ELEMENTI DI TEORIA DELLA COMPUTAZIONE

COMPLESSITÀ (PARTE II)

Introdurremo:

- La classe *EXPTIME*
- La classe *NP*
- Il concetto di riduzione polinomiale
- Il concetto di *NP*-completezza

La 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** ovvero problemi per i quali esistono algoritmi polinomiali per decidere i linguaggi associati e
- problemi **intrattabili** ovvero problemi per i quali esistono algoritmi esponenziali per decidere i linguaggi associati.

Ovviamente $P \subseteq EXPTIME$.

Inoltre esistono linguaggi in $EXPTIME \setminus P$, 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 $EXPTIME \setminus P$ 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 R_1 ed R_2, usando le operazioni $\cup$, $\circ$ e $*$.

Le espressioni regolari generalizzate (o *ERG*) aggiungono l'operazione $\uparrow$: se R è un'espressione regolare e $k \in \mathbb{N}$, $R \uparrow k$ è la concatenazione (o prodotto) di R con se stessa k volte.

Sia

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

risulta $EQ_{REX\uparrow} \in EXPTIME \setminus P$.

La classe *NP*

- I problemi corrispondenti ai linguaggi in $EXPTIME \setminus P$ non sono in genere importanti nelle applicazioni pratiche mentre si incontrano comunemente problemi (ovvero linguaggi) decidibili ma tali che gli algoritmi attualmente noti per decidere tali linguaggi 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.

La classe NP

- Molti di questi problemi hanno una caratteristica in comune che giustificherà l'introduzione di una nuova classe di linguaggi, la classe NP. Non è noto se tale classe sia o meno più estesa della classe P.
- Inoltre, esaminando questa questione, si è scoperto che un algoritmo polinomiale per decidere il linguaggio associato ad alcuni di essi, se esistesse, potrebbe essere utilizzato per decidere ogni linguaggio nella classe.
- Per capire tale caratteristica, consideriamo i seguenti due esempi di problemi di decisione.

HAMPATH

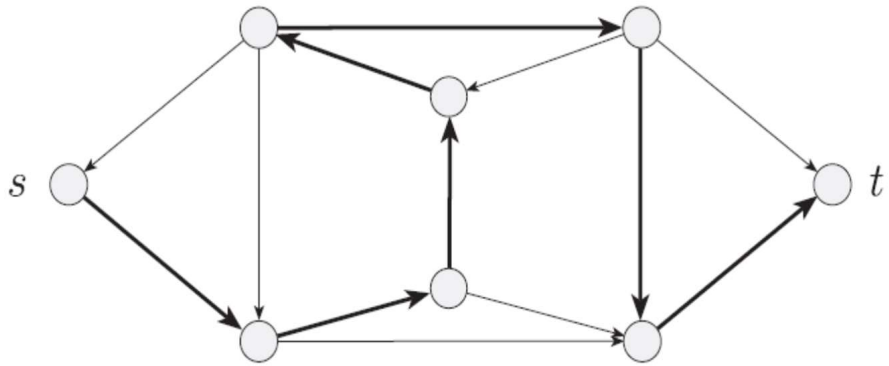

FIGURA 7.17
Un cammino Hamiltoniano attraversa ogni nodo esattamente una volta

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

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*.

HAMPATH

- Il linguaggio associato al problema di decisione del cammino Hamiltoniano:

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

- *HAMPATH* 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 $2 \leq i \leq |V|$,
- $u_1 = s$,
- $u_{|V|} = t$.

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

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

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

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

Esiste un algoritmo **polinomiale** che **verifica** se una sequenza $c = (u_1, \ldots, u_{|V|})$ 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*.

- Il linguaggio associato al problema di decisione di stabilire se un numero non è primo:

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

- Recentemente è stato dimostrato che $COMPOSITES \in P$.
- Comunque, dati x, p, **verificare** che p è un divisore di x, con $p > 1$, $p \neq x$, 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 M è un decisore tale che, sull'input w, esiste una computazione da $q_0 w$ a $u q_{accept} v$ se e solo se $w \in L$).
- Abbiamo introdotto la classe P e il concetto di linguaggio L decidibile in tempo polinomiale:

  $L \in P$, 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.

