Informatica Teorica

Stefano Staffolani

Anno accademico 2022-2023

Contents

1 La macchina di Turing **4**
1.1 Espressività delle macchine di Turing . . . 4
1.2 Linguaggi formali: ripasso . . . 4
1.3 Dai problemi di decisione ai linguaggi formali . . . 5
1.4 Linguaggi decidibili . . . 5
1.5 Linguaggi riconoscibili . . . 5

2 WHILE un linguaggio di programmazione di alto livello **6**
2.1 Semantica . . . 6
2.2 Equivalenza . . . 6

3 Macchina di Turing Universale **7**
3.1 Programmi universali . . . 7
3.2 La macchina di Turing universale (UTM) . . . 7
3.3 Codificare una TM . . . 8
3.4 Osservazioni sulla codifica . . . 8
3.5 Costruzione di una UTM . . . 8
3.6 Considerazioni finali . . . 9

4 Cosa non possono fare le TM **9**
4.1 Ripasso: Linguaggi e TM . . . 9
4.2 Gradi di (in)calcolabilità . . . 9
4.3 Un problema indecidibile: Halting Problem . . . 9
4.4 Problemi non riconoscibili . . . 11
4.5 Osservazione 1. Complemento linguaggio riconoscibile . . . 11
4.6 Osservazione 2. Ridurre un problema ad un altro . . . 12

5 Mapping-reduction **12**
5.1 Perchè . . . 12
5.2 Definizione . . . 12
5.3 Riduzione e Decidibilità . . . 13
5.4 Mapping-reduction in azione: ETH . . . 13
5.5 Il full language problem . . . 13
5.6 L'equivalence problem . . . 14
5.7 Riduzione e Riconoscibilità . . . 15
5.8 l'equivalence problem non è riconoscibile . . . 15
5.9 Ulteriori proprietà della mapping-reduction . . . 16

5.10 Riduzione come Relazione . . . 16
5.11 Riduzione e Complemento . . . 16
5.12 Gerarchia dei problemi . . . 16
5.13 Turing-riducibilità . . . 17

6 La cardinalitá dei problemi irrisolvibili 17
6.1 Obiettivo . . . 17
6.2 Insiemi infiniti numerabili . . . 17

7 Linguaggi sono non numerabili 18
7.1 Riassumendo . . . 19
7.1.1 Quanti linguaggi non riconoscibili ci sono? . . . 19

8 Teorema di Rice 19
8.1 ripasso: linguaggi . . . 20
8.2 Teorema di Rice . . . 20
8.3 Propritá decidibili . . . 21

9 Tiling 21
9.1 Sistema di Tiling . . . 21
9.2 Tiling . . . 21
9.2.1 Esempio . . . 22
9.3 Il problema del tiling . . . 23
9.4 Sistema di tiling, formalmente . . . 23
9.5 Problemi di tiling . . . 23

10 Teoria della complessità computazionale 24
10.1 Misurare la complessità . . . 24
10.1.1 Complessità di tempo . . . 24
10.1.2 Notazione Big-O . . . 24
10.1.3 Classi di complessità . . . 24
10.2 La classe P . . . 25
10.2.1 Perchè P? . . . 25
10.2.2 Tesi di Church-Turing rafforzata . . . 25
10.3 La classe EXP . . . 25
10.4 Esempio: PATH . . . 25
10.5 Una nota sulla codifica . . . 26
10.6 Altri problemi in P: panoramica . . . 26
10.7 La classe NP . . . 26
10.8 Esempio: CLIQUE . . . 26
10.9 Una caratterizzazione alternativa di NP . . . 27
10.10 P vs NP . . . 27
10.10.1 Perchè Knuth propende per P=NP? . . . 27

11 Poly reduction 28
11.1 Poly reduction e P . . . 28
11.2 Formule e soddisfacibilità . . . 28
11.2.1 3cnf . . . 29
11.2.2 3SAT e SAT . . . 29
11.3 3SAT e CLIQUE . . . 29
11.4 NP-completezza . . . 30
11.5 Il teorema di Cook-Levin . . . 30

11.6 Altri problemi NP-completi . . . 32

12 Complessità di Spazio **33**
12.1 PSPACE e NSPACE . . . 33
12.2 (N)P e PSPACE . . . 33
12.3 Un problema PSPACE completo . . . 34
12.4 Riflessioni sul teorema . . . 35
12.5 Il teorema di Savitch . . . 35

13 Gerarchie e problemi intrattabili **37**
13.1 Notazione small-O . . . 37
13.2 Funzioni costruibili . . . 37
13.2.1 Esempio . . . 37
13.3 Gerarchia di spazio . . . 37
13.4 Gerarchia di tempo . . . 38
13.5 Considerazioni finali . . . 39

14 esercitazione 1 Marzo 2023 **40**
14.1 Quesito 1 . . . 40
14.2 Problema 2.1 . . . 40
14.3 Problema 2.2 . . . 40

1 La macchina di Turing

Una macchina di Turing è una tupla $\langle \sum, Q, q_0, H, \delta \rangle$. Dove:

1. $\sum$ è un **alfabeto** di simboli, che include un simbolo speciale $\emptyset$ che indica una cella vuota.
2. Q è un insieme finito di **stati**.
3. $q_0 \in Q$ è lo stato iniziale.
4. $H \subset Q$ è l'insieme degli **stati accettanti (o finali)**.
5. δ è la funzione di transizione $\delta : (Q \backslash H) \times \sum \to Q \times \sum \times \{\to, \leftarrow\}$

δ esprime il programma che governa il funzionamento della TM, ed è una funzione totale. Possimao scrivere la definizione di δ come un insieme di quintuple.

1.1 Espressività delle macchine di Turing

Molto spesso un problema che vogliamo risolvere usando uno strumento di calcolo può essere espresso come un problema di decisione. Ad esempio

- Dato un grafo, è strettamente connesso?
- Dato un insieme di equazioni, ha una soluzione?

Come fare tutto ciò con delle macchine di Turing?
Abbiamo visto che le TM possono calcolare funzioni di tipo $\mathbb{N}^k \to \mathbb{N}$, il passaggio chiave è codificare un probleman di decisione come la $funzione\ caratteristica$ di un $linguaggio\ formale$.

1.2 Linguaggi formali: ripasso

Dato un alfabeto $\sum$ di simboli, un **linguaggio formale** è un sottoinsieme di $\sum^*$. Ad esempio:

- Alfabeto $\sum\ =\ \{1\}$
- Linguaggio delle stringhe di lughezza pari: $L\ =\ \{\epsilon, 11, 1111, ...\}$

La **funzione caratteristica** di un linguaggio L è la funzione

$$\chi_L(X) = \begin{cases} 1, & \text{if } x \in L \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

1.3 Dai problemi di decisione ai linguaggi formali

Il nostro schema di codifica deve essere tale che dato α (dati del problema di decisione) otteniamo $code(\alpha) \in \sum^*$. Il linguaggio che codifica il **problema di decisione** sarà:

$$L \ = \ \{x \in \sum^* | x = code(\alpha) \ per \ qualche \ \alpha \wedge \alpha \in pos(P)\}$$

Dove $pos(P)$ sono le istanze positive del problema P (dati per i quali la risposta è si).

Le proprietà[1] che solitamente vogliamo per $code(-)$ sono:

1. $\alpha \neq \beta \implies code(\alpha) \neq code(\beta)$;
2. Dovremmo poter verificare se $x \in \sum^*$ è $code(\alpha)$ per qualche α;
3. Dovremmo poter calcolare α a partire da $code(\alpha)$.

1.4 Linguaggi decidibili

Come facciamo a ragionare su un problema di decisione utilizzando una macchina di Turing?

Assumiamo una codifica del problema di decisione come un linguaggio L su un alfabeto $\sum'$. Vogliamo una Tm M con le seguenti proprietà:

1. l'alfabeto di input/output $\sum_I$ è $\sum'$.
2. l'insieme degli stati finali è $H \ = \ \{Y, N\}$.

Diciamo che M **accetta** un input $x \in \sum_I^*$ se la computazione ferma in stato Y. M **rigetta** x se ferma in stato N.
M **decide** L se:

- $x \in L \implies M \ accetta \ x.$
- $x \notin L \implies M \ rigetta \ x.$

Un linguaggio (un problema di decisione) è **decidibile** se c'è una TM che lo decide.

1.5 Linguaggi riconoscibili

Assumiamo una codifica del problema di decisione come un linguaggio L su un alfabeto $\sum'$. Vogliamo una TM con alfabeto di input/output $\sum'$.
M **riconosce** L se:

- $x \in L \implies M \ si \ ferma$(= raggiunge uno stato finale).
- $x \notin L \implies M \ non \ si \ ferma$(= entra in un ciclo).

Un linguaggio (un problema di decisione) è **riconoscibile** se c'è una TM che lo riconosce.

$$Decidibile \implies Riconoscibile$$

[1] Quando si guarda alla complessità computazionale, altre proprietà diventano rilevanti: ad esempio quando la codifica é ridondante e calcolata in modo efficiente.

2 WHILE un linguaggio di programmazione di alto livello

Turing-completo: quando il suo potere espressivo e' equivalente ad una macchina di Turing. Quindi quando qualcuno scrive un nuovo linguaggio deve preoccuparsi del fatto che esso sia Turing-complete. Non tutti i linguaggi sono Turning-Completi. Ad esempio i domain specific language. Sintassi in maniera informale:

1. 1) assegnazione di variabili $X := 3$
2. 2) cicli while while $X \neq Y$ do Program
3. 3) sequenziamento di programmi $program1; program2; program3$
4. 4) altri costrutti come $if - then - else$ possono essere definiti a partire da questi primitivi.

1) assegnazione di variabili $X := 3$ NB: NON ci sono $side - effects$(tipo print(42))

2.1 Semantica

Un programma WHILE calcola una funzione paziale da $\mathbb{N}^k \to \mathbb{N}$

2.2 Equivalenza

Theorem 2.1. *Una funzione (parziale) é computabile da un programma WHILE se e solo se é computabile da una macchina di Turing.*

Proof. La dimostrazione e' per induzione sulla struttura di un programmma WHILE, supponiamo basato su variabili $X_1 ... X_k$

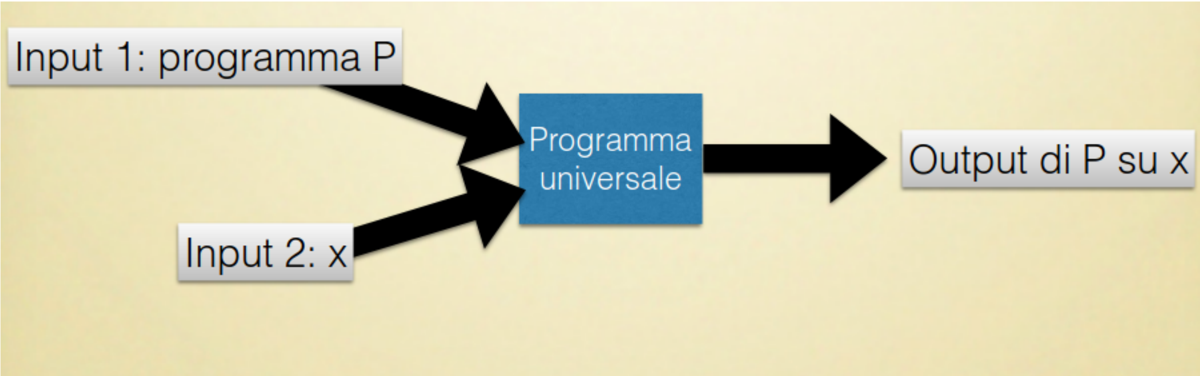
1. Caso base Se il programma e' un'assegnazione di 0 ad una variabile: $beginX_j := 0end$ la TM corrispondente e' quella che su input rimpiazza il valore di X_j con 0. Gli altri casi base sono il programma vuoto e le altre due forme di assegnazione.
2. Caso induttivo Ci sono due casi da considerare: sequenza e ciclo while.
   - sequenza per II abbiamo un programma della forma $beginP_1; P_2; ...; P_jend$ dove $P_1, ...$ sono programmi per i quali abbiamo, da ipotesi induttiva, TM equivalenti M1,... rispettivamente. La TM per $beginP_1; ...; P_jend$ e' definita nel seguente modo a partire da esse, dove l'output di Mi viene dato come input di Mi+1.
   - ciclo while Assumiamo un programma della forma $beginwhileX_i! = X_jdoPend$ dove P e' un programma per il quale, da ipotesi induttiva, abbiamo un aTM M equivalente. Possiamo costruire una TM M_{test} che rigetta l'input se il valore di X_ieX_j e' diverso, ed accetta altrimenti. La TM per il nostro programma e' costruita come segue: (snapshot da inserire)

□

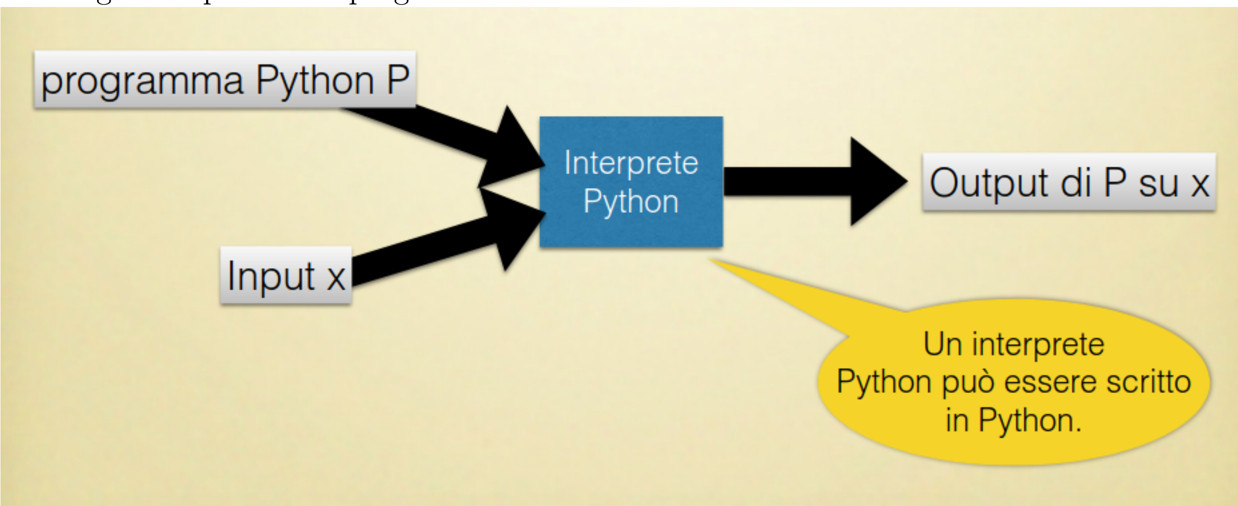
3 Macchina di Turing Universale

3.1 Programmi universali

Un programma universale e' un programma pensato per ricevere altri programmi come input ed eseguirli. I sistemi operativi sono esempi di programmi universali.

