Complessità computazionale

I problemi più difficili di ogni classe: Riduzioni e completezza

Piero A. Bonatti

Università di Napoli Federico II

Laurea Magistrale in Informatica

Tema della lezione I

- Abbiamo visto che molte classi di complessità ne contengono altre che sappiamo contenere problemi "più semplici" da risolvere

$$\mathbf{TIME}(n) \subset \mathbf{P}, \quad \mathbf{TIME}(n^2) \subset \mathbf{P}, ..., \quad \mathbf{P} \subset \mathbf{EXP}$$

- oppure *potrebbero* contenere problemi più semplici

$$\mathbf{P} \subseteq \mathbf{NP}, \quad \mathbf{NP} = \mathbf{NTIME}(n^k) \subseteq \mathbf{TIME}(2^{n^k}) = \mathbf{EXP}$$

- Ad esempio REACHABILITY appartiene sia a **P** che a **NP**
- Invece TSP appartiene a **NP** ma non sappiamo se appartiene a **P**
- Quindi, in qualche misura, TSP "rappresenta" la classe **NP** meglio di REACHABILITY (che non ha bisogno del nondeterminismo per essere risolto efficientemente)

Tema della lezione II

- Ma allora quali sono i problemi "più difficili" di una classe di complessità?
- Quelli "caratterizzanti", che non posso risolvere con meno risorse di quelle che definiscono la classe
  - e TSP fa davvero parte dei problemi più difficili di **NP**?
  - e quali sono gli altri ?
- E in che relazione sono tra loro? Se trovo un algoritmo per uno di essi, posso usarlo per risolvere gli altri?

Tema della lezione III

- In questa lezione introduciamo un criterio di "difficoltà relativa" che dice quando un problema è più difficile da risolvere di un altro
  - mediante le cosiddette riduzioni
- Con questo criterio potremo definire quali sono i problemi "più difficili" di una classe di complessità $\mathcal{C}$
  - detti problemi $\mathcal{C}$-completi

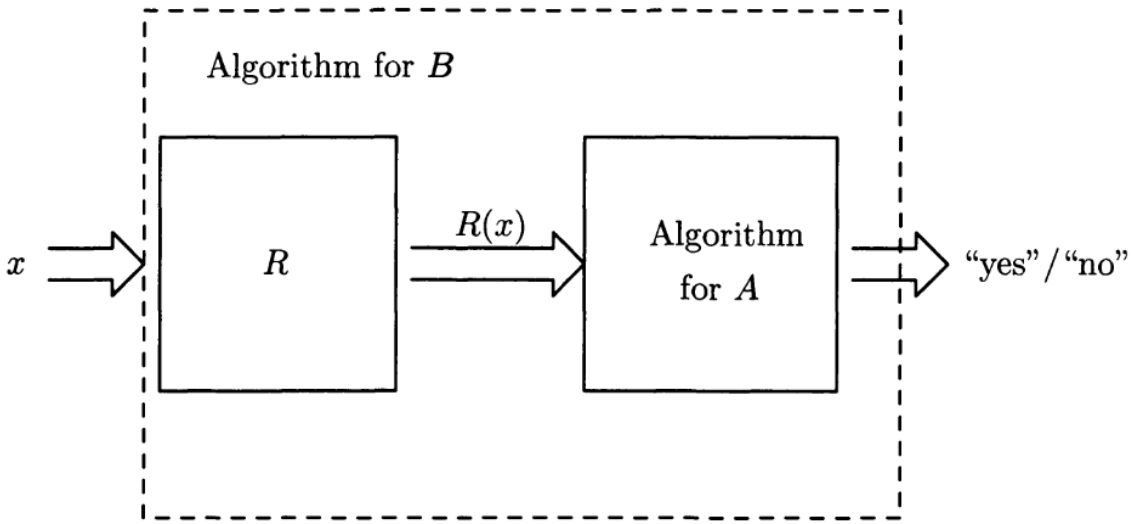
Tema della lezione IV

- Mediante questi strumenti di analisi potremo capire meglio le caratteristiche dei problemi difficili, che oggi sappiamo risolvere efficientemente solo usando il nondeterminismo
  - ovvero studieremo meglio la classe **NP**
  - mostrando una quantità di problemi interessanti in pratica che purtroppo appartengono alla fascia "più complessa" dei problemi in **NP**
- Approfondiremo così la nostra conoscenza di cosa rende un problema *intrinsecamente difficile*
  - e di come riconoscere tali problemi
- Nondimeno, riduzioni e completezza sono strumenti generali, utili per analizzare *qualunque* classe di complessità (non solo **NP**)