Algoritmi di verifica

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, $w \in A$ 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*

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

La stringa c prende il nome di **certificato** *o* **prova**.

A è il **linguaggio verificato** *da V.*

Nota.

Algoritmo = Decider
= Macchina di Turing deterministica (a un nastro) che si arresta su ogni input.

Quindi un verificatore è una macchina di Turing deterministica (a un nastro) che si arresta su ogni input.

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*

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

- *V ha complessità di tempo polinomiale in $|w|$.*

Complessità degli algoritmi di verifica

- 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 7.20

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 $t : \mathbb{N} \to \mathbb{R}^+$ *una funzione. La classe di complessità in tempo non deterministico* $NTIME(t(n))$ *è*

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

Corollario 7.22

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

Nota. Il Corollario 7.22 è una diretta conseguenza del Teorema 7.20 che afferma che un linguaggio L è in NP se e solo se esiste una macchina di Turing non deterministica che decide L in tempo polinomiale.

Esempi di linguaggi in *NP*: *CLIQUE*

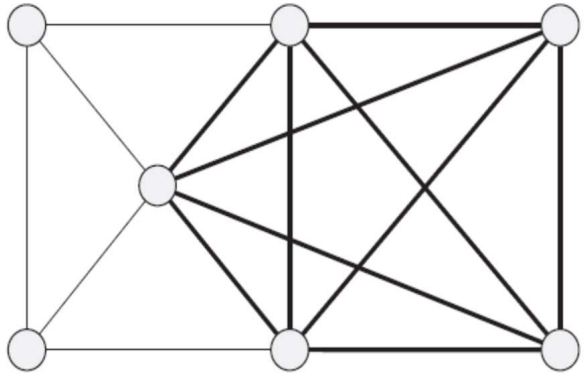

FIGURA 7.23
Un grafo con una 5-clique

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

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.

Esempi di linguaggi in *NP*: *CLIQUE*

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$\}$

Esempi di linguaggi in *NP*: *CLIQUE*

Teorema
$CLIQUE \in NP$

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 finito 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 $= \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}$ ed esiste $S' \subseteq S$ tale che $\sum_{s \in S'} s = t\}$

Esempio: $\langle\{4, 11, 16, 21, 27\}, 25\rangle \in$ *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).

Esempi di linguaggi in *NP*: *SUBSET-SUM*

Teorema
$SUBSET\text{-}SUM \in NP$

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."

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

Prova alternativa: utilizzare le macchine di Turing non deterministiche.

Esempi di linguaggi in *NP*: *HAMPATH*

Teorema
$HAMPATH \in NP$

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, $u_1 = s$, $u_{|V|} = t$ e, per ogni i con $2 \leq i \leq n$, se $(u_{i-1}, u_i) \in E$, accetta in caso affermativo; altrimenti rifiuta."

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

- $P =$ 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 1
$P \subseteq NP$

Dimostrazione
Se $L \in P$, 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.

□

Nota. Sia V un algoritmo di verifica.

La codifica di $\langle w, \epsilon \rangle$ può essere diversa dalla codifica di $\langle w \rangle$.

Teorema
$P \subseteq NP$

Dimostrazione
Se $L \in P$, esiste un macchina di Turing deterministica $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$ che decide L in tempo polinomiale.
Sia $M' = (Q, \Sigma, \Gamma, \delta', q_0, q_{accept}, q_{reject})$ la macchina di Turing non deterministica tale che

$$\delta'(q, \gamma) = \{\delta(q, \gamma)\}$$

per ogni $q \in Q \setminus \{q_{accept}, q_{reject}\}$ e ogni $\gamma \in \Gamma$.
È facile provare che M' è una macchina di Turing non deterministica equivalente ad M e che M' decide L in tempo polinomiale.