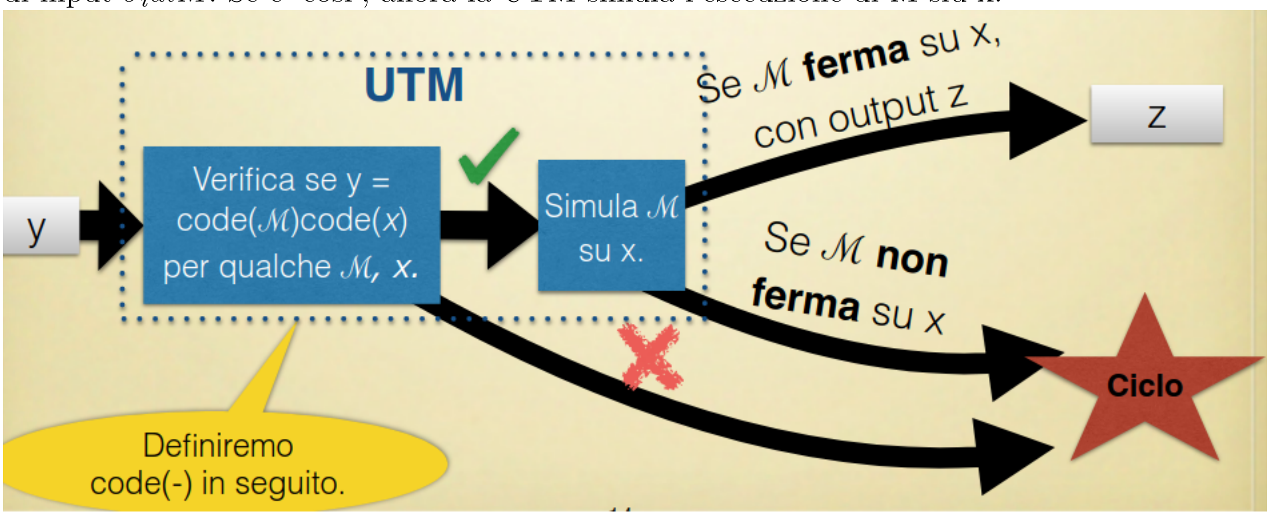


Anche gli interpreti sono programmi universali.



3.2 La macchina di Turing universale (UTM)

La UTM prende in input una stringa y, e per prima cosa verifica che y sia della forma code(M)code(x), dove code() e' una codifica, M e' un TM, e x una stringa nell'alfabeto di input $\sigma_i diM$. Se e' cosi', allora la UTM simula l'esecuzione di M siu x.



3.3 Codificare una TM

$M = (\sigma, Q, q_0, H, \delta)$ Introduciamo alcune convezioni. Gli stati in Q sono ordinati come $q_0, q_1, ...$ con q_0 iniziale. Ordiniamo i simboli che possono apparire nella definizione di δ

$$\sigma 0 = \emptyset \quad \sigma 1 = \rightarrow \quad \sigma 2 = \leftarrow$$

e gli altri simboli in σ come $\sigma 4, ..$ Possiamo codificare gli stati ed i simboli come stringhe unarie:

$$code(q_i) = 11...1 \quad code(\sigma_i) = 11...1$$

Codifichiamo una tupla t di δ come:

$$code(t) = code(q_i)0code(\sigma_n)0code(q_j)0code(\sigma_m)0code(\sigma_0)0$$

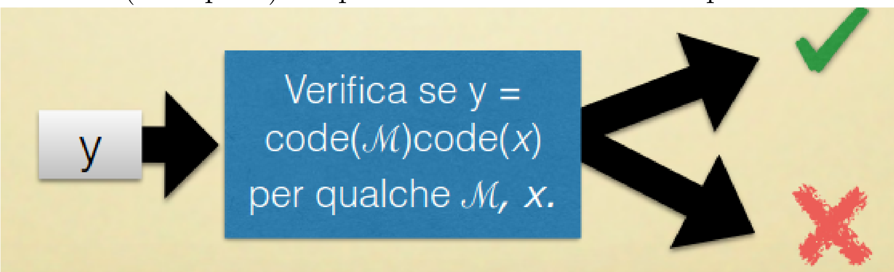
E la funzione di transizione $\delta = t_1, t_2, ..., t_k$ come

$$code(\delta) = code(t_1)0code(t_2)0...0code(t_k)0$$

Possiamo dedurre quali sono gli stati finali di H: sono quelli su cui δ non e' definita (= non occorrono mai in terza posizione in una tupla). INSERIRE ESEMPIO

3.4 Osservazioni sulla codifica

E' possibile che ci siano due o piu' TM che computino la stessa funzione, ma codificate come stringhe differenti (intuitivamente: se esprimono un diverso algoritmo). Nondimeno, la codifica e' *iniettiva*: due macchine differenti saranno codificate da stringhe differenti. Data una stringa su 0,1, e' possibile determinare se sia o meno il codice di una TM (e di quale). In particolare: si tratta di un problema **decidibile**.



3.5 Costruzione di una UTM

La UTM e' definita come una macchina di Turing con tre nastri.

1. Nastro 1 mantiene il nastro di M in forma codificata.
2. Nastro 2 manterra' code(M).
3. Nastro 3 manterra' o stato corrente di M in forma codificata.

Un passo della simulazione di M da parte della UTM funziona nel seguente modo.

1. Cerca in code(M) una tupla $\langle q_i, \sigma_n, q_j, \sigma_m, \sigma_o \rangle$ dove qi concida con lo stato sul nastro 3 e σ_n coincida con il simbolo attualmente esaminato da M.
2. Aggiorna nastro 1 con il nuovo simbolo σ_0 e sposta la testina nella direzione σ_m.
3. Aggiorna nastro 3 con lo stato q_j. Se é finale, fermati.

3.6 Considerazioni finali

Questa costruzione mostra l'esistenza di una macchina di Turing universale, M_U. Niente impedisce a M_U di ricevere la sua stessa codifica code(M_U) come parte dell'input! Questa forma di autoreferenzialita‘ sara‘ utilizzata nella prossima lezione per dimostrare che esiste un problema indecidibile.

4 Cosa non possono fare le TM

Introduciamo il nostro primo problema indecidibile: il problema della **fermata**(*halting problem*). Questo risultato ci informa, più in generale, sui limiti della computazione per algoritmi. Abbiamo visto che l'essere calcolabile da una procedura algoritmica implica essere calcolabile da una TM. Dunque **non** essere calcolabile da una TM implica non essere calcolabile da nessuna procedura algoritmica.

4.1 Ripasso: Linguaggi e TM

Theorem 4.1. *Una TM M* ***decide*** *un linguaggio* ***L*** *se:*

1. *Quando $x \in L$, allora M accetta x(= ferma nello stato Y).*
2. *Quando $x \notin L$, allora M rigetta x(= ferma nello stato N).*

Theorem 4.2. *Una TM M* ***riconosce*** *un linguaggio* ***L*** *se:*

1. *Quando $x \in L$, allora M termina.*
2. *Quando $x \notin L$, allora M non termina.*

4.2 Gradi di (in)calcolabilità

1. Decidibile da una TM se e solo se è calcolabile($\exists$ un algoritmo che risponde "Si" o "No").
2. Non decidibile da nessuna TM ma riconoscibile da una qualche TM se e solo se non è calcolabile(Semi-decidibile: nessun algoritmo saprà calcolare tutte le risposte "No").
3. Non riconoscibile da nessuna TM se e solo se non è calcolabile(del tutto non calcolabile: qualsiasi algoritmo fallirà nel dare sia le risposte "Si" che "No").

4.3 Un problema indecidibile: Halting Problem

Supponiamo che esista un a codifica *code*(−) che presa una TM su alfabeto $\sum$ restituisce le stringhe $x \in \sum^*$. La codifica usata per definire la macchina di Turing universale è un esempio di tale procedura. Definiamo il linguaggio del **problema della fermata**:

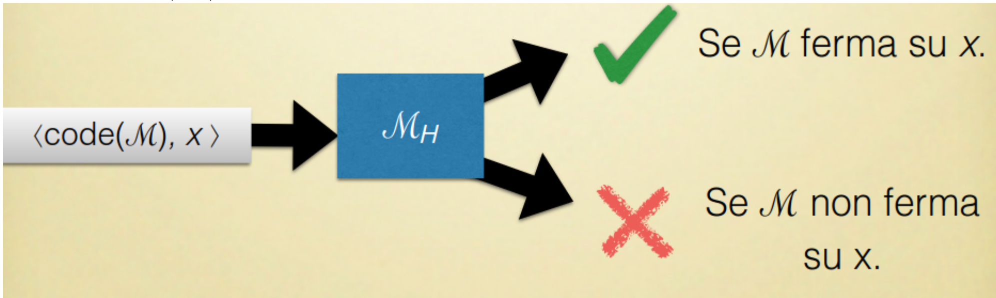
$$\text{HALT} = \{(\text{y,x}) \in \sum^* \text{x} \sum^* \mid \text{y} = \text{code}(M) \; e \; M \text{ ferma su x. } \}$$

Theorem 4.3. *Il problema della fermata è riconoscibile ma non è decidibile.*

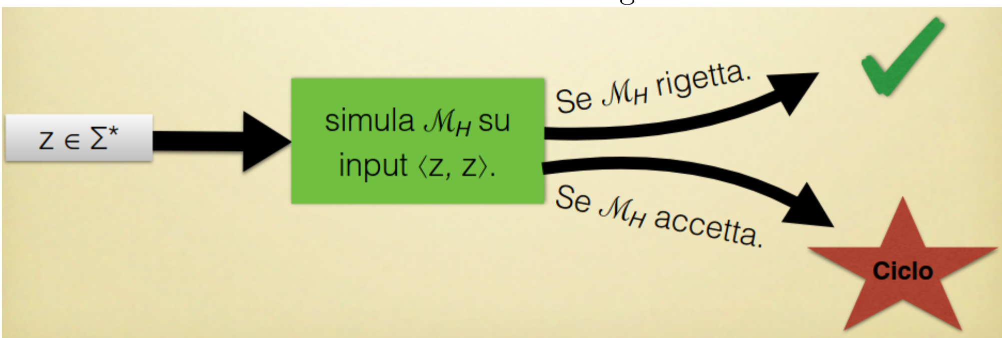
Proof. Dimostriamo che HALT è riconoscibile[1] e che non è decidibile[2].

1. Dimostriamo che HALT è riconoscibile. Dobbiamo costruire una TMU M_H che prende in input una coppia (y, x), se y ferma su x M_H si ferma altrimenti cicla.

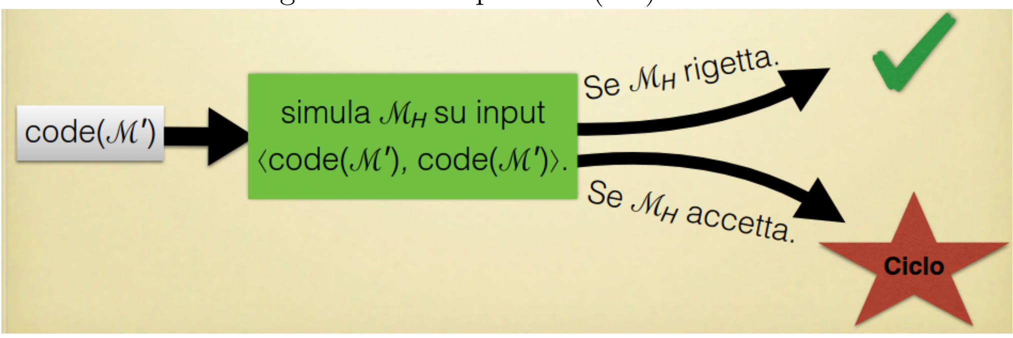
2. dimostriamo che HALT non è decidibile. Assumiamo che HALT sia decidibile, e chiamiamo M_H la TM che decide HALT. Perciò M_H si comporterà come segue. Se $y = code(M)$ per un qualche M:



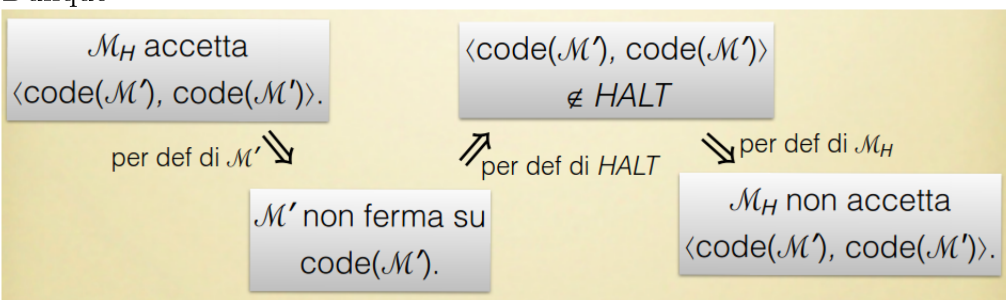
Possiamo definire una nuova TM M' come segue.