Misurare la difficoltà relativa: Riduzioni

Riduzioni

- Abbiamo già incontrato le riduzioni nella prima lezione
- Una riduzione da un problema B a un problema A è una funzione R che traduce ogni istanza x di B in una istanza $R(x)$ di A preservando le risposte:
  - la risposta a x è "yes" se e solo se la risposta a $R(x)$ è "yes"
  - equivalentemente, $x \in B \Leftrightarrow R(x) \in A$
- Così si può risolvere B componendo R e un algoritmo per A:



Riduzioni e difficoltà relativa – I

- Se possiamo risolvere B usando un algoritmo per A è ragionevole convenire che
  - B non è più difficile di A
  - A è almeno tanto difficile quanto B

  ad una condizione: *niente imbrogli*
- **Requisito v.1**: La riduzione non deve fare tutto il lavoro
  - caso limite: R risolve B e A deve solo controllare se $R(x) = yes$
  - un problema banale risulterebbe almeno difficile quanto qualunque altro...

Riduzioni e difficoltà relativa – II

- **Requisito v.2**: La riduzione non dovrebbe cambiare la classe di complessità di A
  - ad es. il tempo totale richiesto è $f_R(n) + f_A(n)$, dove f_R/ f_A sono i tempi richiesti per calcolare R e risolvere A
  - molte classi sono chiuse rispetto all'aggiunta di polinomi, ad es.
  - se R gira in tempo polinomiale, allora
  - se l'algoritmo per A è polinomiale anche la composizione di R e A è polinomiale (la somma di polinomi è un polinomio)
  - se l'algoritmo per A è esponenziale anche la composizione di R e A è esponenziale (f_A domina f_R)

Riduzioni e difficoltà relativa – III

- Quanto appena detto ci va bene per il tempo, in quanto i problemi che richiedono tempo men che lineare non sono molto interessanti
  - non serve nemmeno guardare tutto l'input!

  Però esistono algoritmi interessanti che usano *spazio logaritmico* (ad es. REACHABILITY)...
- Se lo spazio per A fosse logaritmico e quello per R polinomiale, allora R aumenterebbe drasticamente il consumo di memoria, uscendo dalla classe di complessità di A
- quindi ci limitiamo a spazio $O(\log n)$
- Il tempo richiesto resta polinomiale (per le relazioni spazio-tempo)

$$\mathbf{SPACE}(\log n) \subseteq \mathbf{TIME}(k^{\log n}) \subseteq \mathbf{TIME}(n^k)$$

- quindi anche $|R(x)|$ è polinomiale in $|x|$

Riduzioni

Definizione formale

Definizione (riducibilità, riduzione)

Un linguaggio L_1 è riducibile a un linguaggio L_2 se esiste una funzione R su stringhe

- calcolabile da una MdT deterministica in spazio $O(\log n)$
- tale che per ogni stringa x, $x \in L_1$ sse $R(x) \in L_2$

R viene detta riduzione da L_1 a L_2

- Quando un problema L_1 è riducibile a L_2 consideriamo L_2 almeno difficile quanto L_1
  - le risorse necessarie per risolvere L_2 sostanzialmente bastano per risolvere anche L_1
  - a meno del costo di R che è "trascurabile" da **L** in su

Esempio di Riduzione

Richiami di Logica Proposizionale

Espressioni booleane (sintassi)

- Il linguaggio si basa su di un insieme infinito numerabile di *variabili booleane* $X = \{x_1, x_2, \ldots\}$
- e sui connettivi logici $\wedge, \vee, \neg$ (*and*, *or*, *not*)

Richiami di Logica Proposizionale

Truth assignments (semantica)

- Un assegnamento di verità (*truth assignment* o *interpretazione* o *modello del linguaggio*) è una funzione

$$T : X' \rightarrow \{true, false\}$$

dove X' è un sottinsieme finito di X.

- Data una espressione booleana ϕ le cui variabili booleane sono contenute in X', diciamo che T soddisfa ϕ (in simboli $T \models \phi$) sse (ricorsivamente)
  - $\phi = x_i$ e $T(x_i) = true$
  - $\phi = \phi_1 \wedge \phi_2$ e $T \models \phi_1$ e $T \models \phi_2$
  - $\phi = \phi_1 \vee \phi_2$ e $T \models \phi_1$ oppure $T \models \phi_2$
  - $\phi = \neg\phi_1$ e $T \not\models \phi_1$ (T non soddisfa ϕ_1)

Richiami di Logica Proposizionale

Truth assignments (semantica)

- Un assegnamento di verità (*truth assignment* o *interpretazione* o *modello del linguaggio*) è una funzione

$$T : X' \rightarrow \{true, false\}$$

dove X' è un sottinsieme finito di X.