Relazioni tra i modelli: MdT non deterministica

Teorema 7.11

Sia $t(n)$ una funzione tale che $t(n) \geq n$. 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.

Teorema

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

Dimostrazione

Dobbiamo provare che $P \subseteq NP$ e che $NP \subseteq EXPTIME$.

Il Teorema 1 dimostra l'inclusione $P \subseteq NP$.

Sia $L \in NP$. Per il Teorema 7.20, esiste una macchina di Turing non deterministica N che decide L in tempo $O(n^k)$, per qualche $k \geq 1$.

Per il Teorema 7.11, esiste una macchina di Turing deterministica a un nastro M che decide L con complessità di tempo $2^{O(n^k)}$.

Quindi M decide L con complessità di tempo $O(2^{n^h})$, per qualche $h \geq 1$, cioè $L \in EXPTIME$.

$\square$

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

È noto che $P \subsetneq EXPTIME$.

Uno dei più grandi problemi aperti dell'informatica teorica:

$P = NP?$

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, non è noto se la classe NP sia o meno chiusa rispetto al complemento.

Se $\langle G, s, t \rangle \in HAMPATH$ esiste un cammino Hamiltoniano c in G da s a t.

Se tale cammino è stato scoperto, è possibile verificare in tempo polinomiale che $\langle G, s, t \rangle \in HAMPATH$: basta fornire in input al verificatore per $HAMPATH$ la stringa $\langle \langle G, s, t \rangle, c \rangle$.

Ma se $\langle G, s, t \rangle \notin HAMPATH$, tale cammino non esiste e non conosciamo alcun algoritmo polinomiale per verificare la non esistenza di tale cammino.

Nota che se $\langle G, s, t \rangle \notin HAMPATH$ allora $\langle G, s, t \rangle \in \overline{HAMPATH}$.

Le osservazioni precedenti si applicano a qualsiasi linguaggio in NP e pongono il problema del rapporto tra la classe NP e la classe $coNP = \{L \mid \overline{L} \in NP\}$.

$coNP = \{L \mid \overline{L} \in NP\}$

- Esempi di linguaggi in *coNP*: $\overline{HAMPATH}$, $\overline{CLIQUE}$, $\overline{SUBSET\text{-}SUM}$.

  Per ognuno di questi linguaggi non è noto se tale linguaggio appartenga o meno a *NP*.
  (Nota: errore di stampa sul testo, pag. 318, frase dopo Teorema 7.25.)

  Verificare che "qualcosa non c'è" sembra essere più difficile che verificare che "c'è".

$NP = coNP?$

Quattro possibili scenari

Le risposte alle domande

$P = NP?$

$NP = coNP?$

danno luogo ai seguenti quattro possibili scenari.