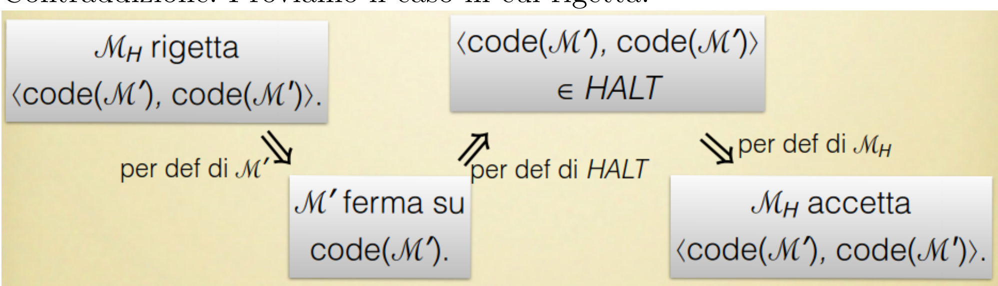
Proviamo ora ad eseguire M' su input $code(M')$.



Dunque



Contraddizione. Proviamo il caso in cui rigetta.



Anche questo caso è in contraddizione. L'unica assunzione utilizzata nel costruire M' è che $\exists M_H$ che decide HALT. Perciò M_H non può esistere: HALT è indecidibile.

□

4.4 Problemi non riconoscibili

Theorem 4.4. *Il complemento $HALT^-$ del problema della fermata non è riconoscibile da nessuna TM.*

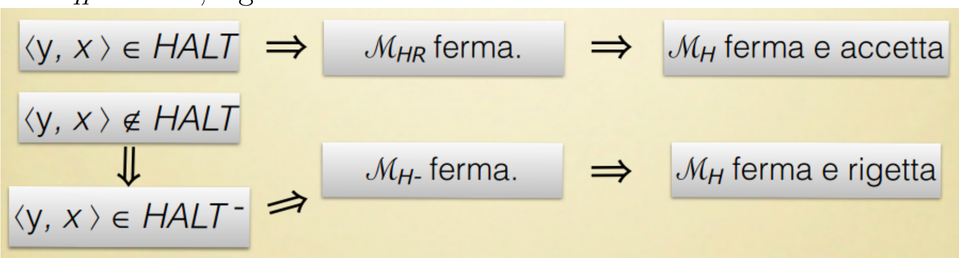
$$HALT = \{\langle y, x\rangle \in \sum^* x \sum^* \mid y{=}code(M) \text{ e } M \text{ ferma su } x.\}$$

$$HALT^- = \{\langle y, x\rangle \in \sum^* x \sum^* \mid y \neq code(M)\ \forall\ M \text{ o } y{=}code(M) \text{ e } M \text{ non ferma su } x\}$$

C'è una dimostrazione diretta, per contaddizione, ma è più interessante mostrare una dimostrazione più astratta. Deriva dal seguente teorema.

Theorem 4.5. *Se $HALT^-$ fosse riconoscibile, allora HALT sarebbe decidibile.*

Proof. Abbiamo già visto che HALT è riconoscibile, diciamo da una TM M_{HR}. Supponiamo per assurdo che anche $HALT^-$ sia riconoscibile, e chiamiamo M_{H^-} la TM che lo riconosce. Possimao ora costruire una TM M_H che decide HALT come segue. Su input $\langle y, x\rangle$, simula M_{HR} e M_{H^-} in parallelo su input $\langle y, x\rangle$. Se M_{HR} ferma, accetta. Se M_{H^-} ferma, rigetta.



Perciò M_H decide HALT. □

Dal momento che HALT è indecidibile, allora $HALT^-$ non può essere riconoscibile.

4.5 Osservazione 1. Complemento linguaggio riconoscibile

La dimostrazione data non sfruttaa in alcun modo il fatto che HALT sia definito nel modo in cui è definito" potremmo sostituire HALT con qualsiasi problema riconoscibile, e funzionerebbe lo stesso. Abbiamo dunque il seguente teorema.

Theorem 4.6. *Se L e L^- sono riconoscibili, allora L è decidibile.*

Proof. La stessa data per $L = HALT$ □

Dai teoremi 4.3 e 4.6 otteniamo il seguente corollario.

Corollary 4.6.1. *I linguaggi riconoscibili non sono chiusi sotto complemento.*

Proof. HALT è riconoscibile ma il suo complemento non è riconoscibile. □

4.6 Osservazione 2. Ridurre un problema ad un altro

La nostra dimostrazione del fatto che $HALT^-$ non sia riconoscibile ha la seguente struttura:

Se potessimo riconoscere L, allora potremmo decidere L'.
Poichè non è decidibile, allora non possiamo riconoscere L.

Come ridurre L a L'? Vedremo come questa intuizione possa essere formalizzata in una tecnica di dimostrazione, che ci permette di ridurre problemi tra di loro al fine di dimostrarne la non calcolabilità.

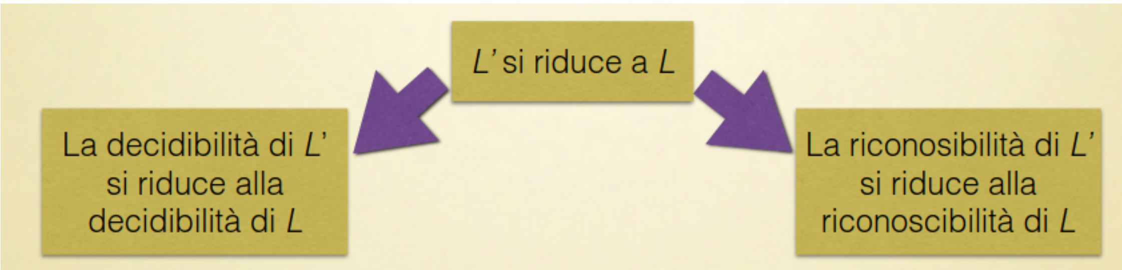
5 Mapping-reduction

Abbiamo dimostrato che

1. il problema della fermata ($HALT$) è indecidibile.
2. Il complemento del problema della fermata ($HALT^-$) non è riconoscibile. Questo può essere dimostrato come conseguenza del punto 1.

Introduciamo dunque una tecnoca semplice e generale per dimostrare l'indecidibilità/non-riconoscibilità di un problema: **mapping-reduction**.
Questa tecnica ci consente di *ridurre* le istanze di un problema a quelle di un altro problema. Nel confrontare due problemi la **decidibilità/riconoscibilità non è essenziale nel processo di prova**. Può essere derivata come una conseguenza.

5.1 Perchè



Rispetto ad un tentativo di investigare la calcolabilità di L o L', questo approccio è maggiormente strutturato: ci consentriamo sulla relazione dei problemi.

5.2 Definizione

Siano L e L' linguaggi sull'alfabeto $\sum$, diciamo che L' è $mapping - riducibile$ a L, scritto $L' \leq L$, se esiste una TM che computa la funzione (totale) $f : \sum^* \to \sum *$ tale che

$$x \in L' \iff f(x) \in L$$

Sostanzialmente, f converte il problema di appartenenza per L' nel problema di appartenenza per L. Intutivamente $L' \leq L$: L è difficile almeno quanto L'.

5.3 Riduzione e Decidibilità

La mapping-reducibility non parla di decidibilità. È però uno strumento efficace per mostrare che un linguaggio è (in)decidibile.

Theorem 5.1. *Se $L' \leq L$ e L è decidibile, allora L' è decidibile.*

Corollary 5.1.1. *Da 5.1 se $L' \leq L$ e L' è indecidibile, allora L è indecidibile. Quindi, per dimostrare che L è indecidibile, è sufficiente mostrare che $HALT \leq L$.*

Corollary 5.1.2. *Da 5.1 se L è decidibile e L' non lo è, allora $L' \nleq L$.*

5.4 Mapping-reduction in azione: ETH

Il problema della fermata su nastro vuoto (ETH) è definito dal linguaggio:

$$ETH = \{x \in \sum^* |x = code(M) \wedge M ferma\ su\ \epsilon\}$$

Theorem 5.2. *ETH è indecidibile.*

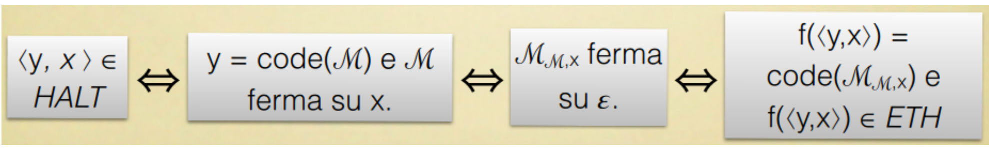
Idea della dimostrazione: ridurre $HALT$ a ETH, così che l'indecibilità di $HALT$ implichi quella di ETH, grazie a 5.1.

Proof. Costruiamo una funzione computabile f tale che:

$$\langle y, x \rangle \in HALT \iff f(\langle y, x \rangle) \in ETH$$

La definizione di f è come segue. Su argomento $\langle y, x \rangle$:

- Se $\forall M.y \neq code(M)$ allora $f(\langle y, x \rangle) = y \neq ETH$.
- Se $y = code(M)$, allora $f(\langle y, x \rangle) = code(M_{M,x})$ è costruita come segue:
  1. $M_{M,x}$ entra in loop su oni stringa non vuota ($\neq \epsilon$).
  2. su input ϵ, scrive x sul nastro e simula M su x.



□

5.5 Il full language problem

Il "full language" problem (FL) è definito dal linguaggio seguente:

$$FL = \{x \in \sum^* |x = code(M) \wedge M ferma\ su\ ogni\ input\}$$

Theorem 5.3. *FL è indecidibile.*

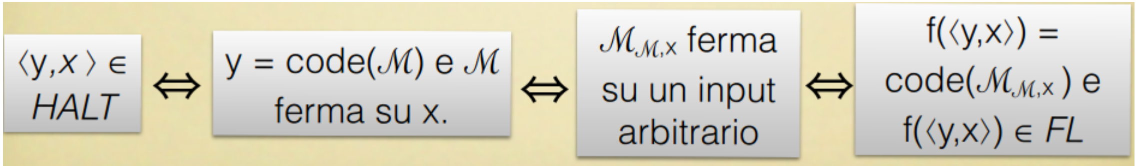
Schema della dimostrazione: Riduciamo $HALT$ a FL.

Proof. Costruiamo una funzione f computabile tale che:

$$\langle y, x \rangle \in HALT \iff f(\langle y, x \rangle) \in FL$$

Definiamo f come segue. Su argomento $\langle y, x \rangle$:

- Se $\forall M.y \neq code(M)$, allora $f(\langle y, x\rangle) = y \in FL$.
- Se $y = code(M)$, allora $f(\langle y, x\rangle) = code(M_{M,x})$, dove $M_{M,x}$ è costruita come segue:
    1. $M_{M,x}$ cancella il suo input.
    2. scrive x sul nastro e simula M su x.



$\square$

5.6 L'equivalence problem

L'**equivalence problem** (EQ) per le TM è definito dal seguente linguaggio:

$$EQ = \{\langle y, x\rangle \in \sum^* \times \sum^* | x = code(M), y = code(M') \wedge f_M = f_{M'}\}$$

Ovvero M e M' computano la stessa funzione parziale.

Theorem 5.4. *EQ è indecidibile.*

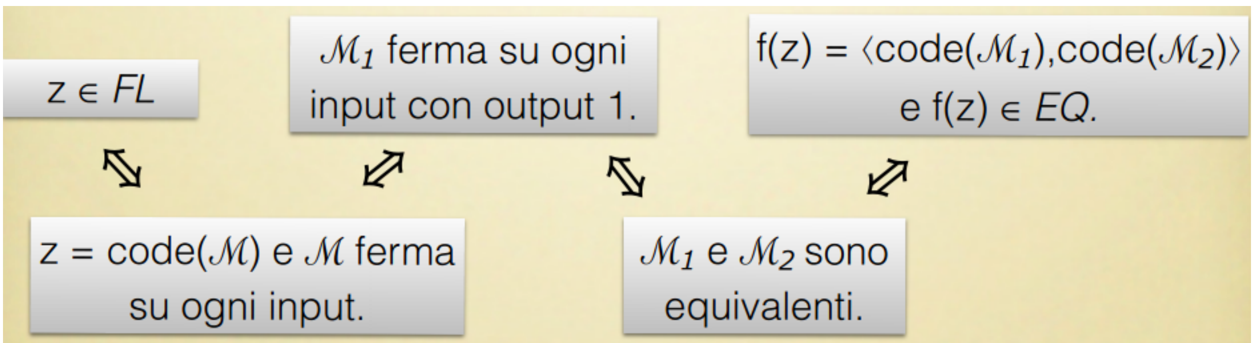
Idea della dimostrazione: Riduciamo FL a EQ.[2]

Proof. È sufficiente costruire una funzione computabile f tale che:

$$z \in FL \iff f(z) \in EQ$$

La definizione di f è come segue. Su argomento z:

- $\forall M.z \neq code(M) \implies f(z) = \langle z, z\rangle \notin EQ$.
- $z = code(M) \implies f(z) = \langle code(M_1), code(M_2)\rangle$, dove M_1 e M_2 sono definite coem segue:
    1. M_1 esegue M sul suo input e restituisce 1 se M si ferma, e va in loop altrimenti.
    2. $M2$ restituisce 1 $\forall input$.



$\square$

[2]Osserva che avremmo potuto usare la riduzione da $HALT$, abbiamo cambiato per mostrare la flessibilità di questo approccio.

5.7 Riduzione e Riconoscibilità

Ciò che è vero per riduzione e decidibilità vale anche per riduzione e riconoscibilità.

Theorem 5.5. *Se $L' \leq L \wedge L$ è riconoscibile $\implies L'$ è riconoscibile.*

Corollary 5.5.1. *Da 5.5 $L' \leq L \wedge L'$ non è riconoscibile $\implies L$ non è riconoscibile.*

Corollary 5.5.2. *Da 5.5 Se L è riconoscibile $\wedge L'$ non lo è $\implies L' \nleq L$.*

5.8 l'equivalence problem non è riconoscibile

Abbiamo dimostrato che EQ è indecidibile. Ora mostriamo che EQ non è riconoscibile. Ricordando che EQ è definito come segue:

$$EQ = \{\langle y, x\rangle \in \sum^* \times \sum^* | x = code(M), y = code(M') \wedge f_M = f_{M'}\}$$

Ovvero M e M' computano la stessa funzione parziale.

