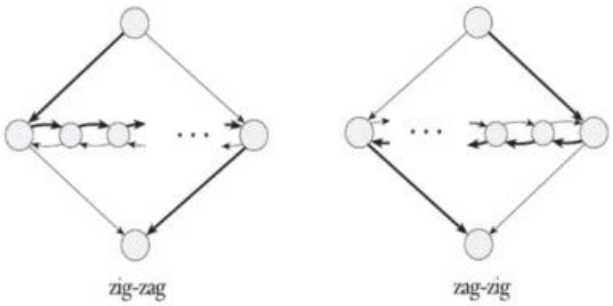


FIGURA 7.53
Zig-zag (da sinistra a destra) o zag-zig (da destra a sinistra), come stabilito dall'assegnamento soddisfacente

Questo cammino copre tutti i nodi di G, eccetto i nodi delle clausole. Per includerli aggiungiamo una deviazione per ognuno di tali nodi in questo modo:

- Per ciascuna clausola c_j scegliamo uno dei letterali in c_j a cui è assegnato il valore 1. Supponiamo che sia x_i oppure $\overline{x_i}$.

- Se selezioniamo x_i nella clausola c_j, attraversiamo la riga i corrispondente nella direzione "corretta" per raggiungere il vertice c_j da uno dei vertici della j-esima coppia nella riga e tornare nell'altro vertice. Cioè possiamo deviare alla j-esima coppia nell'i-esimo rombo. Questo è possibile perche x_i deve essere 1, quindi il cammino procede a zig-zag da sinistra a destra attraverso il rombo corrispondente. Quindi gli archi verso il nodo c_j sono nell'ordine corretto per permettere una deviazione e un ritorno.
- Allo stesso modo, se avessimo selezionato $\overline{x_i}$ nella clausola c_j, avremmo potuto deviare alla j-esima coppia nell'i-esimo rompo perché x_i deve essere 0, quindi il cammino procede a zig-zag da destra a sinistra attraverso il rombo corrispondente. Pertanto gli archi verso il nodo c_j sono nuovamente nell'ordine corretto per permettere una deviazione ed un ritorno.

(Si consideri che ciascun letterale vero in una clausola fornisce una *opzione* di deviazione per raggiungere il nodo clausola. Come risultato, se diversi letterali in una clausola sono veri, viene presa soltanto una deviazione.)

Cosi abbiamo costruito il cammino Hamiltoniano.

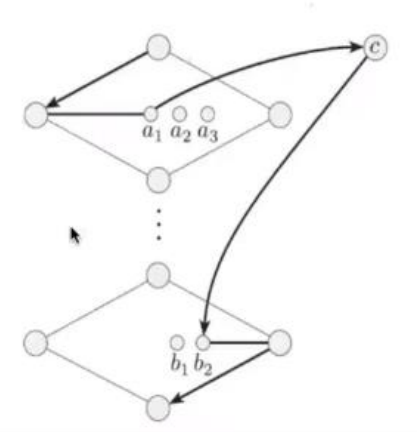
($\Leftarrow$) Mostriamo che se un grafo G così costruito ha una cammino Hamiltoniano da s a t allora esiste un assegnamento che soddisfa ϕ.

Se il cammino Hamiltoniano è *normale*, ossia passa attraverso i rombi in ordine, da quello più in alto a quello più in basso, con deviazioni verso i nodi delle clausole come descritte prima, possiamo facilmente definire un assegnamento che soddisfa ϕ:

- Alla variabile i-esima x_i assegniamo il valore 1 se la riga orizzontale è attraversata da sinistra a destra, cioè se il cammino procede a zig-zag attraverso il rombo,
- Alla variabile i-esima x_i assegniamo il valore 0 se la riga orizzontale è attraversata da destra a sinistra, cioè se il cammino procede a zag-zig attraverso il rombo

L'assegnamento soddisfa la formula poiché ciascun nodo clausola è connesso a una coppia di una riga e quindi contiene un letterale vero. Osservando come la deviazione verso il nodo clausola avviene, possiamo stabilire quale dei letterali nella clausola corrispondente è vero.

Tutto ciò che resta da mostrare è che un cammino Hamiltoniano deve essere normale. La normalità può venir meno solo se il cammino entra in una clausola da un rombo e ritorna in un altro come in figura seguente:



Il vertice a_2 o a_3 deve essere un vertice separatore. Se a_2 è un vertice separatore, l'unico modo per raggiungerlo è da a_3 (non potendo da a_1). Analogamente se a_3 è un vertice separatore, l'unico modo per raggiungere a_2 è da a_3 (non potendo da a_1 e c). Poi però il cammino non ha modo di uscirne e a_2 non può essere l'ultimo vertice del cammino (che deve essere t).