Quattro possibili scenari

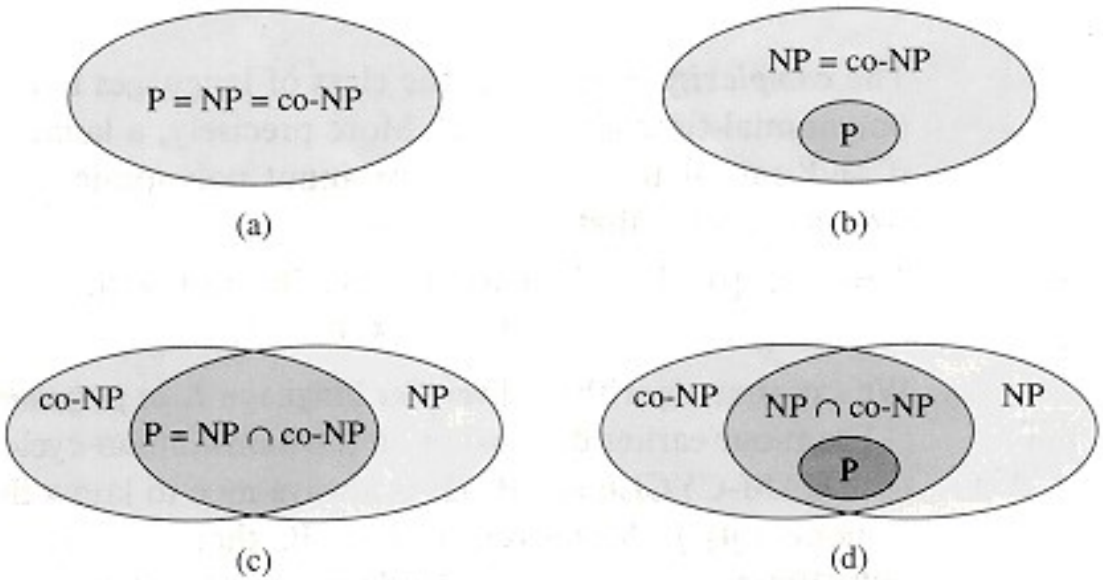

Figura tratta da T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to algorithms.

Quattro possibili scenari

Vi sono quattro possibilità:

1. $P = NP = coNP$
2. $P \subsetneq NP = coNP$
3. $NP \neq coNP$, $P = NP \cap coNP \subsetneq NP$ (e quindi $P = NP \cap coNP \subsetneq coNP$)
4. $NP \neq coNP$, $P \subsetneq NP \cap coNP \subsetneq NP$ (e quindi $P \subsetneq NP \cap coNP \subsetneq coNP$)

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 vari linguaggi appartenenti a NP la cui complessità individuale è correlata a quella dell'intera classe NP.

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.

Il concetto di 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.

Funzioni calcolabili in tempo polinomiale

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.*

- Nota la differenza tra funzione calcolabile e funzione calcolabile in tempo polinomiale.

Funzioni calcolabili in tempo polinomiale

- Esempio 1. Consideriamo la funzione $f : \{0,1\}^* \to \{0,1\}^*$ tale che $f(\langle m \rangle) = \langle m+1 \rangle$,
  dove $m \in \mathbb{N}$ e $\langle m \rangle$ è la rappresentazione binaria di m.

  Abbiamo visto un algoritmo polinomiale per il calcolo del successore in binario.

  La funzione f è calcolabile in tempo polinomiale nella lunghezza dell'input $\langle m \rangle$.

Funzioni calcolabili in tempo polinomiale

- Esempio 2. Consideriamo la funzione $g : \{0,1\}^* \to \{0,1\}^*$ tale che $g(\langle m \rangle) = \langle m \rangle \# 1^m$,
  dove $m \in \mathbb{N}$ e $\langle m \rangle$ è la rappresentazione binaria di m.

  Abbiamo visto che la funzione g non è calcolabile in tempo polinomiale nella lunghezza dell'input $\langle m \rangle$. Tuttavia la funzione g è calcolabile.

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*

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

Riducibilità in tempo polinomiale

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.

Riducibilità in tempo polinomiale

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.

Nota.

Nella definizione di riduzione di tempo polinomiale **non** è richiesto che f sia una funzione iniettiva o suriettiva.

Nota.

- Se A è un linguaggio su un alfabeto Σ e A è associato a un problema di decisione $\mathbb{P}_D$, le stringhe w in Σ^* si dividono in tre gruppi:
  1. w è la codifica di un'istanza di $\mathbb{P}_D$ per la quale $\mathbb{P}_D$ ammette risposta "sì" (e quindi $w \in A$);
  2. w è la codifica di un'istanza di $\mathbb{P}_D$ per la quale $\mathbb{P}_D$ ammette risposta "no" (e quindi $w \notin A$);
  3. w non è la codifica di un'istanza di $\mathbb{P}_D$ (e quindi $w \notin A$).
- 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 $\mathbb{P}_D$) una stringa $f(w)$ che non è in B.

Nota.

- Abbiamo detto che consideriamo codifiche **"ragionevoli"** cioè tali che non vi siano istanze la cui rappresentazione sia artificiosamente lunga.
- Codifiche ragionevoli consentono di codificare e decodificare in tempo polinomiale.
- Quindi se A è il linguaggio associato a un problema di decisione $\mathbb{P}_D$ ed f è una riduzione polinomiale di A a B, la macchina di Turing F che calcola f utilizza inizialmente un algoritmo polinomiale nella lunghezza dell'input w per decidere se w è o non è la codifica di un'istanza di $\mathbb{P}_D$.