- Data una espressione booleana ϕ le cui variabili booleane sono contenute in X', diciamo che T soddisfa ϕ (in simboli $T \models \phi$) sse (ricorsivamente)
  - $\phi = x_i$ e $T(x_i) = true$
  - $\phi = \phi_1 \wedge \phi_2$ e $T \models \phi_1$ e $T \models \phi_2$
  - $\phi = \phi_1 \vee \phi_2$ e $T \models \phi_1$ oppure $T \models \phi_2$
  - $\phi = \neg\phi_1$ e $T \not\models \phi_1$ (T non soddisfa ϕ_1)

Il problema SAT

Definizione

- Le istanze di SAT sono espressioni booleane ϕ in CNF
  - rappresentate con stringhe, ad es. "$(p1 \vee p2) \wedge (\neg p1 \vee p3)$"
- La risposta ad una istanza ϕ è "yes" sse esiste un assegnamento di verità T che soddisfa ϕ
- In forma di linguaggio: $\mathrm{SAT} = \{\phi \mid \exists T : T \models \phi\}$

- Uno dei problemi di ragionamento elementari per l'AI
- applicabile a risoluzione automatica di vincoli (ad es. configurazione di hardware)
- ha una importanza centrale nello studio di **NP**

Il problema SAT

Definizione

- Le istanze di SAT sono espressioni booleane ϕ in CNF
  - rappresentate con stringhe, ad es. "$(p1 \vee p2) \wedge (\neg p1 \vee p3)$"
- La risposta ad una istanza ϕ è "yes" sse esiste un assegnamento di verità T che soddisfa ϕ
- In forma di linguaggio: $\mathrm{SAT} = \{\phi \mid \exists T : T \models \phi\}$

- Uno dei problemi di ragionamento elementari per l'AI
- applicabile a risoluzione automatica di vincoli (ad es. configurazione di hardware)
- ha una importanza centrale nello studio di **NP**

Il problema Hamilton Path

Definizione

- le istanze sono grafi
- la risposta è "yes" sse esiste un *ciclo Hamiltoniano* (che tocca ogni nodo una e una sola volta)
- Formalmente: una permutazione π dei nodi $1, 2, \ldots, n$ tale che per ogni $i = 1, \ldots, n-1$, $(\pi(i), \pi(i+1))$ è un arco di G

- Noto problema difficile
- non si conoscono soluzioni polinomiali

Il problema Hamilton Path

Definizione

- le istanze sono grafi
- la risposta è "yes" sse esiste un *ciclo Hamiltoniano* (che tocca ogni nodo una e una sola volta)
- Formalmente: una permutazione π dei nodi $1, 2, \ldots, n$ tale che per ogni $i = 1, \ldots, n-1$, $(\pi(i), \pi(i+1))$ è un arco di G

- Noto problema difficile
- non si conoscono soluzioni polinomiali

Riduzione da Hamilton Path a SAT

- Nel seguito, per dare un'esempio di riduzione, mostriamo come ridurre Hamilton Path a SAT
- Mostrando così che SAT è almeno altrettanto difficile da risolvere di Hamilton Path
  - la riduzione permette di trasformare un qualsiasi algoritmo per SAT in uno per Hamilton Path
  - non è garantito che questo algoritmo sia ottimo per Hamilton Path
  - quindi ci dice solo che le risorse utilizzate per SAT sono *almeno* pari a quelle minime necessarie per risolvere Hamilton Path
  - In questo senso SAT *è almeno tanto difficile da risolvere di* Hamilton Path

Riduzione da Hamilton Path a SAT

- Nel seguito, per dare un'esempio di riduzione, mostriamo come ridurre Hamilton Path a SAT
- Mostrando così che SAT è almeno altrettanto difficile da risolvere di Hamilton Path
  - la riduzione permette di trasformare un qualsiasi algoritmo per SAT in uno per Hamilton Path
  - non è garantito che questo algoritmo sia ottimo per Hamilton Path
  - quindi ci dice solo che le risorse utilizzate per SAT sono *almeno* pari a quelle minime necessarie per risolvere Hamilton Path
  - In questo senso SAT *è almeno tanto difficile da risolvere di* Hamilton Path