Theorem 5.6. *EQ non è riconoscibile.*

Idea della dimostrazione: Riduciamo $HALT^-$ (che sappiamo essere un problema non-riconoscibile) a EQ.

Proof. Ricordiamo la definizione di $HALT^-$:

$HALT^- = \{\langle y, x\rangle \in \sum^* \times \sum^* | \forall M. y \neq code(M) \vee y = code(M) \wedge M$ non ferma su$x\}$

Per la riduzione, abbiamo la funzione f

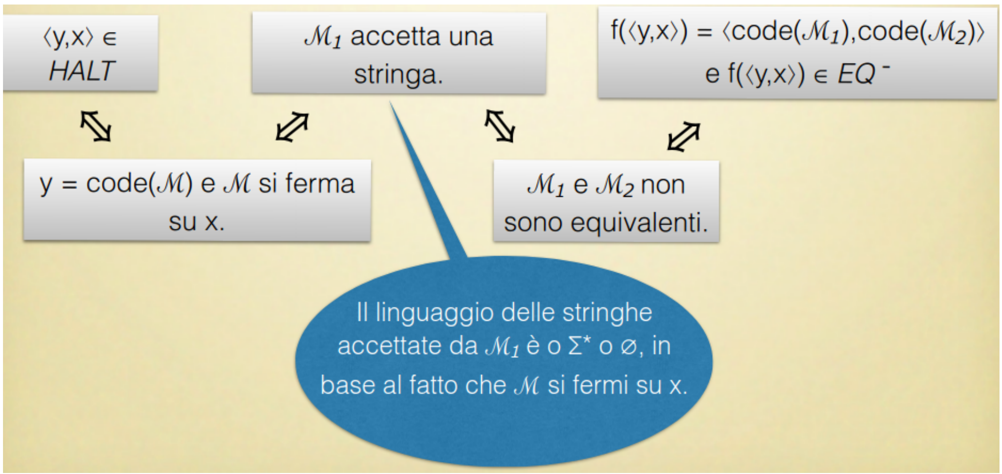
$$\langle y, x\rangle \in HALT^- \iff f(\langle y, x\rangle) \in EQ$$

equivalente a

$$\langle y, x\rangle \in HALT \iff f(\langle y, x\rangle) \in EQ^-$$

Su $\langle y, x\rangle$ definiamo f come segue:

- Se $\forall M. y \neq code(M)$ allora prendiamo M' quialsiasi e consideriamo $f(\langle y, x\rangle) = \langle code(M'), code(M)\rangle$.
- altrimenti $y = code(M)$ e consideriamo $f(\langle y, x\rangle) = \langle code(M_1), code(M_2)\rangle$, dove M_1 e M_2 sono definite come:
  1. M_1 esegue M su x e si ferma se M si ferma, altrimenti entra in un ciclo.
  2. M_2 entra in un ciclo su ogni input.



□

5.9 Ulteriori proprietà della mapping-reduction

Theorem 5.7. *Se $L' \leq L \wedge L$ è decidibile, allora L' è decidibile.*

Corollary 5.7.1. *Da 5.6 se $L' \leq L \wedge L'$ è indecidibile, allora L è indecidibile.*

Corollary 5.7.2. *Da 5.6 se L è decidibile e L' no, allora $L' \nleq L$.*

Se L' è decibile, la riduzione a qualche L è di scarsa utilità.

Theorem 5.8. *Se L è un linguaggio non-triviale ($L \neq \sum^* \wedge L \neq \emptyset$), allora $\forall L'$ decidibile.$L' \leq L$.*

Proof. Poichè L è non-triviale, possiamo prendere $x \in L \wedge y \notin L$. Definiamo la funzione $f : \sum^* \to \sum^*$ come segue:

1. $f(z) = x$ se $z \in L'$

2. $f(z) = y$ altrimenti

Poichè è decidibile, f può essere implementata tramite una TM che simula internamente la TM per L' e restituisce x o y coerentemente. Inoltre, per definizione di f:

$$z \in L' \iff f(z) \in L$$

5.10 Riduzione come Relazione

La riduzione è una relazione tra linguaggi.

1. È **riflessiva**: $\forall L.L \leq L'$.

2. È **transitiva**" $L' \leq L \wedge L \leq L'' \implies L' \leq L''$.

3. Non è però **simmetrica** (sarebbe simmetrica se $L' \leq L \implies L \leq L'$)[3]

Quindi, la riduzione non è una relazione di equivalenza. □

5.11 Riduzione e Complemento

Theorem 5.9. $L_1 \leq L_2 \iff L_1^- \leq L_2^-$.

Corollary 5.9.1. *Da 5.8* $L_1^- \leq L_2 \iff L_1 \leq L_2^-$.[4]

Mettendo insieme questi risultati, abbiamo senza ulteriore lavoro una prova che EQ^- non è riconoscibile. Infatti, $HALT \leq FL \leq EQ$, quindi $HALT^- \leq EQ^-$ per teorema. La non-riconoscibilità di $HALT^-$ implica non-riconoscibilità di EQ^-.

5.12 Gerarchia dei problemi

1. Decidibile.

2. Indecidibile, ma riconoscibile (es. $HALT$).

3. Non riconoscibile, ma il cui complemento è riconoscibile (es. $HALT^-$).

4. Non riconoscibile, e non il cui complemento non è riconoscibile (es. EQ).

[3]Il problema della fermata è un controesempio, per per 5.7 $\forall L'$ decidibile, $L' \leq HALT$, se anche $HALT \leq L'$, allora anche $HALT$ sarebbe decidibile per 5.6. Contraddizione.

[4]Usato implicitamente nella prova di 5.6 con $L_1 = HALT$ e $L_2 = EQ$

5.13 Turing-riducibilità

La mapping-reducibility è solo uno dei modi in cui definire il concetto di problema riducibile a un altro problema. Un altro è la Turing-riducibilità. Un **oracolo** per il linguaggio L è uno strumento esterno ("black box") capace di rispondere alla domanda "$x \in L?$"$\forall x$.
Un linguiaggio L' è **Turning-riducibile** a L se, dato un oracolo per L, possiamo decodere L'.

$$mapping - riducibility \implies Turing - riducibility.$$

Ad esempio sappiamo che $HALT^- \not\leq HALT$, cioè $HALT^-$ non è mapping-riducibile a $HALT$. MA $HALT^-$ è Turing-riducibile ad $HALT$. Infatti, se abbiamo un oracolo per la domanda "$\langle y, x \rangle \in HALT?$", questo può essere usato per decidere se $\langle y, x \rangle \in HALT^-$.

6 La cardinalitá dei problemi irrisolvibili

6.1 Obiettivo

Vogliamo mostrare cjhe la maggior parte dei linguaggi non sono riconoscibili (e, quindi indecidibili). Il nostro argomento consiste nel mostrare che ci sono "*molti*" più linguaggi che TM.

6.2 Insiemi infiniti numerabili

Un insieme S è **infinito numerabile** se c'è una funzione totale biettiva $f : \mathbb{N} \to S$. Ad esempio: l'insieme dei numeri dispari D con la funzione $f : \mathbb{N} \to D$ definita come $f(n) = 2n - 1$.

Lemma 6.1. *Se S_1 e S_2 sono infiniti numerabili, $S_1 \cup S_2$ è infinito numerabile.*

Ad esempio prendiamo l'insieme $\sum^*$ di stringhe su alfabeto finito $\sum$.
Assumi $|\sum| = n$, la biezione con $\mathbb{N}$ è costruita come:

$$f : \mathbb{N} \to \sum^*$$

1. $f(\epsilon) = 1$
2. $\forall i \in \{1..n\}.\ f(\sigma_i) = i + 1$
3. $\forall i \in \{1..n\} \forall j \in \{1..n\}.\ f(\sigma_i \sigma_j) = n \times i + i + j$
4. ecc. per stringhe con lunghezza > 2.

Abbiamo visto che ogni TM può essere codificata come una stringa per un alfabeto $\sum$ con $|\sum| = 2$ (es. $\sum = \{0, 1\}$).
Allora, l' **insieme di tutte le TM** èe inifinito numerabile. Anche l' **insieme di tutti i linguaggi riconoscibili** è infinito numerabile. Questo perchè, per definizione, un linguaggio è riconoscibile se c;è una TM che lo riconosce. Per la stessa ragione, l' **insieme delle funzioni** $\mathbb{N} \to \mathbb{N}$ *computabili* da una TM è infonoto numerabile.

7 Linguaggi sono non numerabili

Sia $S_{\sum}$ l'insieme di tutti i linguaggi sull'alfabeto finito $\sum$.

Theorem 7.1. *L;insieme $S_{\sum}$ non è numerabile.*

Proof. Ricorda che un linguaggio L è un sottoinsieme di $\sum^*$. Abbiamo già visto che $\sum^*$ è infinito numerabile, quindi possiamo scriverlo come $\sum^* = \{\sigma_1, \sigma_2, \sigma_3, ...\}$. Allora un linguaggio, diciamo $L_1 = \{\sigma_1, \sigma_4\}$, può essere rappresentato come una riga in una tabella:

	σ_1	σ_2	σ_3	σ_4	σ_5	....
L_1	1	0	0	1	0	....

□

Ciascun linguaggio su $\sum$ può essere rappresentato in questo modo. Per contaddizione, assumi che $S_{\sum}$ sia un insieme numerabile. Allora possiamo assegnare un numero naturale ai suoi elementi, così che ogni $L_i \in S_{\sum}$ appare come riga a destra.

	σ_1	σ_2	σ_3	σ_4	σ_5	....
L_1	1	0	0	1	0	....
L_2	0	1	1	0	1	....
L_3	0	0	0	0	0	....
L_4	1	1	1	0	1	....
L_5	1	1	1	1	1	....
⋮	⋮	⋮	⋮	⋮	⋮	⋱

Davvero ogni elemento L di $S_{\sum}$ compare su una riga?
Definisci L come 00110..., allora $\sigma_i \in L \iff \sigma_i \notin L_i$. Quindi L è diverso da ogni linguaggio L_i sulla riga. Dunque L non può essere in una riga! **Contraddizione**.
Quindi, $S_{\sum}$ non è numerabile.

	σ_1	σ_2	σ_3	σ_4	σ_5	$\cdots$
L_1	1	0	0	1	0	$\cdots$
L_2	0	1	1	0	1	$\cdots$
L_3	0	0	0	0	0	$\cdots$
L_4	1	1	1	0	1	$\cdots$
L_5	1	1	1	1	1	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

7.1 Riassumendo

Dato un alfabeto finito $\sum$, abbiamo visto che:

1. l'insieme di linguaggi riconoscibili da una TM è infinito numerabile.
2. l'insieme di tutti i linguaggi è non-numerabile.

Quindi, esistono limguaggi che non sono riconoscibili da alcuna TM, ad esempio $HALT^-, EQ, EQ^-$.

7.1.1 Quanti linguaggi non riconoscibili ci sono?

La risposta deriva da un un risultato generale.

Theorem 7.2. *Se S è un insieme infinito, non-numerabile e S' è un sottoinsieme infinito numerabile di S, allora $S \backslash S'$ non è un inifinito numerabile.*

Proof. Assumi $S \backslash S'$ sia infinito numerabile. Allora, poichè i linguaggi infiniti numerabili sono chiusi per unione, $(S \backslash S') \cup S' = S$ è numerabile. Contraddizione! □

Dal 7.2 otteniamo il seguente corollario.

Corollary 7.2.1. *L'insieme di linguaggi non riconoscibili non è infinito numerabile, allora ci sono più linguaggi non riconoscibili che riconoscibili.*

8 Teorema di Rice

Abbiamo visto che non possiamo decidere se una TM:

1. ferma su un dato input
2. ferma su input vuoto (stringa vuota)
3. è equivalente a unàltra TM.

Allora, quali problemi riguardanti le TM sono decidibili? Per esempio:

1. possimo verificare quanti stati ha una macchina, e desumere quanti stati ha
2. se va mai a dx o a sx

Cos'altro?

8.1 ripasso: linguaggi

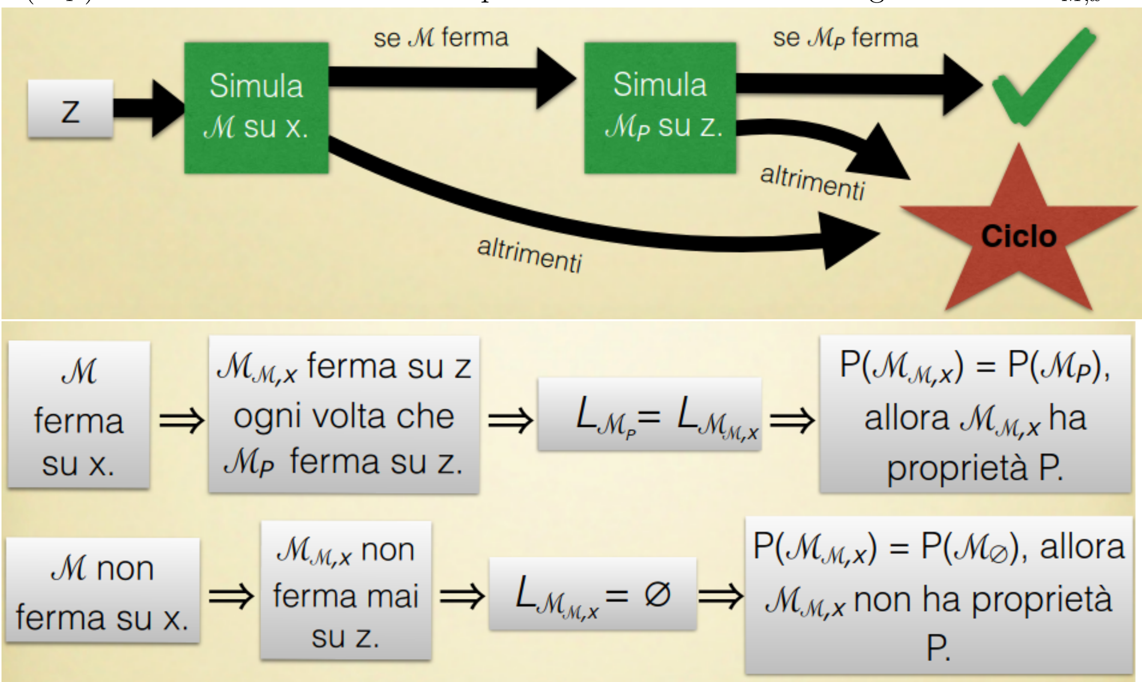
Una **proprietà di linguaggio** P è una funzione da un insieme di TM a 0, 1 (falso/vero), tale che $L_M = L_{M'}$ implica P(M) = P(M'). Questo assicura che P dipenda solo dal linguaggio descritto dalla macchina. Per esempio: "ferma in 42 step" non è proprietà del linguaggio. Questa proprietà è **non-triviale** se esiste una TM M tale che P(M) - 1 e una TM M' tale che P(M')=0. Formalmente, identificheremo le TM che soddisfano la proprietà P con lìnsieme:

$$\{y \in \sum^* | y = code(M) \ e \ P(M) = 1\}$$

8.2 Teorema di Rice

Theorem 8.1. *Se P è una proprietà di lunguaggio non triviale, allora il problema "M ha proprietà P" è indecidibile.*