Riducibilità in tempo polinomiale

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.

Riducibilità in tempo polinomiale - Teoremi

Teorema

Se $A \leq_P B$ e $B \in P$, allora $A \in P$.

Dimostrazione

- Per ipotesi $B \in P$, 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 sull'input w:
  1. simula F su w e calcola $f(w)$,
  2. simula M sull'input $f(w)$ per decidere se $f(w) \in B$.
  3. N accetta w se M accetta $f(w)$, N rifiuta w se M rifiuta $f(w)$.

$$N : w \to \boxed{\boxed{F} \to f(w) \to \boxed{M}}$$

N decide A (correttezza dell'algoritmo N).

N è un decider. Infatti N si ferma su w se si fermano F ed M. Ora, per ogni w, F si ferma con $f(w)$ sul nastro e per ogni w, M si ferma su $f(w)$ perché M è un decider.

Inoltre N riconosce A. Infatti

$$\begin{aligned} w \in L(N) &\Leftrightarrow f(w) \in L(M) \text{ (per la definizione di } N) \\ &\Leftrightarrow f(w) \in B \text{ (perché } M \text{ decide } B) \\ &\Leftrightarrow w \in A \text{ (perché } f \text{ è una riduzione polinomiale di } A \text{ a } B) \end{aligned}$$

Riducibilità in tempo polinomiale - Teoremi

- 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, $|f(w)| \leq cn^k$ perché **la lunghezza dell'output di F è limitata dalla complessità di tempo di F**)

  Al secondo passo M viene eseguito sull'input $f(w)$ e si arresterà dopo $c'|f(w)|^t \leq c'(cn^k)^t$ 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 $A \in P$.

□

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 $(g \circ f)(w) = g(f(w))$.
- Risulta, per ogni $w \in \Sigma^*$:

$$
\begin{aligned}
w \in A \quad &\Leftrightarrow \quad f(w) \in B \text{ (perché } f \text{ è una riduzione polinomiale di } A \text{ a } B\text{)} \\
&\Leftrightarrow \quad g(f(w)) \in C \text{ (perché } g \text{ è una riduzione polinomiale di } B \text{ a } C\text{)}
\end{aligned}
$$

- Inoltre la funzione $g \circ f$ è una funzione calcolabile in tempo polinomiale.

- Infatti, sia F l'algoritmo di complessità $O(n^k)$ 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.

Riducibilità in tempo polinomiale

- L'algoritmo GF calcola $g \circ f$ perchè 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, $|f(w)| \leq cn^k$ perché **la lunghezza dell'output di F è limitata dalla complessità di tempo di F**).
  Al secondo passo G viene eseguito sull'input $f(w)$ e si arresterà dopo $c'|f(w)|^t \leq c'(cn^k)^t$ 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 $g \circ f$ è una riduzione di tempo polinomiale di A a C.

- Variabili Booleane: variabili che possono assumere valore VERO o FALSO. In genere rappresentiamo VERO con 1 e FALSO con 0.
- Operazioni Booleane: $\vee$ (o *OR*), $\wedge$ (o *AND*), $\neg$ (o *NOT*)
- Denotiamo $\neg x$ con $\overline{x}$
- $0 \vee 0 = 0$, $0 \vee 1 = 1 \vee 0 = 1 \vee 1 = 1$
- $0 \wedge 0 = 0 \wedge 1 = 1 \wedge 0 = 0$, $1 \wedge 1 = 1$
- $\overline{0} = 1$, $\overline{1} = 0$

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, $\overline{x}$, *con* $x \in X$, *sono formule booleane.*
- *Se* ϕ, ϕ_1, ϕ_2 *sono formule booleane allora* $(\phi_1 \vee \phi_2)$, $(\phi_1 \wedge \phi_2)$, $\overline{\phi}$ *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: $\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ 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:

- $\phi_1 = (\overline{x} \vee y) \wedge (x \vee \overline{z})$ è soddisfacibile (assegnamento: $x = 0$, $y = 1$, $z = 0$),
- $\phi_2 = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$ è soddisfacibile,
- $\phi_3 = (\overline{x} \vee x) \wedge (y \vee \overline{y})$ 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.