Il cammino va dal nodo a_1 a c; ma invece di tornare in a_2 nello stesso rombo, ritorna in b_2 in un rombo differente. Se questo accade a_2 oppure a_3 è un nodo separatore.

- Se a_2 fosse un nodo separatore, gli unici archi entranti in a_2 sarebbero quelli da a_1 e a_3.
- Se a_3 fosse un nodo separatore, a_1 e a_2 sarebbero nella stessa coppia associata alla clausola c, quindi gli unici archi entranti in a_2 sarebbero quelli da a_1,a_3 e c.

In ogni caso il cammino non potrebbe contenere il nodo a_2. Il cammino non può entrare in a_2 da c o da a_1 perché il cammino va altrove da questi nodi. Il cammino non può entrare in a_2 da a_3 perché a_3 è l'unico nodo disponibile a cui a_2 punta, quindi il cammino deve uscire da a_2 passando per a_3.

Pertanto il cammino Hamiltoniano deve essere normale. Questa riduzione è ovviamente calcolabile in tempo polinomiale e la dimostrazione è completa.

Problema UHAMPATH

È possibile definire una "versione non orientata" del problema del cammino Hamiltoniano.

Un cammino Hamiltoniano in un grafo non orientato è un cammino che passa per ogni vertice del grafo una e una sola volta.

$$UHAMPATH = \{\langle G, s, t\rangle \mid G \text{ è } un\ grafo\ non\ orientato\ e\ ha\ un\ cammino\ Hamiltoniano\ da\ s\ a\ t\}$$

Per mostrare che UHAMPATH è NP-completo, definiamo una riduzione di tempo polinomiale da HAMPATH A UHAMPATH.

Teorema

UHAMPATH ∈ NP.

Dimostrazione

Un algoritmo N che verifica UHAMPATH in tempo polinomiale:

N= "Sull'input $\langle\langle G, s, t\rangle, c\rangle$, dove G=(V,E) è un grafo non orientato:

1. Verifica se $c = (u_1, \dots, u_{|V|})$ è una sequenza di |V| vertici di G , altrimenti rifiuta.
2. Verifica se i nodi della sequenza sono distinti, $u_1 = s, u_{|V|} = t$ e, per ogni i con $2 \leq i \leq n$, se $(u_{i-1}, u_i) \in E$, accetta in caso affermativo, altrimenti rifiuta".

$$\exists c\colon \langle\langle G, s, t\rangle, c\rangle \in L(N) \Leftrightarrow \langle G, s, t\rangle \in UHAMPATH$$

Teorema

UHAMPATH è NP-completo

Dimostrazione

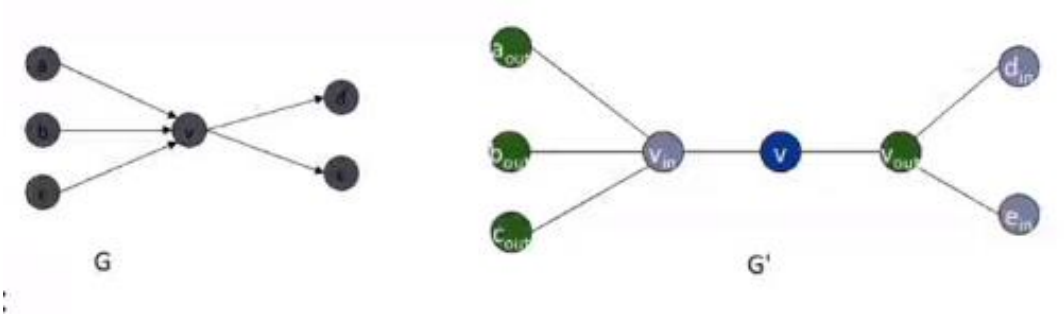
Abbiamo provato che UHAMPATH è in NP. Per concludere la prova, dimostriamo che:

$$HAMPATH \leq_P UHAMPATH$$

La riduzione di tempo polinomiale associa a un grafo orientato G=(V,E) con vertici s e t un grafo non orientato G'=(V',E') con vertici s' e t'.

Il grafo G ha un cammino Hamiltoniano da s a t se e solo se G' ha un cammino Hamiltoniano da s' a t'. Inotre G' deve essere costruito a partire da G in tempo polinomiale.