Proof. Per contraddizione dimostraiamo che se "M ha proprietà P" fosse decidibile, allora il problema della fermata sarebbe decidibile.
Considera una prpprietà P. Assumiamo P($M_\emptyset$) = 0. ($M_\emptyset$ è una TM che riconosce il linuaggio vuoto.) Poichè P è non-triviale, possiamo considerare una TM M_P tale che P(M_P) = 1. Fissiamo M e x come parametri e costruiamo la seguente TM $M_{M,x}$:



Se potessimo decidere se $M_{M,x}$ ha la proprietà P, potremmo decidere il problema della fermata. Allora, $\{y | y = code(M) e P(M) = 1\}$ è indecidibile.
Abbiamo assunto P($M_\emptyset$)=0. Se P($M_\emptyset$)=1? In questo caso, ripetiamo lo stesso argomento, ma per la proprietà $\neg P$ ("M non ha la proprietà P"). Osserva che questo funziona perchè:

1. dato che P è non-triviale, anche $\neg P$ è non-triviale.
2. dato che P($M_\emptyset$) = 1, allora $\neg P(M_\emptyset) = 1\}$ è indecidibile.

Concludiamo che $\{y | y = code(M) \wedge \neg P(M) = 1\}$ è indecidibile. Questo implica che anche $\{y | y = code(M) \wedge P(M) = 1\}$ sia indecidibile. □

NB: lo schema di dimostrazione è:

- Devo dimostrare $A \iff B$
- Dimostriamo $A \Rightarrow B \land \neg A \Rightarrow \neg B$
- Che equivale a $A \Rightarrow B \land B \Rightarrow A$

8.3 Proprità decidibili

Il Teorema di Rice riguarda proprietà di **linguaggio**, non proprietà algoritmiche; riguarda funzioni (specifiche), non programmi (implementazioni). Per esempio non possiamo usare il teorema di Rice per derivare l'indecidibilità del problema della fermata (e simili). In geerale, ci sono tre tipi di proprietà riguardo le TM:

1. **Proprietà di linguaggio**: Quelle non triviali sono indifinibili (teorema di Rice).
2. **Proprietà strutturali**: Queste sono tipicamente decidibili poichè si possono verificare staticamente sulla (codifica della) descrizione di TM. (es. "*M ha* 13 *stati*").
3. **Proprietà comportamentali (o algoritmiche)**: Alcune sono decidibili, altre no, e la classificazione non è ovvia (es. "*M non si mouve a sinistra su input* 0101").

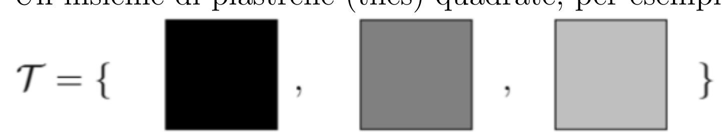
9 Tiling

Mostriamo ora un esmpio di problema non calcolabile in un contesto diverso dalla teoria della computabilità. L'obiettivo generale è quello di mostrare che la non-calcolabilità è un fenomeno **pervasivo**, presente in diverse discipline e problemi.
Dato un insieme di piastrelle e di regole per metterle insieme, esiste una disposizione del piano che rispetti tali regole?

9.1 Sistema di Tiling

Un sistema di tiling è costituito da:

1. Un insieme di piastrelle (tiles) quadrate, per esempio



2. Un elemento scelto $t_0 \in \tau$ detto piastrella d'origine.
3. Un insieme di *regolediadiacenza*, che specificano quali piastrelle possano essere posate le une accanto alle altre.

9.2 Tiling

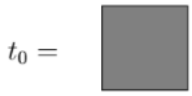
Il **tiling** è una disposizione delle piastrelle in τ con le seguenti proprità:

1. t_0 si trova nell'angolo in basso a sinistra.
2. Ogni piastrella ha una piastrella sopra e una disposta alla sua destra, senza spazi intermedi.
3. Tutte le regole di adiacenza dono rispettate.

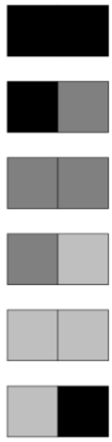
9.2.1 Esempio

L'insieme di piastrelle

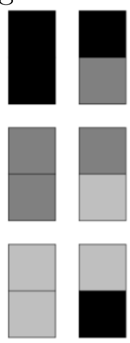
la piastrella d'origine



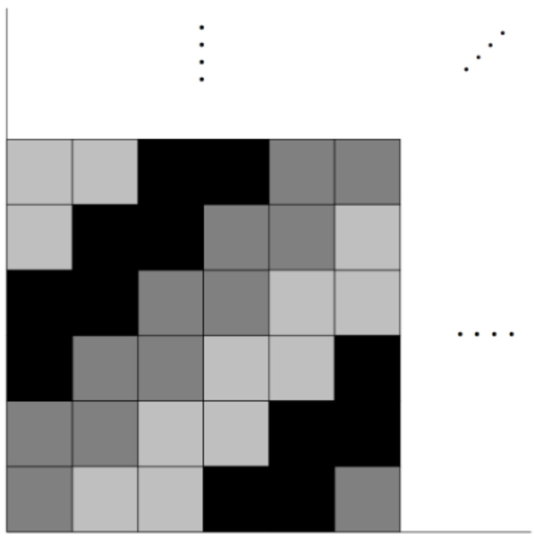
Regole Orizzontali



Regole Verticali



Un tiling per questo sistema è per esempio:



9.3 Il problema del tiling

Vogliamo analizzare il **problema del tiling** (tiling problem):

Domanda: *dato un sistema di tiling, esiste un tiling?*[5]
Risposta: *No.*[6]

Per dimostrare questo, abbiamo bidogno di una definizione formale dei dati del problema.

9.4 Sistema di tiling, formalmente

Un **sistema di tiling** è una tupla $\langle \tau, t_0, H, V \rangle$ tale che:

1. τ è un insieme di piastrelle.
2. $t_0 \in \tau$ è la piastrella d'origine.
3. $H \subset \tau \times \tau$ è un insieme di regole di adiacenza orizzontali e $V \subset \tau \times \tau$ è un insieme di regole di adiacenza verticali.

Assumiamo il quadrante positivo del piano sia diviso in celle identificate dalle proprie coordinate. Il **tiling** è una funzione $f : \mathbb{N} \times \mathbb{N} \to \tau$ tale che:

1. $f(1, 1) = t_0$
2. $\forall n, m \in \mathbb{N}.(f(n, m), f(n, m + 1)) \in V$
3. $\forall n, m \in \mathbb{N}.(f(n, m), f(n + 1, m)) \in H$

9.5 Problemi di tiling

Mostreremo che il tiling problem è irriconoscibile mostrando che il complemento di ETH si riduce a esso.

$$ETH = \{x \in \sum^* | x = code(M) \wedge M \ ferma \ su \ \epsilon\}$$
$$ETH = \{x \in \sum^* | \forall M.x \neq code(M) \vee x = code(M) \wedge M \ non \ ferma \ su \ \epsilon\}$$

Abbiamo visto che ETH è indecidibile ma riconoscibile. Quindi come per il problema della fermata e il suo complemento, ETH^- deve essere non riconoscibile: altrimenti ETH sarebbe in realtà decidibile. Perciò, ridurre ETH^- al tiling problem implica che il tiling problem non sia riconoscibile da nessuna TM. La nostra strategia per dimostrare che la non-riconoscibilità del tiling problem è la seguente:

1. Mostriamo che ogni TM M può essere trasformata in un sistema di tiling τ_M.
2. Questa trasformazione è fatta in modo tale per cui

$$code(M) \in ETH \iff \nexists \text{ un tiling per } \tau_M$$

Tale corrispondenza riduce ETH al complemento del tiling problem, così che ETH^- si riduce al tiling problem.

[5] 1961, Hao Wang.
[6] 1966, Robert Berger

10 Teoria della complessità computazionale

10.1 Misurare la complessità

$$A = \{0^k 1^k | k \geq 0\}$$

Programma della TM che decide A:

1. Leggi l'input e rigetta se trovi uno 0 a destra di un 1.[7]
2. Fino a che ci sono sia valori 0 che valori 1 sul nastro: scansiona il nastro eliminando un singolo 0 e un singolo 1.[8]
3. Se rimangono ancora valori 0 o 1, rigetta. Se invece il nastro è vuoto, accetta.[9]

Supponi che l'input abbia lunghezza n. Quanti passi di computazione richiede l'algoritmo?

10.1.1 Complessità di tempo

Sia M una TM che si ferma su tutti gli input. La sua **complessità di tempo** è definita come la funzione $f : \mathbb{N} \to \mathbb{N}$, dove $f(n)$ è il massimo numero di passi che M impiega a fermarsi su arbitrari input di lunghezza n.
Ad esmpio, la TM appena vista ha complessità: $n + (n/2 \times N) + n$
Quanto è robusto questo metodo di misurare la complessità di un algoritmo?

1. Dipende dal modello di calcolo.
2. Dipende da cosa intendiamo per 'passi' di computazione.
3. È sensibile a variazioni minime (ad es. una sub-routine che richiede un tempo costante, indipendente dall'input).
4. Dipende dalla codifica dell'input e come misuriamo la sua lunghezza.

10.1.2 Notazione Big-O

Un modo più efficace di esprimere la complessità di un algoritmo è usando una notazione che astrae dettagli implementativi poco rilevanti, e fornisce solo una **stime** significativa del suo tempo di esecuzione.

date funzioni $f, g : \mathbb{N} \to \mathbb{N}$ scriviamo $f(n) = O(g(n))$ e diciamo che $g(n)$ è un **bound superiore asintotico** se esistono $c \in \mathbb{N}$, e $m \in \mathbb{N}$ tali che $\forall n. n \geq m. f(n) \leq c \times g(n)$
In altre parole, $g(n)$ è sempre grande almeno quanto $f(n)$ per n sufficientemente grandi e modulo un fattore costante c.

10.1.3 Classi di complessità

Data una funzione $f : \mathbb{N} \to \mathbb{N}$, definiamo **la classe di complessità**(di tempo) **TIME(t(n))** come la collezione di tutti i linguaggi decidibili da una TM (deterministica, a un nastro) in te,mpo $O(t(n))$.

Ad esempio il linguaggio A è nella classe $TIME(n^2)$.

[7] n passi.
[8] al più $n/2 \times n$ passi.
[9] al più n passi.

10.2 La classe P

P è la classe dei linguaggi decidibili da una TM (deterministica, a un nastro) in tempo **polinomiale**. Ovvero:
$P = \bigcup_k TIME(n^k)$

10.2.1 Perchè P?

Nella pratica dello sviluppo software, un algoritmo che lavora in tempo polinomiale viene solitamente consoderato "ragionevole".
Il bound polinomiale può essere in realtà molto alto (es. $O(n^{10})$), ma l'esperienza ci ha rivbelato che, qualora un algoritmo polinomiale è conosciuto, è solitamente possibile renderlo più efficiente.
Un altro aspetto riguarda la codifica, la maggior parte delle codifiche per strutture come grafi, alberi, matrici, automi, ecc. come stringhe richiede tempo polinomiale, e produce una stringa di output di lunghezza polinomiale rispetto all'input.

10.2.2 Tesi di Church-Turing rafforzata

Ogni modello di calcolo deterministico fisicamente realizzabile può essere simulato da una TM (deterministica, su nastro singolo) con overhead al più polinomiale.
M passi di computazione del modello in questione possono essere simulati dalla TM in al più $O(m^c)$ passi per una costante c.
Se vera, la tesi asserisce che la classe P è robusta, nel senso di essere invariante rispetto al modello di computazione deterministico scelto.

10.3 La classe EXP

P è la classe dei linguaggi decidibili in un tempo "ragionevole".

$$EXP = \bigcup_k TIME(2^{n^k})$$

EXP è la classe dei linguaggi decidibili in un tempo "irragionevole".
La classe EXP intutivamente è propria degli algoritmi che eseguono un'analisi brute-force, esplorando lo spazio di tutte le possibili soluzioni a un problema.

10.4 Esempio: PATH

$PATH = \{\langle G, s, t\rangle | \ G \ grafo \ diretto \ che \ ha \ un \ percorso \ diretto \ da \ s \ a \ t.\}$

Theorem 10.1. *PATH è in P.*

Proof. Considera il seguente algoritmo: Su input $\langle G, s, t\rangle$:

1. Contrassegna il node s.
2. Ripeti fino a che nessuno nuovo nodo è contrassegnato: [scansiona tutti gli archi di G. Se trovi un arco da nodo a contrassegnato ad uno b non contrassegnato, contrassegna il nodo b.]
3. Se t è contrassegnato, accetta. Se no, rigetta.

