Calcolabilità e Complessità

Andrea Asperti

Department of Computer Science, University of Bologna
Mura Anteo Zamboni 7, 40127, Bologna, ITALY
asperti@cs.unibo.it

Content

Lezioni 1-3: Introduzione

- La teoria della complessità
- L'algoritmo di Euclide
- Cosa misurare e come farlo
- Funzioni d'ordine
- Esempi di problemi con complessità polinomiale
- Riducibilità
- Ricerca vs. Verifica
- Altri esempi di problemi in NP

La Teoria della Complessità

Ramo dell'Informatica Teorica focalizzato sulla classificazione dei problemi computazionali in funzione della loro inerente difficoltà, ovvero delle risorse necessarie alla loro risoluzione.

Richiede una definizione precisa di un modello di calcolo che permetta la quantificazione delle risorse (ad esempio tempo e spazio) necessarie alla computazione.

Le principali tematiche di studio riguardano:

- indipendenza dal modello di calcolo
- relazioni tra misurazioni basate su risorse di natura differente
- relazioni tra modelli di calcolo deterministici e non deterministici (ricerca vs. verifica)
- definizione e studio di classi di complessità
- loro strutturazione gerarchica
- teoremi di separazione

L'algoritmo di Euclide (mcd)

Siano $div(m, n)$ e $rem(m, n)$ il quoziente ed il resto della divisione tra m e n, i.e.

$$m = div(m, n) * n + rem(m, n)$$

Euclide

Il massimo comun divisore $gcd(m, n)$ tra due numeri naturali m e n $(m \geq n)$ è calcolato dalla seguente funzione:

$$\begin{cases} gcd(m, 0) = m \\ gcd(m, n) = gcd(n, rem(m, n)) \end{cases}$$

Se l divide m e n allora divide anche $rem(m, n) = m - div(m, n) * n$.
Se l divide n e $rem(m, n)$ allora divide anche $m = div(m, n) * n + rem(m, n)$

Complessità dell'algoritmo di Euclide

Consideriamo un generico passo del calcolo di $gcd(r_i, r_{i+1})$

$$gcd(r_i, r_{i+1}) = gcd(r_{i+1}, r_{i+2}) \text{ dove } r_i = q_{i+1} * r_{i+1} + r_{i+2}$$

Per ogni i, $r_{i+1} < r_i$ e $q_i > 0$, e dunque

$$r_i = q_{i+1} * r_{i+1} + r_{i+2} > 2r_{i+2}$$

r_{i+2} è il primo parametro della chiamata successiva di gcd: questo significa che ogni due chiamate il primo parametro viene almeno dimezzato, e dunque che il calcolo di $gcd(a, b)$ si arresta dopo al più $2 * log_2(a)$ iterazioni.

Complessità dell'algoritmo di Euclide

Consideriamo un generico passo del calcolo di $gcd(r_i, r_{i+1})$

$$gcd(r_i, r_{i+1}) = gcd(r_{i+1}, r_{i+2}) \text{ dove } r_i = q_{i+1} * r_{i+1} + r_{i+2}$$

Per ogni i, $r_{i+1} < r_i$ e $q_i > 0$, e dunque

$$r_i = q_{i+1} * r_{i+1} + r_{i+2} > 2r_{i+2}$$

r_{i+2} è il primo parametro della chiamata successiva di gcd: questo significa che ogni due chiamate il primo parametro viene almeno dimezzato, e dunque che il calcolo di $gcd(a, b)$ si arresta dopo al più $2 * log_2(a)$ iterazioni.

Poiche $log_2(a)$ è la dimensione dell'input, il calcolo richiede un numero lineare di iterazioni.

L'algoritmo ha dunque complessità polinomiale.

Rappresentazione binaria

Utilizzeremo le seguenti notazioni:

- $bin(n)$ per la rappresentazione binaria del natuale n:

$$bin(n) = a_k a_{k-1} \dots a_0 \quad \Leftrightarrow \quad n = \sum_{i=0}^{k} a_i * 2^i$$

- $log(n) := |bin(n)| - 1 = log_2(max(1, n))$.

n	0	1	2	3	4	5	6	7	8	...
$log(n)$	0	0	1	1	2	2	2	2	3	...
$bin(n)$	0	1	10	11	100	101	110	111	1000	...

Importanza della Rappresentazione

L'algoritmo naif per il test di primalità (verificare che per ogni $i < n$, $rem(n, i) \neq 0$) richiede un numero lineare di divisioni in n, ed è dunque un algoritmo esponenziale (rispetto alla dimensione dell'input).

Nel 2002 tre informatici indiani (Agrawal, Kayal e Saxena) hanno definito un algoritmo di complessità polinomiale in $bin(n)$ (con una crescita dell'ordine di $log(n)^{11}$).

Notazioni d'Ordine

Sia $f : \mathcal{N} \to \mathcal{N}$:

- $O(f) := \{g : \mathcal{N} \to \mathcal{N} | \exists c \forall n, g(n) \leq cf(n) + c\}$
  è la classe delle funzioni che crescono al più come f
- $o(f) := \{g : \mathcal{N} \to \mathcal{N} | \forall c \exists n_0 \forall n \geq n_0, cg(n) + c \leq f(n)\}$
  è la classe delle funzioni che crescono meno rapidamente di f
- $\Omega(f) := \{g : \mathcal{N} \to \mathcal{N} | f \in O(g)\}$
  è la classe delle funzioni che crescono almeno quanto f
- $\Theta(f) := O(f) \cap \Omega(f)$
  è la classe delle funzioni che crescono come f

Equivalentemente, $O(f)$ può essere definita come l'insieme delle funzioni g per cui $\exists c, \exists n_0$ tale che $\forall n \geq n_0$

$$g(n) \leq cf(n)$$

Proprietà delle relazioni d'Ordine

- $\forall c > 0, O(cf) = O(f)$
- se $f_1 \in O(g_1)$ e $f_2 \in O(g_2)$ allora $f_1 + f_2 \in O(g_1 + g_2)$
- se $f_1 \in O(g_1)$ e $f_2 \in O(g_2)$ allora $f_1 f_2 \in O(g_1 g_2)$

Inoltre, supposto $g(n) > 0$ per ogni n

- se $lim_{n\to\infty} \frac{f(n)}{g(n)} = l \neq 0$ allora $f \in O(g)$ e $g \in O(f)$
- se $lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ o $lim_{n\to\infty} \frac{g(n)}{f(n)} = \infty$, allora $f \in O(g)$ $g \notin O(f)$
- $lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ se e solo se $f \in o(g)$.

Osservazione: $f \in o(g)$ implica $f \in O(g)$ ma non il viceversa!

Esempi

- $6n^4 - 2n^3 + 5 \in O(n^4)$ (polinomio di grado massimo)
- per ogni costante c, $n^c \in O(c^n)$ ma $c^n \notin O(n^c)$ (l'esponenziale cresce più rapidamente di ogni polinomio)
- per ogni a, b, $log_a(n) \in O(log_b(n))$ (la base del logaritmo è ininfluente)
- se $a < b$, $b^n \notin O(a^n)$ (la base dell'esponente è influente)
- $O(nlog(n)) = O(log(n!)), O(n^n) = O(n!)$

Ricordiamo che

$$e\left(\frac{n}{e}\right)^n \leq n! \leq en\left(\frac{n}{e}\right)^n$$

L'ordine di alcuni noti algoritmi

ordine	nome	esempio
$O(1)$	constante	operazioni su strutture dati finite
$O(\log n)$	logaritmico	ricerca di un elemento in un array ordinato o in un albero di ricerca bilanciato
$O(n)$	lineare	ricerca di un elemento in array disordinati/liste; somma di due interi con la tecnica del riporto
$O(n \log n)$	quasi-lineare	Fast Fourier transform; merge-sort, quicksort (caso medio)
$O(n^2)$	quadratico	prodotto di due interi; bubble sort e insertion sort
$O(n^c)$ per $c > 1$	polinomiale	parsing di grammatiche contestuali; simplesso (caso medio)
$O(c^n)$ per $c > 1$	esponenziale	soluzione del problema del commesso viaggiatore con tecniche di programmazione dinamica; costruire la tabella di verità di una proposizione
$O(n!)$	fattoriale	soluzione del problema del commesso viaggiatore mediante ricerca esaustiva

Grafi

Un grafo finito è una coppia (V, E) dove

- V è un insieme finito di vertici
- $E \subseteq V \times V$ è una relazione che definisce l'insieme degli archi.

Un grafo si dice non orientato quando la relazione E è simmetrica e irriflessiva.

Sia $G = (V, E)$ un grafo.

- due vertici $u, v \in V$ sono adiacenti se esiste un arco $(u, v) \in E$.
- un cammino è una sequenza di vertici dove tutte le coppie di vertici consecutivi sono adiacenti.
- un cammino è semplice se tutti i vertici sono distinti.
- un ciclo è un cammino semplice il cui ultimo vertice è adiacente al primo.

Grafi - Definizioni

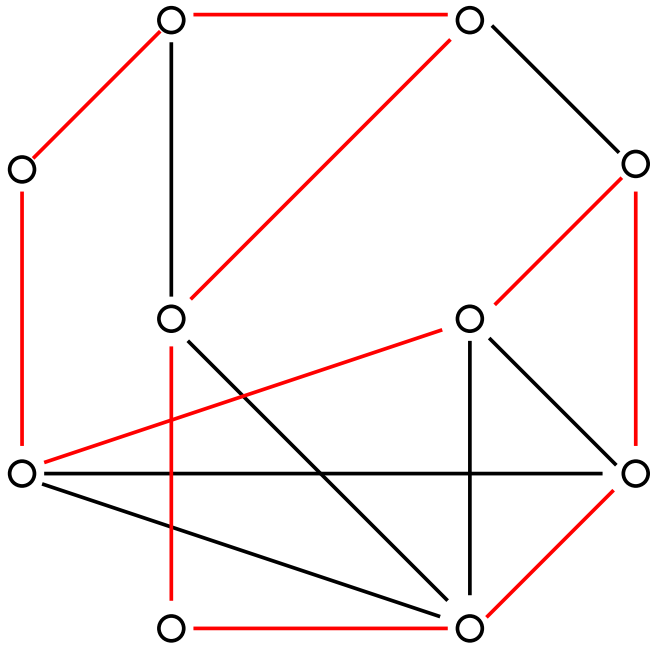
Sia $G = (V, E)$ un grafo.

- un cammino (ciclo) Hamiltoniano in G è un cammino (ciclo) che comprende tutti i vertici del grafo.
- un ricoprimento di vertici per G è un sottoinsieme $V_0 \subseteq V$ tale che ogni arco $e \in E$ ha almeno una estremità in V_0.
- G è n-colorabile, se esiste una funzione di colorazione $col : V \to c_1, \ldots, c_n$ tale che vertici adiacenti hanno colori diversi, ovvero

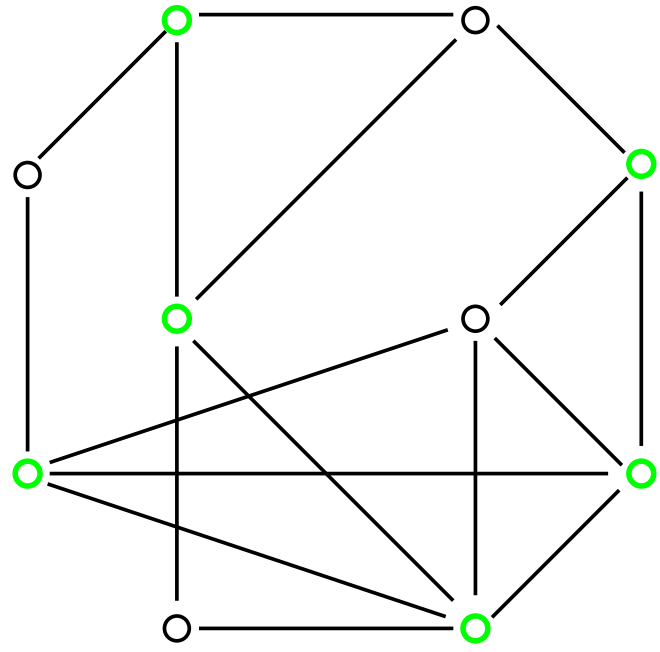
$$(u, v) \in E \Rightarrow col(u) \neq col(v)$$

- G è completo se ogni coppia di nodi distinti è connessa da un arco
- una cricca (clique) di G è un suo sottografo completo $G' = (V', E')$, ovvero $V' \subseteq V$ e $E' = V' \times V' \subseteq E$.
- un insieme indipendente in G è un sottoinsieme di vertici $V' \subseteq V$ tale che per ogni coppia di vertici $u, v \in V' \Rightarrow (u, v) \notin E$.

Grafi - Esempi

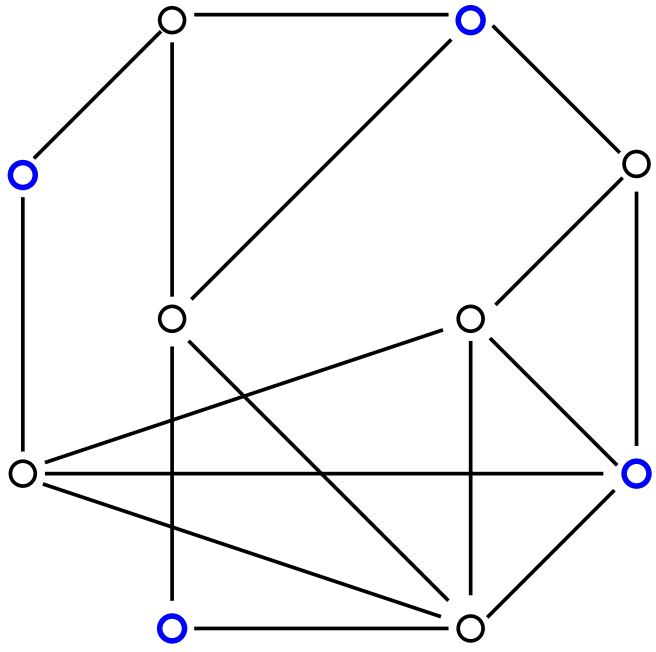


Cammino Hamiltoniano

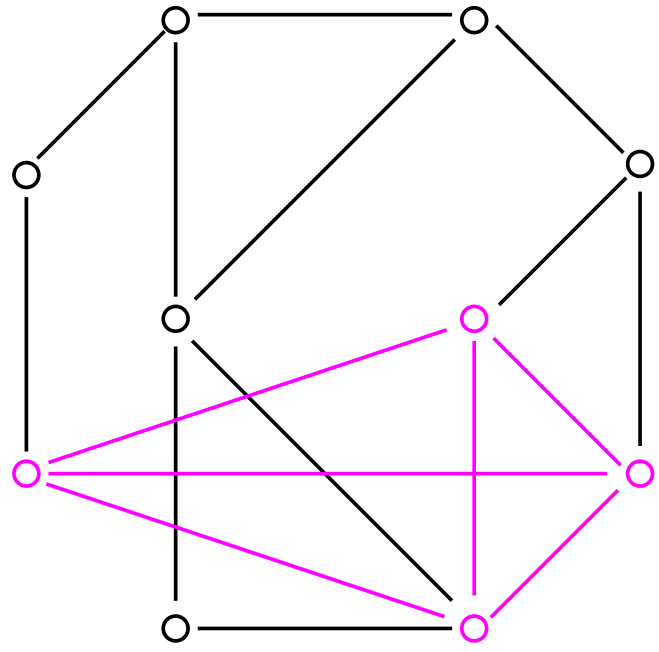


Ricoprimento

Grafi - Esempi



Insieme indipendente



Cricca

Osservazioni

Sia $G = (V, E)$ un grafo.

- V è un caso degenere di ricoprimento. Inoltre se R è un ricoprimento, ogni suo soprainsieme lo è. Siamo interessati a trovare **ricoprimenti minimi**.
- Per ogni $v \in V$, $\{v\}$ è un caso degenere di insieme indipendente (come anche l'insieme vuoto). Ogni sottoinsieme di un insieme indipendente è ancora indipendente. Siamo interessati a trovare **insieme indipendenti massimi**.
- Per ogni $v \in V$, $\{v\}$ è un caso degenere di cricca (come anche l'insieme vuoto). Ogni coppia di nodi $\{u, v\}$ connessi da un arco formano una cricca. Ogni sottoinsieme di una cricca è ancora una cricca. Siamo interessati a trovare **cricche massime**.

Rappresentazione dei grafi

La matrice di adiacenza M_G di un grafo $G = (V, E)$ è la matrice definita nel modo seguente:

$$M_G(u, v) = 1 \Leftrightarrow (u, v) \in E$$

Se, $|V| = n$, la rappresentazione del grafo è al più quadratica in n.

Se la matrice è sparsa, altre rappresentazioni possono essere più convenienti.

Raggiungibilità

Dati due vertici u e v di un grafo, determinare se esiste un cammino da u a v

Raggiungibilità

Dati due vertici u e v di un grafo, determinare se esiste un cammino da u a v

Partizioniamo i vertici V in tre sottoinsiemi D (done) C (current) T (todo).

1. Inizialmente $D = \emptyset$, $C = \{u\}$ e $T = V \setminus \{u\}$
2. Se $v \in C$ abbiamo trovato il cammino e terminiamo con successo;
3. Se $C = \emptyset$ terminiamo con fallimento;
4. Selezioniamo un vertice $x \in C$, e consideriamo i vertici ad esso adiacenti Ad(x); poniamo $D = D \cup \{x\}$, $C := (C \cup (Ad(x) \cap T)) \setminus \{x\}$, $T = T \setminus Ad(x)$
5. ripetiamo dal passo 2.

Raggiungibilità

Dati due vertici u e v di un grafo, determinare se esiste un cammino da u a v

Partizioniamo i vertici V in tre sottoinsiemi D (done) C (current) T (todo).

1. Inizialmente $D = \emptyset$, $C = \{u\}$ e $T = V \setminus \{u\}$
2. Se $v \in C$ abbiamo trovato il cammino e terminiamo con successo;
3. Se $C = \emptyset$ terminiamo con fallimento;
4. Selezioniamo un vertice $x \in C$, e consideriamo i vertici ad esso adiacenti Ad(x); poniamo $D = D \cup \{x\}$, $C := (C \cup (Ad(x) \cap T)) \setminus \{x\}$, $T = T \setminus Ad(x)$
5. ripetiamo dal passo 2.

Ogni arco del grafo è visitato una sola volta; se $|V| = n$, la complessità è $O(n^2)$($equivalentemente$, $O(|G|)$).

2-colorabilità

Determinare se un grafo è 2-colorabile

2-colorabilità

Determinare se un grafo è 2-colorabile

Siano $\{0, 1\}$ i due colori. Partizioniamo i vertici V in tre sottoinsiemi D (done) C (current) T (todo). I vertici in $D \cup C$ sono già colorati; quelli in T da colorare. Si fissi un vertice u del grafo in modo aribtrario e gli si attribuisca un colore (ad esempio $col(u) = 0$).

1. Inizialmente $D = \emptyset$, $C = \{u\}$ e $T = V \setminus \{u\}$.
2. Se $T = \emptyset$ l'algoritmo termina con successo.
3. Altrimenti, selezioniamo un vertice $x \in C$ (se $C = \emptyset$ - G non è connesso - si prende un nuovo elemento di T e lo si colora arbitrariamente) e consideriamo i vertici ad esso adiacenti $Ad(x)$. Per ogni $v \in Ad(x)$,
   - se $v \in D \cup C$, verifichiamo che $col(v) = \overline{col(x)}$ e terminiamo con fallimento altrimenti
   - se $v \in T$, poniamo $col(v) = \overline{col(x)}$ e rimuoviamo v da T e lo aggiungiamo a C
4. rimuoviamo x da C, lo aggiungiamo a D e ripetiamo dal passo 2.

2-colorabilità

Determinare se un grafo è 2-colorabile

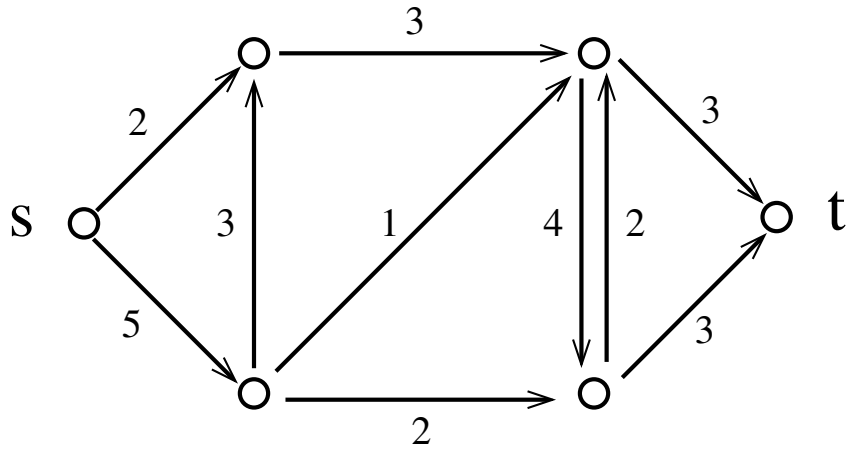
Siano $\{0, 1\}$ i due colori. Partizioniamo i vertici V in tre sottoinsiemi D (done) C (current) T (todo). I vertici in $D \cup C$ sono già colorati; quelli in T da colorare. Si fissi un vertice u del grafo in modo aribtrario e gli si attribuisca un colore (ad esempio $col(u) = 0$).

1. Inizialmente $D = \emptyset$, $C = \{u\}$ e $T = V \setminus \{u\}$.
2. Se $T = \emptyset$ l'algoritmo termina con successo.
3. Altrimenti, selezioniamo un vertice $x \in C$ (se $C = \emptyset$ - G non è connesso - si prende un nuovo elemento di T e lo si colora arbitrariamente) e consideriamo i vertici ad esso adiacenti $Ad(x)$. Per ogni $v \in Ad(x)$,
   - se $v \in D \cup C$, verifichiamo che $col(v) = \overline{col(x)}$ e terminiamo con fallimento altrimenti
   - se $v \in T$, poniamo $col(v) = \overline{col(x)}$ e rimuoviamo v da T e lo aggiungiamo a C
4. rimuoviamo x da C, lo aggiungiamo a D e ripetiamo dal passo 2.

Ogni arco del grafo è visitato una sola volta; se $|V| = n$, la complessità è $O(n^2)(equivalentemente, O(|G|))$.

Flusso massimo

Una rete N è un grafo orientato con una sorgente s, un pozzo t, e una capacità $c(u, v)$ associata ad ogni arco.



Un flusso in N è una funzione $f(u, v)$ che ad ogni arco (u, v) associa un intero positivo tale che

- $f(u, v) \leq c(u, v)$
- la somma dei flussi entranti in ogni nodo (a parte s e t deve essere uguale alla somma dei flussi uscenti

Il problema consiste nel determinare il flusso massimo dalla sorgente al pozzo.

"Sottrarre" un flusso da una rete

Data una rete N e un flusso f relativa ad essa, definiamo una nuova rete $N \setminus f$ che ha gli stessi vertici di N e archi e capacità definite nel modo seguente:

- decrementiamo la capacità di ogni arco per una quantità pari al flusso e rimuoviamo gli archi con capacità nulla:

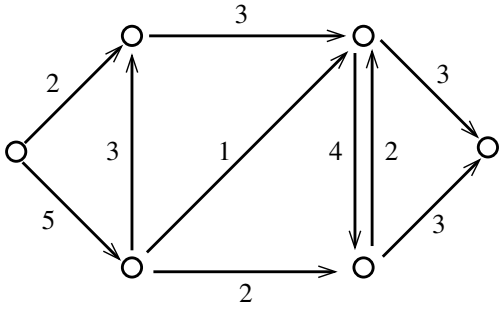
$$c'(i,j) = c(i,j) - f(i,j)$$

- aumentiamo la capacità dell'arco (j,i) per una quantità pari al flusso $f(i,j)$, o aggiungiamo un nuovo arco di capacità $f(i,j)$ se tale arco non esisteva:

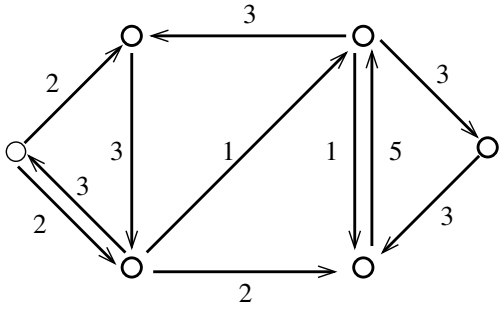
$$c'(j,i) = c(j,i) + f(i,j)$$

L'ultima regola permette di ridiminuire il flusso da i a j per una quantità pari a $f(i,j)$ considerando un flusso opposto

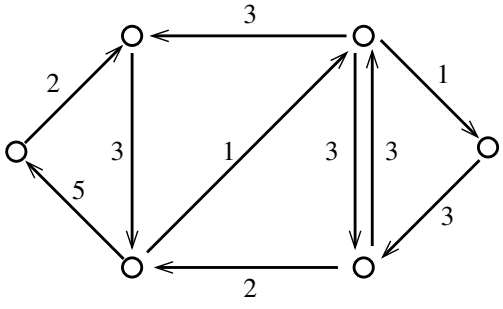
Esempio (1)



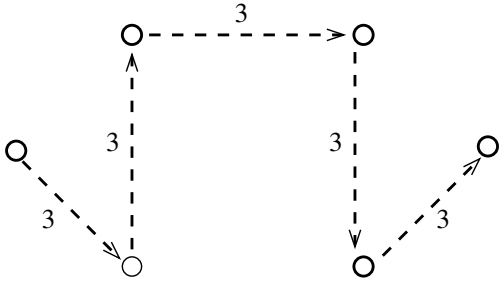
(a)



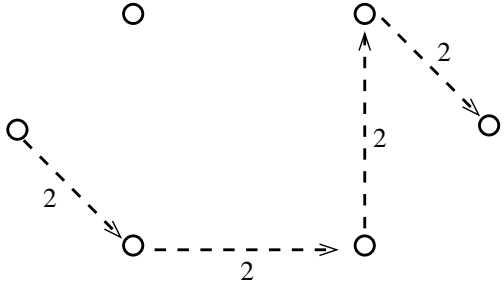
(c)



(e)

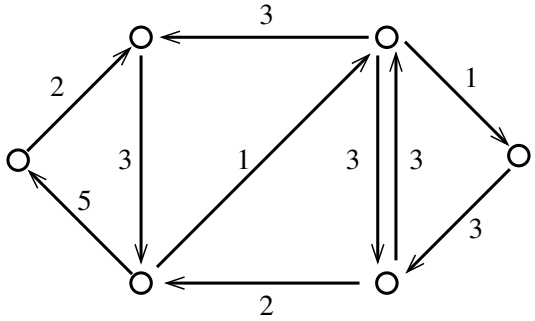


(b)

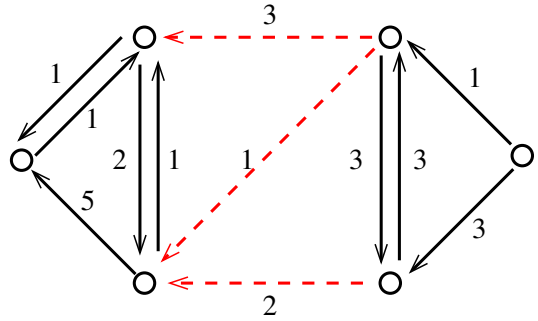


(d)

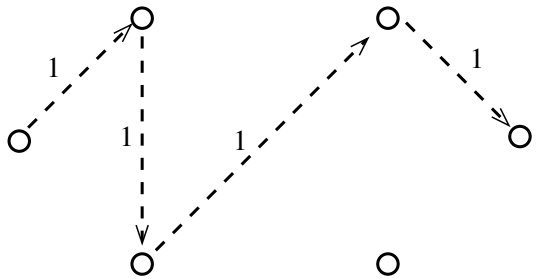
Esempio (2)



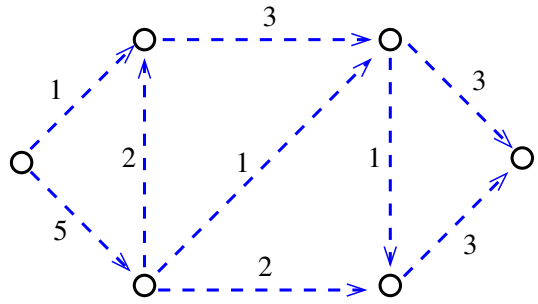
(e)



(g)



(f)



(h)

L'algoritmo

Algoritmo

1. si parte con un flusso $maxf$ inizialmente nullo.
2. si cerca un cammino da s a t nella rete N e si considera il flusso f lungo tale cammino determinato dalla capacità massima dei suoi archi; se tale cammino non esiste si restituisce $maxf$.
3. Si pone $maxf := maxf + f$; $N := N \setminus f$ e si ripete dal passo 2.