(⇒)Costruiamo un grafo G' con 3n vertici:



- Ogni vertice u di G, diverso da s e t è sostituito da tre vertici u^{in}, u^{mid}, u^{out} in G'.
- I vertici s e t sono sostituiti dai vertici s^{out} *e* t^{in} in G'.
- Per ogni $u \in V\backslash\{s,t\}$, (u^{in}, u^{mid}) *e* (u^{mid}, u^{out}) sono in E'
- Se $(u,v) \in E$ *allora* $\left(u^{out}, v^{in}\right) \in E'$.

Dimostriamo che G ha un cammino Hamiltoniano da s a t se e solo se G' ha un cammino Hamiltoniano da s^{out} *a* t^{in}.

Se G ha un cammino Hamiltoniano P da s a t:

$$P = s, u_1, u_2, \dots u_k, t$$

Allora

$$P' = s^{out}, u_1^{in}, u_1^{mid}, u_1^{out}, u_2^{in}, u_2^{mid}, u_2^{out}, \dots, u_k^{in}, u_k^{mid}, u_k^{out}, t^{in}$$

Sarà un cammino Hamiltoniano in G' da $s^{out} a\ t^{in}$.

(⇐)Se G' ha un cammino Hamiltoniano P' da $s^{out}\ a\ t^{in}$, è facile vedere che P' deve essere della forma:

$$P' = s^{out}, u_1^{in}, u_1^{mid}, u_1^{out}, u_2^{in}, u_2^{mid}, u_2^{out}, \dots, u_k^{in}, u_k^{mid}, u_k^{out}, t^{in}$$

La prova è per induzione su k. Infatti P' ha come primo vertice s^{out} il quale è connesso solo a vertici della forma u_i^{in}. Quindi il secondo vertice è u_i^{in} per qualche vertice i. I vertici successivi devono essere u_i^{mid} e u_i^{out} perchè u_i^{mid} è connesso solo a u_i^{in} e u_i^{out}. Ma se P' ha la forma suddetta allora:

$$P = s, u_1, u_2, \dots u_k, t$$

È un cammino Hamiltoniano da s a t.

Problema SUBSET-SUM

Dato un insieme S di numeri interi e un numero intero t, esiste un sottinsieme S' di D tale che la somma dei suoi numeri sia uguale a t?

$$SUBSET - SUM = \left\{\langle S, t\rangle \mid \exists S' \subset S\ tale\ che \sum_{s \in S'} s = t\right\}$$

Esempio: {⟨4,11,16,21,27⟩, 25} ∈ SUBSET-SUM perché 4+21 = 25.

Nota. È possibile definire SUBSET-SUM in cui S, S' sono multinsiemi (cioè insiemi in cui alcuni elementi si ripetono).

Teorema

SUBSET-SUM ∈ NP

Dimostrazione.

Un algoritmo V che verifica SUBSET-SUM in tempo polinomiale:

V=”Sull’input $\langle\langle S, t\rangle, c\rangle$:

1. Verifica se c è un insieme di numeri la cui somma è t, altrimenti rifiuta.
2. Verifica se S contiene tutti i numeri in c, accetta in caso affermativo altrimenti rifiuta.”

$$\exists c: \langle\langle S, t\rangle, c\rangle \in L(V) \Leftrightarrow \langle S, t\rangle \in SUBSET - SUM$$

Prova alternativa : utilizzare le macchine di Turing non deterministiche.

Teorema

SUMSET-SUM è NP-completo.

Dimostrazione

Abbiamo già provato che SUBSET-SUM è in NP. Per concludere la prova basta provare che

$$3SAT \leq_P SUBSET - SUM$$

($\Rightarrow$)Sia ϕ una formula 3CNF con variabili $x_1, x_2, \dots, x_l$ e clausole $c_1, \dots, c_k$. La riduzione associa a ϕ un insieme di S di numeri e un numero t, espressi nella notazione decimale ordinaria.