Analizziamo la complessità: 1 e 3 sono chiaramente in P. La subroutine di 2 è in P ed è ripetuta al massimo per un numero di nodi di G, quindi 2 nel suo compleso è in P. Perciò l'algoritmo è in P. □

10.5 Una nota sulla codifica

Abbiamo dimostrato che $PATH$ è in P assumendo che la complessità venga rispetto al numero di nodi nel grafo. Tuttavia una TM non lavora direttamente su un grafo ma su una sua codifica come stringa di un alfabeto. Per esempio, possiamo codificare un grafo come una matrice di adiacenza, e una matrice di adiacenza come un numero reale.
In che modo possiamo essere certi che questo non influenzi la nostra analisi della complessità di PATH? Perchè ciascuna di queste codifiche impiega tempo al più polinomiale, e modifica la dimensione di questo input di un fattore al più polinomiale.

10.6 Altri problemi in P: panoramica

1. n è primo? Sì.
2. Il grafo G è connesso? Sì.
3. Data una regex r e la stringa s, $s \in L(r)$? Probabilmente no (PSPACE-completo).
4. Problema del commesso viaggiatore? Probabilmente no (NP-completo).
5. Dalla posizione P in una fila di s cacchi, quale giocatore ha una strategia vincente? Probabilmente no (PSPACE-completo).

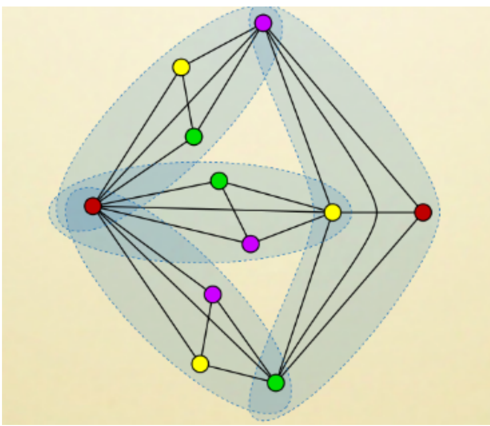
10.7 La classe NP

Data una funzione $f : \mathbb{N} \to \mathbb{N}$, definiamo **la classe di complessità** (di tempo) **NTIME(t(n))** come la collezione di tutti i linguaggi decidibili da una TM (non-deterministica, a un nastro) in tempo O(t(n)).

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

10.8 Esempio: CLIQUE

Dato un grafo indiretto, un clique è un sotto-grafo dove tutti i nodi sono collegati tra loro da un arco.



$$CLIQUE = \{\langle G, k\rangle | Il\ grafo\ G\ contiene\ un\ CLIQUE\ di\ k\ nodi.\}$$

CLIQUE è in NP. Ecco una algoritmo non-deterministico polinomiale che lo calcola.
Su input $\langle G, k\rangle$

1. Seleziona non-deterministicamente un sottoinsieme C di k nodi di G.
2. Verifica che G colleghi tutti i nodi di C tramite archi. Se si, accetta. Se no, rigetta.

10.9 Una caratterizzazione alternativa di NP

Un linguaggio A è **verificabile** se esiste una TM M (che termina sempre, ed accetta o rigetta) con la proprietà:

$$w \in A \iff \exists c \text{ t.c. } M \text{ accetta } \langle w, c \rangle$$

Se M lavora in tempo polinomiale, diciamo che A è **verificabile polinomialmente**. Intuitivamente, c è un certificato del fatto che $w \in A$. Nota che, se M lavora in tempo polinomiale, può accedere un certificato di lunghezza al più polinomiale in w.

Theorem 10.2. *Un linguaggio è in NP se e solo se è verificabile polinomialmente.*

Idea della dimostrazione.

Proof. In una direzione, sia M una TM non-deterministica che decide un linguaggio L in tempo polinomiale. Costruiamo un verificatore V polinomiale per L nel seguente modo: Su input $\langle w, c \rangle$:

1. Simula M su w, trattiamo c come descrizione codificata delle scelte non-deterministiche da prendere nell'albero di computazione di M.
2. Se il ramo di computazione esplorato è accettante, accetta.

Nella direzione opposta, sia V un verificatore polinomiale per L. Costruiamo una TM M non-deterministica che decide L nel seguente modo: Su input w di lunghezza n:

1. Seleziona non-deterministicamente una stringa c di lunghezza al più n^k.
2. Simula V su input $\langle w, c \rangle$.
3. Se V accetta, accetta. Se no, rigetta.

□

10.10 P vs NP

1. P è la classe dei linguaggi A per cui la domanda "$w \in A$?" può essere **risposta** in maniera efficiente.
2. Np è la classe dei linguaggi A per cui la correttezza di una risposta alla domanda "$w \in A$?" può essere **verificata** in maniera efficiente.

10.10.1 Perchè Knuth propende per P=NP?

1. Gli informatici hanno cercato di dimostrare che P è diverso da NP per lungo tempo, senza successo. Questo é un indizio che il contrario (P = NP) é vero.
2. D'altro canto, altrettante energie sono state impiegate per trovare algoritmi polinomiali per problemi NP, senza successo. É questo un indizio che P é diverso da NP? Secondo Knuth no: questa strategia potrebbe essere sbagliata, un vicolo cieco.
3. Infatti, la dimostrazione che P = NP potrebbe essere non-costruttiva. Dato un problema in NP, un algoritmo polinomiale potrebbe esistere, ma essere incredibilmente complicato, fuori dalla portata umana. Ad esempio, potrebbe essere di complessità O(ng), dove g é il numero di Graham.

4. C'è un precedente significativo per quanto sostiene Knuth: il teorema di Roberston-Seymour, il quale dimostra che un certo problema in teoria dei grafi é in P...ma non dà l'algoritmo polinomiale in sé (e non é chiaro come potremmo derivarlo in maniera esplicita).

5. In altre parole, secondo Knuth, anche qualora dimostrassimo che P = NP, la dimostrazione sarebbe quasi sicuramente non-costruttiva, e di nessuna utilità pratica.

Perciò, in conclusione, Knuth ritiene che P vs NP non sia un problema così interessante, o quantomeno è malposto.

11 Poly reduction

Siano L e L' linguaggi sull'alfabeto $\sum$ Diciamo che L è **poly (mapping-)riducibile** a L, scritto $L' \leq_p L$, se esiste una TM che computa in tempo polinomiale una funzione (totale) $f : \sum^* \to \sum^*$ tale che

$$x \in L' \iff f(x) \in L$$

In altre parole, $L' \leq_p L$ se $L' \leq L$ e la riduzione che lo testimonia è computabile in tempo polinomiale.

11.1 Poly reduction e P

Theorem 11.1. *Se $L' \leq_p L$ e $L \in P$, allora $L' \in P$.*

Proof. Da aggiungere. Fatta in classe alla lavagna □

Theorem 11.2. *Per $L \in P$, $L^- \leq_p L$.*

Proof. Da aggiungere alla lavagna. $L^- \leq_p L$ $x \in L^- \iff f(x) \in L$ $x \notin L \iff f(x) \in L$ Problema cos'è f? f prende x e calcola se x appartiene ad L (in tempo poly) se sì ritorna b non in L , altrimenti a in L. (da correggere) □

Domanda: Se $L \in NP$, $L^- \leq_p L$?

11.2 Formule e soddisfacibilità

Una formula booleana è costuita a partire da variabili $x, y, z, ...$, loro negazione $\neg x, \neg y, \neg z, ...$ (chiamati collettivamente letterali) e combinazioni di letterali tramite congiunzione e disgiunzione $(\wedge, \vee)$. Le variabili possono ricevere valore vero (1) o falso (0), da cui deriviamo il valore di verità dell'intera formula.
Esempio: $(\neg x \vee y) \wedge (x \vee z)$.
Una formula è **soddisfacibile** se esiste un assegnamento di valore alle sue variabili che le dia valore 1. La formula di esempio è soddisfacibile, come testimoniato dall'assegnamento $x = 1, y = 1, z = 0$.

11.2.1 3cnf

Una formula booleana è una **clausula** se è una disgiunzione di variabili (positive o negate).

$$\neg x \vee y \vee z \vee \neg z$$

Una formula booleana è in **forma normale congiunta (cnf)** se e una congiunzione di clausole.

$$(x \vee y) \wedge (\neg z \vee z) \wedge (\neg z \vee y \vee z \vee \neg z)$$

Una formula booleana è **3cnf** se è in cnf e ogni sua clausula contiene esattamente tre letterali.

11.2.2 3SAT e SAT

$$SAT = \{\langle F \rangle | F \ formula\ booleana\ soddisfacibile\}$$
$$3SAT = \{\langle F \rangle | F \ formula\ booleana\ 3cnf\ soddisfacibile\}$$

Perchè sono importanti?

1. ragioni storiche.
2. Ragioni pratiche: molti problemi informatici possono essere formulati come istanze di SAT e 3SAT, ad esmpio constraint programming.

11.3 3SAT e CLIQUE

Dimostreremo che $3SAT \leq_p CLIQUE$. La riduzione è interessante perchè collega problemi all'apparenza molto diversi (uno sulle formule logiche e l'altro sui grafi).

Theorem 11.3. *$3SAT \leq_p CLIQUE$*

Proof. L'idea della dimostrazione è tradurre formule in grafi, dove $f(\langle F \rangle)$ sarà costruito in modo tale da 'mimare' il comportamento di variabili e clausole. Un clique in $f(\langle F \rangle)$ corrisponderà ad un assegnamento che soddisfa $\langle F \rangle$.

$$\langle F \rangle \in 3SAT \iff f(\langle F \rangle) \in CLIQUE$$

Sia F una formula con k clausole definita come:

$$F = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2)...$$

Definiamo $f(\langle F \rangle) = \langle G, k \rangle$, dove G è il seguente grafo:

- **Nodi**: G ha 3k nodi, suddivisi in k gruppi $t_1, ...t_k$ chiamati triple. L'idea è che ogni tripla corrisponde ad una clausula di F: perciò etichettiamo ciascun nodo di G con il letterale corrispondente in F.
- **Archi**: gli archi di G collegano tutti le coppie (n_1, n_2) di nodi tra di loro, eccetto (I) se n_1 e n_2 sono nella stessa tripla, oppure (II) n_1 ha etichetta x e n_2 ha etichetta $\overline{x}$ per qualche variabile x.

Dimostraimo che questa costruzione soddisfa l'equivalenza.

$$\langle F \rangle \in 3SAT \rightarrow f(\langle F \rangle) \in CLIQUE$$

Se F è soddisfacibile, almeno un letterale per ogni clausola è vero. Selezioniamo il nodo corrispondente nel grafo G: avremo così selezionato un nodo in ciascuna tripla. Questa selezione ci da un clique di grandezza k: notiamo infatti che (1) ci sono k triple e (2) ogni nodo selezionato è collegato agli altri della selezione, per virtù delle regole di costruzione di G.

$$\langle F \rangle \in 3SAT \leftarrow f(\langle F \rangle) \in CLIQUE$$

Se $f(\langle F \rangle) = \langle G, k \rangle$ e G contiene un clique S di k nodi, definiamo un assegnamento A di valori di verità alle variabili in F dando semplicemente valore vero ad ogni letterale che etichetta un nodo in S. Per costruzione di G, nota che (1) A non è contraddittoria, perchè nodi etichettati con *non* sono collegati e perciò non possono far parte di S; (2) ogni nodo di S appartiene ad una tripla diversa rispetto agli altri in S, perciò A assegna valore vero esattamente ad un letterale per clausola. Perciò A rende vera F. □

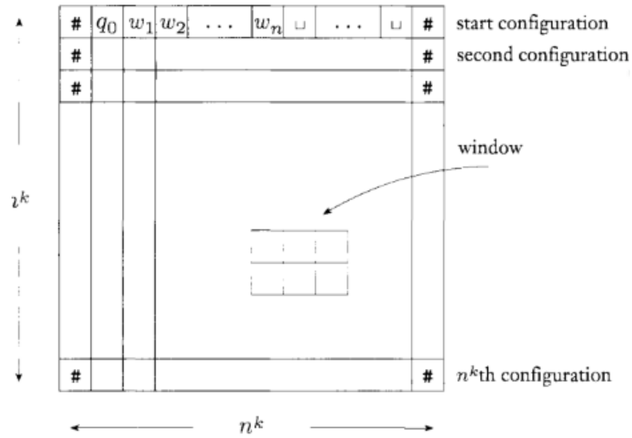
11.4 NP-completezza

Un linguaggio L è NP-completo se è in NP e ogni altro linguaggio L' in NP è poly-riducibile ad esso ($L' \leq_p L$).
Esistono linguaggi NP-completi?
$TMSAT = \{\langle x, w, s, t \rangle |$ x = $code(M)$ per qualche TM M e M accetta $\langle w, c \rangle$ per qualche c di lunghezza al più s, in tempo al più t.$\}$
Per ogni L' in NP, $L' \leq_p TMSAT$.
Tuttavia, TMSAT non ci fa scoprire 'nulla di nuovo' su NP.

11.5 Il teorema di Cook-Levin

Theorem 11.4. *SAT è NP-completo.*

Proof. Dimostrare che SAT é in NP é immediato: data una formula F, possiamo costruire una TM che scelga in maniera non-deterministica un assegnamento A, e accettare se A rende F vera. Rimane da dimostrare che ogni linguaggio L in NP é riducibile a SAT. Supponi che M sia la TM non-deterministica che decide L. L'idea é di tradurre qualsiasi stringa w in una formula F_w, tale che assegnamenti di valore A a F_w rappresentano computazioni di M su w. In particolare, vogliamo che A renda F_w vera se e solo se M accetta w (quindi w é in L). La parte 'laboriosa' (ma concettualmente semplice) è costruire F_w con tale proprietà. Supponiamo che la TM $M = \langle \sum, Q, q_0, \delta, \{Y, N\} \rangle$ decida L in tempo N^k per qualche costante k. Definiamo un **tableau** per M come una tabella $n^k \times n^k$ dove le righe descrivono le configurazioni di un ramo di computazione di M su input w. Il problema di stabilire se M accetta w è equivalente a stabilire se esiste un tableau accettante (cioè uno che arrivi ad una configurazione in uno stato finale). La formula F_w sarà la congiunzione di quattro sottoformule.



$$F_w := F_{cell} \wedge F_{start} \wedge F_{move} \wedge F_{accept}$$

Ricordiamo la proprietà che vogliamo garantire tramite la definizione di F_w:

F_w è soddisfacibile $\iff$ $\exists$ un tableau accettante ($\iff$ $\exists$ una computrazione accettante di M su w.)