L'algoritmo

Algoritmo

1. si parte con un flusso $maxf$ inizialmente nullo.
2. si cerca un cammino da s a t nella rete N e si considera il flusso f lungo tale cammino determinato dalla capacità massima dei suoi archi; se tale cammino non esiste si restituisce $maxf$.
3. Si pone $maxf := maxf + f$; $N := N \setminus f$ e si ripete dal passo 2.

Complessità

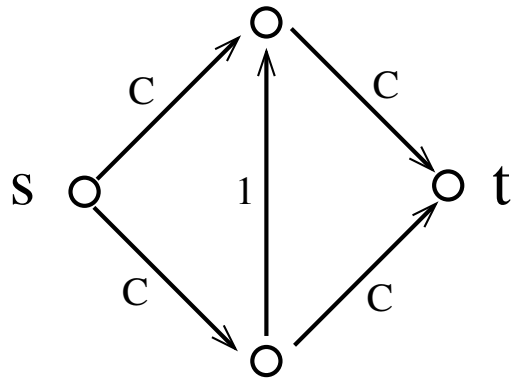
Osserviamo innanzitutto che se C è la massima capacità degli archi, il flusso massimo è sicuramente inferiore a nC in quanto ci sono meno di n archi che partono dalla sorgente.

- la ricerca del cammino costa $O(n^2)$
- il flusso aumenta ad ogni iterazione del ciclo; dunque viene ripetuto al più nC volte.

In conclusione, la complessità è $O(n^3 C)$.

Warning

L'algoritmo precedente dipende in modo lineare da C, e dunque potrebbe dipendere in modo esponenziale rispetto alla descrizione della capacità degli archi della rete.



Caso pessimo

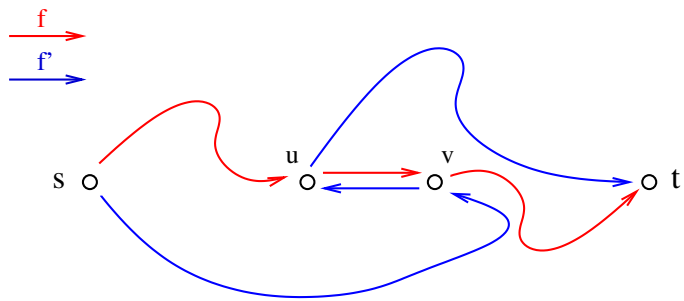
È possibile ovviare al problema selezionando ad ogni iterazione il *cammino più corto* da s a t. Questo fa si che un arco risulterà essere un *collo di bottiglia* in un numero limitato di casi, permettendo di maggiorare il numero di iterazioni con n^3. Dunque la complessità dell'algoritmo è $O(n^5)$.

Dettagli

Un collo di bottiglia è un arco su di un cammino da s a t con capacità minima.

Remark 1: La distanza di un nodo u da s non può decrementare passando da N a $N \setminus f$: $d_{N \setminus f}(u) \geq d_N(u)$.

Remark 2: Se l'arco (u, v) è il collo di bottiglia del flusso f allora se (u, v) verrà mai attraversato nuovamente da un flusso successivo f' (necessariamente nella direzione opposta) la distanza $d_{N'}(u)$ di u da s sarà aumentata.
Infatti, $d_{N'}(u) > d_{N'}(v) \geq d_N(v)$; se $d_{N'}(u) = d_N(u)$ avremmo $d_N(u) > d_N(v)$ contraddicendo la minimalità di f.



Le distanze sono maggiorate dal numero dei nodi, dunque ogni ogni arco può essere un collo di bottiglia al più n volte, e il numero dei flussi è al più $O(n \cdot |E|) \leq O(n^3)$.

Problemi decisionali

La 2-colorazione e la raggiungibilità sono esempi di problemi di decisione, cioè problemi che richiedono una risposta booleana (definiscono dunque un linguaggio).

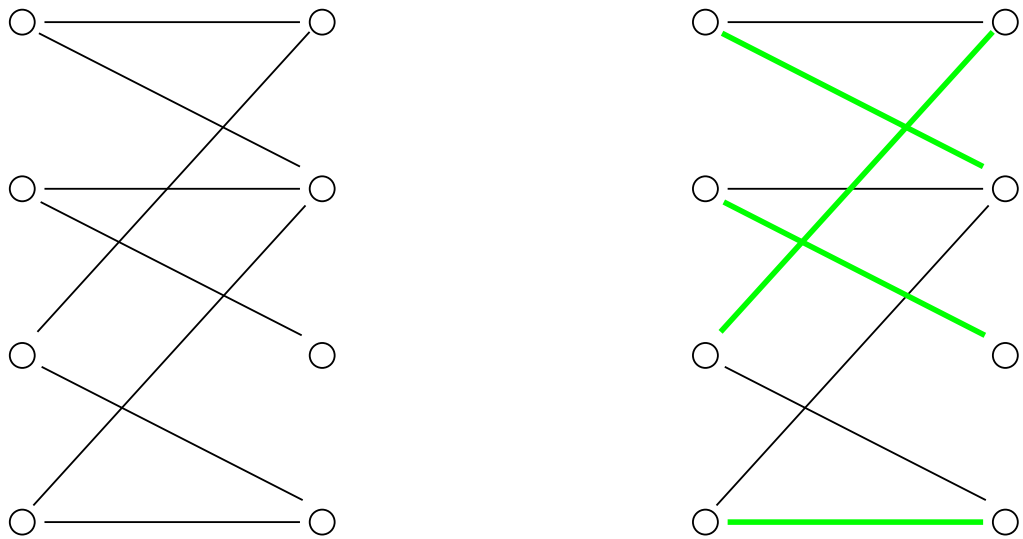
Il problema del flusso massimo è un tipico esempio di problema di ottimizzazione, cioè un problema che richiede la scelta della migliore soluzione tra un insieme di risposte ammissibili rispetto ad una data funzione di costo.

È possibile fornire una versione decisionale di un problema di ottimizzazione, semplicemente chiedendo se esiste una soluzione non peggiore di un valore prefissato. Spesso i due problemi hanno complessità comparabili (è possibile determinare la complessità dell'uno in funzione dell'altro).

Matching bipartito

Un grafo bipartito è una tripla $B = (U, V, E)$ dove U e V sono due insiemi di nodi di uguale cardinalità e $E \subseteq U \times V$ è un insieme di archi.

Un matching è un insieme $M \subseteq E$ che associa ad ogni elemento in U uno e un solo elemento in V

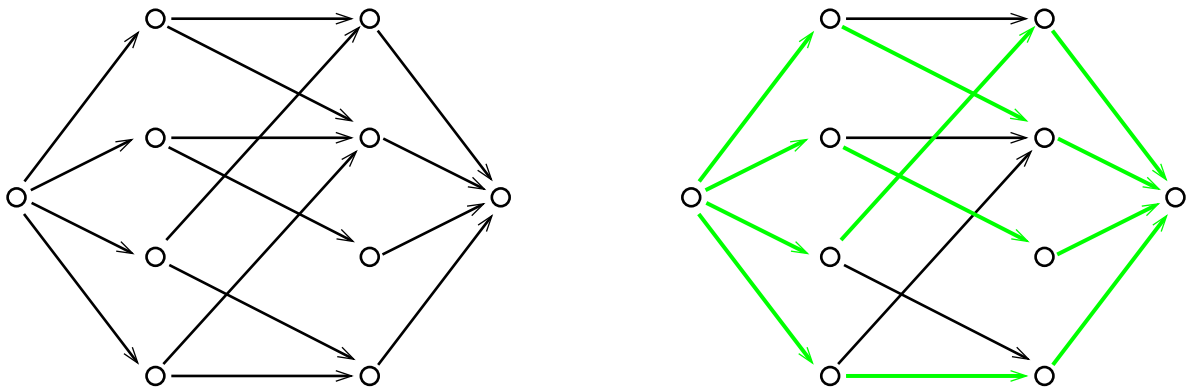


Il problema consiste nel determinare l'esistenza o meno di un matching.

Riduzione a un problema noto

Riduciamo il problema ad un problema di flusso, orientando gli archi da U a V, aggiungendo una sorgente s, un pozzo t, un arco $(s, u), \forall u \in U$ e un arco $(v, t), \forall v \in t$. Tutti gli archi hanno capacità unitaria.

È facile vedere che il grafo bipartito ammette un matching se e solo se la rete così ottenuta ammette un flusso di entità n, dove $n = |U| = |V|$.



Poichè la rete può essere costruita in tempo lineare nella dimensione del grafo, il problema del matching in un grafo bipartito è $O(n^3)$ (si conoscono soluzioni migliori).

Altri esempi di riducibilità

Dato un grafo $G = (V, E)$ e un intero k determinare se

1. esiste un insieme indipendente $V' \subseteq V$ tale che $k \leq |V'|$;
2. esiste un ricoprimento $V' \subseteq V$ tale che $|V'| \leq k$;
3. esiste una cricca $V' \subseteq V$ tale che $k \leq |V'|$;

Questi tre problemi sono riducibili l'uno all'altro.

Insieme indipendente, ricoprimento e cricca

Ricordiamo le definizioni:

- un insieme $V' \subseteq V$ è detto indipendente se

$$\forall u, v \in V' \Rightarrow (u, v) \notin E$$

- un sottoinsieme $V' \subseteq V$ è detto ricoprimento se

$$\forall (u, v) \in E, u \in V' \vee v \in V'$$

- una cricca di G è un sottoinsieme completo $V' \subseteq V$, tale cioè che

$$\forall u, v \in V' \Rightarrow (u, v) \in E$$

Insieme indipendente, ricoprimento e cricca

Ricordiamo le definizioni:

- un insieme $V' \subseteq V$ è detto indipendente se

$$\forall u, v \in V' \Rightarrow (u, v) \notin E$$

- un sottoinsieme $V' \subseteq V$ è detto ricoprimento se

$$\forall (u, v) \in E, u \in V' \vee v \in V'$$

- una cricca di G è un sottoinsieme completo $V' \subseteq V$, tale cioè che

$$\forall u, v \in V' \Rightarrow (u, v) \in E$$

È facile vedere che

- $V' \subseteq V$ è indipendente se e solo se $V \setminus V'$ è un ricoprimento
- $V' \subseteq V$ è indipendente se e solo se V' è una cricca nel grafo $G' = (V, \overline{E})$

Ricerca vs. Verifica

Cercare in modo esaustivo un insieme indipendente (un ricoprimento, o una cricca) di cardinalità k richiede l'esame di un numero di casi pari a

$$\frac{n!}{k!(n-k)!}$$

Per valori di $k \approx n/2$ questa quantità cresce in modo esponenziale in k.

Non è noto se è possibile avere algoritmi polinomiali per risolvere questi problemi.

Al contrario, **verificare** se un sottoinsieme dato di vertici è indipendente (un ricoprimento, o una cricca) richiede un tempo polinomiale nel numero dei vertici.

I problemi che ammettono soluzioni di dimensione polinomiale rispetto all'input e algoritmi polinomiali di verifica della loro correttezza sono detti problemi **NP**.

Certificati

$x \in A?$ $\longrightarrow$

Certificati

$x \in A?$ $\longrightarrow$

$\longleftarrow$ *Si*!

Certificati

$x \in A?$ $\longrightarrow$

$\longleftarrow$ *Si!*

lo puoi dimostrare? $\longrightarrow$

Certificati

$x \in A?$ $\longrightarrow$

$\longleftarrow$ *Si!*

lo puoi dimostrare? $\longrightarrow$

$\longleftarrow$ *certo*: c_x

Certificati

$x \in A?$ $\longrightarrow$

$\longleftarrow$ *Si!*

lo puoi dimostrare? $\longrightarrow$

$\longleftarrow$ *certo*: c_x

nuovo problema: verificare la correttezza del certificato.

$$\langle x, c_x \rangle \textit{ corretto} \Leftrightarrow x \in A$$

NP: problemi di facile verifica

NP come proiezione di P

Un insieme A è in NP se e solo se esiste un insieme B (insieme dei certificati per A) decidibile in tempo polinomiale e un polinomio p tale che per ogni x,

$$x \in A \Leftrightarrow \exists c_x, |c_x| \leq p(|x|) \wedge \langle x, c_x \rangle \in B$$

Osservazioni:

- il certificato deve avere **dimensione polinomiale** in x (se c_x è enorme, il fatto che la sua verifica sia semplice è irrilevante).
- L'esistenza del certificato fornisce una tecnica di soluzione per il problema A: cerco il certificato (generate and test).
  Tuttavia, siccome la dimensione del certificato è polinomiale in x, il numero dei certificati è esponenziale.

Esempio: soddisfacibilità

Data una formula proposizionale, determinare se è soddisfacibile, cioè se esiste una attribuzione di valori di verità alle variabili proposizionali (funzione di valutazione) che rende vera la formula.

Esempio:

- $(P \rightarrow Q) \wedge \neg P \wedge Q$ è soddisfacibile ($P = 0, Q = 1$)
- $(Q \rightarrow P) \wedge \neg P \wedge Q$ è insoddisfacibile

Il numero totale di funzioni di valutazione è 2^n dove n è il numero di variabili proposizionali nella formula.

Esistono pocedimenti migliori (e.g. risoluzione), ma non si conoscono algoritmi polinomiali.

Verificare che una data funzione di valutazione rende vera la formula ha un costo lineare nella dimensione della formula.

Esempio: il problema del commesso viaggiatore

Sono date n città e una distanza intera $d_{ij} = d_{ji} > 0$ tra ciascuna esse. Il problema consiste nel determinare il ciclo di lunghezza minima, ovvero una permutazione π tale la quantità

$$d_\pi = \sum_{i=1}^{n} d_{\pi(i),\pi(i+1)}$$

è la più piccola possibile.

La versione decisionale consite nel determinare se esiste un ciclo di lunghezza inferiore o uguale ad una distanza data d.

Il numero complessivo delle permutazioni è $\frac{(n-1)!}{2}$. È possibile migliorare il bound (ad un "semplice" esponenziale) con tecniche di programmazione dinamica, ma non si conoscono algoritmi polinomiali.

D'altra parte, verificare che un dato cammino π ha una lunghezza $d_\pi \leq d$ richiede un tempo lineare in n.

Esempio: il problema dello zaino (knapsak)

Dato uno zaino di volume V e n oggetti $O = \{1, \ldots, n\}$ cascuno con un volume $v(i)$ (con V, v_i interi positivi), determinare se esiste un sottoinsieme $I \subseteq O$ tale che

$$V = \sum_{i \in I} v(i)$$

Ad esempio, dati 7 oggetti di volume $\{6, 7, 8, 9, 10, 11, 13\}$ è possibile riempire uno zaino di dimensione $V = 52$?

Una ricerca esaustiva impone di considerare tutti i sottoinsiemi, ovvero 2^n casi.

Anche in questo caso, verificare che un dato sottoinsieme rispetta la condizione voluta richiede un costo al più lineare nel numero degli oggetti.

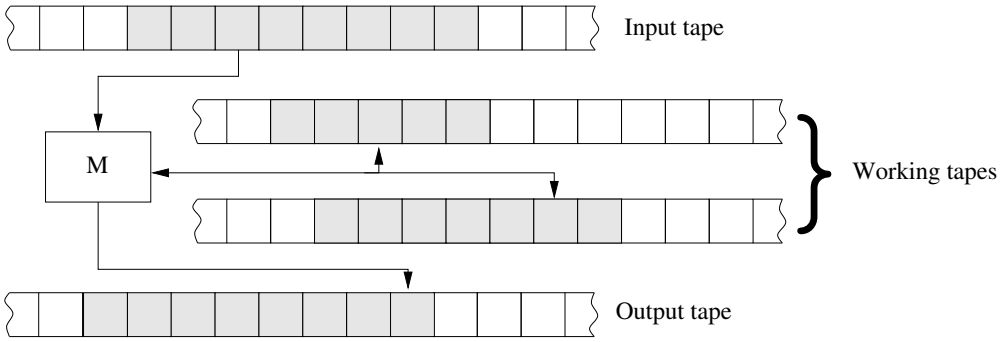
Riassumendo:

- La classe P è la classe dei problemi che ammettono un algoritmo rapido di ricerca della soluzione
- La classe NP è la classe dei problemi che ammettono un algoritmo rapido di verifica della correttezza soluzione

Lezioni 4-5: Classi deterministiche di complessità

- Il Modello di calcolo
- Complessità in tempo e spazio per computazioni deterministiche
- Influenza del modello di calcolo sugli aspetti di complessità
- Relazioni tra tempo e spazio

La Macchina di Turing



Hardware

- nastri di memoria illimitati, divisi in celle di dimensione fissata.
  Ogni cella può contenere un carattere di un alfabeto dato, compreso un carattere *b* (bianco) di inizializzazione.
- una testina di lettura mobile per ogni nastro
- un automa di controllo a stati finiti.

Operazioni elementari

- leggere e scrivere il carattere individuato dalla testina (il nastro di input è di sola lettura, quello di output di sola scrittura)
- spostare la testina di una posizione verso destra o verso sinistra (sui nastri di input/output la testina può solo muoversi verso destra)
- modificare lo stato interno dell'automa

La Macchina di Turing: definizione formale

Una Macchina di Turing (multi-tape, deterministica) è una tupla $\langle Q, \Gamma, b, \Sigma, k, \delta, q_0, F \rangle$ dove

- Q è un insieme finito di stati
- Γ è l'alfabeto finito del nastro
- $b \in \Gamma$ è il carattere bianco
- $\Sigma \subseteq \Gamma \setminus \{b\}$ è l'insieme dei caratteri di input/output
- $k \geq 1$ è il numero dei nastri
- $q_0 \in Q$ è lo stato iniziale
- $F \subseteq Q$ è l'insieme degli stati finali (o di accettazione)
- $\delta : Q \setminus F \times \Gamma^k \to Q \times \Gamma^k \times \{L, R\}^k$ è la funzione di transizione.

L (left) e R (right) denotano le possibili mosse della testina.

Convenzioni di input-output

- **input:** si suppone che il nastro di input sia inizializzato con la stringa di input (un carattere per ogni cella); la testina è posizionata sul primo carattere dell'input; tutte le altre celle del nastro sono inizializzate col carattere speciale b.
- **output:** nel momento in cui la macchina si arresta l'output è la più lunga stringa di caratteri in Σ (in particolare, senza b) alla sinistra della testina sul nastro di output
- **tapes:** se $k > 1$ il nastro 1 è un nastro di sola lettura (input tape); se $k > 2$ il nastro k è un nastro di sola scrittura (output tape)
- **spostamenti:** le sole mosse consentite sui nastri di input/output sono spostamenti verso destra.

Configurazioni istantanee

Una configurazione è una descrizione dello stato della computazione ad un dato istante della computazione. Questa è definita come una tupla

$$q, (\sigma_1, \tau_1), \ldots (\sigma_k, \tau_k)$$

dove q è lo stato dell'automa e σ_i, τ_i sono due stringhe di caratteri che descrivono la porzione non (definitivamente) bianca del nastro i alla sinistra e alla destra della relativa testina. Il carattere in lettura è il primo carattere di τ_i.

La computazione avviene per passi discreti: una transizione tra due configurazioni è una relazione $\vdash$ governata dalla funzione di transizione:

$$(q, (\sigma_1 b_1, a_1 \tau_1), \ldots (\sigma_k b_k, a_k \tau_k)) \vdash (q', (\sigma_1 \beta_1, \alpha_1 \tau_1), \ldots (\sigma_k \beta_k, \alpha_k \tau_k))$$

se

- $\delta(q, a_1, \ldots, a_k) = (q', a'_1, \ldots, a'_k, D_1, \ldots, D_k)$
- se $D_i = R$ allora $\beta_i = b_i a'_i$ e $\alpha_i = \epsilon$
- se $D_i = L$ allora $\beta_i = \epsilon$ e $\gamma_i = b_i a'_i$

Il nastro può essere esteso "on demand" con caratteri bianchi se necessario.

Computazioni

La relazione $\vdash^*$ denota la chiusura transitiva e riflessiva della relazione $\vdash$.

Una funzione $f : \Sigma^* \to \Sigma^*$ è calcolata da una macchina di Turing M se per ogni α esiste $q_f \in F$ tale che

$$q_0, (\epsilon, \alpha), \ldots (\epsilon, \epsilon) \vdash^* q_f, (\gamma_1, \tau_1), \ldots (\gamma_k, \tau_k)$$

e $f(\alpha)$ è il più lungo suffisso di γ_k appartenente a Σ^*

Classi deterministiche di Complessità

Sia data una macchina di Turing M:

- $\text{time}_M(x)$ è il tempo di esecuzione di M su input x, i.e. il numero di passi richiesti per la computazione
- $\text{space}_M(x)$ è il numero massimo di celle visitate da una qualche testina durante la computazione (per macchine a più nastri si considerano solo i nastri di lavoro)
- $t_M(n) := max\{\text{time}_M(x) : |x| = n\}$
- $s_M(n) := max\{\text{space}_M(x) : |x| = n\}$

Data una funzione $f : N \to N$ introduciamo le seguenti classi di complessità:

- $\text{DTIME}(f) := \{L \subseteq \Sigma^* : \exists M, L = L_M \wedge t_M \in O(f)\}$
- $\text{DSPACE}(f) := \{L \subseteq \Sigma^* : \exists M, L = L_M \wedge s_M \in O(f)\}$

Aggiungeremo un pedice k per esplicitare che M ha k nastri.

Tempo e Spazio

Tempo e spazio

Per ogni $f : N \to N$ si ha:

$$\mathrm{DTIME}(f) \subseteq \mathrm{DSPACE}(f) \subseteq \bigcup_{c \in N} \mathrm{DTIME}(2^{c(log+f)})$$

La prima inclusione vale in quanto la Macchina di Turing ha bisogno di almeno un passo per visitare una nuova cella.

La seconda inclusione vale in quanto il numero di configurazioni della macchina con spazio fissato è finito, e la computazione deve arrestarsi entro un numero di passi pari al più a queste configurazioni (altrimenti si avrebbe un ciclo).

Calcoliamo il numero di configurazioni. Sia $M = \langle Q, \Gamma, b, \Sigma, k, \delta, q_0, F \rangle$ e $L_M \subseteq DSPACE(f)$. Ricordiamo che una configurazione è una tupla

$$q, (\sigma_1, \tau_1), \ldots (\sigma_k, \tau_k)$$

Abbiamo allora, per un oportuno $c \in N$,

$$t_M(n) \leq |Q| \cdot k \cdot |\Gamma|^{s_M(n)} \leq 2^{c(log(n)+f(n))+c}$$

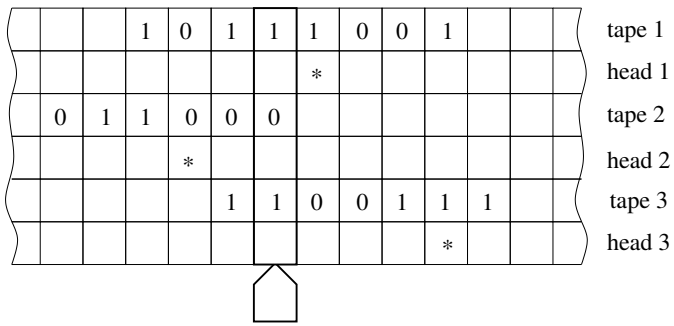
Riduzione dei nastri

Riduzione ad un nastro

Per ogni $f : N \to N$ si ha:

$$DSPACE(f) \subseteq DSPACE_1(f) \wedge DTIME(f) \subseteq DTIME_1(f^2)$$

L'idea è di simulare k nastri in uno solo arricchendo l'alfabeto. Utilizzare altre k "tracce" per rappresentare la posizione della testina sul nastro corrispondente:



Se Σ è l'alfabeto originale, il nuovo alfabeto è $\Sigma' =:= (\Sigma \times \{*, B\})^k$.
La crescita quadratica è dovuta al fatto che ogni passo simulato richiede la ricerca della posizione della testina per ogni nastro: questa ricerca è limitata dalla dimensione del nastro, che a sua volta, per 1., è limitata dal tempo.

Riduzione dei nastri da k a 2

Riduzione a due nastri

Per ogni $f : N \to N$ si ha:

$$DTIME(f) \subseteq DTIME_2(f \cdot \log f)$$

La simulazione in tempo $n\log n$ di una MdT a k nastri su di una con due soli nastri richiede un algoritmo piuttosto intricato.
Le idee principali sono le seguenti:

- anche in questo caso i k nastri sono simulati con tracce multiple su uno dei due nastri (l'altro e' un nastro di lavoro),
- invece di muovere la testina, si ricopiano i contenuti dei nastri.
- il contentuo dei nastri è organizzato in "slot" N_i di dimensione esponenzialmente crescente 2^{i+1}.
- vengono lasciati periodicamente degli slot vuoti che essenzialmente operano come dei buffer
- la complessità in tempo cresce solo di un fattore logaritmico in quanto slot N_i di dimensioni grandi devono essere ricopiati raramente

Riduzione a 2 nastri (invarianti)

Sia C la cella corrente del nastro; indichiamo con R_i e L_i gli slot destri e sinistri sul nastro, ciascuno di dimensione 2^{i+1}. Un simbolo particolare $\square$ indica posizioni vuote negli slots.
Durante l'esecuzione, vengono preservati i seguenti invarianti:

1. Ognuno degli slots è completamente vuoto, completamente pieno oppure pieno esattamente a metà;
2. R_i e L_i sono l'uno pieno e l'altro vuoto, oppure entrambi pieni a metà; il numero totale dei simboli significativi (diversi da $\square$) in $R_i \cup L_i$ è dunque 2^{i+1};
3. il simbolo in C non è mai $\square$

Riduzione a 2 nastri (shift)

L'operazione di shift verso sinistra opera nel modo seguente (simmetricamente per lo shift verso destra):

1. si cerca il più piccolo i_0 tale R_{i_0} non è vuoto (tutti gli L_i per $i < i_0$ saranno pieni, per il secondo invariante);
2. si sposta il primo carattere in C e altri $2^{i_0} - 1$ caratteri negli slots vuoti $R_0, R_1 \ldots R_{i_0-1}$, riempiendo ciascuno a metà;
3. si opera ora sulla parte sinistra, svuotando per metà ciascuno degli slots pieni $L_0, L_1 \ldots L_{i_0-1}$, e mettendo tali caratteri in L_i (che è sicuramente almeno vuoto a metà, sempre per l'invariante 2)
4. si muovo il vecchio simbolo in lettura modificato opportunamente in nello slot L_0

È facile vedere che gli invarianti sono mantenuti dall'operazione di shift.
La complessità della singola operazione di shift è proporzionale alla dimensione delle aree coinvolte e dunque dell'ordine di 2_{i_0}.

Riduzione a 2 nastri (complessità)

Dopo aver eseguito un operazione di shift relativa ad uno slot in posizione i, tutti gli slot (sia destri che sinistri) con indice inferiore a i sono mezzo pieni.

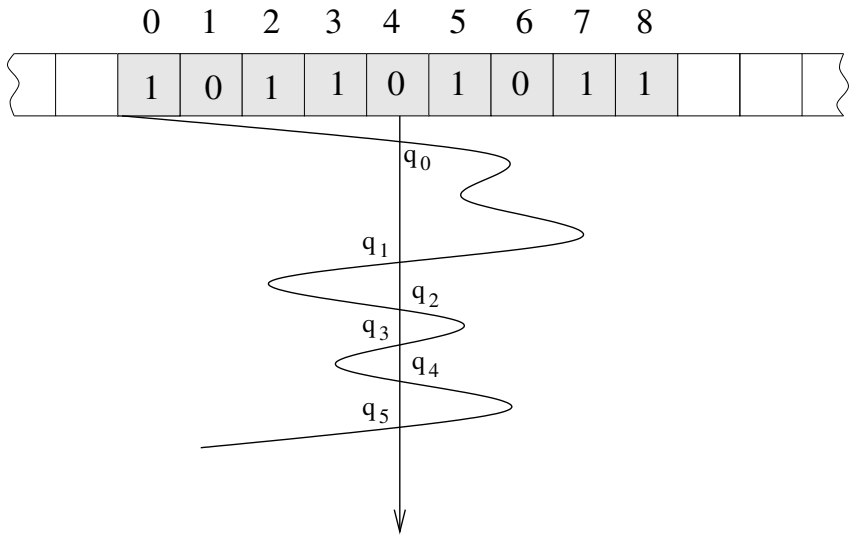
Questo significa che le successive $2^i - 1$ operazioni avranno un indice inferiore a i, ovvero le operazioni di shift con indice i sono al più una frazione pari a $1/2^i$ del numero complessivo di shift.