	1	2	3	4	⋯	l	c_1	c_2	⋯	c_k
y_1	1	0	0	0	⋯	0	1	0	⋯	0
z_1	1	0	0	0	⋯	0	0	0	⋯	0
y_2		1	0	0	⋯	0	0	1	⋯	0
z_2		1	0	0	⋯	0	1	0	⋯	0
y_3			1	0	⋯	0	1	1	⋯	0
z_3			1	0	⋯	0	0	0	⋯	1
⋮					⋱	⋮	⋮		⋮	⋮
y_l						1	0	0	⋯	0
z_l						1	0	0	⋯	0
g_1							1	0	⋯	0
h_1							1	0	⋯	0
g_2								1	⋯	0
h_2								1	⋯	0
⋮									⋱	⋮
g_k										1
h_k										1
t	1	1	1	1	⋯	1	3	3	⋯	3

Per ogni variabile x_i con $1 \leq i \leq l$ definiamo due numeri y_i e z_i di l+k cifre decimali:

- y_i ha un 1 in posizione i (da sinistra) e in ogni posizione l+j (da sinistra) per cui x_i appare nella clausola c_j, con $1 \leq j \leq k$,
- z_i ha un 1 in posizione i (da sinistra) e in ogni posizione l+j (da sinistra) per cui $\overline{x_i}$ appare nella clausola c_j, con $1 \leq j \leq k$,
- Per ogni clausola c_j con $1 \leq i \leq k$, definiamo due numeri uguali g_j e h_j di l+k cifre decimali, tali che g_j e h_j hanno un 1 in posizione l+j (da sinistra).
- Le cifre non specificate uguali a 1 saranno uguali a 0.

$$S = \{y_i, z_i \mid 1 \leq i \leq l\} \cup \{g_j, h_j \mid 1 \leq j \leq l\}$$

Definiamo **t** un numero di l+k cifre decimali che ha un 1 in posizione i (da sinistra), per $1 \leq i \leq l$ e ha un 3 in posizione l+j (da sinistra) per $1 \leq j \leq l$.

- Esempio: $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2 \vee \overline{x_3})$

Numero	1	2	3	1	2	3
y_1	1	0	0	1	0	0
z_1	1	0	0	0	1	1
y_2	0	1	0	1	0	1
z_2	0	1	0	0	1	0
y_3	0	0	1	1	1	0
z_3	0	0	1	0	0	1
g_1	0	0	0	1	0	0
h_1	0	0	0	1	0	0
g_2	0	0	0	0	1	0
h_2	0	0	0	0	1	0
g_3	0	0	0	0	0	1
h_3	0	0	0	0	0	1
t	1	1	1	3	3	3

Sia ϕ soddisfacibile e sia τ un assegnamento che soddisfa ϕ. Consideriamo il sottinsieme S' di S che contiene y_i se τ assegna a x_i valore 1, z_i altrimenti. Se sommiamo ciò che abbiamo scelto finora, otteniamo un 1 in ciascuna delle prime l cifre perché abbiamo selezionato y_i o z_i per ciascun i.

Inoltre ciascuna delle ultime k cifre è un numero da 1 a 3 perché ciascuna clausola è soddisfatta e quindi contiene da 1 a 3 letterali veri. Quindi scegliamo un numero sufficiente di g_j, h_j da aggiungere a S' per portar ciascuna delle ultime k cifre fino a 3 e ottenere:

$$\sum_{s \in S'} s = t$$

($\Leftarrow$)Supponiamo che esista un sottoinsieme S' di S tale che $\sum_{s \in S'} s = t$

Due osservazioni:

- Tutte le cifre negli elementi di S sono 0 o 1.
- Ciascuna colonna nella tabella che descrive S contiene al più cinque 1.

Per ogni i con $1 \leq i \leq l$, S' deve contenere y_i o z_i ma non entrambi. Sia τ l'assegnamento definito come segue: per ogni i con $1 \leq i \leq l$ assegnaimo a x_i il valore:

- 1 se S' contiene y_i
- 0 se S' contiene z_i.

Questo assegnamento τ soddisfa ϕ. Infatti poiché le ultime k cifre di t sono uguali a 3, in ciascuna delle k colonne finali, la somma è sempre 3.

Per ogni j con $1 \leq j \leq k$, almeno un 1 nella colonna c_j deve venire da qualche y_i o z_i nel sottoinsieme S' perché da g_j ed h_j può venire al più 2.

Per ogni j nella colonna c_j vi deve essere una cifra uguale a 1 corrispondente ad un y_i o z_i in S'.

- Se è y_i allora x_i è presente in c_j e gli viene assegnato 1, quindi c_j è soddisfatta.
- Se è z_i allora $\overline{x_i}$ è presente in c_j e a x_i viene assegnato 0, quindi c_j è soddisfatta.

Pertanto ϕ è soddisfatta e la riduzione può essere effettuata in tempo polinomiale.

VAFFANCULO ETC