L'insieme di variabili che appaiono in F_w è dato da:

$$\{x_{i,j,s} | (i,j) \in n^k \times n^k \wedge s \in Q \cup \sum \cup \{\#\}\}$$

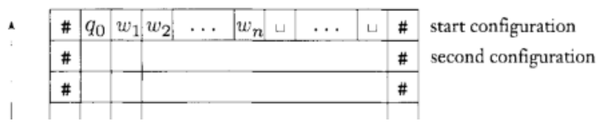
Intuitivamente, se vogliamo assegnare valore 1 (vero) a $x_{i,j,s}$ se $cell[i,j]$ contiene il valore s, e 0 (false) altrimenti. La prima sottoformula di F_w, F_{cell}, assicura che per rendere vera F_w dobbiamo rendere vera esattamente una variabile per ogni cella del tableau.

$$F_{cell} := \bigwedge_{1 \leq i,j \leq n^k} [(\bigvee_{s \in C} x_{i,j,s}) \wedge (\bigvee_{s,t \in C, s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}))]$$

La seconda sottoformula di F_w, F_{start}, assicura che nel rendere vera F_w il tableau considerato deve avere sulla prima riga la configurazione iniziale di una computazione di M su w.

$$F_{start} := x_{1,1,\#} \wedge x_{1,2,q_0} \wedge ... \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\emptyset} \wedge ... \wedge x_{1,n^k-1} \wedge x_{1,n^k,\#}$$

ovvero:



La quarta sottoformula di F_w, F_{accept}, assicura che nel rendere vera F_w almeno una delle configurazioni rappresentate nel tableau raggiunge lo stato accettante Y.

$$F_{accept} := \vee_{1 \leq i,j \leq n^k} x_{i,j,Y}$$

Rimane da definire la terza sottoformula di F_w, F_{move}, la quale assicura che F_w può essere resa vera solo se i simboli nelle varie celle del tableau descrivono una computazione di M su w che rispetti la funzione di transizione δ di M. La computazione di una TM è locale, per cui è sufficiente esprimere una condizione che riguardi porzioni del tableau di dimensione 2×3. Le chiamiamo **finestre**. Per esempio, supponiamo che δ includa le transizioni:

$$\delta(q_1, a) = \{(q_1, b, \rightarrow)\}$$
$$\delta(q_1, b) = \{(q_2, c, \leftarrow), (q_2, a, \rightarrow)\}$$

Le seguenti sono finestre ammesse nel tableau:

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Le seguenti sono finestre non ammesse nel tableau:

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

$$F_{move} :=$$
$$\bigwedge_{1<i\leq n^k, 1<j<n^k} \left(\bigvee_{a_1,\ldots,a_6} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \right)$$

Abbiamo dunque verificato che F_w è soddisfacibibile $\iff$ $\exists$ un tableau accettante. Dobbiamo ora dimostrare che la costruzione di F_w avviene in tempo polinomiale rispetto alla lunghezza di w.

1. Il tableau è una tabella $n^k \times n^k$, perciò contiene n^{2k} celle. Ogni cella contiene uno tra m simboli diversi, dove m è la cardinalità di $Q \cup \sum \cup \{\#\}$. Perciò il numero di variabili $x_{i,j,s}$ è $m \times n^{2k}$. In notazione Big-O, $O(n^{2k})$.

2. F_{cell} contiene una sottoformula di lunghezza costante $\forall x_{i,j,s}$, perciò la sua lunghezza è $O(n^{2k})$.

3. F_{start} contiene una sottofrmula di lunghezza costante per ognuna delle n^k celle nella prima riga, perciò la sua lunghezza è $O(n^k)$.

4. Sia F_{move} che F_{accept} contengono una sottofrmula di lunghezza costante per ogni cella del tableau, perciò la loro lunghezza è $O(n^{2k})$

5. Perciò la lunghezza complesivadi F_w è polinomiale: $O(n^{2k}) + O(n^k) + O(n^{2k}) + O(n^{2k}) = O(n^{2k})$

Ora che abbiamo appurato che generare ogni carattere di F_w richiede tempo polinomiale, non è difficile ottenere un TM che costruisce in tempo polinomiale F_w nel modo descritto, a partire dalla definiaione di M e di w. □

Corollary 11.4.1. *Da 11.4 se SAT è in P, allora P = NP.*

11.6 Altri problemi NP-completi

Theorem 11.5. *3SAT è NP-completo.*

Corollary 11.5.1. *11.5 CLIQUE è NP-completo.*

Altri esempi:

- TSP
- olorare un grafo con k colori diversi

- vincere a battaglia navale
- fare mining di bitcoin in maniera ottimale
- Serializzabilità della conologia di un database

12 Complessità di Spazio

Sia M una TM che si ferma su tutti gli input. La sua **complessità di spazio** è definita come la funzione $t : \mathbb{N} \to \mathbb{N}$, dove $t(n)$ è il massimo numero di celle del nastro che la testina di M vista su arbitrari input di lunghezza n.

Data una funzione $t : \mathbb{N} \to \mathbb{N}$, definiamo la calsse di **complessità** (di spazio) **SPACE(t(n))** come la collezione di tutti i linguaggi decidibili da una TM (deterministica, a un nastro) in spazio $O(t(n))$.

In modo analogo, considerare le TM non-deterministiche ci dà una classe **NSPACE(t(n))**.

12.1 PSPACE e NSPACE

PSPACE è la classe dei linguaggi decidibili da una TM (deterministica, a un nastro) in spazio **polinomiale**. Ovvero:

$$PSPACE = \bigcup_k SPACE(n^k)$$

Analogamente, abbiamo:

$$NPSPACE = \bigcup_k NPSPACE(n^k)$$

12.2 (N)P e PSPACE

Il fatto che P sia inclusa in PSPAE è ovvio per definizione. È meno ovvio (ma verop) che NP sia inclusa in PSPACE. Dimostriamolo come corollario del seguente lemma.

Lemma 12.1. *$SAT \in PSPACE$.*

Proof. Considera una TM M definita nel seguente modo: Su input $\langle F \rangle$, dove F è una formula booleana:

1. $\forall$ *assegnamento* A di valore alle variabili di F:
2. Valuta F rispetto ad A. Se il valore di F è 1 (vero), accetta, se no rigetta.

Ogni iterazione di 1 può essere eseguita in spazio lineare rispetto a $\langle F \rangle$, dal momento che l'assegnamento avrà lunghezza $O(m)$ dove m è il numero di variabili in n. Nota che M nella sua interezza lavora in spazio $O(m)$: infatti possiamo eseguire ogni iterazione di 1 sulla stessa porzione di nastro, una volta cancellato il risultato dell'iterazione precedente. □

Corollary 12.1.1. *NP è inclusa in PSPACE.*

Proof. Sia L in NP. Per il teorema di Cook-Levin, $L \leq_p SAT$. Per 12.1 abbiamo che SAT è in PSPACE. Unendo questi due fatti, possiamo facilmente dimostrare che L è in PSPACE. □

12.3 Un problema PSPACE completo

Un linguaggio L è PSPACE-completo se è in PSPACE e ogni altro linguaggio L' in PSPACE è poly-riducibile ad esso ($L' \leq_p L$).

$$TQBF = \{\langle F \rangle | F \text{ è un enunciato booleano vero.}\}$$

Theorem 12.2. *TQBF è PSPACE-completo.*

Proof. Verifichiamo prima che TQBF sia in PSPACE. Ecco un algoritmo ALG che lo decide:
Su input $\langle F \rangle$, dove F è un enunciato booleano:

1. Se F non contiene quantificatori, allora contiene solo costanti (nessuna variabile). Valutiamola e accettiamola $\iff$ è vera.

2. Se $F = \exists x G$, chiama ALG ricorsivamente su G, prima valutando $x = 1$, poi valutando $x = 0$. Se almeno una computazione è accettante, accetta. Altrimenti, rigetta.

3. Se $F = \forall x G$, chiama ALG ricorsivamente su G, prima valutando $x = 1$, poi valutando $x = 0$, se entrambi le computazioni sono accettanti, accetta. Altrimenti rigetta.

Questo algoritmo decide TQBF. Inoltre il numero delle chiamate ricorsive è al più il numero di variabili in F. Ogni chiamata ha bisogno di memorizzare solo il valore di una variabile, perciò lo spazio utilizzato è $O(m)$, dove m è il numero di variabili in F. In conclusione, TQBF è in PSPACE. Vogliamo ora dimostrare che $L \leq_p TQBF$. Chiamiamo M la TM che decide L in spazio polinomiale, diciamo n^k per qualche costante k. Per dimostrare $L \leq_p TQBF$, è sufficiente costruire in tempo polinomiale una formula F_w tale che:

$$\text{M accetta w} \iff F_w \text{ è un enunciato booleano vero.}$$

Per costruire un tale F_w costruiamo prima un tipo di formula $F_{c,c',t}$ più generale con la seguente proprietà.

$$\text{M può andare dalla configurazione c alla configurazione c' in al più t passi}$$
$$\iff$$
$$F_{c,c',t} \text{ è un enunciato booleano vero.}$$

Basterà definire F_w come $F_{c_{init},c_{acc},t}$, dove c_{init} è la configuarzione iniziale, c_{acc} codifica una qualsiasi configurazione accettante, e $t = 2^{dn^k}$ per una costante d tale che M non ha più di t configurazioni possibili su input di lunghezza n. In $F_{c,c',t}$ c e c' saranno insiemi di tali variabili. Ciascuno codifica una configurazione (riga del tableau, vedi sopra teorema di Cook-Levin). Costruiamo $F_{c,c',t}$ per induzione su t. Se $t = 1$, abbiamo solo due casi:

1. la computazione ha 0 passi. Costruiamo un enunciato F_1 che esprime (nel modo visto nel teorema di Cook-Levin 11.4) che $cell[i, j]$ in c ha lo stesso valore di $cell[i, j]$ in c'.

2. la computazione ha 1 passo. Costruiamo un enunciato F_2 che esprime che la configurazione c' è raggiungibile in un passo dalla configurazione c (come visto nel teorema di Cook-Levin 11.4, tramite finestre).

$F_{c,c',1}$ sarà definita come $F_1 \vee F_2$.

3. Se $t > 1$, un'idea potrebbe essere di definire $F_{c,c',t}$ come $\exists m_1(F_{c,m,\frac{t}{2}} \wedge F_{m_1,c',\frac{t}{2}})$. Qui $\exists m_1$ abbrevia $\exists x_1, ..., x_l$, dove $x_1, ..., x_l$ sono le variabili che codificano la configurazione di m_1. Le sottoformule $F_{c,m_1,\frac{t}{2}}$ e $F_{m_1,c',\frac{t}{2}}$ sono definite per ipotesi induttiva. Il significato è quello giusto, ma la formula risultante ha lunghezza esponenziale in n. Infatti, ogni passo della costruzione induttiva di $F_{c,c',t}$ dimezza t, ma raddoppia la lunghezza della formula. PEr cui nel caso che ci interessa $t = 2^{dn^k}$, la lunghezza di $F_{c,c',t}$ sarà esponenziale in n. Con uso sapiente dei quantificatori otteniamo una formula della lunghezza giusta:

$$F_{c,c',t} := \exists m_1 \forall (c_3, c_4) \in \{(c, m_1), (m_1, c')\} F_{c_3,c_4,\frac{t}{2}}$$

Qui $\forall x \in y, z$ è notazione per $\forall x((x = y \vee y = z) \implies ...)$. Intuitivamente, la formula dice che le variabili in c_3 e c_4 possono avere lo stesso valore che le variabili in c e m_1, oppure lo stesso che le variabili in m_1 e c'. La nuova definizione ha lo stesso significato della precedente, ma ogni passo induttivo aggiunge una parte di lunghezza $O(n^k)$ (la quantificazione su c_3, c_4), e vi sono $log(2^{dn^k}) = O(n^k)$ passi induttivi. La sua lunghezza è dunque polinomiale in n.

Abbiamo così concluso la definizione di $F_{c,c',t}$ per induzione su t. Per concludere basta definire F_w come $F_{c_{init},c_{acc},t}$ dove c_{init} è la configurazione iniziale, c_{acc} codifica una qualsiasi configurazione accettante, e $t = 2^{dn^k}$ per una costante d tale cche M non ha più di t configurazioni possibili su input di lunghezza n. □

12.4 Riflessioni sul teorema

1. Il fatto che TQBF sia PSPACE-completo ci da un'indicazione di che tipo di problemi siano in PSPACE.

2. Il problema di trovare una strategia vincente nei giochi posizionali (scacchi, dama, Go, ...) può essere espresso mediante un'alternanza di quantificatori: per ogni mossa dell'avversario, esiste una mia mossa tale che etc.

3. Non é dunque sorprendente che questa tipologia di problemi é solitamente dimostrabile essere in PSPACE attraverso la costruzione di una Poly riduzione a TQBF.

4. Molti problemi in robotica possono essere formulati come giochi, dove l'avversario é l'ambiente in cui si muove il robot.

12.5 Il teorema di Savitch

Theorem 12.3. *Per qualsiasi funzione* $t : \mathbb{N} \to \mathbb{N}$, *abbiamo:*

$$NSPACE(t(n)) \subset SPACE(t^2(n))$$

Proof. Supponi che M sia una TM non-deterministica che decida un linguaggio L in spazio $t(n)$ rispetto alla lunghezza n dell'input. Vogliamo costruire una TM deterministica M' che decide L in spazio $t^2(n)$.
La costruzione di M' usa la stessa idea utilizzata per dimostrare che TQBF è PSPACE-completo. Definiremo una procedura REACH che lavora in spazio $O(t^2(n))$, tale che:

$$M \text{ può andare dalla configurazione } c \text{ alla configurazione } c' \text{ in al più } t \text{ passi.}$$
$$\iff$$
$$REACH(c, c', t) \text{ dà output ACCETTA.}$$

Definiremo poi M' come la TM che su input di lunghezza n esegue $REACH(c_{init}, c_{acc}, 2^{dt(n)})$[10] e accetta se l'output è ACCETTA.
REACH è definita nel modo seguente. Su input c, c', T:

1. Se $T = 1$, verifica se $c = c'$ o se la configurazione c' è raggiungibile da c in un passo di computazione di M. ACCETTA se almeno uno dei due test ha successo, se no RIGETTA.