Se n è il numero di passi (shift) richiesto dalla computazione, allora il massimo indice per gli slots è $log(n)$ e il lavoro complessivo per ogni tape è

$$O(\sum_{i=1}^{log(n)} \frac{n}{2^i} 2^i) = O(nlog(n))$$

Sequenza di attraversamento

Sia $M = \langle Q, \Gamma, b, \Sigma, 1, \delta, q_0, F \rangle$ una macchina di Turing ad un nastro e sia x un input di lunghezza $|x| = n$. Numeriamo in modo progressivo $0, 1, ..., n-1$ le celle che contengono l'input. La sequenza di attraversamento $CS(x, i)$ di x a i è la sequenza (eventualmente infinita) di stati $q_1, q_2, \cdots \in Q$ che occorrono durante la computazione di M su input x al passaggio della testina dalla cella $i-1$ alla cella i o viceversa.



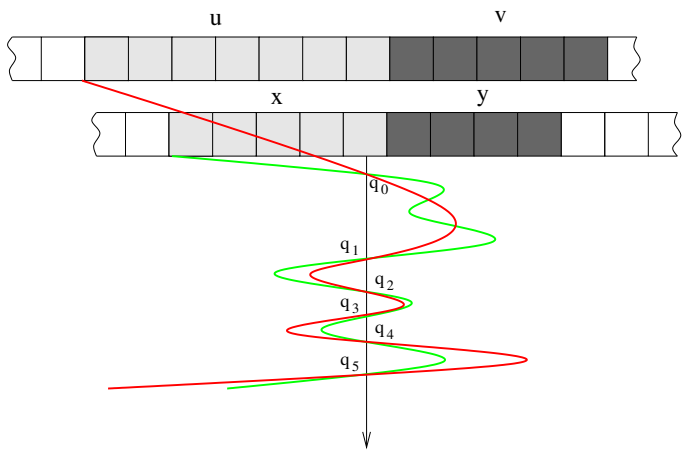
Lemma di attraversamento

Sia M una macchina di Turing a un nastro e siano $u, v, x, y \in \Sigma^*$ tali che $uv \in L_M \Leftrightarrow xy \in L_M$. Supposto

$$CS(uv, |u|) = CS(xy, |x|)$$

allora $uv \in L_M \Leftrightarrow uy \in L_M$.

Dimostrazione. Per simmetria basta dimostrare che $uv \in L_M \Rightarrow uy \in L_M$. Consideriamo la computazione di uv e xy e dividiamo entrambe in corrisponenza della rispettiva posizione di crossing (i.e. $|u|$ e $|x|$). A questo punto accoppiamo la computazione "di sinistra" di uv con quella "di destra" di xy. Questo porta ad una computazione che accetta uy.



Un limite inferiore per la riduzione dei nastri

Sia $L := \{w\#^{|w|}w | w \in \{0,1\}^*\}$. Allora:

1. $L \in DTIME_2(n) \subseteq DTIME_1(n^2)$
2. $L \notin DTIME_1(t)$ per nessun $t \in o(n^2)$

Il primo claim è una semplice verifica.
Per 2., sia M una MdT ad un nastro per cui $L_M = L$; notiamo le cose seguenti:

1. $time_M(w\#^m w) \geq m \cdot l(w)$ dove $l(w) := min_{m \leq i < 2m} |CS(w\#^m w, i)|$, in quanto ogni attraversamento richiede almeno un passo
2. ogni crossing sequence descrive in modo univoco una stringa $w \in L$. Supponiamo infatti che $CS(w\#^{|w|}w, i) = CS(w'\#^{|w'|}w', j)$ per $|w| \leq i < 2|w|$ e $|w'| \leq j < 2|w'|$. Allora per il lemma di attraversamento la stringa $w\#^m w' \in L$ (per m opportuno), che implica $w = w'$.
3. è possibile descrivere una qualunque stringa w come quell'unica stringa di lunghezza minore o uguale a m riconosciuta da M con una crossing sequence data (e.g. minima).

   La dimensione di questa descrizione è al più

$$C_M \frac{time_M(w\#^m w)}{m} + 2log(m)$$

Un limite inferiore per la riduzione dei nastri (2)

La complessità di Kolmogorov di una stringa w è dunque almeno pari a

$$C_M \frac{time_M(w\#^m w)}{m} + 2log(m)$$

Ma se w è un numero random (ed esistono infiniti numeri random), allora $k(w) \geq |w| = m$, e quindi

$$C_M \frac{time_M(w\#^m w)}{m} + 2log(m) \geq m$$

da cui

$$t_M(3m) \geq time_M(w\#^m w) \geq (1/C_M)(m^2 - 2mlog(m))$$

ed in particolare

$$t_M \notin o(n^2)$$

Macchine ad accesso casuale

Una macchina ad accesso casuale (RAM) dispone di un insieme numerabile di locazioni di memoria in grado di contenere interi di dimensione arbitraria. Il comportamento della macchina è descritto da una sequenza di istruzioni (programma):

istruzione	**operando**	**semantica**
READ	j	$r_0 := i_j$
READ	$\uparrow j$	$r_0 := i_{r_j}$
STORE	j	$r_j := r_0$
STORE	$\uparrow j$	$r_{r_j} := r_0$
LOAD	x	$r_0 := x$
ADD	x	$r_0 := r_0 + x$
SUB	x	$r_0 := r_0 - x$
HALF		$r_0 := div\ r_0\ 2$
JUMP	j	$k := j$
JPOS	j	if $r_0 > 0$ then $k := j$
JZERO	j	if $r_0 = 0$ then $k := j$
HALT		k := 0

- r_i è il contenuto della locazione i (la locazione 0 è utilizzata come accumulatore)
- i_j è l'i-esimo input
- k è il program counter (inizializzato alla prima istruzione)

Modello di costo per RAM

Ogni istruzione ha costo unitario, indipendentemente dal fatto che opera su interi di dimensione arbitraria!

Warning Se l'input è $I = (i_1, \ldots, i_k)$ la dimensione non è k, ma

$$\ell(I) = \sum_{j=1}^{k} |bin(i_j)|$$

Simulazione di una RAM mediante MdT

Simuliamo una RAM con una MdT a 6 nastri:

- il primo nastro contiene l'input
- il secondo nastro descrive il contenuto dei registri sotto forma di una lista di coppie $i : r_i$ separate da caratteri bianchi e concluse con $\triangleleft$. Quando un registro è aggiornato, viene cancellato e riappeso in coda.
- il terzo nastro contiene il program counter
- i tre nastri rimanenti sono utilizzati per le istruzioni aritmetiche (due per gli operandi e il terzo per il risultato)

Gli stati della MdT sono suddivisi in m gruppi, uno per ogni singola istruzione del programma da simulare.

Crescita del contenuto dei registri

Lemma

Al passo t di una computazione su input I, il contenuto di ogni registro ha una dimensione minore o uguale a $t + \ell(I) + c$, dove c è la dimensione della massima costante che appare nel programma.

La dimostrazione si effettua per induzione su t. Al passo 0 il lemma è evidente. Al passo induttivo si procede per casi sull'ultima istruzione eseguita. Le istruzioni che possono aumentare sensibilmente il contenuto dei registri sono quelle aritmetiche, ma anche somma e sottrazione al più richiedono un solo carattere extra per il riporto finale, incrementando la dimensione dei risultati di una sola unità.

Si noti che il lemma non varrebbe se ad esempio tra le istruzioni aritmetiche si comprendesse anche la moltiplicazione.

Costo della simulazione

RAM vs. MdT

Ogni programma RAM P con complessità in tempo $t_P(n)$ può essere simulato da una MdT in tempo $O(t_P(n)^3)$.

Il costo di esecuzione di ogni singola istruzione è dovuto principalmente al costo di reperimento degli operandi sul nastro 2 di simulazone della memoria.

Per ogni operando sono necessarie al più due scansioni del nastro (nel caso di riferimenti indiretti).

Il nastro ha dimensione $O(t_P(n)^2)$, in quanto contiene al più $O(t_P(n))$ coppie, ciascuna di dimensione $O(t_P(n))$.

Dunque, la simulazione di ogni istruzione ha un costo pari a $O(t_P(n)^2)$, e la computazione complessiva è in $O(t_P(n)^3)$.

Conclusioni

La complessità computazionale dipende dal modello di calcolo e dalla definizione delle funzioni di costo.

Definizioni "ragionevoli" delle funzioni di costo permettono comunque la mutua simulazione di modelli differenti in tempi polinomiali.

Lezioni 6-8: Gerarchie in tempo e spazio

- Macchine di Turing normalizzate
- Macchina di Turing universale
- Funzioni costruibili di complessità
- Il Teorema della Gerarchia in tempo e spazio
- Alcuni risultati di separabilità

Macchine di Turing normalizzate

Diremo che una macchina di Turing è normalizzata se è della forma

$$\langle Q = \{0, 1, \ldots, n\}, \Gamma = \{0, 1, 2\}, b = 2, \Sigma = \{0, 1\}, k, \delta, q_0 = 0, F = \{n\}\rangle$$

Lemma

Per ogni MdT ad un nastro sull'alfabeto $\Sigma = \{0, 1\}$ ne esiste una normalizzata M' tale che $L_M = L_{M'}$, $t_{M'} \in O(t_M)$ e $s_{M'} \in O(s_M)$.

Sia $M = \langle Q_M, \Gamma_M, b_M, \Sigma, 1, \delta_M, q_{M_0}, F_M\rangle$. Ogni simbolo in Γ_M può essere codificato da una sequenza di lunghezza l opportuna di simboli in $\Gamma_{M'} = \{0, 1, 2\}$ (riservando 2^l come codifica di b). $\delta_{M'}$ Si ottiene da δ_M rimpiazzando ogni operazione su di un simbolo mediante una piccola sequenza di operazioni sui rispettivi blocchi. Possiamo numerare gli stati in modo crescente, ed assumere senza perdita di generalità di avere un solo stato finale. Il tempo e lo spazio addizionali richiesti dalla macchina normalizzata sono approssimativamente l volte quelli della macchina orginaria.

Remark: Operando in modo inverso, ovvero arricchendo l'alfabeto del nastro, abbiamo un fenomeno di speed-up lineare.

Macchina di Turing universale

Esiste un encoding totale e suriettivo M_x delle MdT normalizzate in stringhe $x \in \{0,1\}^*$, tale che esiste $u \in \{0,1\}^*$ e $c \in N$ per cui:

1. $L(M_u) = \{\langle x, y\rangle | y \in L(M_x)\}$
2. $\forall x, y, time_{M_u}\langle x, y\rangle \leq c \cdot (|x|^2 + |x| \cdot time_{M_x}(y))^2 + c$.

Ogni transizione $\delta(i,j) = (k,l,m)$ può essere codificata da una stringa della forma

$$w = 0^{i+1}10^{j+1}10^{k+1}10^{l+1}10^{m+1}$$

dove $m = 0$ per L e $m = 1$ per R. Sicome Q è finito, la funzione di transizione è codififcata da un'unica stringa $x = 1w_011w_111...11w_h$ che identifica in modo univoco una MdT M_x. Alle stringhe $x \in \{0,1\}^*$ che non sono della forma precedente associamo arbitrariamente una MdT M_x che riconosce il linguaggio vuoto.
La MdT universale opera nel modo seguente:

Macchina di Turing universale

Utilizziamo per semplicità una macchina di Turing M a 5 nastri. Dato l'input $\langle x, y \rangle$ la macchina ricopia innanzitutto x e y su nastri di lavoro (nastro del programma e nastro della computazione). Quindi si verifica che x sia un codice valido rispetto alla codifica precedente (parsing). In caso negativo, l'input viene rifiutato; altrimenti M comincia a simulare M_x su input y, utilizzando un ulteriore nastro per memorizzare lo stato interno di M_x (nastro dello stato). La simulazione richiede ad ogni passo la scansione della funzione di transizione per ricercare la mossa corretta da effettuare sul nastro della computazione. Infine il risultato della computazione viene ricopiato sul nastro di output.
Complessivamente:

- Copiare l'input richiede un tempo $O(|x|) + O(|y|)$
- Verificare che x sia valido richiede un tempo $O(|x|^2)$
- La simulazione di ogni passo di M_x richiede un tempo $O(|x|)$, e dunque l'intera computazione un tempo $O(|x| \cdot time_{M_x}(y))$
- Ricopiare l'output richiede un tempo $O(time_{M_x}(y))$

Per il teorema della riduzione dei nastri, la precedente MdT può essere simulata da una MdT ad un nastro con la complessità richiesta.

Macchina di Turing universale a due nastri

Nel caso di una macchina di Turing universale a due nastri, lo slow down invece di essere quadratico si riduce ad una fattore $\log n$.

Per quanto riguarda lo spazio, l'interpetazione mediante macchina universale non introduce modifiche apprezzabili (sia con un nastro che con due nastri), ovvero esiste una costante c tale che

$$space_{M_u}\langle x, y\rangle \leq c \cdot (|x| + space_{M_x}(y)) + c$$

Funzioni costruibili

Una funzione $f : N \to N$ è detta costruibile in tempo se esiste una MdT M sull'alfabeto $\Sigma = \{0\}$ che calcola una funzione f_M per cui:

- per ogni n, $f_M(0^n) = 0^{f(n)}$,
- $t_M \in O(f)$

Analogamente, f è costruibile in spazio se valgono le stesse condizioni con s_m al posto di t_M.

Lemma

Ogni funzione costruibile in tempo è costruibile in spazio.

Esempi Le seguenti funzioni sono costruibili in tempo:

$$n, n \cdot \log(n), n^c, c^n, n \cdot \sqrt{n}$$

La funzione $f(n) = log(n)^c$ è costruibile in spazio ma non in tempo.

Il Teorema della gerarchia (spazio)

I teoremi della gerarchia riflettono l'idea intuitiva che disponendo di una maggiore quantità di risorse è effetivamente possibile affrontare problemi più complessi (ad esempio, che esistono funzioni calcolabili in tempo quadratico, ma non in tempo linerare).

Questi risultati forniscono una delle poche tecniche di separabilità tra classi di complessità attualmente conosciute.

Il Teorema della gerarchia (spazio)

Teorema della gerarchia (spazio)

Sia s una funzione costruibile in spazio e $log \in O(s)$. Allora

$$O(s) \subseteq O(s') \Leftrightarrow DSPACE(s) \subseteq DSPACE(s')$$

L'implicazione

$$O(s) \subseteq O(s') \Rightarrow DSPACE(s) \subseteq DSPACE(s')$$

è ovvia: aumentando le risorse a disposizione riusciamo almeno a calcolare quello che calcolavamo prima.

Il viceversa non lo è: voglio dimostrare che aumentando l'ordine di grandezza delle risorse riesco effettivamente a risolvere problemi che prima non ero in grado di risolvere.

Ad esempio, esistono problemi risolubile in spazio $nlog(n)$, ma non in spazio n.

Il Teorema della gerarchia in spazio (solo se)

Dobbiamo dimostrare che

$$O(s) \not\subseteq O(s') \Rightarrow DSPACE(s) \not\subseteq DSPACE(s')$$

L'ipotesi (equivalente a dire che $s \notin O(s')$) ci dice che

$$\forall f \in O(s') \text{ esistono infiniti } n \text{ tali che } s(n) > f(n) \qquad (*)$$

Ad esempio, per visualizzare le idee, potete immaginare che $s' \in o(s)$.

Sotto l'ipotesi precedente devo dimostrare che

$$DSPACE(s) \not\subseteq DSPACE(s')$$

cioè devo esibire un linguaggio L tale che $L \in DSPACE(s)$ ma $L \notin DSPACE(s')$.

Procediamo a definire tale linguaggio.

Il linguaggio L (primo tentativo)

Definiamo L come un insieme di codici $\lceil M \rceil$ di opportune macchine di Turing:

$$L := \{ \lceil M \rceil \mid \underbrace{space_M(|\lceil M \rceil|) \leq s(|\lceil M \rceil|)}_{\text{complessità}} \text{ e } \underbrace{\lceil M \rceil \notin L_M}_{\text{diagonalizzazione}} \}$$

- il vincolo di diagonalizzazione vuole garantire che il linguaggio L sia diverso da tutti i linguaggi riconosciuti da macchine in L
- il vincolo di complessità vuole garantire che L possa appartenere a $DSPACE(s)$

Supponiamo di avere una macchina M_0 che riconosce L, cioè per cui $L = L_{M_0}$. Se $\lceil M_0 \rceil \in L$, per diagonalizzazione avremmo che $\lceil M_0 \rceil \notin L_{M_0}$, che è una contraddizione. Dunque $\lceil M_0 \rceil \notin L$.

Vorremmo poter concludere che, suppondendo $space_{M_0} \in O(s')$, anche l'ipotesi $\lceil M_0 \rceil \notin L$ fosse contraddittoria, cosa che ci permetterebbe di conlcudere che $L \notin DSPACE(s')$.

Il linguaggio L (primo tentativo)

Purtroppo il fatto che $space_{M_0} \in O(s')$ non basta per rispettare il vincolo di complessità nella definizione di L:

$$space_{M_0}(|\lceil M_0 \rceil|) \leq s(|\lceil M_0 \rceil|)$$

poichè la consocenza della complessità asintotica non ci dice nulla sulla complessità puntuale.
Dunque, **non riusciamo** a conlcudere che

$$\lceil M_0 \rceil \notin L \Rightarrow \lceil M_0 \rceil \in L$$

che darebbe la contraddizione cercata.

Il linguaggio L (primo tentativo)

Purtroppo il fatto che $space_{M_0} \in O(s')$ non basta per rispettare il vincolo di complessità nella definizione di L:

$$space_{M_0}(|\lceil M_0 \rceil|) \leq s(|\lceil M_0 \rceil|)$$

poichè la consocenza della complessità asintotica non ci dice nulla sulla complessità puntuale.
Dunque, **non riusciamo** a conlcudere che

$$\lceil M_0 \rceil \notin L \Rightarrow \lceil M_0 \rceil \in L$$

che darebbe la contraddizione cercata.

Dobbiamo complicare la definizione del linguaggio L, facendo il **padding** dei codici con **infinite stringhe** x di dimensione crescente, in modo da catturare un **comportamento asintotico**.

Il linguaggio L (corretto)

Definiamo L come un insieme di coppie $\langle \lceil M \rceil, x \rangle$ dove $\lceil M \rceil$ è il codice di una macchina di Turing e x è una stringa di padding

$$L := \{ \langle \lceil M \rceil, x \rangle \mid \underbrace{space_M(|\langle \lceil M \rceil, x \rangle|) \leq s(|\langle \lceil M \rceil, x \rangle|)}_{\text{complessità}} \text{ e } \underbrace{\langle \lceil M \rceil, x \rangle \notin L_M}_{\text{diagonalizzazione}} \}$$

Voglio dimostrare che $L \notin DSPACE(s')$.
Supponiamo di avere una macchina M_0 che riconosce L, e tale per cui $space_{M_0} \in O(s')$.
Per la proprià $(*)$ esistono infiniti n per cui $s(n) > space_{M_0}(n)$.
Possiamo quindi scegliere x_0 tale

$$space_{M_0}(|\langle \lceil M_0 \rceil, x_0 \rangle|) \leq s(|\langle \lceil M_0 \rceil, x_0 \rangle|)$$

e quindi avremmo

$$\begin{aligned} \langle \lceil M_0 \rceil, x_0 \rangle \in L \quad & \Leftrightarrow space_{M_0}(|\langle \lceil M_0 \rceil, x_0 \rangle|) \leq s(|\langle \lceil M_0 \rceil, x_0 \rangle|) \text{ e } \langle \lceil M_0 \rceil, x_0 \rangle \notin L_{M_0} \\ & \Leftrightarrow \langle \lceil M_0 \rceil, x_0 \rangle \notin L_{M_0} \end{aligned}$$

Il Teorema della gerarchia (spazio)

Dobbiamo ancora dimostrare che $L \in DSPACE(s)$. Consideriamo una macchina multi-tape che opera nel modo seguente su input y:

1. se y è della forma $\langle \lceil M \rceil, x \rangle$ allora calcola $0^{s(n)}$ per $n = |y|$, e fallisce altrimenti
2. calcola un timeout $t := |Q| \cdot 2 \cdot |\Sigma|^{s(n)} \cdot |y|$ dove Q è l'insieme degli stati della macchina M e Σ è il suo alfabeto
3. simula passo passo il comportamento di M su input y fino a terminazione oppure finchè ha utilizzato più di $s(n)$ spazio (ad esempio utilizzando $0^{s(n)}$ come area di lavoro) o un tempo superiore a t.
4. la macchina accetta l'input y se la simulazione di M termina rifiutandolo, oppure se si è esaurito il tempo, ma non lo spazio.

Il Teorema della gerarchia (spazio)

Osserviamo che se M richiede più di t passi su input y allora è in un ciclo infinito.
Siccome s è **costruibile in spazio**, allora il punto 1. richiede spazio $O(s)$. Il calcolo di t al punto 2 richiede spazio $O(log|y| + s(n))$.
La simulazione di M al punto 3 richiede spazio aggiuntivo rispetto a $s(n)$ solo per memorizzare il codice (costante) di M.
Poichè $log(n) \in O(s)$, tutti i passi richiedono spazio $O(s)$, il che dimostra che $L \in DSPACE(s)$.

Osservazione Il Teorema della Gerarchia non vale senza l'assunzione di costruibilità. È possibile dimostrare che per ogni funzione g ricorsiva non decrescente esiste una funzione totale calcolabile $f : N \to N$ tale

$$DSPACE(f) = DSPACE(g \circ f)$$

Questo risultato, noto come Gap Theorem vale anche per DTIME.

Il Teorema della gerarchia (tempo)

Teorema della gerarchia (tempo)

Sia t una funzione costruibile in tempo e $n \in O(t)$. Allora

$$O(t) \subseteq O(t') \Rightarrow DTIME(t) \subseteq DTIME(t') \Rightarrow O(\frac{t}{\log t}) \subseteq O(t')$$

La prima implicazione è una conseguenza delle definizioni. La seconda implicazione si dimostra in modo simile al teorema della gerarchia in spazio, considerando il linguaggio

$$L := \{0^k 1x | time_{M_{bin(k)}}(0^k 1x) \leq \frac{t(|0^k 1x|)}{log(t(|0^k 1x|))} \text{ e } 0^k 1x \notin L_{M_{bin(k)}}\}$$

Per $k \geq 2$ lo statement può essere rafforzato:

$$O(t) \subseteq O(t') \Leftrightarrow DTIME_k(t) \subseteq DTIME_k(t')$$

Alcune Classi di Complessità deterministica

- $P := \bigcup_{c \in N} DTIME(n^c)$
- $EXP := \bigcup_{c \in N} DTIME(2^{cn})$
- $LOGSPACE := DSPACE(log)$
- $PSPACE := \bigcup_{c \in N} DSPACE(n^c)$
- $EXPSPACE := \bigcup_{c \in N} DSPACE(2^{cn})$

Come corollario dei risultati precedenti abbiamo in particolare:

- $LOGSPACE \subseteq P \subseteq PSPACE \subset EXPSPACE$,
- $LOGSPACE \subset PSPACE$
- $P \subset EXP \subseteq EXPSPACE$

Per nessuna inclusione $\subseteq$ si conosce se sia stretta.

Il gap theorem

Borodin 1969

Per ogni funzione totale calcolabile g tale che $g(x) \geq x$ per ogni x, esiste una funzione $t(n)$ tale che non esiste i per cui $t(n) \leq t_{M_i}(n) \leq g(t(n))$ per un numero infinito di input.

Fissiamo una enumerazione delle MdT e definiamo t nel modo seguente: $t(0) = 1$ e $t(n) = t(n-1) + k_n$, dove $k_n = \mu(k \geq 1)\forall(i \leq n)P(i,k)$ per

$$P(i,k) = t_{M_i}(n) < t(n-1) + k \vee t_{M_i}(n) > g(t(n-1) + k)$$

- k_n esiste per ogni n in quanto, per ogni input x di dimensione inferiore a n o $M_i(x)$ diverge e $t_{M_i}(n) > g(t(n-1)+k)$ per ogni k, oppure ogni $M_i(x)$ converge ed esiste k sufficientemente grande per cui $t_{M_i}(n) < t(n-1)+k$
- t è calcolabile in quanto $P(i,k)$ è ricorsivo.
- per definizione, $P(i,k_n)$ è vero per ogni $i \leq n$, e dunque

$$t_{M_i}(n) < t(n) \vee t_{M_i}(n) > g(t(n)))$$

Per l'ultima condizione, per ogni $n \geq i$, se $t_{M_i}(n) \leq g(t(n))$, allora:

$$t_{M_i}(n) < t(n)$$

Il gap theorem - discussione

Il gap theorem viene correntemente considerato come una stranezza teorica, derivante dal considerare funzioni "esoteriche" di complessità.

In particolare, la funzione t del gap theorem

- cresce ad una velocità vertiginosa
- non è costruibile in tempo/spazio

Padding

EXP vs. PSPACE

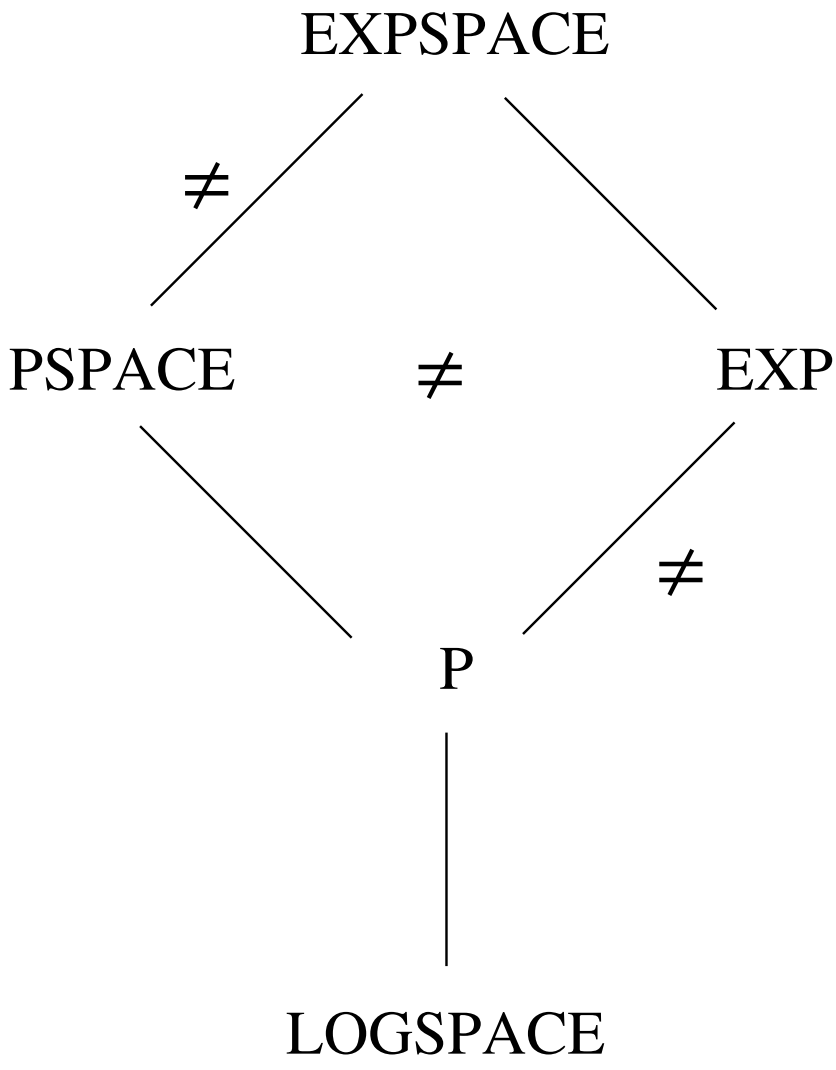
$EXP \neq PSPACE$.