Il problema della **soddisfacibilità di una formula booleana**:
Data una formula booleana ϕ, ϕ è soddisfacibile?

Il linguaggio associato è:

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

Teorema

$SAT \in NP$.

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."

$\exists c : \langle \langle \phi \rangle, c \rangle \in L(V) \Leftrightarrow \langle \phi \rangle \in SAT$ □

Definizione

Una clausola è un OR di letterali.

Esempio: $(\overline{x} \vee x \vee y \vee z)$

Definizione

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.

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

$$3SAT = \{\langle \phi \rangle \mid \phi \text{ è una formula 3CNF soddisfacibile}\}$$

$3SAT$ è riducibile in tempo polinomiale a $CLIQUE$

Teorema

$$3SAT \leq_P CLIQUE$$

$3SAT = \{\langle\phi\rangle \mid \phi$ è una formula 3CNF soddisfacibile$\}$

Una formula 3CNF è un *AND* di clausole e tutte le clausole hanno tre letterali.

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

$3SAT$ è riducibile in tempo polinomiale a $CLIQUE$

Nel descrivere la riduzione polinomiale f da $3SAT$ a $CLIQUE$ applicheremo la nostra convenzione: non specificheremo il valore di f sulle stringhe w tali che $w \neq \langle \phi \rangle$, ϕ formula booleana 3CNF.

3*SAT* è riducibile in tempo polinomiale a *CLIQUE*

Dobbiamo definire una funzione f, calcolabile in tempo polinomiale, che a $\langle \phi \rangle$, dove ϕ è una formula booleana 3CNF, associa $\langle G, k \rangle$, dove G è un grafo non orientato e k è un intero.

Inoltre f deve essere tale che

- Se ϕ è soddisfacibile allora il grafo associato G ha una k-clique.
- Se il grafo associato G ha una k-clique allora ϕ è soddisfacibile.

3*SAT* è riducibile in tempo polinomiale a *CLIQUE*

La dimostrazione inizia con la costruzione del grafo non orientato G e la definizione di k che la funzione f associa alla formula 3CNF ϕ, cioè tali che $f(\langle\phi\rangle) = \langle G, k\rangle$.

Poi si dimostra che f è calcolabile in tempo polinomiale e tale che

- Se ϕ è soddisfacibile allora il grafo associato G ha una k-clique.
- Se il grafo associato G ha una k-clique allora ϕ è soddisfacibile.

3*SAT* è riducibile in tempo polinomiale a *CLIQUE*

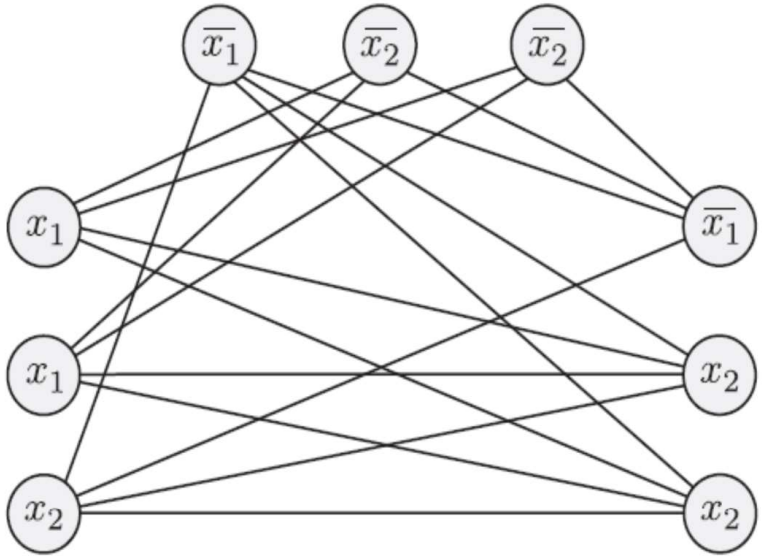

FIGURA 7.33
Il grafo che la riduzione produce per $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$