2. Se $T > 1$, $\forall c_m$ configurazione di M che usa spazio $t(n)$:

3. Esegui $REACH(c, c_m, \frac{T}{2})$

4. Esegui $REACH(c_m, c, \frac{T}{2})$

5. Se entrambi hanno dato output ACCETTA, ACCETTA.

6. Se nonn ha accettato in nessun passo precedente, RIGETTA.

Rimane da dimostrare che $REACH$ lavora in spazio $O(t^2(n))$.

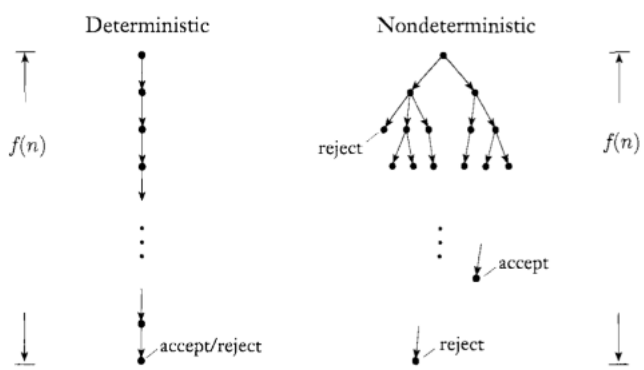
- La ricorsione in 2 divide ogni volta T per due. Dunque per $T = 2^{dt(n)}$ ci sono $O(log 2^{dt(n)}) = O(t(n))$ chiamate ricorsive.

- Ciascuno dei passi 3 e 4 usa spazio $O(t(n))$. Quando abbiamo eseguito 3, possiamo cancellare ed eseguire 4.

- Pertanto $REACH$ nel suo complesso richiede spazio $O(t(n)) \times O(t(n)) = O(t^2(n))$.

Come anticipato, definiamo M' come la TM che su input di lunghezza n esegue $REACH(c_{init}, c_{acc}, 2^{dt(n)})$ e accetta se l'output è ACCETTA. Da momento che $REACH(c_{init}, c_{acc}, 2^{dt(n}$ lavora in tempo $O(t^2(n))$ abbiamo dimostrato che L è in $SPACE(t^2(n))$. □

È soprendente che $PSCAPE = NPSPACE$, oppure no?



Corollary 12.3.1. $NSPACE = PSPAPE$

[10]scegli d tale che M non ha più di $2^{dt(n)}$ configurazioni possibili su input di lunghezza n,

13 Gerarchie e problemi intrattabili

La nostra intuizione é che non tutte le classi di complessità di tempo e di spazio corrispondono allo stesso insieme di problemi. Più risorse computazionali sono allocate, più dovrebbe essere grande il numero di problemi che é possibile calcolare. Per esempio, una TM dovrebbe poter risolvere strettamente più problemi in tempo n^3 che in tempo n^2. I risultati che dimostrano tali inclusioni strette sono detti teoremi di gerarchia. Tali risultati hanno come conseguenza che alcuni problemi non appartengono a classi come P o PSPACE. Si tratta dunque di problemi **intrattabili**: nonostante siano decidibili, non esistono algoritmi efficienti.

13.1 Notazione small-O

Date due funzioni $f, g : \mathbb{N} \to \mathbb{N}$, scriviamo $f(n) = o(g(n))$ e diciamo che $g(n)$ è un **bound superiore asintotico stretto** se esistono $c \in \mathbb{N}$ e $m \in \mathbb{N}$ tali che $\forall n, m.n \geq m$

$$f(n) < c \times g(n)$$

In altre parole, $g(n)$ é strettamente più grande di $f(n)$ per n sufficientemente grandi e modulo un fattore costante c.

13.2 Funzioni costruibili

$f : \mathbb{N} \to \mathbb{N}$, dove $f(n) \geq O(log(n))$, e SPACE-costruibile se possiamo calcolare in spazio $O(f(n))$ la funzione:

$$1^n \longmapsto \langle f(n) \rangle$$

dove $\langle f(n) \rangle$ è la codifica binaria di $f(n)$.
Tutte le misure più comuni per misurare la complessità sono SPACE-costruibili.

13.2.1 Esempio

n^2 é SPACE-costruibile. La TM che lo testimonia riceve in input 1^n , traduce in binario, $\langle n \rangle$, contando il numero degli 1, e ritorna come output $\langle n^2 \rangle$ usando qualsiasi metodo standard per moltiplicare n con sé stesso. Lo spazio utilizzato é $O(n)$.

13.3 Gerarchia di spazio

Theorem 13.1. *Per ogni funzione $f : \mathbb{N} \to \mathbb{N}$ SPACE-costruibilie, esiste un linguaggio L decidibile in spazio $O(f(n))$ ma non in spazio $o(f(n))$.*

Proof. Il linguaggio L verrà definito non da una proprietà ma da un algoritmo ALG, che usa spazio $O(f(n))$ ed é costruito in modo tale da assicurare che L é diverso da ogni linguaggio decidibile in spazio $o(f(n))$. Come ci riesce? Per diagonalizzazione. Su input code(M), ALG simula M su code(M) in una porzione di nastro f(n), e accetta se e solo se M ferma e rigetta. Da notare che:

1. M potrebbe usare più di f(n) spazio per n sufficientemente piccolo;
2. La simulazione di M su code(M) potrebbe richiedere più di f(n) spazio, e quindi essere rigettata a prescindere.

Definiamo ALG. Su input di lunghezza n:

1. Calcola f(n), e contrassegna tale quantità di nastro. Se i passi successivi provano ad usare più di f(n) celle, rigetta.
2. Se $w \neq code(M)10*$[11] per qualche TM M, rigetta.
3. Simula M su w. Se non ha concluso dopo $2^{f(n)}$[12] passi, rigetta.
4. Se M accetta, rigetta. Se M rigetta, accetta.

ALG richiede spazio $O(f(n))$. Definiamo L come il linguaggio

$$L = \{w \mid ALG \ accetta \ w\}$$

Pertanto L è decidibile in spazio $O(f(n))$. Dobbiamo dimostrare che non esiste TM che lo decida in spazio $o(f(n))$.
Per contraddizione, supponiamo che esista N tale che N decide L in spazio $g(n) = o(f(n))$. Per costruzione, ALG simula N in spazio $d \times g(n)$ per qualche costante d. Poiché $g(n) = o(f(n))$, esiste una costante m tale che, per tutti gli $n \geq m$, $(d \times g(n)) < f(n)$. Perciò la simulazione da parte di ALG di N su input code(N)10 m andrà a buon fine nello spazio allocato f(n). Abbiamo:

$$code(N)10^m \in L \iff \text{ALG accetta } code(N)10^m \iff \text{N rigetta } code(N)10^m$$

Perciò non può decidere L, contraddizione. □

Corollary 13.1.1. *Da 13.1* $\forall k, j.k < j$*:*

$$SPACE(n^k) \subset SPACE(n^j)$$

Corollary 13.1.2. $SPACE(log(n)) \subset SPACE(n)$

Definiamo $EXPSPACE = \bigcup_k SPACE(2^{n^k})$

Corollary 13.1.3. $PSPACE \subset EXPSPACE$

Quindi i problemi EXPSPACE-completi sono **intrattabili** (non hanno algoritmi PSPACE).

13.4 Gerarchia di tempo

$f : \mathbb{N} \to \mathbb{N}$, dove $f(n) \geq O(n \times log(n))$, è TIME-costruibile se possiamo calcorare in tempo $O(f(n))$ la funzione

$$1^n \longmapsto \langle f(n) \rangle$$

dove $\langle f(n) \rangle$ è la codifica binaria di $f(n)$.

Theorem 13.2. *Per ogni* $f : \mathbb{N} \to \mathbb{N}$ *TIME-costruibile, esiste un linguaggio L decidibile in tempo* $O(f(n))$ *ma non in tempo* $o(f(n)/logf(n))$*.*

Proof. La dimostrazione é simile a quella data per la gerarchia di spazio. Il fattore $logf(n)$ é dovuto al fatto che la simulazione di una TM richiede un overhead di tempo logaritmico (mentre nel caso dello spazio, l'overhead era costante). Per dettagli confrontare Sipser, capitolo 8. □

[11] Aggiungiamo una stringa $s \in 10*$ a code(M) affinchè la lunghezza dell'input diventi grande abbastanza perchè valga il bound asintotico $o(f(n))$ sullo spazio di computazione di M.

[12] Vogliamo assicurarci che ALG termini sempre.

Corollary 13.2.1. $\forall k, j. 1 < k < j$*:*

$$TIME(n^k) \subset TIME(n^j)$$

Corollary 13.2.2. $P \subset EXPTIME$

Quindi i problemi EXPTIME-completi sono **intrattabili** (non hanno algoritmi polinomiali).

13.5 Considerazioni finali

I teoremi di gerarchia sono entrambi basati sull'uso della diagonalizzazione per separare classi di complessità. In astratta, qualsiasi tecnica detta di diagonalizzazione si basa su due principi:

1. la possibilità di rappresentare TM come stringhe.
2. L'abilità di una TM di simulare un'altra TM di cui riceve il codice come input senza 'troppo' overhead in termini di spazio e tempo.

Quanto lontano ci può portare la diagonalizzazione? Perché non la utilizziamo per dimostrare che $P \neq NP$?
Consideriamo una variante della TM: una TM con oracolo $O \subseteq \{0,1\}^*$ può, in qualunque momento della computazione, chiedere $w \in O$? e ricevere risposta in tempo/spazio costante. L'osservazione che ci interessa é che, indipendentemente dalla scelta di O, i due presupposti indicati per la diagonalizzazione valgono anche per le TM con oracolo O. Perciò i risultati dimostrati usando la diagonalizzazione (ad esempio, i teoremi di gerarchia) valgono anche per le TM con oracolo.

Theorem 13.3 (Baker, Gill, Solovay 1975)**.** *Esistono oracoli* O *e* O' *tali che* $P^O = NP^O$ *e* $P^{O'} \neq NP^{O'}$.[13]

Perciò, se mai trovassimo una risposta al quesito $P = NP$?, o non usa la diagonalizzazione, o la utilizza insieme ad altre assunzioni che non valgono per TM con oracolo. Tali risultati si chiamano **non-relativizzanti** (nel senso che non relativizzano a TM con oracolo). Per esempio i teoremi di gerarchia relativizzano, il teorema di Cook-Levin (11.4) no.
In conclusione, la diagonalizzazione é un metodo potente, ma con limiti. Per attaccare domande come $P = NP$?, non é sufficiente simulare computazioni, trattandole come 'scatole nere'. Dobbiamo fare un'analisi più concreta, che riguardi l'effettivo contenuto della computazione
. Queste osservazioni hanno portato allo studio di modelli di calcolo differenti, dove l'analisi degli algoritmi é più flessibile. Un esempio importante sono i circuiti booleani.

[13] Classe dei problemi decidibili in tempo polinomiale da una TM con oracolo (deterministica se P, non-deterministica se NP).

14 esercitazione 1 Marzo 2023

14.1 Quesito 1

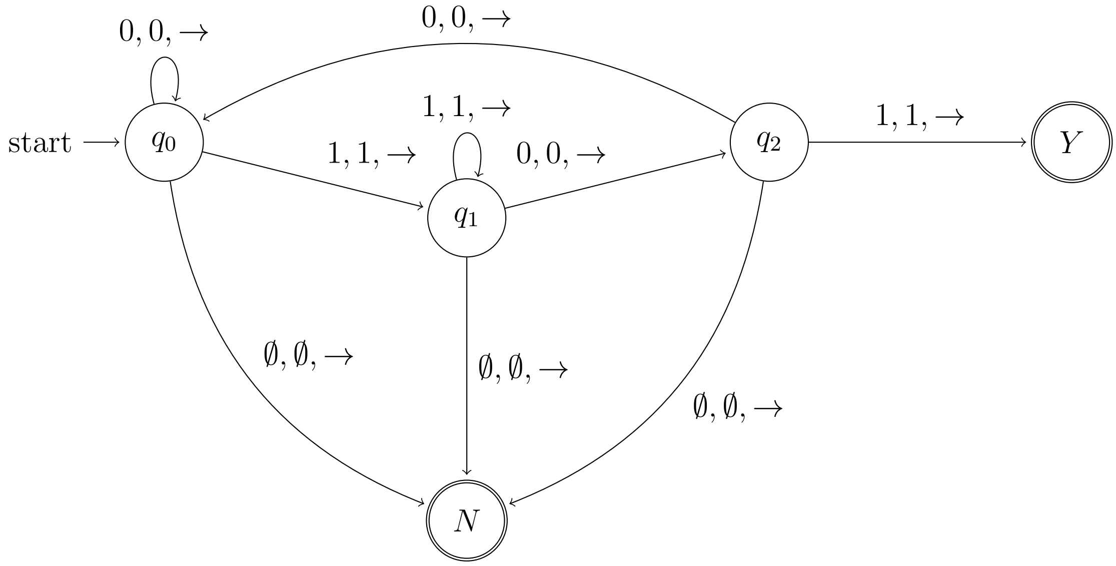
Un problema che vogliamo risolvere usando uno strumento di calcolo puo' essere espresso come un problema di decisione. Questi problemi hanno la componente dei dati e la risposta SI/NO. Per poter risolvere questi problemi con un calcolatore e' necessario codificare un problema di decisinoe come la funzione caratteristica di un linguaggio formale.

Dato a e b dati le cui codifiche sono code(a) e code(b) $\in \sum^*$ Le proprieta' che la codifica deve rispettare sono:

1. $a \neq b$ $code(a) \neq code(b)$
2. deve essere verificabile che se x $\in \sum^*$ è $code(a)$ per qualche a
3. deve essere calcolabile a partire da $code(a)$.

14.2 Problema 2.1

Alfabeto $\sum = \{0, 1\}$ La TM deve accettare l'input quando esso contiene 101 e rifiutare altrimenti.



14.3 Problema 2.2