Per il teorema della gerarchia in tempo

$$EXP \subseteq DTIME(2^{n^{1.5}}) \subset DTIME(2^{n^2})$$

Supponiamo $EXP = PSPACE$ e sia $L \in DTIME(2^{n^2})$.
Allora $L' := \{x\#^t : x \in L \text{ e } |x| + t = |x|^2\} \in DTIME(2^n) \subseteq EXP$.
Per ipotesi $L' \in PSPACE$, ovvero esiste $k > 0$ tale che $L' \in DSPACE(n^k)$.
Sia M la MdT corrispondente e consideriamo un'altra MdT M' che su input x opera nel modo seguente: (1) copia x su un nastro di lavoro; (2) estende x con $|x|^2 - |x|$ caratteri $\#$; (3) lancia una simulazione di M sull'input $x\#^{|x|^2-|x|}$.
Chiaramente, $L_{M'} = L$ e M' opera in spazio $O(n^{2k})$. Dunque, $L \in PSPACE$ e $DTIME(2^{n^2}) \subseteq PSPACE = EXP$. Contraddizione!

Inclusioni tra Classi di Complessità Deterministiche



Lezione 9: Complessità non deterministica

- Macchine di Turing non deterministiche
- Classi di complessità non deterministiche
- Riduzione dei nastri per MdT non deterministiche

Macchine di Turing non deterministiche

Una MdT non deterministica (MdTN) è definita in modo analogo alla versione deterministica, con la sola eccezione che la funzione di transizione δ è multi-valore, ovvero è una relazione

$$\delta \subseteq Q \times \Sigma^k \times Q \times (\Sigma \times \{L, R\})^k$$

- la nozione di configurazione resta invariata;
- la nozione di passo tra transizione tra configurazioni è adattata nel modo ovvio; ogni configurazione ammette ora più continuazioni possibili, e le computazioni non sono più sequenze lineari ma alberi;
- la macchina si arresta su input x se esiste almeno una computazione (un cammino nell'albero) che conduce a terminazione;
- la macchina riconosce l'input se esiste almeno una computazione terminante che si arresta in uno stato di riconoscimento.

Classi di Complessità Non Deterministica

Sia data una macchina di Turing non deterministica M:

- $\text{time}_M(x)$ è il minimo numero di passi richiesti da una qualche computazione di M relativa ad x
- $\text{space}_M(x)$ è il minimo spazio richiesto da una qualche computazione di M relativa ad x (per macchine a più nastri si considerano solo i nastri di lavoro)
- $t_M(n) := max(\{n\} \cup \{\text{time}_M(x) : |x| = n\}$
- $s_M(n) := max(\{1\} \cup \{\text{space}_M(x) : |x| = n\}$

Data una funzione $f : N \to N$ introduciamo le seguenti classi di complessità:

- $\text{NTIME}(f) := \{L \subseteq \Sigma^* : \exists \text{ MdTN } M, L = L_M \wedge t_M \in O(f)\}$
- $\text{NSPACE}(f) := \{L \subseteq \Sigma^* : \exists \text{ MdTN } M, L = L_M \wedge s_M \in O(f)\}$

Aggiungeremo un pedice k per esplicitare che M ha k nastri.

Alcune Classi di Complessità Non Deterministica

- $NP := \bigcup_{c \in N} NTIME(n^c)$
- $NEXP := \bigcup_{c \in N} NTIME(2^{cn})$
- $NLOGSPACE := NSPACE(log)$
- $NPSPACE := \bigcup_{c \in N} NSPACE(n^c)$

$D \subseteq N$

Per ogni $f : N \to N$

$$DTIME(f) \subseteq NTIME(f) \quad \wedge \quad DSPACE(f) \subseteq NSPACE(f)$$

Ovvio, in quanto le macchine deterministiche sono un caso particolare di quelle non deterministiche.

Come corollario:

$$P \subseteq NP \qquad LOGSPACE \subseteq NLOGSPACE$$
$$EXP \subseteq NEXP \qquad PSPACE \subseteq NPSPACE$$

Riduzione dei nastri per MdTN

Riduzione dei nastri per MdTN

Per ogni $f : N \to N$ abbiamo:

$$NSPACE(f) \subseteq NSPACE_1(f) \quad \wedge \quad NTIME(f) \subseteq NTIME_1(f^2)$$

La dimostrazione è identica a quella per il caso deterministico: i k nastri sono simulati su di uno solo utilizzando un alfabeto esteso che codifica le differenti "tracce".

Nel caso di MdTN, facendo un pesante uso del nondeterminismo, è anche possibile dimostrare che

$$NTIME(f) \subseteq NTIME_1(f)$$

Lezioni 10-11: Simulazione del nondeterminismo

- Simulazione del nondeterminismo
- Il Teorema di Savitch
- Il Teorema di Immerman and Szelepscènyi
- Inclusioni tra classi di complessità

Simulazione del nondeterminismo

NTIME vs. DSPACE

Per ogni $f : N \to N$ costruibile in spazio

$$NTIME(f) \subseteq DSPACE(id + f)$$

Sia M una MdTN che riconosce L in tempo $t_M \leq cf + c$. L'idea è quella di simulare le varie computazioni una alla volta, ricordando su un tape ausiliario le scelte effettuate. A questo scopo viene utilizzato un nuovo alfabeto $\Sigma_b = \{b_1, ..., b_m\}$ dove il numero dei caratteri dipende dal fattore massimo di branching della funzione δ (ed è dunque finito).

In maggiore dettaglio, la macchina M' che simula deterministicamente M opera nel modo seguente:

Simulazione del nondeterminismo (spazio)

1. si ricopia l'input x in un nastro di lavoro,
2. si scrive $0^{cf(n)+c}$ su di un altro nastro,
3. si generano progressivamente tutte le stringhe $w \in \{b_1, ..., b_m\}^*$ (opzioni) di lunghezza $|w| \leq 0^{cf(n)+c}$,
4. per ogni opzione w si simula la MdTN M su altri nastri di lavoro per un numero di passi pari al più a $cf(n) + c$; al passo i si usa il carattere $w(i) = b_j$ per selezionare l'alternativa j tra le possibili transizioni (si usi la prima come default se non esiste j).
5. se M termina in un uno stato di accettazione, allora si accetta x, altrimenti si seleziona la parola w successiva, fino ad esaurimento.
6. se si arriva ad esaurimento delle opzioni, x non è riconosciuta.

Si noti che l'arresto delle computazioni dopo al più $cf(n) + c$ passi è giustificato dal fatto che, se x è riconosciuta da M, esiste almeno una computazione che la riconosce entro tale tempo.
Per la complessità: il punto 1. richiede spazio $O(n)$; poichè f è costruibile in spazio, il secondo passo richiede spazio $O(f)$, così come i punti successivi.

Simulazione del nondeterminismo (tempo)

NSPACE vs. DTIME

Per ogni $f : N \to N$ costruibile in spazio

$$NSPACE(f) \subseteq DTIME(2^{c(log+f)})$$

Sia M una MdTN che accetta L in spazio $s_M \leq cf + c$. Il massimo numero di configurazioni differenti attraversate durante il riconoscimento dell'input x (con minimo spazio) è limitato da $2^{c'(log(n)+f(n))+c'}$ per un qualche c' opportuno (come nel caso delle macchine deterministiche). Costruiamo una macchina M' che simula deterministicamente M eseguendo una visita "in larghezza" dell'albero delle computazioni relative ad un input x, mantenendo gli insiemi delle configurazioni da visitare e di quelle già visitate. In particolare M' esegue i passi seguenti:

Simulazione del nondeterminismo (tempo)

1. ricopia la configurazione iniziale su di un nastro di lavoro (frontiera),
2. calcola $0^{cf(n)+c}$ su di un altro nastro,
3. seleziona una configurazione ξ dalla frontiera; se ξ è in un uno stato di riconoscimento m' si arresta con successo; altrimenti, verifica che ξ non sia già stata presa in considerazione confrontandola con una lista di configurazioni già visitate memorizzate in un nastro opportuno (nodi interni).
4. aggiunge le configurazioni raggiungibili da ξ alla frontiera e sposta ξ sui nodi interni (le nuove configurazioni più lunghe di $cf(n)+c$ vengono ignorate),
5. se la frontiera è vuota si termina con fallimento, altrimenti torna a 3.

Il primo passo richiede tempo $O(n) = O(2^{log(n)})$. Poichè f è costruibile in spazio, il secondo passo richiede un tempo $O(2^{c''(log+f)})$ per qualche $c'' \in N$. Come già osservato, il numero massimo $maxc$ di configurazioni differenti di lunghezza minore o uguale a $cf(n)+c$ da analizzare è $2^{c'(log(n)+f(n))+c'}$ e l'intera simulazione richiede al più tempo $O(maxc^2) = O(2^{c'''(log(n)+f(n))})$ per un opportuno $c''' \in N$.

Simulazione del nondeterminismo

Come corollari dei risultati precedenti abbiamo le inclusioni seguenti, per ogni funzione f costruibile in spazio:

$$NTIME(f) \subseteq \bigcup_{c \in N} DTIME(2^{c(id+f)})$$

$$NSPACE(f) \subseteq \bigcup_{c \in N} DSPACE(2^{c(log+f)})$$

È possibile migliorare queste maggiorazioni?

Raggiungibilità

Sia (V, E) un grafo di dimensione n. È possibile determinare se due nodi u, v sono connessi da un cammino di lunghezza inferiore a 2^i con un consumo di spazio dell'ordine di $i * log(n)$.

$$\begin{array}{l} \text{let rec reachable}(i, u, v) ::= \\ \quad \text{if } i = 0 \text{ then } u = v \vee \langle u, v \rangle \in E \\ \quad\quad \text{else } \exists z.\text{reachable}(i-1, u, z) \wedge \text{reachable}(i-1, z, v) \end{array}$$

Se esiste un cammino tra due nodi deve avere lunghezza inferiore a n e dunque si può determinare la raggiungibilità tra due nodi con complessità in spazio pari a $log(n)^2$.

Si noti che la complessità in tempo dell'algoritmo precedente è dell'ordine di $(2n)^i$ (ovvero $n \cdot n^{log(n)}$ per $i = log(n)$).

Il Teorema di Savitch

Teorema di Savitch

Sia $s : N \to N$ una funzione costruibile in spazio e tale che $log \in O(s)$. Allora

$$NSPACE(s) \subseteq DSPACE(s^2)$$

Sia $L \in NSPACE(s)$. Poichè $NSPACE(s) \subseteq NSPACE_1(s)$, possiamo assumere che esista una MdtN M ad un nastro che riconosce L in spazio $s_M \leq cs + c$ per qualche $c \in N$. Costruiamo una macchina deterministica M' che riconosce L in spazio $s_{M'} \in O(s^2)$.
Supponiamo per semplicità che M abbia un unica configurazione di arresto Ξ_h. Come osservato in precedenza, la lunghezza delle computazioni di M su input $x \in L$ con $|x| = n$ è limitata da $maxc = 2^{cs(n)+c}$ per c opportuno (in quanto $log \in O(s)$). Inoltre, ognuna di queste configurazioni ha una dimensione limitata da $cs(n) + c$.
Il nostro scopo è dimostrare l'esistenza di almeno una computazione dalla configurazione iniziale Ξ_0 a quella di arresto Ξ_h che utilizza uno spazio $O(s^2)$ e fare in modo che M' segua questo cammino.

Il Teorema di Savitch (2)

Definiamo il seguente predicato di raggiungibilità in al più k passi tra due configurazioni β e γ:

$$\begin{aligned} reach(\beta, \gamma, k) \quad \Leftrightarrow \quad & \exists \beta_0, \ldots, \beta_j, j \leq k \wedge \beta_0 = \beta \wedge \beta_j = \gamma \wedge \\ & \forall i = 0, ..., j-1, \beta_i \vdash_M \beta_{i+1} \end{aligned}$$

In particolare, per ogni i,

$$reach(\beta, \gamma, 2i) \Leftrightarrow \exists \alpha, reach(\beta, \alpha, i) \wedge reach(\alpha, \gamma, i)$$

che permette una implementazione ricorsiva dell'algoritmo di ricerca. Quindi una MdT M', dato l'input x con $|x| \leq n$ è in grado di determinare

$$reach(\Xi_0, \Xi_f, 2^{cs(n)+c})$$

con la piú $O(s(n))$ chiamate attive innestate: l'intero stack dei record di attivazione richiede al piú uno spazio $O(s^2)$.

PSPACE and NPSPACE

Come ovvio orollario otteniamo:

$$PSPACE = NPSPACE$$

Poichè inoltre le classi deterministiche sono chiuse rispetto alla complementazione, abbiamo anche:

$$NPSPACE = coNPSPACE$$

Possiamo estendere il risultato a classi di complessità inferiori?

Il Teorema di Immerman and Szelepscènyi

Teorema di Immerman and Szelepscènyi (su grafi)

Dato un grafo G con n nodi, e un nodo x il numero dei nodi raggiungibili da x in G può essere calcolato in $NSPACE(\log n)$.

Costruiremo una macchina non deterministica che calcola la funzione data nel senso che *tutte* le computazioni che terminano con successo concordano sul risultato, ed *almeno* una computazione è garantita terminare (alcune computazioni possono però fallire).

Sia n_k il numero di nodi raggiungibili da x in al più k passi.
La macchina opera con quattro cicli annidati.

Il ciclo esterno calcola n_k per ogni k. Il calcolo di n_{k+1} richiede di conoscere n_k (il cui valore è tenuto in una variabile Nk) ma non valori precedenti.
Inizialmente $Nk = n_0 = 1$.

$$Nk := 1;\ \text{for } k = 0 \text{ to } n-1 \text{ do } Nk := next\_no$$

Il Teorema di Immerman and Szelepscènyi (2)

Il calcolo di $next\_no$ richiede di esaminare ogni nodo v del grafo. Si inizializza un contatore c a 0 e lo si incrementa tutte le volte che si trova un cammino dal nodo iniziale x a v. La funzione termina restituendo c:

$$netx\_no() = c := 0;\ \text{for } v \in V \text{ do } \ \text{if } reach(x, v) \text{ then } c := c + 1 :\ \text{return } c$$

Il calcolo di $reach(x, v)$ avviene in modo non deterministico. Nuovamente si opera su ogni nodo v' del grafo. Si inizializza un nuovo contatore c' a 0 e una variabile booleana b a false. Si tenta non determisticamente un cammino da x a v'; se questo esiste e se $(v, v') \in E$, allora si incrementa c' e si pone $b := true$. Alla fine del loop si verifica che $c' = Nk$; se questo non succede, abbiamo perso qualche cammino per via della scelta non deterministica; il valore di b non è affidabile e dunque si abortisce la computazione. Altrimenti si restituisce b.

$reach(x, v) =$
$c' := 0; b := false;$
for $v' \in V$ do
if $try\_a\_path(x, v', k)$ then $(c' := c' + 1;\ \text{if } (v, v') \in E \text{ then } b := true)$
if $b = false$ and $c' < Nk$ then abort else return b

Il Teorema di Immerman and Szelepscènyi (3)

Il calcolo di $try\_a\_path(x, y, k)$ è il ciclo più interno. Semplicemente richiede di indovinare il cammino, tentando non deterministicamente i nodi intermedi:

$try\_a\_path(x, y, k) =$
  $s := x; b' := true$
  for $i := 1$ to k
    if b' then
      $t :=$ guess a node ;
      if $s = t$ or $(s, t) \in E$ then $s := t$ else $b' := false$
  $b' := (b' \wedge s = y)$; return b'

L'intero algoritmo richiede di memorizzare k, Nk, c, c',v,v', b,b',s,t e i. Su queste variabili operiamo con confronti e incrementi unitari; numericamente sono tutte limitate da n, dunque richiedono una dimensione $log\ n$.

Il Teorema di Immerman and Szelepscènyi (4)

Teorema di Immerman and Szelepscènyi

Sia $s : N \to N$ una funzione costruibile in spazio e tale che $log \in O(s)$. Allora

$$NSPACE(s) = coNSPACE(s)$$

Sia $L \in NSPACE(s)$. Poichè $NSPACE(s) \subseteq NSPACE_1(s)$, possiamo assumere che esista una MdTN M ad un nastro che riconosce L in spazio $s_M \leq cs + c$ per qualche $c \in N$. Dato un input x di lunghezza $|x| = n$ consideriamo l'insieme C_x delle configurazioni di M relative ad x e di dimensione inferiore o uguale a $cs(n) + c$. Siccome $log \in O(s)$ possiamo supporre che la dimensione di C_x (e dunque del suo più lungo cammino) sia $m \leq 2^{c's(n)+c'}$ per una opportuna costante c'.

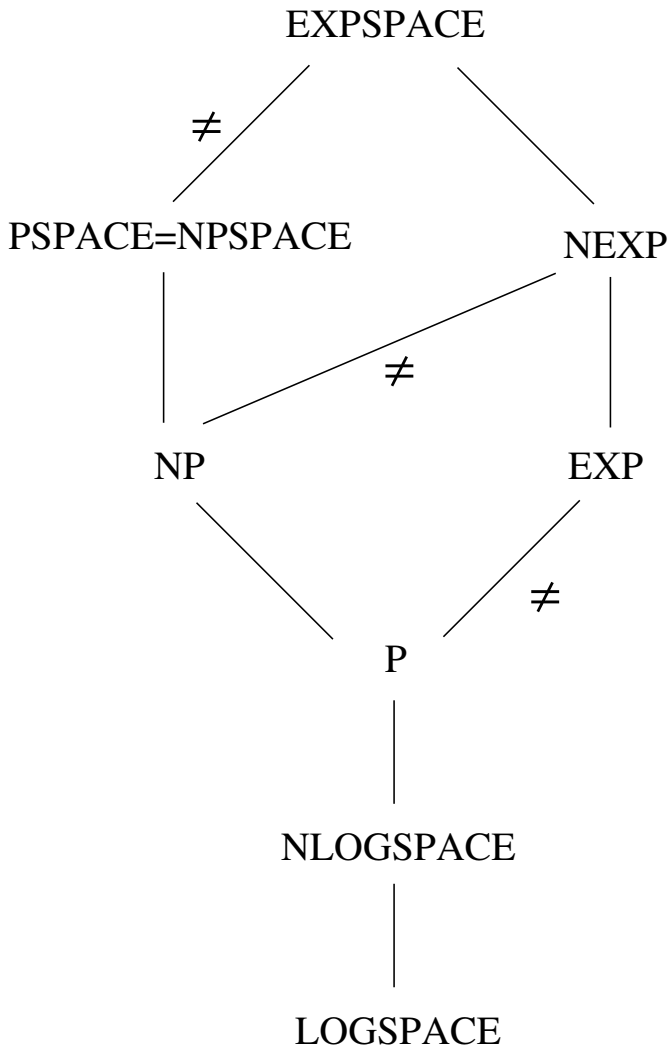
Il Teorema di Immerman and Szelepscènyi (5)

Dobbiamo dimostrare che

- $\Sigma^* \setminus L$ può essere accettato da un MdTN M' in spazio $O(s)$.

Data la macchina M per L, la macchina M' calcola innanzitutto il numero $n = n_m(\alpha_0)$ delle configurazioni raggiungibili dalla configurazione iniziale α_0 di M nel tempo m. Consideriamo quindi tutte le possibili configurazioni in C_x e per ognuna di queste verifichiamo (non deterministicamente) che sia effettivamente raggiungibile da α_0 e che non sia uno stato di accettazione. Se abbiamo trovato n configurazioni raggiungibili, nessuna delle quali di accettazione, allora M' accetta e altrimenti fallisce (relativamente a quella particolare computazione).
Chiaramente, anche M' opera in spazio $O(s)$.

Inclusioni tra Classi di Complessità



Lezione 12: Nondeterminismo e Verifica

- Il Teorema della Proiezione
- Classi di complessità per funzioni
- Riducibilità
- NP-Completezza

Il Teorema della Proiezione

Teorema della Proiezione

Dato un linguaggio $A \subseteq \Sigma^*$, $A \in NP$ se e solo se esiste un linguaggio $B \in P$ e un polinomio p tale che per ogni $x \in \Sigma^*$,

$$x \in A \Leftrightarrow \exists y, |y| \leq p(|x|) \wedge \langle x, y \rangle \in B$$

Sia $A \in NP$. Per il teorema della riduzione dei nastri possiamo assumere che esista un MdTN ad un nastro che riconosce A in tempo $t_M \leq p$ per un qualche polinomio p. Eliminiamo il nondeterminismo utilizzando una tecnica simile a quella adottata nella prova che $NTIME(f) \subseteq DSPACE(id + f)$.
Sia δ la funzione di transizione nondeterministica di M e sia m il branching massimo. Costruiamo la macchina deterministica M' che accetta B in tempo $t_M \in O(p)$. M' lavora con un alfabeto arricchito Σ' con simboli $b1, ..., b_m$ utlizzati per decidere ad ogni passo il ramo della computazione da seguire.

Il Teorema della Proiezione (2)

In particolare, M' su input $\langle x, y \rangle$ simula la macchina M per al più $|y|$ passi, utizzando ad ogni passo i il carattere y_i della stringa y per decidere la mossa da seguire.
È evidente che

$$x \in A = L_M \Leftrightarrow \exists y, |y| \leq p(|x|) \wedge \langle x, y \rangle \in B = L_{M'}$$

Viceversa, sia dato $B \in P$, un polinomio p che soddisfa le ipotesi, e una MdT M tale che $B = L_M$ e $t_M \in O(q)$ per un qualche polinomio q. La MdTN M' che riconosce A opera nel modo seguente: estende in modo non deterministico l'input x con le $p(|x|)$ stringhe y e quindi simula il comportamento di M sull'input $\langle x, y \rangle$. Chiaramente, M' opera in tempo $t_M \in O(p + q)$.

Programmazione per tentativi e verifica

- Il teorema della proiezione permette di pensare alle computazioni nondeterministiche in modo deterministico.
- L'input $\langle x, y \rangle$ consiste di due componenti: l'istanza del problema x e un testimone (traccia, certificato, prova) che permette di verificare rapidamente l'appartenenza di x al linguaggio

Lezioni 12-13: Riducibilità e Completezza

- Classi di complessità per funzioni
- Riducibilità
- Problemi Ardui e Completi
- Un problema NP-Completo

Classi di Complessità per funzioni

Fino ad ora abbiamo parlato di complessità relativamente al riconoscimento di linguaggi. Per ragioni di composizionalità è bene estendere la teoria per poter trattare funzioni.
Data una funzione $t : N \to N$ introduciamo le seguenti classi di complessità:

- $FTIME(t) := \{f : \Sigma^* \to \Sigma^* | \exists \text{ Mdt } M, f = f_M \wedge t_M \in O(t)\}$
- $FSPACE(t) := \{f : \Sigma^* \to \Sigma^* | \exists \text{ Mdt } M, f = f_M \wedge s_M \in O(t)\}$

Utilizzeremo in particolare le seguenti classi di funzioni:

- $FP := \bigcup_{c \in N} FTIME(n^c)$
- $FLOGSPACE := FSPACE(log)$

Lemma

$FLOGSPACE \subseteq FP$

Il lemma si dimostra in modo del tutto analogo alla dimostrazione che $LOGSPACE \subseteq P$.

Chiusura per composizione

Chiusura per composizione

Le classi FP e $FLOGSPACE$ sono chiuse per composizione.

Supponiamo che f sia calcolata da un MdT M in tempo $t_M \leq p$ per un qualche polinomio p. Allora, per ogni input x di lunghezza $|x| = n$ anche l'output deve avere lunghezza polinomiale, ovvero $|f(x)| \leq p(n)$. Supponiamo inoltre che g sia calcolata da una MdT M' in tempo $t_{M'} \leq q$ per un altro polinomio q. Sia M'' la MdT ottenuta componendo sequenzialmente M e M'. Abbiamo allora:

$$time_{M''}(x) \leq time_M(x) + time_{M'}(f(x))$$

e dunque

$$t_{M''}(n) \leq t_M(n) + t_{M'}(p(n)) \in O(qp)$$

Dunque $g \circ f \in FP$.

Supponiamo ora che $f, g \in FLOGSPACE$ siano rispettivamente calcolate dalle MdT M e M'. Non possiamo operare come in precedenza poichè l'output di una funzione $f \in FLOGSPACE$ non ha necessariamente lunghezza logaritmica (si contano solo i nastri di lavoro!).

Chiusura per composizione (logspace)

Tuttavia, poichè $FLOGSPACE \subseteq FP$ l'output $f(x)$ relativo a un input x di lunghezza $n = |x|$ ha al più lunghezza polinomiale $|f(x)| \leq p(n)$. Si noti che la memorizzazione di questo output potrebbe richiedere uno spazio più che logaritmico.
La soluzione è quella di cominciare l'esecuzione di M' e solo allorchè questa computazione richiede un nuovo carattere di input si procede nella simulazione di M finché non produce un nuovo carattere di output che viene **sovrascritto** al precedente. Questa strategia funziona poichè i nastri di input e output sono nastri su cui la testina può solo avanzare!
La nuova macchina richiede il seguente spazio:

$$space_{M''}(x) \leq space_M(x) + space_{M'}(f(x))$$

e dunque

$$s_{M''}(n) \leq s_M(n) + s_{M'}(p(n)) \in O(log + log \circ p) \subseteq O(log)$$

Dunque $g \circ f \in LOGSPACE$.

Riducibilità

Siano $A, B \in \Sigma^*$ due linguaggi.

- Diremo che A è riducibile in tempo polinomiale a B, e scriveremo $A \leq_P B$, se esiste $f \in FP$ tale che per ogni $x \in \Sigma^*$, $x \in A \Leftrightarrow f(x) \in B$
- analogamente, diremo che A è riducibile in spazio logaritmico a B, e scriveremo $A \leq_L B$ se esiste $f \in FLOGSPACE$ che realizza l'equazione precedente.

Lemma

Le relazioni $\leq_P$ e $\leq_L$ sono preordini.

La riflessività deriva dalla definizione e la transitività dalla chiusura rispetto alla composizione di FP e $FLOGSPACE$.

Indicheremo con $\equiv_P$ e $\equiv_L$ le equivalenze indotte.

Chiusura per riducibilità

Sia $\mathcal{C}$ una classe di linguaggi e $\leq$ un qualche preordine. Diremo che $\mathcal{C}$ è chiusa rispetto a $\leq$, se

$$A \leq B \wedge B \in \mathcal{C} \rightarrow A \in \mathcal{C}$$

Lemma

- *Le classi $P, NP, PSPACE$ sono chiuse rispetto alla riducibilità in tempo polinomiale $\leq_P$.*
- *Le classi $LOGSPACE, NLOGSPACE, P, NP, PSPACE$ sono chiuse rispetto alla riducibilità in spazio logaritmico $\leq_L$*

La dimostrazione è simile a quella della chiusura di FP e $FLOGSPACE$ rispetto alla composizione.

Problemi ardui e completi

Sia $\mathcal{C}$ una classe di linguaggi e $\leq$ un preordine tra di essi. Sia B un linguaggio:

- B è *mathcalC*-arduo rispetto a $\leq$, se ogni $A \in \mathcal{C}$ è riducibile a B, ovvero $A \leq B$.
- B è $\mathcal{C}$-completo rispetto a $\leq$, se $B \in \mathcal{C}$ e B è $\mathcal{C}$-arduo

Nel caso in cui non sia esplicitato altrimenti, si assume che il preordine sia la riducibilità polinomiale $\leq_P$.

MdTN Universale

Esistenza di MdTN Universali

Esiste un encoding totale e suriettivo M_x delle MdTN normalizzate, ad un solo nastro mediante stringe $x \in \{0,1\}^*$ tale che esiste $u \in \{0,1\}^*$ con le seguenti proprietà:

- $L_{M_u} = \{\langle x, y\rangle : |y \in L_{M_x}\}$
- $\forall x, y, \exists c, time_{M_u}\langle x, y\rangle \leq c \cdot (time_{M_x}(y))^2 + c$

La prova è analoga a quella del caso deterministico.
Il non determinismo dell'interprete permette di simulare il non determinismo descritto dalla funzione di transizione delle singole macchine.

Il problema limitato della fermata

Il problema limitato della fermata

$$BHP := \{\langle x, y, 0^t\rangle | \text{MdTN } M_x \text{ accetta } y \text{ in tempo } time_{M_x}(y) \leq t\}$$

è NP-completo.

La macchina universale M_u può simulare M_x su input y per t passi e verificare se arriva in uno stato di accettazione in un tempo inferiore a $p(t)$ per un qualche polinomio p.

Viceversa se $A \in NP$ allora esiste una MdTN M_x che riconosce A in tempo $t_{M_x} \leq p$ per un qualche polinomio p. La funzione f di riduzione che mostra che $A \leq_P BHP$ è semplicemente

$$f(y) := \langle x, y, 0^{p(|y|)}\rangle$$

e $f \in FP$ poichè i polinomi sono costruibili in tempo.