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

$3SAT$ è riducibile in tempo polinomiale a $CLIQUE$

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 $3 \times k$ 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 in $(a_j \vee b_j \vee c_j)$. Quindi $V = \{a_1, b_1, c_1, \ldots, a_k, b_k, c_k\}$.
- 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 $\overline{x}$.
- Ogni altra coppia di vertici è connessa da un arco.

$3SAT$ è riducibile in tempo polinomiale a $CLIQUE$

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 $\langle \phi \rangle \in 3SAT$ se e solo se $\langle G, k \rangle \in CLIQUE$ cioè

ϕ è soddisfacibile se e solo se G ha una k-clique.

3*SAT* è riducibile in tempo polinomiale a *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 $x, \overline{x}$ perché corrispondono a letterali veri nell'assegnamento di soddisfacibilità. Quindi due qualsiasi vertici in G' sono connessi da un arco in G.

$3SAT$ è riducibile in tempo polinomiale a $CLIQUE$

- 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 $x, \overline{x}$.
- Ogni tripla contiene un nodo di G' e quindi ogni clausola contiene un letterale vero.
- Questo è un assegnamento di soddisfacibilità per ϕ cioè $\langle \phi \rangle \in 3SAT$.

Il concetto di NP-completezza

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 di linguaggio *NP*-completo

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.*

Perché la classe dei linguaggi *NP*-completi è importante?

Teorema

Se B è NP-completo e B è in P allora $P = NP$.

Dimostrazione.

- Siccome B è NP-completo, per ogni $A \in NP$, risulta $A \leq_P B$
- Ma abbiamo provato che se $A \leq_P B$ e $B \in P$ allora $A \in P$
- Quindi $NP \subseteq P$ e siccome $P \subseteq NP$ risulta $P = NP$.

□

Ricorda: Indecidibilità

- 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.

Teorema

Se B è NP-completo e $B \leq_P C$, con $C \in NP$, allora C è NP-completo.

Dimostrazione.

- Per ipotesi:
  1. $C \in NP$,
  2. Per ogni $A \in NP$, $A \leq_P B$ (B è NP-completo)
  3. $B \leq_P C$
- Allora, utilizzando la proprietà transitiva di $\leq_P$:
  1. $C \in NP$,
  2. Per ogni $A \in NP$, $A \leq_P C$
- Cioè C è NP-completo.

□

Riducibilità polinomiale ed NP-completezza

Una possibile strategia per provare che un linguaggio B è NP-completo:

1. Mostrare che $B \in NP$
2. Scegliere un linguaggio A che sia NP-completo
3. Definire una riduzione di tempo polinomiale di A a B.

Teorema (Cook-Levin)

SAT è NP-completo.

- Conseguenza: $SAT \in P$ se e solo se $P = NP$.
- Sappiamo che $SAT \in NP$. La prova del teorema di Cook-Levin consiste nel mostrare che ogni $A \in NP$ 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.

$SAT_{CNF} = \{\langle\phi\rangle \mid \phi$ è una formula booleana soddisfacibile in $CNF\}$

$SAT_{CNF} \in NP$,

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

$$3SAT = \{\langle\phi\rangle \mid \phi \text{ è una formula 3CNF soddisfacibile}\}$$

Teorema

3SAT è NP-completo.

$3SAT$ 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.

CLIQUE è *NP*-completo

Teorema
CLIQUE è NP-completo.

Dimostrazione.

- Sappiamo che $CLIQUE \in NP$.
- Inoltre, $3SAT$ è NP-completo e $3SAT \leq_P CLIQUE$
- Quindi *CLIQUE* è *NP*-completo.

□