P vs. NP

Lemma

Se per qualche problema NP-completo A si ha $A \in P$ allora $P = NP$.

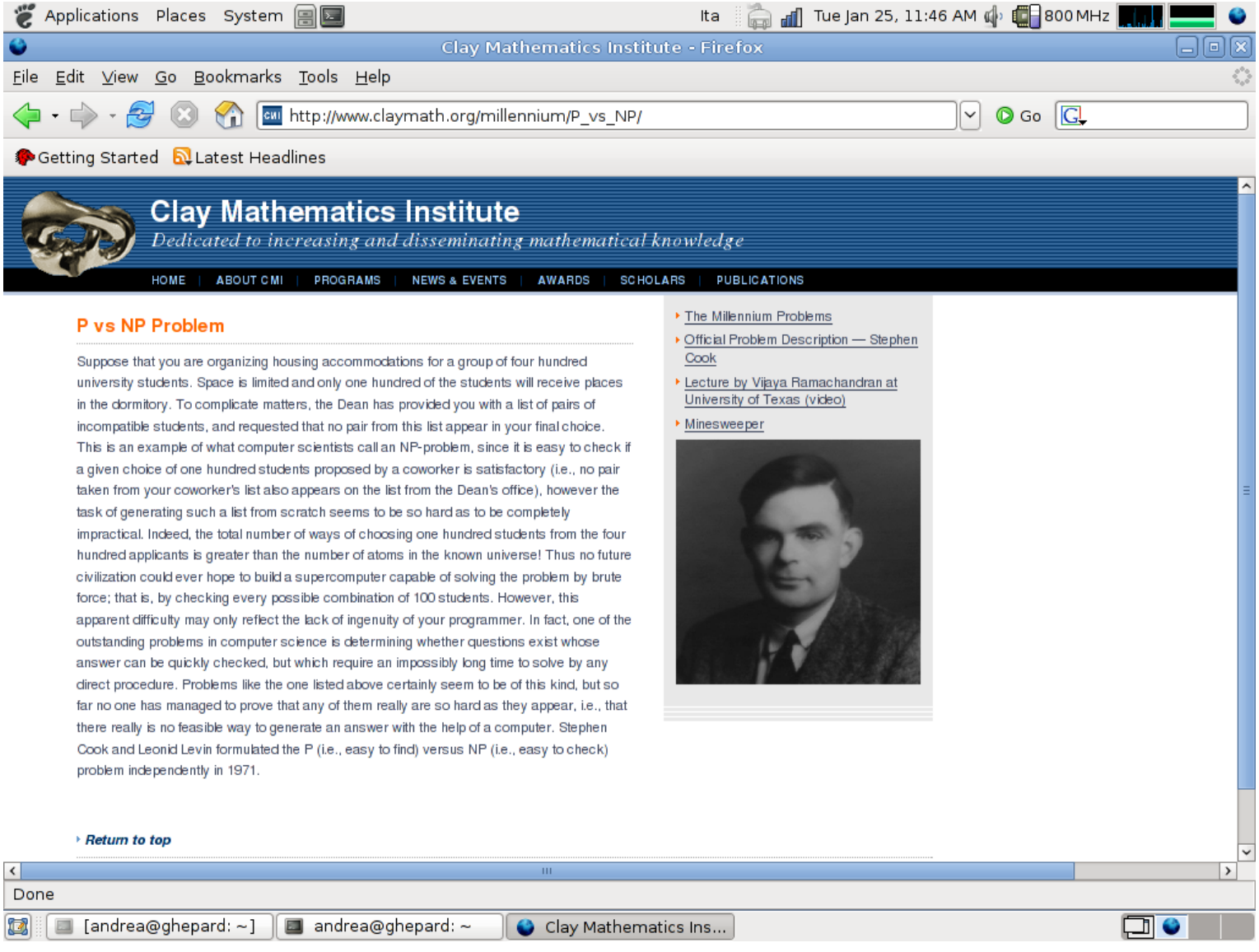
$P \subseteq NP$, quindi dobbiamo dimostrare l'inclusione opposta. Se $B \in NP$, per la NP-completezza di A si ha $B \leq_P A$. Siccome P è chiusa per la riducibilità in tempo polinomiale e per ipotesi $A \in P$, anche $B \in P$.

Lemma

Se $P = NP$ allora ogni problema non banale $A \in NP$ è NP-completo.

Basta osservare che per ogni coppia di problemi non banali $B, A \in P$ si ha $B \leq_P A$ (supposto $a_0 \in A$ e $a_1 \notin A$ basta modificare l'algoritmo di decisione per B in un algoritmo che restituisca a_0 al posto di 1 e a_1 al posto di 0).

The Millenium Problems



Lezioni 14-15

- Il Problema della Soddisfacibilità
- Il problema del Ricoprimento
- Il problema dell'Insieme indipendente
- Il problema della Cricca
- Equazioni Diofantine quadratiche
- Il Teorema di Manders e Adleman

Il Problema della Soddisfacibilità

Sia $\Sigma := \{0, 1, \neg, \wedge, \vee, \rightarrow, (,), [,]\}$. $SAT \subseteq \Sigma^*$ è l'insieme delle formule booleane soddisfacibili, cioè l'insieme delle formule che sono vere rispetto ad un qualche assegnamento di valori di verità ai simboli proposizionali. Il simbolo proposizionale P_k è codificato da una stringa della forma $[bin(k)]$, mentre gli altri simboli di Σ hanno la loro interpretazione logica usuale.

Ad esempio,$[0] \rightarrow [1] \in SAT$ ma $[0] \wedge \neg[0] \notin SAT$.

$3SAT \subseteq SAT$ è l'insieme delle formule soddisfacibili in forma normale congiuntiva e tali per cui ogni clausola contiene esattamente tre variabili distinte.

SAT è NP-completo

Teorema di Cook

SAT è NP-completo.

È facile vedere che $SAT \in NP$. Dobbiamo dimostrarne la completezza.

Sia $L \in NP$ e consideriamo una MdTN con un nastro $M = (Q, q_0, F, \Sigma, \Gamma, B, 1, \delta)$ tale che $L_M = L$ (senza perdita di generalità, supporremo che M operi sul seminastro destro).

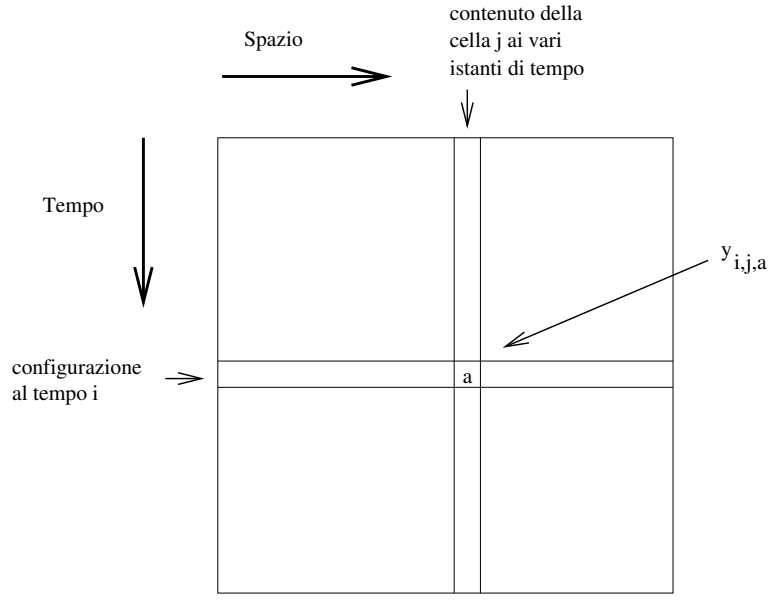
Supponiamo inoltre che $t_M \leq p$ per un qualche polinomio p. Il nostro obiettivo è definire una formula ψ_x per ogni x tale che $x \in L \Leftrightarrow \psi_x \in SAT$.

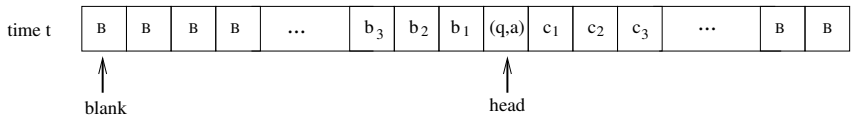
Supposto $|x| = n$, deve esistere una computazione che porta al riconoscimento di x attraversando al più $p(n)$ configurazioni. Inoltre, ogni configurazione ha una dimensione che non eccede $p(n)$.

Codificheremo le configurazioni con stringhe sull'alfabeto $\Gamma' = \Gamma \cup (Q \times \Gamma)$, dove utilizziamo il carattere in $Q \times \Gamma$ per codificare lo stato interno e la posizione della testina.

Il Teorema di Cook

Rappresentiamo la computazione relativa all'input x con una matrice di dimensione $p(n) \times (p(n) + 2)$; due caratteri aggiuntivi delimitano la fine del nastro.





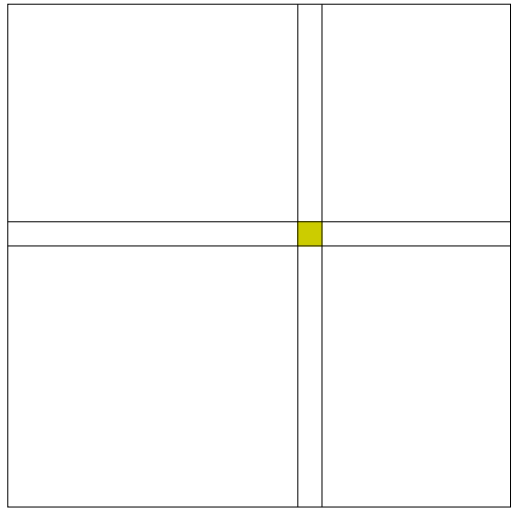
Il contenuto della matrice è descritto mediante variabili proposizionali $y_{i,j,a}$, con la seguente interpretazione intuitiva:

$$y_{i,j,a} = \mathit{true} \quad \Leftrightarrow \quad \text{il carattere } j \text{ della } i\text{-esima configurazione contiene il carattere } a$$

Un carattere per cella

Dobbiamo ora vincolare le variabili proposizionali in modo tale che abbiano l'interpretazione attesa.

Il primo vincolo esprime il fatto che, ad ogni istante di tempo i, ogni cella del nastro j può contenere uno e solo un carattere.

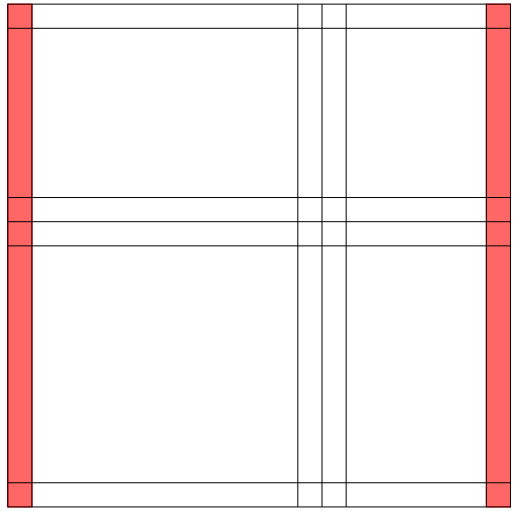


$$\psi_0 := \bigwedge_{i=0}^{p(n)} \bigwedge_{j=1}^{p(n)} \left(\bigvee_{a \in \Gamma'} y_{i,j,a} \wedge \bigwedge_{a,b \in \Gamma', a \neq b} (\neg y_{i,j,a} \vee \neg y_{i,j,b}) \right)$$

Bordi invalicabili

Il secondo vincolo esprime il fatto che i bordi del nastro non possono essere valicati.

Facciamo questo forzando il carattere blank in posizione 0 e posizione $p(n) + 1$.

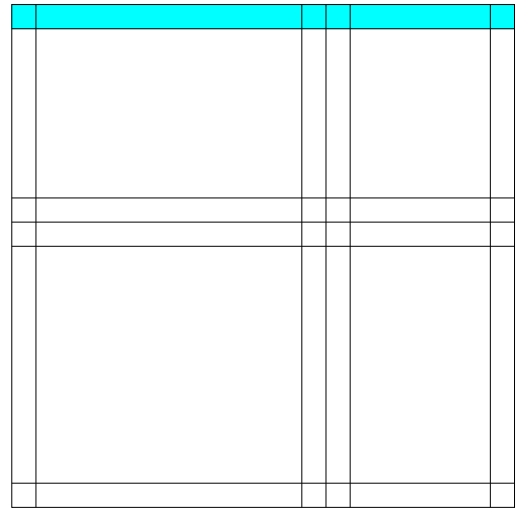


$$\psi_1 \quad := \quad \bigwedge_{i=0}^{p(n)} (y_{i,0,B} \wedge y_{i,p(n)+1,B})$$

Configurazione iniziale

Il terzo vincolo definisce la configurazione iniziale, che è nota:

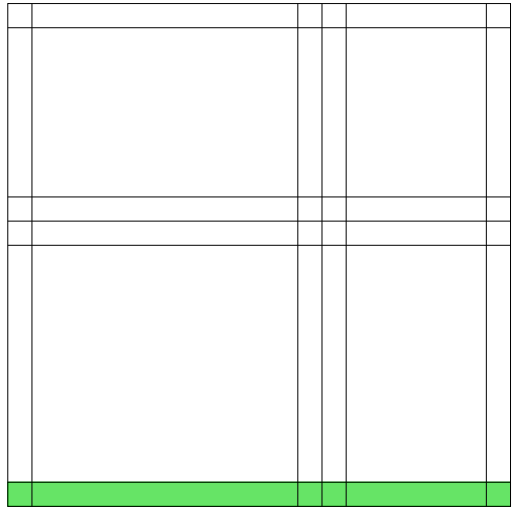
- il nastro deve contenere l'input $x = x_0 x_1 \ldots x_n$
- la porzione restante del nastro è inizializzata a blank
- la testina è sul carattere x_0



$$\psi_2 \quad := y_{0,1,(q_0,x_1)} \wedge \bigwedge_{j=2}^{n} y_{0,j,x_j} \wedge \bigwedge_{j=n+1}^{p(n)} y_{0,j,B}$$

Configurazione finale

Il quarto vincolo esprime il fatto che lo stato interno della macchina all'istante $t = p(n)$ deve essere uno degli stati di accettazione F



$$\psi_3 \quad := \bigvee_{j=1}^{p(n)} \bigvee_{a \in F \times \Gamma} y_{p(n),j,a}$$

Passaggio tra configurazioni

Infine, deve essere possibile transire da una configurazione alla successiva.

Innanzi tutto, la porzione del nastro **non adiacente alla testina** al tempo $t+1$ è identica a quella al tempo t:

time t	B	B	B	B	...	b_3	b_2	b_1	(q,a)	c_1	c_2	c_3	...	B	B
time t+1	B	B	B	B	...	b_3	b_2				c_2	c_3	...	B	B

In modo più esplicito, se al tempo i la testina non si trova in nessuna delle posizioni $j-1, j, j+1$, allora il contenuto della cella j al tempo $i+1$ coincide con quello al tempo i:

$$\psi_4 := \bigwedge_{i=0}^{p(n)-1} \bigwedge_{j=1}^{p(n)} \bigwedge_{a,b,c \in \Gamma} (y_{i,j-1,a} \wedge y_{i,j,b} \wedge y_{i,j+1,c} \rightarrow y_{i+1,j,b})$$

Evoluzione attorno alla testina

Le celle adiacenti alla testina devono essere modificate in accordo alla funzione di transizione δ della macchina

$$\psi_5 := \bigwedge_{i=0}^{p(n)-1} \bigwedge_{j=1}^{p(n)} \bigwedge_{(q,a)\in Q\times\Gamma} \Delta_{q,a,i,j}$$

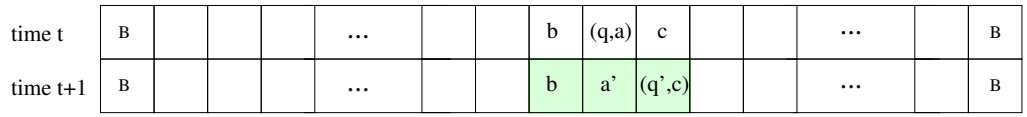
La formula $\Delta_{q,a,i,j}$ descrive le **possibili** evoluzioni della configurazione al passo i conseguenti all'esecuzione di una mossa **non deterministica** della macchina.

Spieghiamo la formula utilizzando un esempio di transizione.
Supponiamo ad esempio che $\delta(q,a) = \{(q',a',R),(q'',a'',L)\}$.

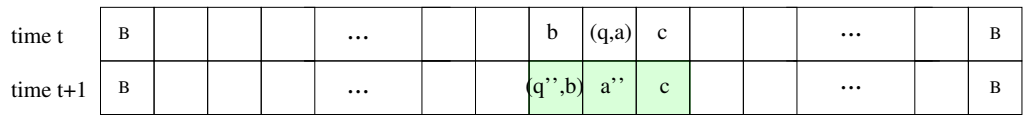
Se la macchina è nello stato q e sta leggendo a avremo due possibili evoluzioni della computazione verso due diverse configurazioni.

La formula $\Delta_{q,a,i,j}$

Se si segue la mossa (q', a', R), allora



Se si segue la mossa (q'', a'', L), allora



Le due possibilità sono descritte mediante la **disgiunzione** logica dei due casi:

$$\begin{aligned}\Delta_{q,a,i,j} \quad := \quad & (y_{i,j-1,b} \wedge y_{i,j,(q,a)} \wedge y_{i,j+1,c} \rightarrow \\ & ((y_{i+1,j-1,b} \wedge y_{i+1,j,a'} \wedge y_{i+1,j+1,(q',c)}) \\ & \vee (y_{i+1,j-1,(q'',b)} \wedge y_{i+1,j,a''} \wedge y_{i+1,j+1,c})))\end{aligned}$$

Per altri valori della funzione δ, $\Delta_{q,a,i,j}$ è definita in modo analogo.

Summing up

La congiunzione $\psi_x := \psi_0 \wedge \psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \wedge \psi_5$ è soddisfacibile se e solo se esiste una computazione non deterministica della MdTN M che porta alla accettazione di x, e quindi, dato che per ipotesi M riconosce il linguaggio L:

$$x \in L \Leftrightarrow \psi_x \in SAT$$

Resta da appurare la complessità computazionale della funzione $f : x \mapsto \psi_x$.

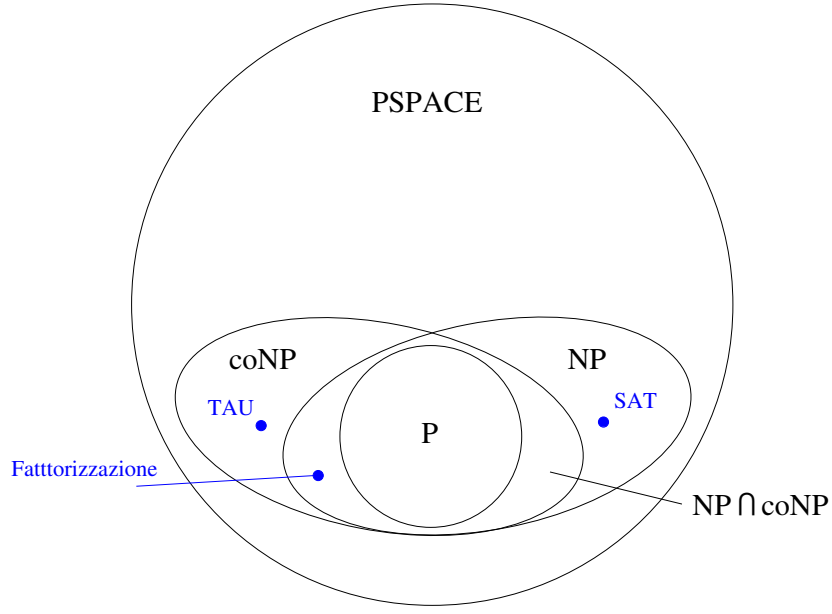
È evidente che tutte la formule ψ_i pssono essere costruite in tempo $O(p)$ e dunque

$$f \in FP$$

Remark È in effetti possibile dimostrare che $f \in LOGSPACE$.

Analogie tra Calcolabilità e Complessità

Calcolabilità	Complessità
RE	*NP*
Ricorsivo	*P*
$\leq_m$	$\leq_P$
K	*SAT*



Differenze:

- si sa che RE $\neq$ Ricorsivo
- RE $\cap$ coRE $=$ Ricorsivo
- si conoscono insiemi RE non completi

Tecniche per dimostrare che un problema è arduo

Due tecniche generali per dimostrare che A è NP-arduo

- **dimostrazione diretta**: uso la definizione e cerco di dimostrare che **per ogni** problema $B \in NP$,

$$B \leq_p A$$

- **dimostrazione indiretta**: scelgo un problema A' che so essere NP-arduo e dimostro che:

$$A' \leq_p A$$

La tecnica indiretta è abitualmente più semplice, in quanto devo fare la riduzione per un unico problema specifico, che inoltre può essere scelto convenientemente (si conoscono innumerevoli problemi NP completi).

Nei prossimi slides, useremo questa tecnica per dimostrare la NP-completezza di svariati problemi.

specializzazioni di SAT

Possiamo aggiungere vincoli riguardo al formato delle formule logiche.

Un rappresentazione canonica tipica è la **forma normale congiuntiva** (cnf).

L'idea è di propagare la negazione verso l'interno usando le formule

$$\neg(A \wedge B) = \neg A \vee \neg B \qquad \neg(A \vee B) = \neg A \wedge \neg B \qquad \neg\neg A = A$$

Alla fine le negazioni si arrestano su simboli atomici. Un simbolo atomico o il negato di un simbolo atomico è detto **letterale**.

Usando la formula di De Morgan

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

ci si può ridurre a congiunzioni di disgiunzioni di letterali (cnf).
Un insieme di letterali inteso in forma disgiuntiva è detto **clausola**.
Una formula in cnf è dunque un insieme (congiuntivo) di clausole.

Es: $\{\{A, \neg B\}, \{\neg A, \neg B\}, \{B\}\} = (A \vee \neg B) \wedge (\neg A \vee \neg B) \wedge B$

SAT a clausole è NP completo?

Il problema è chiaramente in NP. Per dimostrare che è NP-arduo ci basta ridurre SAT verso Sat a clausole. Ovvero dobbiamo definire una trasformazione $\varphi \rightsquigarrow \varphi_c$ tale che

- la trasformata φ_c sia in forma a clausole
- φ_c sia soddisfacibile se e solo se lo era φ
- la trasformazione prenda tempo polinomiale

Potrei pensare di utilizzare la forma normale congiuntiva, ma questo non è possibile in quanto tale trasformazione può talvolta provocare una **esplosione esponenziale** della dimensione della formula (e quindi non potrebbe essere calcolata in tempo polinomiale).

Esplosione dovuta a De Morgan

Ad esempio, la formula

$$(*) \qquad (x_1 \wedge y_1) \vee \ldots (x_n \wedge y_n)$$

genera le 2^n clausole

$$\{x_1, \ldots, x_{n-1}, x_n\}, \{x_1, \ldots, x_{n-1}, y_n\}, \ldots \{y_1, \ldots, y_{n-1}, y_n\}$$

Tuttavia la trasformazione via De Morgan mira a **preservare l'equivalenza logica**, mentre siamo solo interessati a **preservare le soddisfaciblità**.

In questo caso, possiamo trasformare $(*)$ nel modo seguente:

$$\{z_1, \ldots, z_n\}, \{\neg z_1, x_1\}, \{\neg z_1, y_1\}, \ldots \{\neg z_n, x_n\}, \{\neg z_n, y_n\}$$

Adottando la tecnica precedente è possibile dimostrare che ogni formula logica può essere trasformata in forma a clausole preservando la soddisfacibilità con una crescita al più polinomiale della sua dimensione.

NP-completezza di 3SAT

Possiamo aggiungere ulteriori vincoli riguardo al numero dei letterali in ogni clausola: nSAT è il problema della soddisfacibilità per formule composte da clausole con esattamente n letterali.

Casi interessanti sono 2SAT (polinomiale!) e 3SAT.

Teorema di Cook per 3SAT

3SAT è *NP*-completo.

Ci basta definire la riduzione da SAT a clausole verso 3SAT, cioè trasformare una clausola arbitraria in un insieme di clausole a tre-letterali, preservandone la soddisfacibilità. Casi paradigmatici:

$$\{A, \neg B\} \rightsquigarrow \{A, \neg B, C\} \wedge \{A, \neg B, \neg C\}$$

$$\{A, B, C, \neg D\} \rightsquigarrow \{A, B, E\} \wedge \{\neg E, C, \neg D\}$$

Riassumendo

- SAT: NP-completo
- SAT a clausole: NP-completo
- 3SAT: NP-completo
- 2SAT: Polinomiale

Il problema del Ricoprimento

Ricordiamo che, dato un grafo $G = (V, E)$, un ricoprimento è un sottoinsieme $V' \subseteq V$ tale che

$$\forall (u, v) \in E, u \in V' \vee v \in V'$$

Presi in input un grafo G e un intero k, il **problema decisionale del ricoprimento** $(G, k) \in VC$ consiste nel decidere se il grafo G ammette un ricoprimento V' di cardinalità $|V'| \leq k$.

Complessità del Ricoprimento (Vertex Cover)

Il problema del ricoprimento $(G, k) \in VC$ è NP-completo.

Abbiamo già dimostrato che il problema del ricoprimento è di facile *verifica* e dunque appartiene a NP. Per dimostrarne la completezza facciamo vedere che $3SAT \leq_P VC$.

3SAT vs. Ricoprimento

Sia data una formula F in 3FNC, siano $C_1, C_2, ..., C_m$ le sue clausole, con variabili proposizionali in $X_1, ..., X_n$ (possiamo supporre che $n \leq 3m$).

Costruiamo un grafo GF con $2n + 3m$ nodi nel modo seguente. Abbiamo $2n$ vertici x_i e $\overline{x_i}$ e $3m$ vertici $c_{j,k}$ per $k = 1, 2, 3$ connessi tra di loro nel modo seguente:

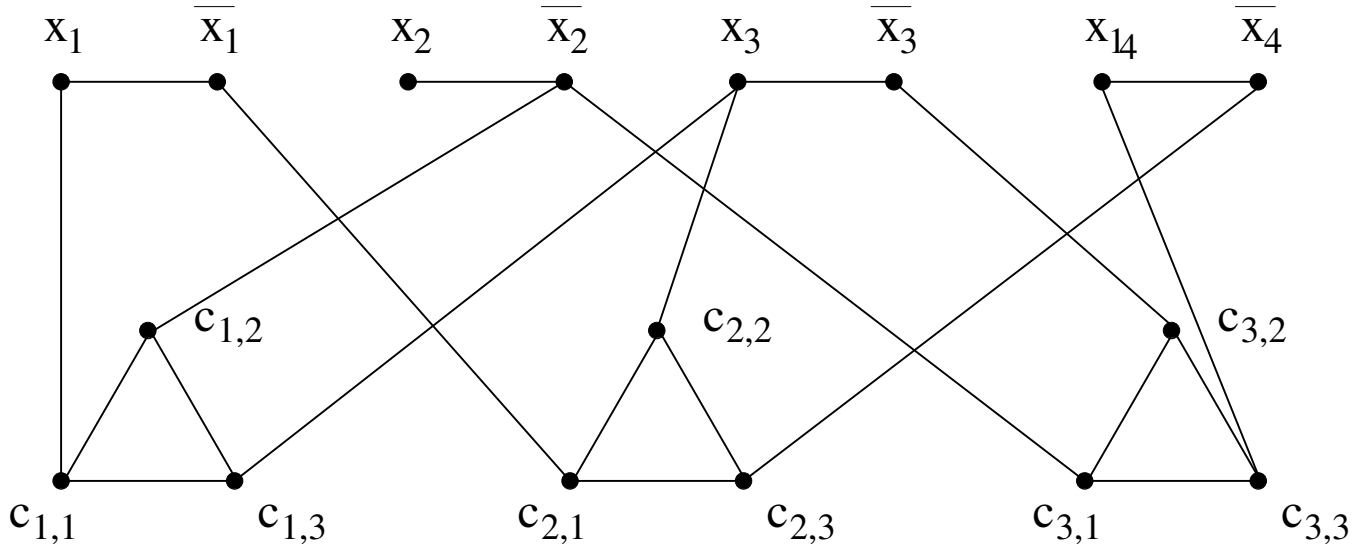
- x_i è connesso a $\overline{x_i}$ per tutti ogni $i = 1, ..., n$.
- $c_{j,1}, c_{j,2}, c_{j,3}$ sono connessi in circolo per ogni $j = 1, ..., m$
- Per ogni $C_j = (Y_1 \vee Y_2 \vee Y_3)$, per $k = 1, 2, 3$ il vertice $c_{j,k}$ è connesso a x_i se $Y_k = X_i$, ed è connesso a $\overline{x_i}$ se $Y_k = \neg X_i$

3SAT vs. Ricoprimento (2)

Ad esempio, la formula

$$F = (X_1 \vee \neg X_2 \vee X_3) \wedge (\neg X_1 \vee X_3 \vee \neg X_4) \wedge (\neg X_2 \vee \neg X_3 \vee X_4)$$

genera il grafo:



Vogliamo dimostrare che la formula F è soddisfacibile se e solo se esiste un ricoprimento S di GF di cardinalità $|S| \leq n + 2m$
(si osservi che il grafo può essere costruito in tempo polinomiale).

3SAT vs. Ricoprimento (3)

Supponiamo che F sia soddisfacibile e sia v un assegnamento di valori di verità alle variabili che rende vera F. Sia $S_1 = \{x_i | v(X_i) = true\} \cup \{\overline{x_i} | v(X_i) = false\}$. Se inoltre v soddisfa F deve soddisfare ognuna delle sue clausole, e dunque per ogni $j = 1, ..., m$ esiste $k_j \in \{1, 2, 3\}$ tale che c_{j,k_j} è adiacente a qualche vertice in S_1. Sia $S_2 = \{c_{j,k} | j = 1, ..., m \wedge k \neq k_j\}$. Allora $S := S_1 \cup S_2$ è un ricoprimento e la sua cardinalità è $n + 2m$.

Viceversa, supponiamo che esista un ricoprimento S di dimensione minore o uguale a $n + 2m$. Allora ogni triangolo $c_{j,1}, c_{j,2}, c_{j,3}$ ha almeno due vertici in S e ogni arco $(x_i, \overline{x_i})$ ha almeno un vertice in S. Dunque S ha *esattamente* $n + 2m$ elementi, con esattamente due vertici per ogni triangolo e un vertice per ogni arco $(x_i, \overline{x_i})$. Posto allora

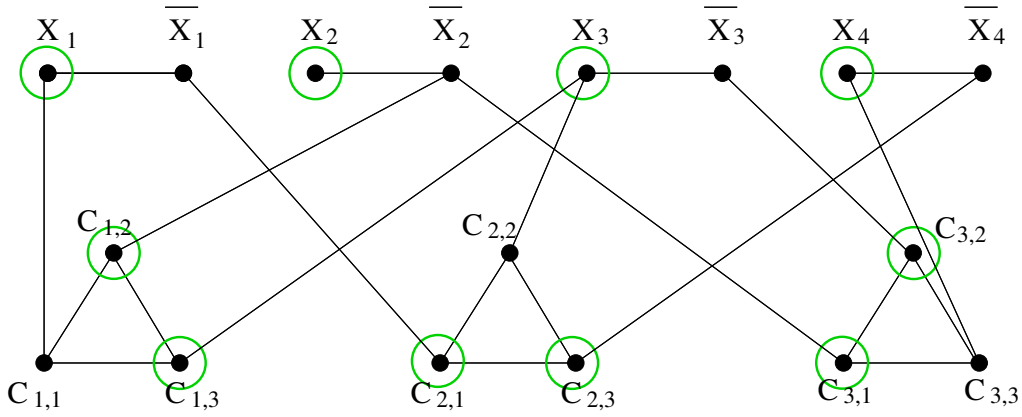
$$v(X_i) = true \Leftrightarrow x_i \in S$$

v verifica ogni clausola C_j, in quanto rende vero il letterale adiacente al vertice $c_{j,k} \notin S$. Dunque, $F \in 3SAT$.

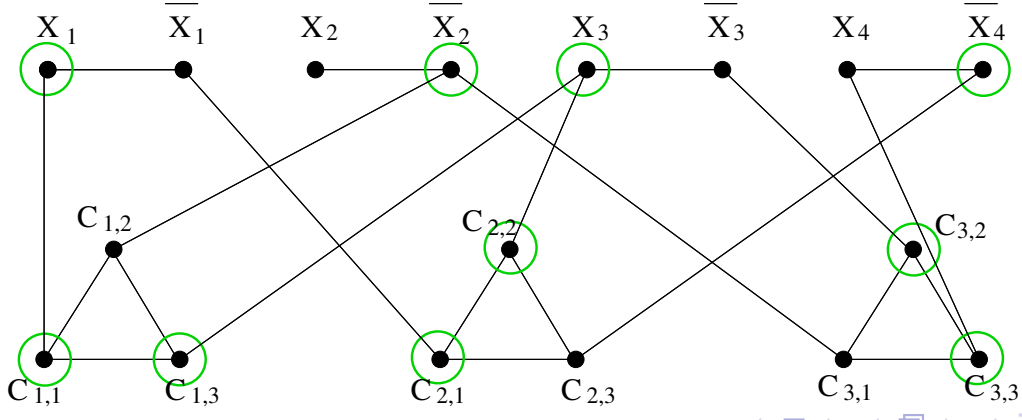
Esempi di interpretazioni/ricoprimenti

$$F = (X_1 \vee \neg X_2 \vee X_3) \wedge (\neg X_1 \vee X_3 \vee \neg X_4) \wedge (\neg X_2 \vee \neg X_3 \vee X_4)$$

- $v(X_1) = 1, v(X_2) = 1, v(X_3) = 1, v(X_4) = 1$



- $v(X_1) = 1, v(X_2) = 0, v(X_3) = 1, v(X_4) = 0$



Problema dell'insieme indipendente

Ricordiamo che, dato un grafo $G = (V, E)$, un insieme $I \subseteq V$ si dice indipendente se

$$\forall u, v \in I \Rightarrow (u, v) \notin E$$

Esistenza di insiemi indipendenti

Il problema IS di determinare se un grafo ammette un insieme indipendente I di cardinalità $|I| \geq k$ è NP-completo.

Basta ricordare che I è indipendente se e solo se $V \setminus I$ è un ricoprimento; poichè $|I| \geq k$ se e solo se' $|V \setminus I| \leq |V| - k$, questo permette di ridurre IS a VC.

Problema della Cricca

Ricordiamo che, dato un grafo $G = (V, E)$, una cricca di G è un sottoinsieme completo $C \subseteq V$, tale cioè che

$$\forall u, v \in C \Rightarrow (u, v) \in E$$

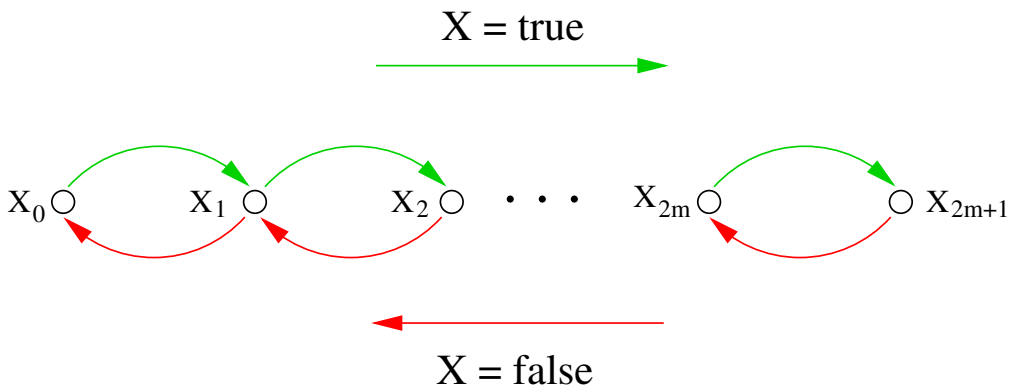
Esistenza di una cricca

Il problema Clique di determinare se un grafo ammette una cricca C di cardinalità $|C| \geq k$ è NP-completo.

Ricordiamo che C è una cricca in G se e solo se C è un insieme indipendente nel grafo $G' = (V, \overline{E})$. Poichè G' può essere costruito in tempo lineare nella dimensione di G, questo dimostra che $Clique \leq_P IS$.

Riduzione da SAT a cammino Hamiltoniano diretto

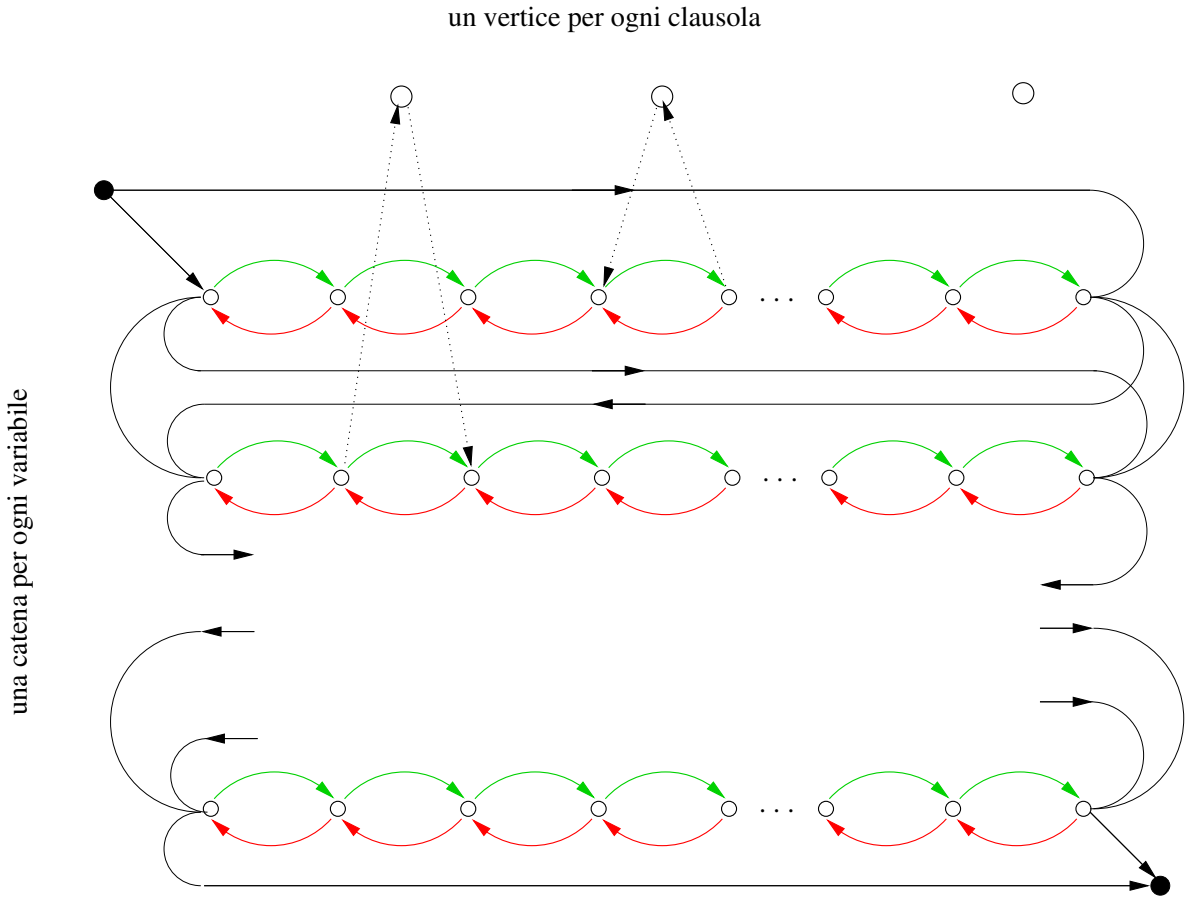
Supponiamo che il problema sia in forma di clausole $C_1, \ldots C_m$.
Dobbiamo costruire un grafo diretto che ammette un cammino Hamiltoniano se e solo se l'insieme delle clausole è soddisfacibile.
Ad ogni variabile X associamo una catena di lunghezza $2m + 2$ fatta nel modo seguente:



L'attraversamento della catena da sinistra verso destra o viceversa corrisponde ai possibili valori di verità di X.

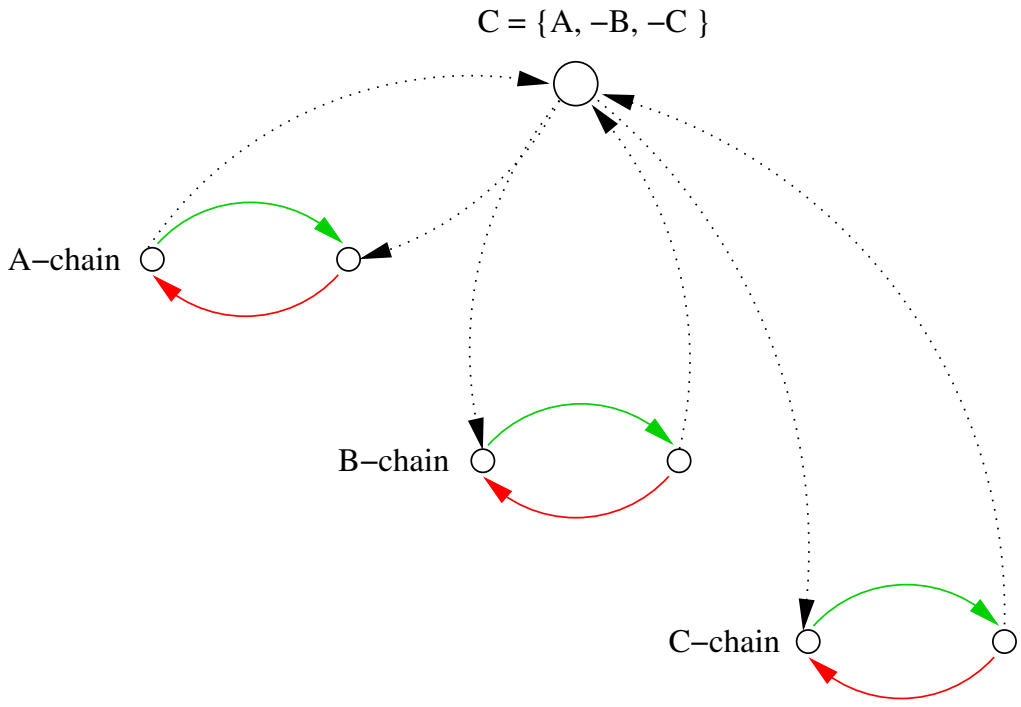
Riduzione da SAT a cammino Hamiltoniano diretto (2)

Le catene sono connesse tra di loro nel modo seguente:



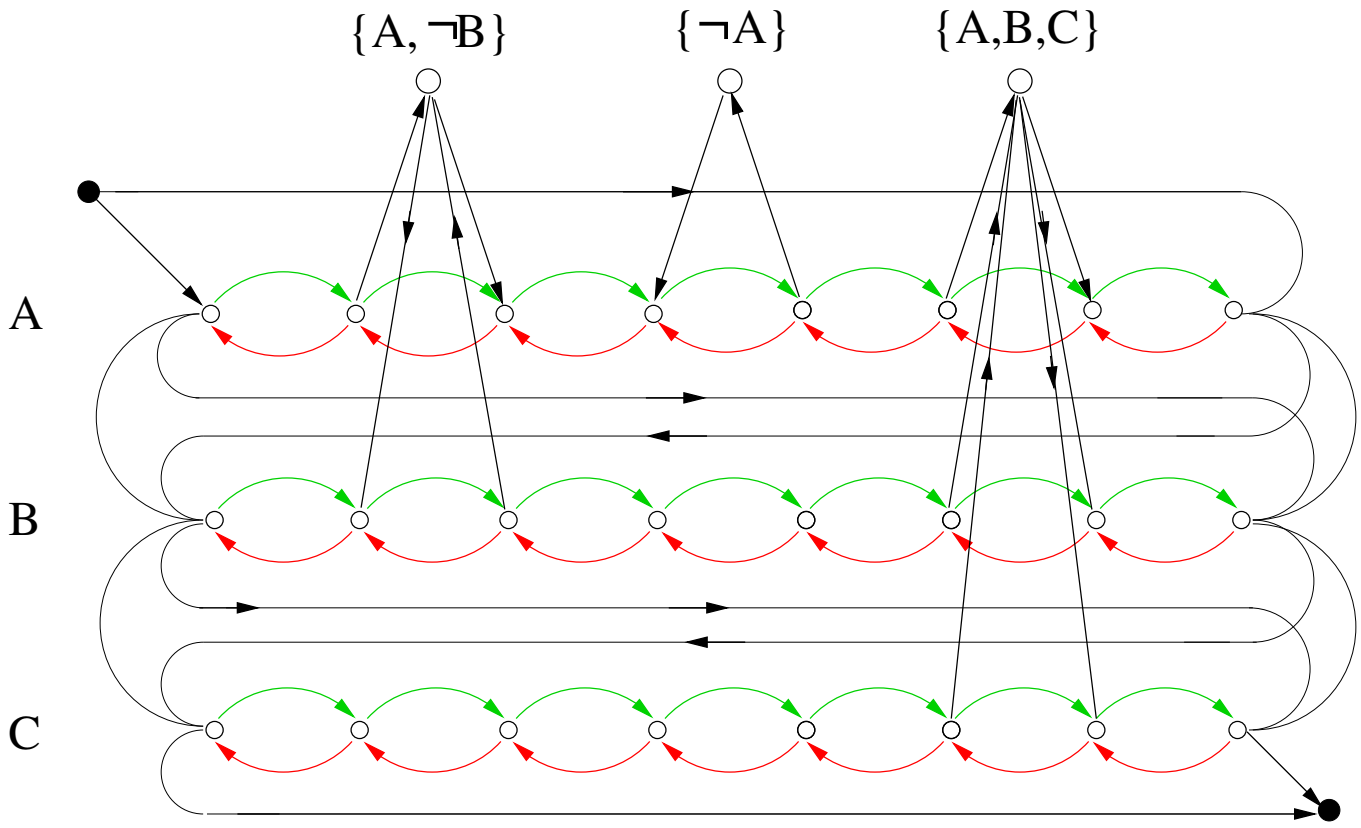
Riduzione da SAT a cammino Hamiltoniano diretto (3)

I nodi corrispondenti alle clausole sono connessi alle catene nel modo seguente



In questo modo il vertice può essere attraversato solo se la catena A è attraversata in senso positivo, **oppure** se la catena B è attraversata in senso negativo, **oppure** se la catena C è attraversata in senso negativo.

Esempio completo



Riduzione da SAT a cammino Hamiltoniano diretto (4)

E' facile convincersi che se una clausola è soddisfacibile allora il grafo ammette un cammino Hamiltoniano.

Il viceversa è meno ovvio e dipende dalla seguente osservazione: se un cammino hamiltoniano esce da una catena sul nodo u per andare verso una clausola c, deve necessariamente rientrare subito nella stessa catena e sul nodo immediatamente adiacente u'. In caso contrario il cammino sarebbe ostruito quando u' sara' visitato, in quanto gli unici nodi adiacenti sarebbero già stati tutti visitati.

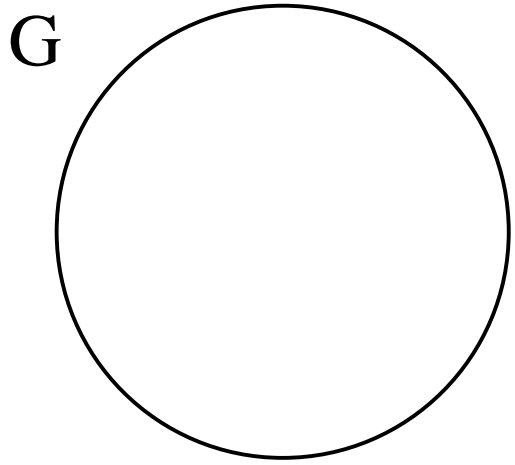
Cammino Hamiltoniano vs. Ciclo Hamiltoniano

Entrambi i problemi del Cammino Hamiltoniano (Ham-P) e del Ciclo Hamiltoniano (Ham-C) vogliono in input solo un grafo.

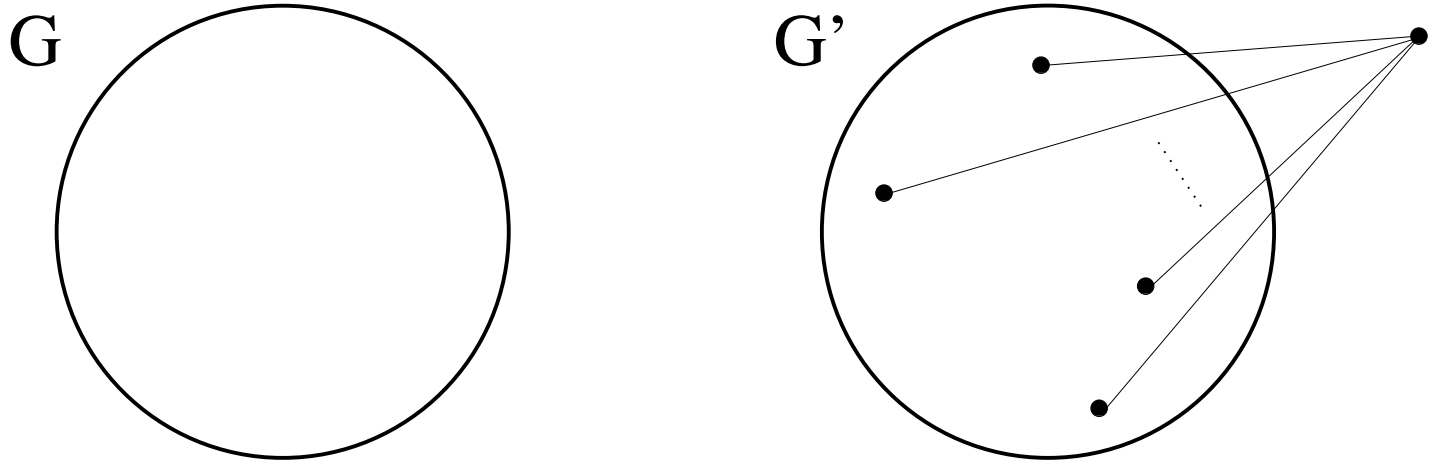
Per ridurre l'uno all'altro devo quindi immaginare delle **trasformazioni di grafi** $G \rightsquigarrow G'$, in modo tale che:

- $Ham - P \leq_p Ham - C$: G ammette un cammino hamiltoniano se e solo se G' ammette un ciclo hamiltoniano
- $Ham - C \leq_p Ham - P$: G ammette un ciclo hamiltoniano se e solo se G' ammette un cammino hamiltoniano

$Ham - P \leq_p Ham - C$

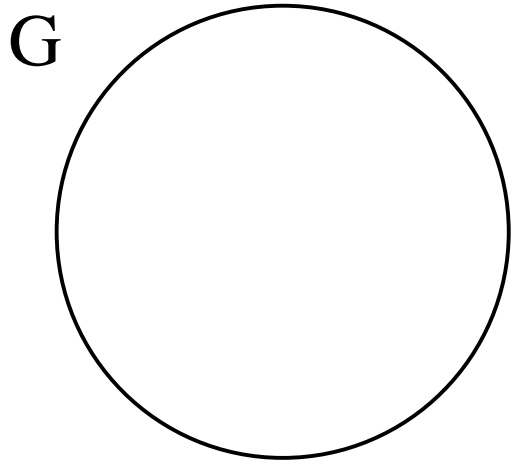


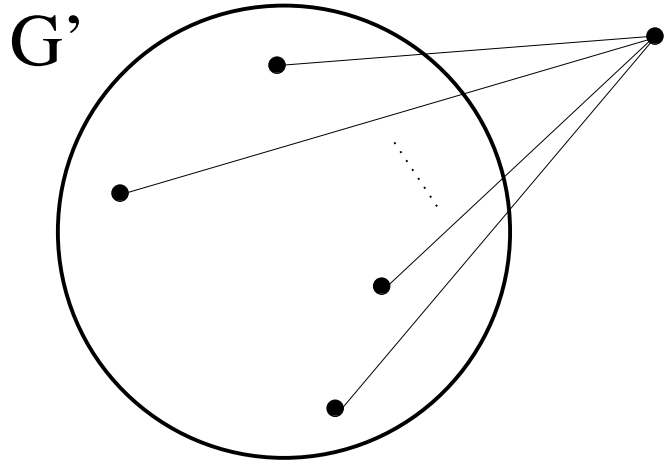
$Ham - P \leq_p Ham - C$



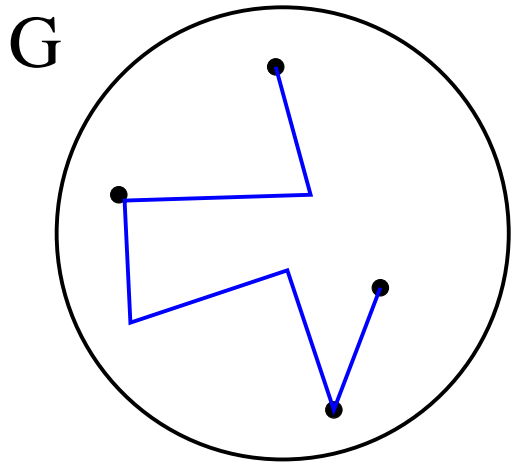
Si aggiunge un nodo e lo si connette a tutti gli altri nodi del grafo G

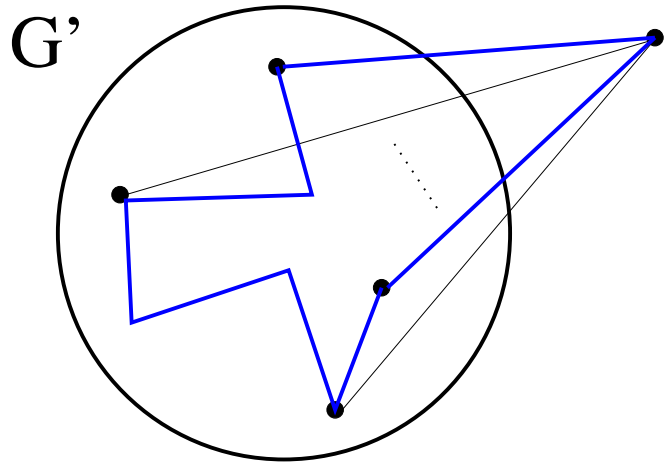
$Ham - P \leq_p Ham - C$





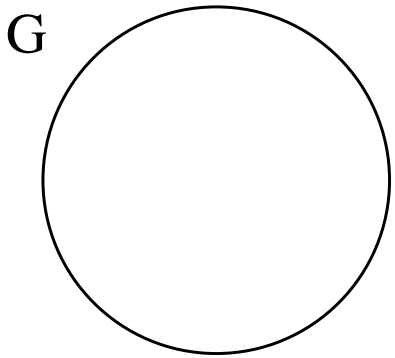
Si aggiunge un nodo e lo si connette a tutti gli altri nodi del grafo G



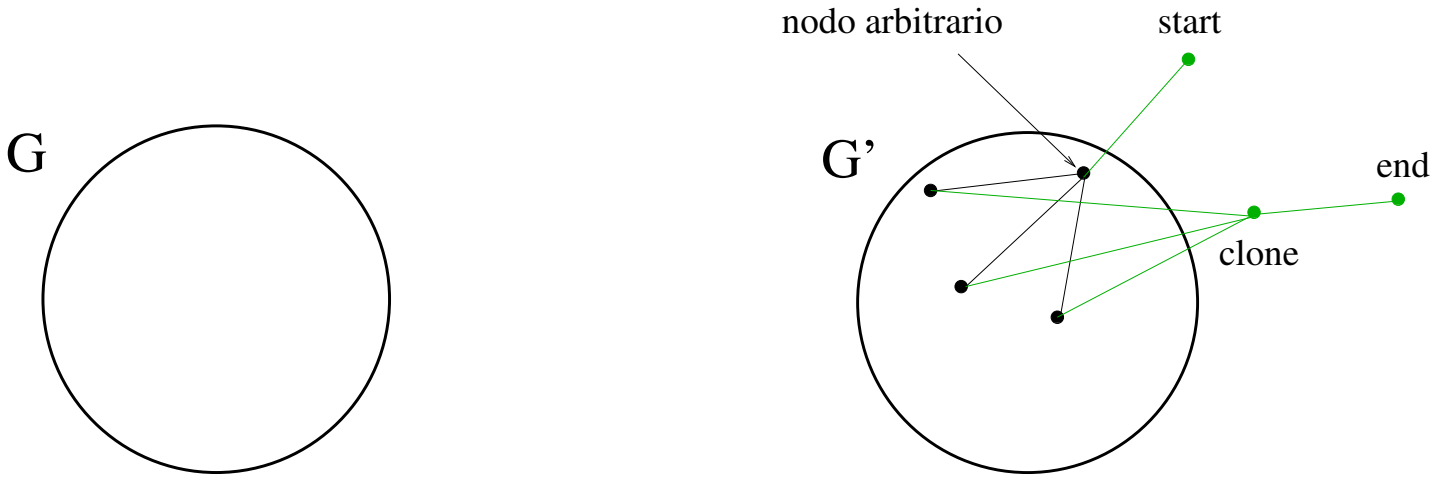


un cammino diventa un ciclo, e viceversa

$Ham - C \leq_p Ham - P$

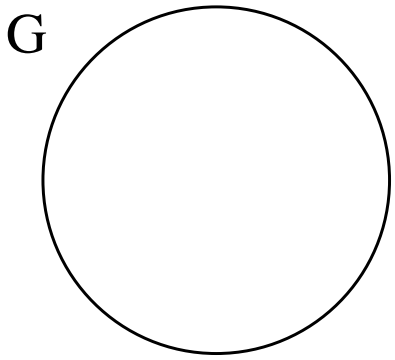


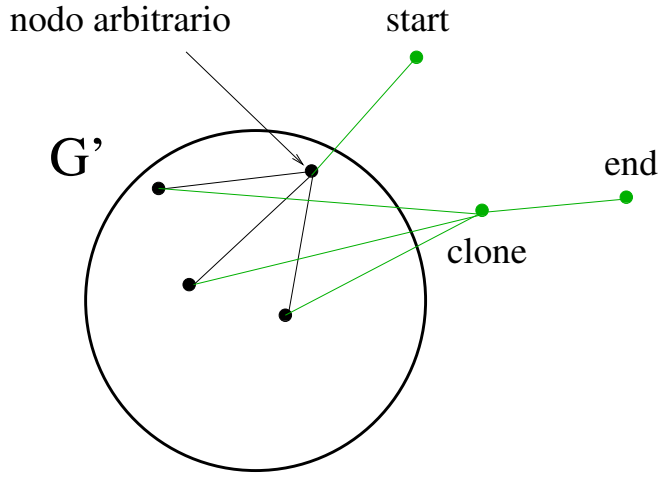
$Ham - C \leq_p Ham - P$



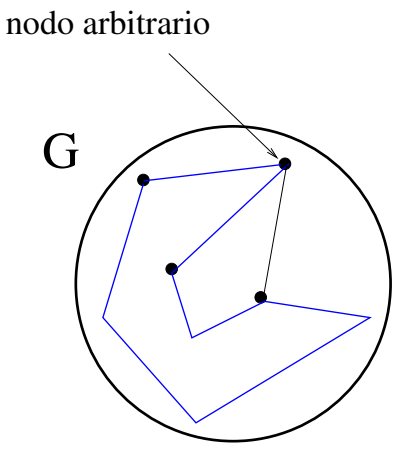
clonare un nodo i, connettere i a un nodo start, e il suo clone a un nodo end

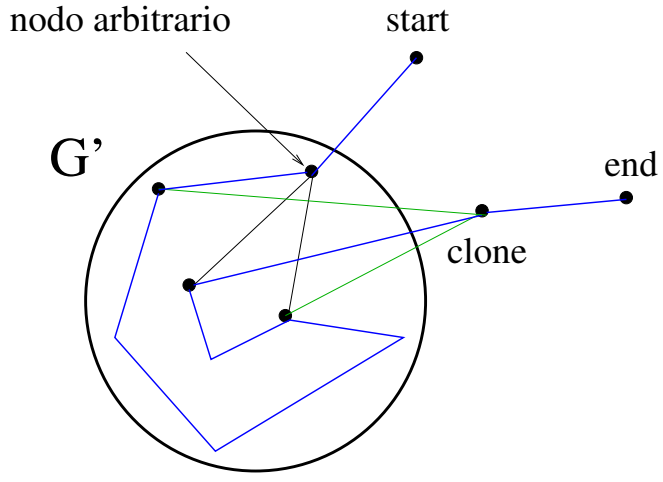
$Ham - C \leq_p Ham - P$





clonare un nodo i, connettere i a un nodo start, e il suo clone a un nodo end





un ciclo diventa un cammino, e viceversa

Cammini lunghi

Il problema di determinare se un Grafo G ammette cammini semplici di lunghezza maggiore o oguale a k è NP-completo.

È ovvio che il problema è in NP. La completezza deriva dal fatto che **generalizza** il problema del cammino hamiltoniano: basta prendere k = n-1, dove n è il numero dei nodi del grafo.

Cammini lunghi

Il problema di determinare se un Grafo G ammette cammini semplici di lunghezza maggiore o oguale a k è NP-completo.

È ovvio che il problema è in NP. La completezza deriva dal fatto che **generalizza** il problema del cammino hamiltoniano: basta prendere k = n-1, dove n è il numero dei nodi del grafo.

DA RICORDARE

- cercare cammini di lunghezza $\leq K$ un problema in P: visita in larghezza
- cercare cammini di lunghezza $\geq K$ è un problema NP-completo

Hitting Set

Sia S un insieme e un sia $C = S_1, \ldots, S_n$ una collezione di sottoinsiemi non vuoti di S. Un **hitting set** per C è un sottoinsieme H di S tale che per ogni i, $H \cap S_i \neq \emptyset$.

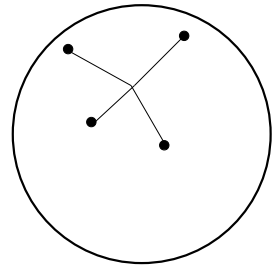
Esempi:

- S è un hitting set per C (banale).
- sia $C = \{\{1,2,3\},\{0,1\},\{0,4,5\},\{2,5\},\{6\}\}$. $H = \{1,5,6\}$ è un hitting set per C di cardinalità 3 (non unico). C non ha hitting set di cardinalità 2.

Il problema di determinare, dato C, se esiste un hitting set di cardinalità minore o uguale a k è NP-completo.

VC $\leq_p$ Hitting Set

Si può vedere S come un insieme di nodi e $C = S_1, \ldots, S_n$ come un insieme di **iperarchi tra questi nodi**.



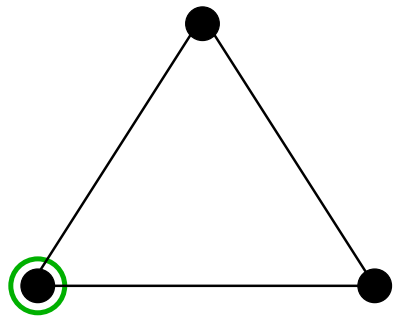
È evidente allora che Hitting Set **generalizza** Vertex Cover: un hitting set è un sotto insieme di nodi $H \subseteq S$ tale che ogni iperarco risulta coperto: $H \cap S_i \neq \emptyset$.

Formalmente, dato un grafo $G = (V, E)$, sia C la collezione delle coppie $\{u, v\}$ tali che $(u, v) \in E$. Per definizione, un ricoprimento per G è anche un hitting set per C, e dunque G ha un ricoprimento di dimensione minore o uguale a k se e solo se C ha un hitting set di dimensione minore o uguale a k.

Insieme dominante

Dato un grafo $G = (V, E)$ un **insieme dominante** è un sottoinsieme di nodi $V' \subset V$ tale che ogni nodo del grafo può essere raggiunto da un nodo in V' in al piú un passo.

La nozione è simile al ricoprimento di vertici; in particolare, ogni ricoprimento è un insieme dominante, ma non vale il viceversa.

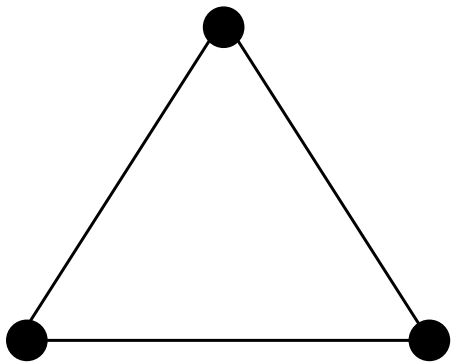


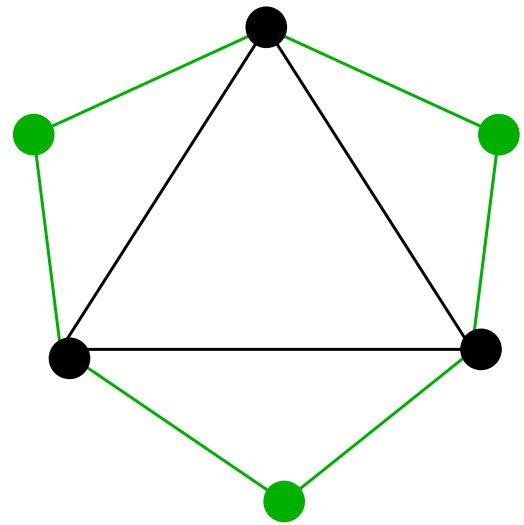
Il problema di determinare se esiste un insieme dominante di dimensione minore o uguale a k (DOM) è NP-completo.

Lo dimostriamo riducendo ad esso il problema del ricoprimento.

VC $\leq_p$ DOM

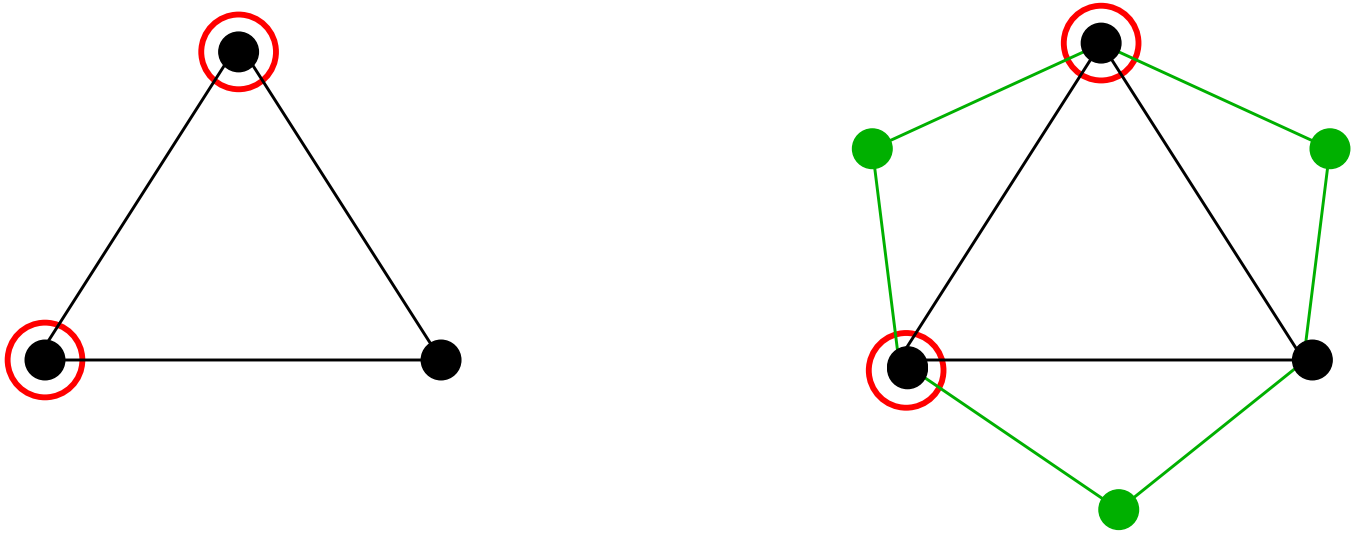
Dato un grafo $G = (V, E)$, costruiamo un nuovo grafo G' aggiungendo un nuovo nodo n_{uv} per ogni arco $(u, v) \in E$ e connettendo n_{uv} sia ad u che a v.





VC $\leq_p$ DOM

Un ricoprimento V' in G è anche un insieme dominante in G'.

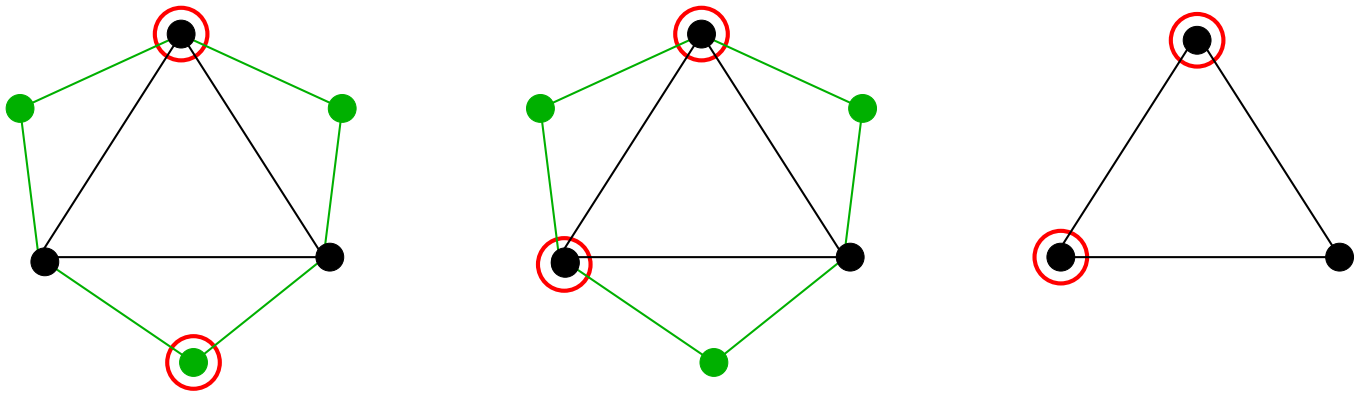


Questo perchè per ogni arco $(u, v) \in E$ almeno una delle due estremità appertiene al ricoprimento V' e da quella estremità in un passo si raggiunge il nuovo nodo n_{uv}.

VC $\leq_p$ DOM

Viceversa sia dato un insieme dominante V' in G'. Se questo comprende dei nodi aggiuntivi n_{nv} li si può rimpiazzare con u o v (indifferentemente) senza perdere in dominanza e senza aumentare la cardinalità.

L'insieme risultante V'' è un ricoprimento in G.



Questo perchè ogni nodo n_{uv} deve essere dominato da V'', e l'unico modo che ho di farlo è da una delle due estremità u o v dell'arco (u, v) soggiacente a n_{uv}.

Commesso viaggiatore

Il problema TSP del **commesso viaggiatore** consiste nel determinare l'esistenza o meno di un ciclo di lunghezza data k in un grafo di n città con distanze assegnate d_{ij} (intere positive) tra ogni coppia di esse.

È chiaro che il problema è in NP.

Commesso viaggiatore

Il problema TSP del **commesso viaggiatore** consiste nel determinare l'esistenza o meno di un ciclo di lunghezza data k in un grafo di n città con distanze assegnate d_{ij} (intere positive) tra ogni coppia di esse.

È chiaro che il problema è in NP.

Possiamo dimostarne la completezza riducendo ad esso il problema del ciclo hamiltoniano HAM-C.

Dato un grafo G di n nodi generiamo il grafo G' totalmente connesso con distanze d_{ij} definite nel modo seguente:

$$d_{ij} = \begin{cases} 1 & \text{se } (i,j) \in G \\ 2 & \text{se } (i,j) \notin G \end{cases}$$

È immediato osservare che esiste un cammino hamiltoniano in G se e solo se esiste un ciclo di lunghezza inferiore a $n+1$ in G'.

Programmazione intera

Per **programmazione intera** si intende il problema di stabilire se un insieme finito di equazioni e disequazioni (di solito lineari - ILP) a coefficienti interi ammette una soluzione intera.

Il problema è chiaramente in NP, in quanto, data la soluzione, è banale verificarne la correttezza.

Per dimostrarne la completezza è sufficiente ridurre un qualche problema NP-completo a ILP. ILP è molto duttile e c'e' solo l'imbarazzo della scelta.

Vediamo qualche esempio.

SAT a clausole $\leq_p$ ILP

Data una formula con n variabili e m clausole, costruiamo un sistema di disequazioni nel modo seguente.

- per ogni variabile prop. A, consideriamo due variabili intere A e $\overline{A}$.
- aggiungiamo i seguenti vincoli

$$A \geq 0 \qquad \overline{A} \geq 0 \qquad A + \overline{A} = 1$$

- per ogni clausola $C = \{L_1, \ldots L_k\}$, aggiungiamo il vincolo

$$L_1 + L_2 \cdots + L_k \geq 1$$

È evidente che la formula è soddisfacibile se e solo se il sistema ammette una soluzione.

TSP $\leq_p$ ILP

Consideriamo variabili $x_{i,j}$ con l'interpretazione che alla tappa i il commesso si trova nella città j. Queste sono vincolate nel seguente modo:

- ad ogni tappa i il commesso è in una e una sola città:

$$\sum_j x_{i,j} = 1$$

- ogni città j deve essere attraversata una e una sola volta:

$$\sum_i x_{i,j} = 1$$

- sommiamo le distante condizionate dal percorso:

$$\sum_{i,j,k} x_{i,j} * x_{i+1,k} * d_{jk} \leq k$$

L'ultimo vincolo non è lineare. Si può costruire la congiunzione $v_{i,j,k}$ che esprime il transito da j a k alla tappa i con le seguenti condizioni (oltre a $0 \leq v_{i,j,k} \leq 1$):

$$v_{i,j,k} \leq x_{i,j} \quad v_{i,j,k} \leq x_{i+1,k} \quad v_{i,j,k} \geq x_{i,j} + x_{i+1,k} - 1$$

Mezza cricca

Spesso, casi particolari di problemi NP-completi continuano a esserlo.

Un esempio tipico è il problema della **mezza cricca** che consiste nel determinare l'esistenza di una cricca di dimensione pari alla metà del numero dei nodi del grafo.

Facciamo vedere come il problema generale della cricca può sempre essere ridotto a questo caso particolare.

Sia dato un grafo di n nodi. Dobbiamo distinguere due casi a seconda che la cricca cercata abbia dimensione k, con $2k \geq n$ o meno.

Il principio che utilizziamo è che aggiungendo nodi **isolati** aumentiamo la dimensione del grafo senza aumentare la dimensione della cricca.

Viceversa, aggiungendo nodi **densamente connessi** aumentiamo sia la dimensione del grafo che quella della cricca.

Mezza cricca (2)

- caso $2k \geq n$:
  Supponiamo di aggiungere m **nodi isolati**. Il grafo assume dimensione $n + m$, e la cricca non aumenta la sua dimensione k.

  Quanti nodi devo aggiungere?

Mezza cricca (2)

- caso $2k \geq n$:
  Supponiamo di aggiungere m **nodi isolati**. Il grafo assume dimensione $n + m$, e la cricca non aumenta la sua dimensione k.

  Quanti nodi devo aggiungere?

  Dobbiamo fare in modo che $n + m = 2k$ e dunque

$$m = 2k - n$$

Mezza cricca (2)

- caso $2k \geq n$:
  Supponiamo di aggiungere m **nodi isolati**. Il grafo assume dimensione $n + m$, e la cricca non aumenta la sua dimensione k.

  Quanti nodi devo aggiungere?

  Dobbiamo fare in modo che $n + m = 2k$ e dunque

  $$m = 2k - n$$

- caso $2k \leq n$:
  Supponiamo di aggiungere m nodi **densamente connessi**. Il grafo assume dimensione $n + m$, e abbiamo una cricca di dimensione $k + m$.

  Quanti nodi devo aggiungere?

Mezza cricca (2)

- caso $2k \geq n$:
  Supponiamo di aggiungere m **nodi isolati**. Il grafo assume dimensione $n + m$, e la cricca non aumenta la sua dimensione k.

  Quanti nodi devo aggiungere?

  Dobbiamo fare in modo che $n + m = 2k$ e dunque

  $$m = 2k - n$$

- caso $2k \leq n$:
  Supponiamo di aggiungere m nodi **densamente connessi**. Il grafo assume dimensione $n + m$, e abbiamo una cricca di dimensione $k + m$.

  Quanti nodi devo aggiungere?

  Dobbiamo fare in modo che $n + m = 2(k + m)$ e dunque

  $$m = n - 2k$$

Knapsack

Consideriamo un caso particolare, noto come Subset Sum.

Subset Sum problem: dati n interi positivi $w_1, \ldots, w_n$ e una somma totale W, decidere se esiste un sottoinsieme $I \subseteq \{1, ..., n\}$ tale che

$$\sum_{i \in I} w_i = W$$

È chiaro che il problema è in NP.

Per dimostrare che è anche NP-arduo riduciamo ad esso il problema 3SAT.

3SAT $\leq_P$ Knapsack

Supponiamo di avere m clausole $c_0, \ldots, c_{m-1}$ e n variabili proposizionali $A_0, \ldots A_{n-1}$.

Ad ogni variabile A_i assoceremo due numeri n_{A_i} e $n_{\overline{A_i}}$ corrispondenti ai due possibili valori di verità per A.

In particolare:

- le prime n cifre (meno significative) identificano via one-hot encoding la variabile in questione: per A_i, la cifra i è 1, e tutte le altre sono 0
- le successive m cifre indicano quali clausole sono soddisfatte dai rispettivi valori di verità per la variabile. Nel numero n_L, la cifra $j + n$, per $0 \leq j < m$, è 1 se il letterale $L \in c_j$ e 0 altrimenti.

Esempio

Consideriamo le seguenti clausole:

$$c_3 = \{A, \overline{B}, C\}\ c_2 = \{\overline{A}, B, \overline{C}\}\ c_1 = \{\overline{A}, \overline{B}, C\}\ c_0 = \{A, B, \overline{C}\}$$

Allora:

numero		*clausole*				*variabili*		
n_A	=	1	0	0	1	0	0	1
$n_{\overline{A}}$	=	0	1	1	0	0	0	1
n_B	=	0	1	0	1	0	1	0
$n_{\overline{B}}$	=	1	0	1	0	0	1	0
n_C	=	1	0	1	0	1	0	0
$n_{\overline{C}}$	=	0	1	0	1	1	0	0

N.B. I numeri sono composti solo da 0 e 1 ma **non** lavoriamo in base binaria. Dobbiamo invece prendere una base sufficientemente grande da escludere la possibilità di riporti quando faremo la somma dei numeri. La base 10 è sufficiente (basta anche base 6)

numeri riempitivi

Per ogni calusola c_j aggiungiamo infine **due** numeri che hanno cifra 1 in posizione $j + n$ (la posizione della clausola j) e le cui altre cifre sono 0.

Nel nostro caso, abbiamo 4 clausole e dobbiamo aggiungere 2*4=8 numeri:

numero		*clausole*				*variabili*		
$n_{c'_0}$	=	0	0	0	1	0	0	0
$n_{c''_0}$	=	0	0	0	1	0	0	0
$n_{c'_1}$	=	0	0	1	0	0	0	0
$n_{c''_1}$	=	0	0	1	0	0	0	0
$n_{c'_2}$	=	0	1	0	0	0	0	0
$n_{c''_2}$	=	0	1	0	0	0	0	0
$n_{c'_3}$	=	1	0	0	0	0	0	0
$n_{c''_3}$	=	1	0	0	0	0	0	0

Abbiamo quindi un totale di 6+8 numeri. Il numero target è

$$W = 3333111$$

riduzione (se)

$n.$		*clausole*				*variabili*		
n_A	$=$	1	0	0	1	0	0	1
$n_{\overline{A}}$	$=$	0	1	1	0	0	0	1
n_B	$=$	0	1	0	1	0	1	0
$n_{\overline{B}}$	$=$	1	0	1	0	0	1	0
n_C	$=$	1	0	1	0	1	0	0
$n_{\overline{C}}$	$=$	0	1	0	1	1	0	0
$n_{c'_0}$	$=$	0	0	0	1	0	0	0
$n_{c''_0}$	$=$	0	0	0	1	0	0	0
$n_{c'_1}$	$=$	0	0	1	0	0	0	0
$n_{c''_1}$	$=$	0	0	1	0	0	0	0
$n_{c'_2}$	$=$	0	1	0	0	0	0	0
$n_{c''_2}$	$=$	0	1	0	0	0	0	0
$n_{c'_3}$	$=$	1	0	0	0	0	0	0
$n_{c''_3}$	$=$	1	0	0	0	0	0	0

Supponiamo che le clausole siano soddisfatte dalla attribuzione di verità v.

Per ogni var A prendiamo n_A o $n_{\overline{A}}$ a seconda che $v(A) = 1$ o $v(A) = 0$.

Ogni clausola j è soddisfatta, dunque la loro somma in posizione $n+j$ è una cifra compresa tra 1 e 3. Lo portiamo a tre sommando 0, 1, o 2 rispettivi numeri riempitivi.

Ad esempio, per A=1,B=0,C=0, prendiamo i numeri evidenziati, la cui somma è

3333111

riduzione (solo se)

Viceversa, supponiamo che la somma sia 3333111.

$n.$		*clausole*				*variabili*		
n_A	$=$	1	0	0	1	0	0	1
$n_{\overline{A}}$	$=$	0	1	1	0	0	0	1
n_B	$=$	0	1	0	1	0	1	0
$n_{\overline{B}}$	$=$	1	0	1	0	0	1	0
n_C	$=$	1	0	1	0	1	0	0
$n_{\overline{C}}$	$=$	0	1	0	1	1	0	0
$n_{c'_0}$	$=$	0	0	0	1	0	0	0
$n_{c''_0}$	$=$	0	0	0	1	0	0	0
$n_{c'_1}$	$=$	0	0	1	0	0	0	0
$n_{c''_1}$	$=$	0	0	1	0	0	0	0
$n_{c'_2}$	$=$	0	1	0	0	0	0	0
$n_{c''_2}$	$=$	0	1	0	0	0	0	0
$n_{c'_3}$	$=$	1	0	0	0	0	0	0
$n_{c''_3}$	$=$	1	0	0	0	0	0	0

La somma di un sottoinsieme dei numeri dati non può mai causare un riporto.

Quindi possiamo ragionare indipendentemente sulle singole cifre.

Per quanto riguarda le cifre da 1 a n, possiamo prendere uno e uno solo dei due numeri n_A e $n_{\overline{A}}$, per ogni A.

Per quanto riguarda le cifre tra n e $n + m$, siccome possiamo al più selezionare 2 numeri riempitivi per la clausola j, la quantità restante deve venire dai numeri n_L, che dunque soddisfano la clausola.

Equazioni Diofantee

Teorema di Manders e Adleman (1978)

Il problema di decidere se un un polinomio di secondo grado a coefficienti interi ammette soluzioni intere è *NP*-completo.

Si dimostra per riduzione di *3SAT* e richiede un po' di teoria dei numeri.

Typo di equazione	Complessità
lineare	polinomiale (Algoritmo di Euclide)
quadratica	NP-completo (Manders e Adleman)
arbitraria	indecidibile (Davis, Robinson e Matiyasevich)

Lezioni 16-17:Complessità relativizzata

- Macchine di Turing con oracolo
- Classi di Complessità con oracolo

Macchine ad oracolo

Una Macchina di Turing non deterministica con oracolo è definita da un ennupla $M = (Q, q0, q?, q+, q-, F, \Sigma, \Gamma, B, k, \delta)$ con il significato inteso per le MdTN, a parte le seguenti caratterisitche:

- M è equipaggiata con un nastro particolare, detto nastro di interrogazione;
- M ha tre stati speciali $q?, q+, q- \in Q \setminus F$, dove $q?$ è lo stato di interrogazione, e $q+, q-$ sono gli stati di risposta.
- la funzione di transizione δ non è definita sullo stato di interrogazione

Semantica delle MdTN ad oracolo

La nozione di computazione è definita nel modo abituale, ad eccezione delle regole seguenti:

- la macchina può scrivere sul nastro di interrogazione come su di un nastro abituale
- nel momento in cui la macchina entra nello stato di interrograzione $q?$ lo stato successivo non è determinato dalla funzione δ ma da un oracolo esterno O. In particolare se nello stato $q?$ il nastro di interrogazione contiene la parola $y \in \Sigma^*$ alla sinistra della testina, allora lo stato successivo della macchina è $q+$ se $y \in O$ e $q-$ se $y \notin O$
- Il contenuto del nastro di interrogazione è cancellato automaticamente non appena la macchina rientra nello stato $q+$ o $q-$

Denotiamo con L_O (f_M) il linguaggio accettato (la funzione calcolata) dalla macchina M con oracolo O.

Tempo e Spazio per MdTN ad oracolo

Sia M una MdTN con oracolo A:

- $time_M^A(x)$ è definito come nel caso deterministico, dove la transizione dallo stato di interrogazione a quello di risposta ha costo unitario.
- $t_M^A(n)$ è il massimo $time_M^A(x)$ al variare di x su tutti le stringhe di lunghezza $|x| = n$
- $t_M(n)$ è il massimo $t_M^A(n)$ per $A \subseteq \Sigma^*$.

$space_M^A(x)$, $s_M^A(n)$ e $s_M(n)$ sono definite in modo analogo (lo spazio richiesto per la scrittura sul nastro di interrogazione è rilevante).

Classi di complessità con oracolo

Sia C una classe di complessità e sia $O \subset \Sigma^*$ un oracolo. La classe C^O è definita in modo analogo a C con la differenza che si considerano macchine con oracolo O invece delle macchine abituali.

Ad esempio P^{SAT} è l'insieme dei linguaggi che ammettono un algoritmo di decisione polinomiale, ammesso di avere un oracolo per SAT.

Se inoltre C' è una classe di complessità, allora

$$C^{C'} := \bigcup_{O \in C'} C^O$$

ovvero è la classe dei problemi che hanno complessità C ammesso di avere un opportuno oracolo per i problemi di complessità C'.

Ad esempio NP^{PSPACE} è l'insieme dei linguaggi riconoscibili in tempo polinomiale non deterministico mediante una qualche macchina che utilizza un oracolo relativo ad un problema in $PSPACE$.

Alcune proprietà delle Classi con oracolo

Per tutti i linguaggi $A, B, C \in \Sigma^*$

- $A \in P^A$
- $A \in P^B \Rightarrow A \in NP^B$
- $A \in NP^B \Rightarrow A \in NP^{\overline{B}}$
- $A \in P^B, B \in P^C \Rightarrow A \in P^C$
- $A \in NP^B, B \in P^C \Rightarrow A \in NP^C$

N.B.

$$A \in P^B, B \in NP^C \quad \not\Rightarrow \quad A \in P^C \text{ (nè } A \in NP^C)$$
$$A \in NP^B, B \in NP^C \quad \not\Rightarrow \quad A \in NP^C$$

Ad esempio, $TAU \in P^{SAT}, SAT \in NP^{\emptyset} = NP$, ma (probabilmente) $TAU \notin NP$.

Alcune Classi di Complessità con oracolo

Lemma

1. $P^P = P$
2. $NP^P = NP$
3. $NP^{PSPACE} = PSPACE$

Dimostrazione:

1. $P \subseteq P^P$ poichè $A \in P^A$; $P^P \subseteq P$ segue da $A \in P^B, B \in P^C \Rightarrow A \in P^C$ con $C = \emptyset$.
2. Analoga alla precedente
3. $PSPACE \subseteq NP^{PSPACE}$ poichè $A \in NP^A$.
   $NP^{PSPACE} \subseteq PSPACE$ si ottiene modificando al caso con oracolo la dimostrazione che $NTIME(f) \subseteq DSPACE(f)$. Vediamo i dettagli:

$NP^{PSPACE} \subseteq PSPACE$

Supponiamo che M sia un MdTN con oracolo $A \in PSPACE$ e che $t_M \leq p$ per qualche polinomio p. Otteniamo una macchina deterministica M' eliminando il nondeterminismo con la tecnica utilizzata nella dimostrazione che $NTIME(f) \subseteq DSPACE(f)$ (ovvero esplorando esaustivamente l'albero delle computazione di M).

Abbiamo che $L^A_{M'} = L^A_M$ e $s^A_{M'} \in O(p)$. Poichè ogni interrogazione $y \in A$ posta dalla macchina M' su input x (analoghe a quelle di M) ha lunghezza $|y| \leq p(|x|)$ e $A \in PSPACE$, possiamo rimpiazzare l'oracolo con un sottoprogramma per A utilizzando una qualche MdT M'' tale che $L_{M''} = A$ e $s_{M''} \leq q$ per un qualche polinomio q.

La macchina risultante M''' è deterministica, $L_{M'''} = L^A_{M'} = L^A_M$ e $s_{M'''} \in O(pq)$. Dunque,$L^A_M \in PSPACE$.

$NP \subseteq P^{NP} \subseteq NP^{NP}$

Lemma

$NP^P = NP \subseteq P^{NP} \subseteq NP^{NP}$

Abbiamo già dimostrato la prima uguaglianza.

- $NP \subseteq P^{NP}$ segue dal fatto che $A \in P^A$.
- $P^{NP} \subseteq NP^{NP}$ segue dal fatto che $A \in P^B \Rightarrow A \in NP^B$.

NP e coNP

NP e coNP

$NP^{NP} = NP$ se e solo se $NP = coNP$.

Dimostriamo che $NP = coNP$ implica $NP^{NP} = NP$.
Sappiamo che $NP \subseteq NP^{NP}$; dobbiamo quindi imostrare che $NP^{NP} \subseteq NP$.

Sia data una MdTN M che opera in tempo $t_M \in O(q)$ per qualche polinomio q e un qualche oracolo $A \in NP$. Siccome $A \in NP = coNP$, esistono due MdTN $M+$ e $M-$ tali che $M+$ accetta A e $M-$ accetta $\overline{A}$, entrambe in tempo polinomiale p.

Possiamo allora costruire una nuova MdTN M' che opera nel modo seguente: ogni interrogazione $y \in A$ di M è rimpiazzata da una chiamata simultanea a $M+$ e $M-$ su input y. Se una delle due macchine $M+$ o $M-$ termina con accettazione, allora si entra rispettivamente nello stato $q+$ o $q-$ e si riprende la simulazione di M. La macchina M' riconosce L_M in tempo $O(qp)$.

NP e coNP

Mostriamo ora che $NP^{NP} = NP$ implica $NP = coNP$.
Ricordiamo innanzi tutto che per ogni A

$$NP^A = NP^{\overline{A}}$$

Abbiamo allora le seguenti proprietà:

- se $A \in coNP$, allora $\overline{A} \in NP$ e dunque

  $$A \in NP^A = NP^{\overline{A}} \subseteq NP^{NP} = NP$$

  che dimostra che $coNP \subseteq NP$
- se $A \in NP$, allora

  $$\overline{A} \in NP^{\overline{A}} = NP^A \subseteq NP^{NP} = NP$$

  e dunque $A \in coNP$, che dimostra che $NP \subseteq coNP$

Lezioni 18-19: La gerarchia polinomiale

- La gerarchia polinomiale
- Il problema relativizzato della fermata limitata
- Il Teorema di Cook relativizzato
- PSPACE-Completezza

La gerarchia polinomiale

Per $n \in N$ si definiscono le seguenti classi:

- $\Sigma_0^P := \Pi_0^P := \Delta_0^P := P$
- $\Sigma_{n+1}^P := NP^{\Sigma_n^P}$
- $\Pi_{n+1}^P := co\Sigma_{n+1}^P$
- $\Delta_{n+1}^P := P^{\Sigma_n^P}$

La classe

$$PH := \bigcup_{n \in N} \Sigma_n^P$$

è detta gerarchia polinomiale.
Osservazioni:

- $\Sigma_1^P = NP^P = NP$, $\Sigma_2^P = NP^{NP}$, $\Sigma_3^P = NP^{NP^{NP}}$, $\ldots$
- $\Delta_1^P = P^P = P$, $\Delta_2^P = P^{NP}$, $\Delta_3^P = P^{NP^{NP}}$, $\ldots$
- se $P = NP$, allora $PH = P$
- se $coNP = NP$, allora $PH = NP$.

Il teorema della gerarchia

Teorema della gerarchia polinomiale

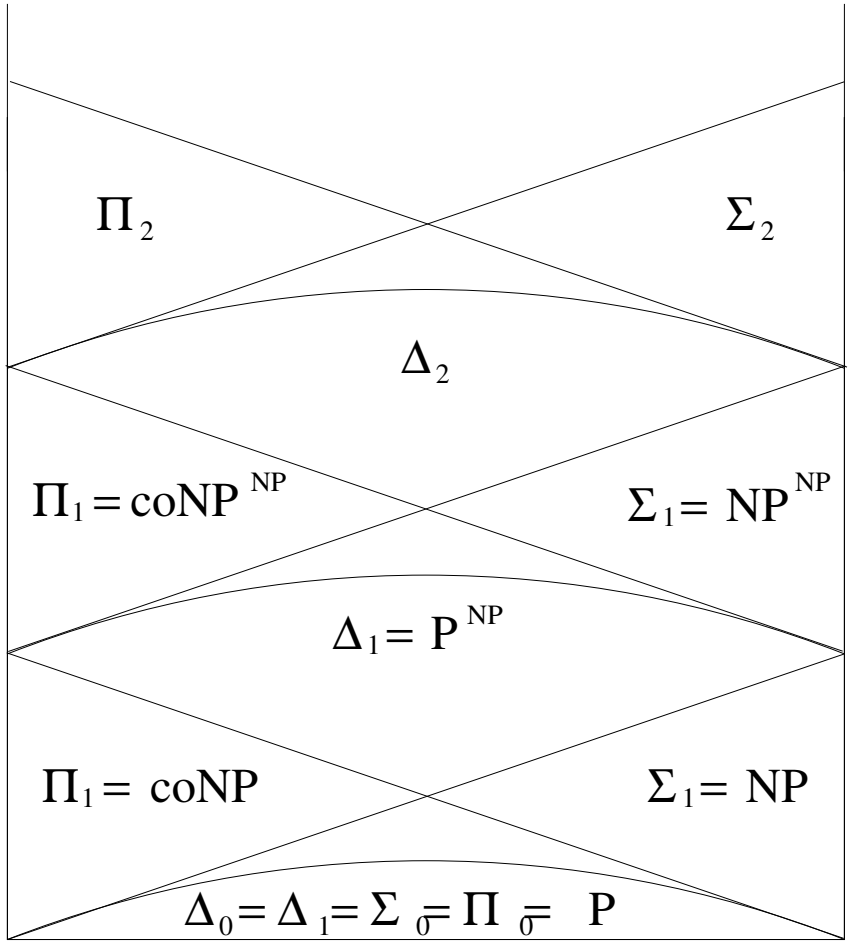
Per ogni $n \in N$ valgono le seguenti inclusioni:

$$\Sigma_n^P \cup \Pi_n^P \subseteq \Delta_{n+1}^P \subseteq \Sigma_{n+1}^P \cap \Pi_{n+1}^P \subseteq PSPACE$$

La dimostrazione è una conseguenza delle seguenti osservazioni:

- $A \in P^A \Rightarrow \Sigma_n^P \subseteq P^{\Sigma_n^P} = \Delta_{n+1}^P$
- $A \in P^A = P^{\overline{A}} \Rightarrow \Pi_n^P \subseteq P^{\Pi_n^P} = P^{\Sigma_n^P} = \Delta_{n+1}^P$
- $P^A \subseteq NP^A \Rightarrow \Delta_{n+1}^P = P^{\Sigma_n^P} \subseteq NP^{\Sigma_n^P} = \Sigma_{n+1}^P$
- $\Delta_{n+1}^P = P^{\Sigma_n^P} = coP^{\Sigma_n^P} \subseteq coNP^{\Sigma_n^P} = \Pi_{n+1}^P$
- $\Sigma_0^P = P \subseteq PSPACE$
- $NP^{PSPACE} = PSPACE \Rightarrow \Sigma_{n+1}^P = NP^{\Sigma_n^P} \subseteq PSPACE$

La gerachia polinomiale



Alcune proprietà di chiusura

Per ogni $n \geq 0$:

- se $A, B \in \Sigma_n^P$, allora $A \cup B, A \cap B \in \Sigma_n^P$, e $\overline{A} \in \Pi_n^P$
- se $A, B \in \Pi_n^P$, allora $A \cup B, A \cap B \in \Pi_n^P$, e $\overline{A} \in \Sigma_n^P$
- se $A, B \in \Delta_n^P$, allora $A \cup B, A \cap B$ e $\overline{A} \in \Delta_n^P$

Quantificatori limitati in spazio

Dato un alfabeto Σ, una proprietà $Q \subseteq \Sigma^*$, un polinomio p e un naturale n adotteremo la seguente notazione:

- $\exists^{p(n)} x, x \in Q(x) \Leftrightarrow, \exists x, |x| \leq p(n) \wedge x \in Q(x)$
- $\forall^{p(n)} x, x \in Q(x) \Leftrightarrow, \forall x, |x| \leq p(n) \rightarrow x \in Q(x)$

Lemma

Per ogni $n \in N$, $A \in \Sigma^P_{n+1}$, se e solo se esiste $B \in \Pi^P_n$ e un qualche polinomio p tale che per ogni stringa x

$$x \in A \Leftrightarrow \exists^{p(|x|)} y, \langle x, y \rangle \in B$$

Dimostriamo il risultato per induzione su n. Se $n = 0$ l'asserto coincide con il teorema della proiezione in tempo polinomiale.
Vediamo il caso induttivo.

Teorema della proiezione generalizzato

- ($\Leftarrow$) Siamo dati $B \in \Pi_n^P$ e p che realizzano l'equazione precedente. Vogliamo dimostrare che allora $A \in \Sigma_{n+1}^P$. Osserviamo che $NP^B = NP^{\overline{B}} \subseteq NP^{\Sigma_n^P} = \Sigma_{n+1}^P$, dunque basta dimostrare che $A \in NP^B$. A questo scopo basta considerare una MdTN che sceglie nodeterministicamente una stringa di lunghezza $|y| \leq p(|x|)$ e termina con accettazione se l'oracolo B risponde positivamente all'interrogazione $\langle x, y \rangle$.
- ($\Rightarrow$) Supponiamo di avere una MdTN M che riconosce A con oracolo $D \in \Sigma_n^P$ in un tempo polinomiale q. Dobbiamo definire un linguaggio $B \in \Pi_n^P$ e un polinomio p tale che

$$x \in A \Leftrightarrow \exists^{p(|x|)} y, \langle x, y \rangle \in B$$

L'idea è di vedere y come una computazione di M relativa ad un input di x di lunghezza $|x| = n$. Questa computazione è descritta da una sequenza $y = \alpha_0, ..., \alpha_m$ di configurazioni con $m \leq q(n)$ e $|\alpha_i| \leq q(n)$ (dunque $|y| \leq q(n)^2$). Dobbiamo solo dimostrare che possiamo *verificare* che y sia una computazione corretta per x mediante un algoritmo in Π_n^P.

Analizziamo i passi che dobbiamo compiere.

Teorema della proiezione generalizzato (2)

1. verificare che α_0 sia una configurazione iniziale e α_m una configurazione di accettazione richiede un tempo polinomiale
2. verificare che, se α_i non è in uno stato di query $\alpha_i \vdash \alpha_{i+1}$ ha complessità polinomiale
3. se α_i è in uno stato di interrogazione, allora la configurazione successiva dipende dalla interrogazione $v_i \in D$ posta all'oracolo. Distinguiamo due casi a seconda che lo stato di risposta nella configurazione successiva sia $q+$ o $q-$. Nel secondo caso, dobbiamo verificare che $v_i \in coD$ che é un problema in Π_n^P in quanto $D \in \Sigma_n^P$.

   Nel primo caso, per ipotesi induttiva, esiste $E \in \Pi_{n-1}^P \subseteq \Pi_n^P$ e un polinomio r tale che

   $$v_i \in D \Leftrightarrow \exists^{r(|v_i|)} w_i, \langle v_i, w_i \rangle \in E$$

   Aggiungiamo l'insieme di questi testimoni $w = \langle w_0, \ldots w_i \rangle$ alla descrizione della computazione.

Definiamo B come l'insieme stringhe $\langle x, \langle y, w \rangle \rangle$ che soddisfano le condizioni precedenti. Allora, $B \in \Pi_n^P$ e per un polinomio p opportuno

$$x \in A \Leftrightarrow \exists^{p(|x|)} y, \langle x, y \rangle \in B$$

Versione polinomiale del Teorema di Kuratowski-Tarski

Caraterizzazione di Kuratowski-Tarski per la gerarchia polinomiale

Sia $n \in N$.

- Σ^P_{n+1} è la classe dei linguaggi A per cui esiste un linguaggio $B \in P$ tale che

$$A = \{x : \exists^{p(|x|)} y_1, \forall^{p(|x|)} y_2, \ldots, Q^{p(|x|)} y_n, \langle x, y_1, y_2, ..., y_n \rangle \in B\}$$

  dove $Q = \forall$ se n è pari e $Q = \exists$ se n è dispari

- Π^P_{n+1} è la classe dei linguaggi A per cui esiste un linguaggio $B \in P$ tale che

$$A = \{x : \forall^{p(|x|)} y_1, \exists^{p(|x|)} y_2, \ldots, Q^{p(|x|)} y_n, \langle x, y_1, y_2, ..., y_n \rangle \in B\}$$

  dove $Q = \exists$ se n è pari e $Q = \forall$ se n è dispari

Per induzione su n utilizzando il lemma generalizzato della proiezione.

Versione relativizzata della fermata limitata

Dat un oracolo A, il problema relativizzato ad A della fermata limitata (Bounded Halting Problem) è:

$$BHP^A := \{\langle x, y, 0^t\rangle \quad | \quad \text{MdTN } M_x \text{ accetta } y \text{ con oracolo } A \text{ in tempo } time^A_{M_x}(y) \leq t\}$$

Lemma

Per ogni oracolo A, BHP^A è NP^A-completo.

La dimostrazione è del tutto analoga a quella della NP-completezza del problema limitato della fermata.

Problemi completi nella gerarchia polinomiale

Lemma

Per ogni $k > 0$, se A è Σ_k^P-completo allora $NP^A = \Sigma_{k+1}^P$.

Se $A \in \Sigma_k^P$, allora $NP^A \subseteq NP^{\Sigma_k^P} = \Sigma_{k+1}^P$.
Viceversa, sia $B \in \Sigma_{k+1}^P$; allora esiste $C \in \Sigma_k^P$ tale che $B \in NP^C$.
Siccome A è Σ_k^P-completo, $C \in P^A$ e per transitività $B \in NP^A$.

Salti nella gerarchia polinomiale

Poniamo

$$BHP_1 := BHP^{\emptyset} \quad BHP_{k+1} := BHP^{BHP_k}$$

Lemma

Per ogni $k > 0$, BHP_k è Σ_k^P-completo.

Per induzione su k.
Per $k = 1$, $BHP_1 = BHP^{\emptyset} = BHP$ che sappiamo essere NP-completo.
Nel caso induttivo, supponiamo che BHP_k sia Σ_k^P-completo.
Per la proposizione precedente, questo implica che $\Sigma_{k+1}^P = NP^{BHP_k}$.
Sappiamo inoltre che per ogni oracolo A, BHP^A è NP^A completo, quindi in particolare $BHP_{k+1} = BHP^{BHP_k}$ è $NP^{BHP_k} = \Sigma_{k+1}^P$-completo.

Il teorema di Cook relativizzato

Dato $k > 0$, denotiamo con SAT_k il problema seguente:

- dati k insiemi di variabili $X_1, \ldots X_k$ e una formula proposizionale F con variabili libere in $X = \bigcup_{1 \leq i \leq k} X_i$, determinare se vale la condizione seguente:

$$\exists v_1 : X_1 \to \{0,1\}, \forall v_2 : X_2 \to \{0,1\},$$
$$\ldots Q v_k : X_k \to \{0,1\}, \ [\![F]\!]_{v_1, v_2, \ldots, v_k} = 1$$

  dove $Q = \forall$ se k è pari e $Q = \exists$ se k è dispari.

Teorema di Cook relativizzato

Per ogni $k > 0$ il problema relativizzato della soddisfaciblitá SAT_k è Σ_k^P-completo.

La prova è una versione relativizzata della dimostrazione del Teorema di Cook.

Il problema della fermata con spazio limitato

Space Bounded Halting Problem

Il problema della fermata con spazio limitato

$$SBHP := \{\langle x, y, 0^s\rangle | \text{ MdT } M_x \text{ accetta } y \text{ in spazio } space_{M_x}(y) \leq s\}$$

è PSPACE-completo.

La MdT universale M_u può simulare M_x su input y con spazio limitato a s richiedendo uno spazio polinomiale in s.
D'altra parte, se $A \in PSPACE$ esiste una MdT M_x che accetta A utilizzando uno spazio $s_{M_x} \leq p$ per un qualche polinomio p. La funzione di riduzione che dimostra che $A \leq_P SBHP$ è semplicemente la funzione f definita da

$$f(y) := \langle x, y, 0^{p(|y|)}\rangle$$

$f \in FP$ poichè i polinomi sono costruibili in tempo.

Altri problemi PSPACE-completi

- Formule Proposizionali Quantificate: data una formula proposizionale le cui variabili sono quantificate, determinarne la verità
- Lo stesso problema ma con formule in 3CNF
- Totalità delle espressioni regolari: data un espressione regolare su di un dato alfabeto, determinare se genera tutte le parole.

Collasso della gerarchia polinomiale

PH = PSPACE implica il collasso della gerarchia polinomiale

Se $PH = PSPACE$, allora esiste un $k \in N$ tale che $PH = \Sigma_k^P$.

$SBHP \in PSPACE$ e se $PH = PSPACE$ allora esiste un qualche $k \in n$ tale che $SBHP \in \Sigma_k^P$. Questo implica che $PSPACE \subseteq \Sigma_k^P$ poichè Σ_k^P è chiuso rispetto all riducibilità polinomiale.

Lezione 20:P-NP relativizzato

- Il problema P-NP relativizzato
- Il Teorema di indipendenza di Hartmanis e Hopcroft

Il problema P-NP relativizzato

Teorema di Baker, Gill e Solovay

Esistono due linguaggi ricorsivi A e B tali che $P^A = NP^A$ e $P^B \neq NP^B$.

Per A possiamo scegliere un qualunque insieme $PSPACE$-completo (ad esempio $SBHP$) in quanto

$$NP^A \subseteq NP^{PSPACE} = PSPACE \subseteq P^A$$

Veniamo all'altro caso.
Dato un linguaggio B definiamo $L_B := \{0^{|x|} | x \in B\}$. Allora $L_B \in NP^B$ poichè data un input 0^n ci basta scegliere non deterministicamente una x di lunghezza $|x| = n$ e verificare che $x \in B$. Il nostro obiettivo è costruire un linguaggio $B \subseteq \{0,1\}^*$ tale che $L_B \notin P^B$ poichè in tal caso $L_B \in NP^B \neq P^B$.
Costruiremo B con una tecnica di diagonalizzazione. Sia data una enumerazione (M_i, p_i) delle MdT polinomiali, e dei relativi bound. Vogliamo costruire B in modo tale che per ogni i esiste una stringa x_i tale che

$$x_i \in L_B \Leftrightarrow x_i \notin L^B_{M_i}$$

Il teorema di Baker, Gill e Solovay

Costruiamo l'insieme B per approssimazioni successive B_i con $i = 0, 1, 2, 3, \ldots$. Ad ogni passo definiamo un numero $n_i > n_{i-1}$ e un insieme $B_i \subseteq \{x \in \{0,1\}^* \mid |x| \leq n_i\}$. L'insieme finale sarà l'unione $B = \bigcup_{i \in N} B_i$ di queste approssimazioni.

- passo 0. Definiamo $n_0 = 0$ e $B_0 = \emptyset$
- passo $i > 0$. Scegliamo un $n \in N$ tale che
  1. $2^n > p_i(n)$ e
  2. $n > 2^{n_{i-1}}$,

  Poniamo $n_i := n$ e $x_i := 0^n$. Simuliamo la macchina M_i su input x_i con oracolo B_{i-1} per $p_i(n_i)$ passi.

  Se M_i accetta x_i, allora poniamo $B_i := B_{i-1}$ (quindi B_i non conterrà stringhe di lunghezza n_i).

  Se M_i non riconosce x_i, cerchiamo una stringa $y \in \{0,1\}^*$ di lunghezza $|y| = n_i$ che non sia oggetto di interrogazioni all'oracolo durante la computazione. Tale stringa deve esistere per 1.: infatti esistono 2^{n_i} stringhe di lunghezza n_i ma la macchina M_i può interrogare l'oracolo al più $p_i(n_i)$ volte. Poniamo allora $B_i := B_{i-1} \cup \{y\}$.

Il teorema di Baker, Gill e Solovay (2)

Per costruzione,

$$x_i \in L_B \Leftrightarrow x_i \in L_{B_i} \Leftrightarrow x_i \notin L_{M_i}^{B_{i-1}}$$

Infatti, $x_i = 0^{n_i} \in L_{B_i}$ se e solo se esiste $y \in B_i$ di lunghezza $|y| = n_i$ e tale y viene aggiunto a B_i se e solo se $x_i \notin L_{M_i}^{B_{i-1}}$.
Il nostro claim è che, per ogni $i \in N$,

$$x_i \in L_{M_i}^{B_{i-1}} \Leftrightarrow x_i \in L_{M_i}^{B}$$

(supposto che la macchina M_i operi in tempo p_i).
Il punto è che le successive estensioni a B_i sono ininfluenti per M_i in quanto riguardano stringhe troppo lunghe per essere oggetto di query. Più precisamente, le condizioni 1. e 2. implicano che $n_{i+1} > 2^{n_i} > p_i(n_i)$ per ogni i e dunque nei passi succesivi a i si possono aggiungere a B solo stringhe di lunghezza $> p_i(n_i)$, mentre M_i può al più fare interrogazioni su stringhe di questa lunghezza. D'altra parte allo stesso passo i la definizione di B assicura di non aggiungere una stringa utile al funzionamento di M_i in tempo p_i su input x_i.

Rilevanza del teorema di Baker, Gill e Solovay

Molte tecniche di separazione per Classi di Complessità possono essere relativizzate al caso con oracoli.

Il Teorema precedente mostra che nessuna di queste tecniche può essere applicata al problema $P = NP$.

Estensioni al teorema di Baker, Gill e Solovay

Generalizzando la tecnica precedente, Baker, Gill e Solovay hanno anche dimostrato che esistono linguaggi A, B, C per cui

- $P^A \neq (NP^A \cap coNP^A) \neq NP^A$
- $P^B \neq NP^B = coNP^B$
- $P^C = (NP^C \cap coNP^C) \neq NP^C$

Il Teorema di Indipendenza di Hartmanis e Hopcroft

Hartmanis e Hopcroft

Data una teoria formale e consistente della complessità che comprenda risultati di base sulle Macchine di Turing, possiamo trovare in modo effettivo una MdT M_i tale che $L_{M_i} = \emptyset$ ma nè $P^{L_{M_i}} = NP^{L_{M_i}}$, nè $P^{L_{M_i}} \neq NP^{L_{M_i}}$ è dimostrabile.

Siano A e B due insiemi ricorsivi tali che $P^A = NP^A$ e $P^B \neq NP^B$. Definiamo una MdT M che su input (x, j) opera nel modo seguente: enumera sistematicamente le prime $|x|$ dimostrazioni della teoria formale e accetta (x, j) se tra queste dimostrazioni ne esiste una di $P^{L_{M_j}} = NP^{L_{M_j}}$ e $x \in B$ oppure ne esista una $P^{L_{M_j}} \neq NP^{L_{M_j}}$ e $x \in A$

Il Teorema di Indipendenza di Hartmanis e Hopcroft (2)

Per il teorema s-m-n e il teorema del punto fisso, possiamo trovare un indice $i \in N$ tale che M_i accetta x se e solo se M accetta (x, i). Se esiste una prova che $P^{L_{M_i}} = NP^{L_{M_i}}$ allora per ogni $x > k$ per un k fissato M accetta (x, i) se e solo se $x \in B$. Dunque la differenza tra L_{M_i} e B è finita e $P^{L_{M_i}} = P^B \neq NP^B = NP^{L_{M_i}}$. Siccome la teoria è consistente, $P^{L_{M_i}} = NP^{L_{M_i}}$ non è dimostrabile.
In modo simile, se esiste una prova che $P^{L_{M_i}} \neq NP^{L_{M_i}}$ allora per ogni $x > k$ per un k fissato M accetta (x, i) se e solo se $x \in A$. Dunque la differenza tra L_{M_i} e A è finita e $P^{L_{M_i}} = P^A = NP^A = NP^{L_{M_i}}$. Siccome la teoria è consistente, $P^{L_{M_i}} \neq NP^{L_{M_i}}$ non è dimostrabile.
Ma se nessuno dei due asserti è dimostrabile, allora la macchina M_i non accetta nessun input x, ovvero $L_{M_i} = \emptyset$.

Osservazione Non è detto che la teoria permetta di dimostrare che $P^{L_{M_i}} = P$. Dunque il teorema di Indipendenza **non dimostra** l'indipendenza di $P = NP$.