Riduzione da Hamilton Path a SAT

- Nel seguito, per dare un'esempio di riduzione, mostriamo come ridurre Hamilton Path a SAT
- Mostrando così che SAT è almeno altrettanto difficile da risolvere di Hamilton Path
  - la riduzione permette di trasformare un qualsiasi algoritmo per SAT in uno per Hamilton Path
  - non è garantito che questo algoritmo sia ottimo per Hamilton Path
  - quindi ci dice solo che le risorse utilizzate per SAT sono *almeno* pari a quelle minime necessarie per risolvere Hamilton Path
  - In questo senso SAT *è almeno tanto difficile da risolvere di* Hamilton Path

Riduzione da Hamilton Path a SAT

Come funziona

- Sia dato un grafo G con nodi $1, 2, \ldots, n$
- La riduzione trasforma G in una espressione booleana $R(G)$
  - soddisfacibile sse G ha un ciclo Hamiltoniano
- $R(G)$ usa n^2 var. booleane x_{ij}, con $i, j \in \{1, \ldots, n\}$
  - faremo in modo che x_{ij} valga *true* sse il nodo j è l'i-esimo del ciclo Hamiltoniano trovato
- $R(G)$ è in *conjunctive normal form* (CNF)
  - noi mostreremo le disgiunzioni di letterali (dette *clausole*) che, messe in *and* costituiscono $R(G)$
  - le clausole codificano le proprietà che le x_{ij} devono avere per rappresentare effettivamente un ciclo Hamiltoniano

Riduzione da Hamilton Path a SAT– I

Le clausole

- ogni nodo j deve apparire nel ciclo, quindi per ogni j aggiungiamo una clausola

$$x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj}$$

- ogni nodo j compare non più di una volta: per ogni j e per ogni $i, k \in \{1, \ldots, n\}$ tali che $i \neq k$,

$$\neg x_{ij} \vee \neg x_{kj}$$

(forse più intuitivo come $\neg(x_{ij} \wedge x_{kj})$)

- ogni posizione i nel cammino va coperta con un nodo: per ogni $i = 1, \ldots, n$,

$$x_{i1} \vee x_{i2} \vee \cdots \vee x_{in}$$

Riduzione da Hamilton Path a SAT– II

Le clausole

- non ci sono 2 nodi nella stessa posizione: per ogni i e per ogni $j, k \in \{1, \ldots, n\}$ tali che $j \neq k$,

$$\neg x_{ij} \vee \neg x_{ik}$$

- se (i, j) *non* è un arco di G allora j non può seguire i nel cammino: per ogni $k = 1, \ldots, n-1$ e per ogni $i, j \in \{1, \ldots, n\}$,

$$\neg x_{ki} \vee \neg x_{k+1,j}$$

(forse più intuitivo come $\neg(x_{ki} \wedge x_{k+1,j})$)

- Infine $R(G)$ è la congiunzione (*and*) di tutte queste clausole

Riduzione da Hamilton Path a SAT

Correttezza della riduzione 1: Se $R(G)$ è soddisfacibile allora G ha un ciclo Hamiltoniano

- Supponiamo che $R(G)$ sia soddisfatta da un truth assignment T

1. Per ogni j esiste esattamente un i tale che $T(x_{ij}) = \mathit{true}$
   - altrimenti $(x_{1j} \vee x_{2j} \vee \cdots \vee x_{nj})$ e/o $(\neg x_{ij} \vee \neg x_{kj})$ non sarebbero soddisfatte
2. Similmente, per ogni posizione i esiste esattamente un nodo j tale che $T(x_{ij}) = \mathit{true}$
   - usando le due clausole relative alle posizioni
3. Quindi ponendo $\pi(i) = j$ sse $T(x_{ij}) = \mathit{true}$ otteniamo una permutazione dei nodi
4. Infine T soddisfa le clausole $(\neg x_{ki} \vee \neg x_{k+1,j})$, dove (i, j) *non* è un arco di G, quindi
   - ogni coppia $(\pi(k), \pi(k+1))$ è un arco di G
   - $\Rightarrow \pi(1), \ldots, \pi(n)$ è un ciclo Hamiltoniano

Riduzione da Hamilton Path a SAT

Correttezza della riduzione 2: Se G ha un ciclo Hamiltoniano allora $R(G)$ è soddisfacibile

- Supponiamo che G abbia un ciclo Hamiltoniano $\pi(1), \ldots, \pi(n)$

1. Definiamo $T(x_{ij}) = \mathit{true}$ sse $\pi(i) = j$
2. Si può dimostrare che T soddisfa $R(G)$
   - fare a casa come esercizio, enucleando tutti i dettagli
   - poi far correggere a ricevimento
   - sarà oggetto di domanda di esame

- A questo punto abbiamo dimostrato che G ha un ciclo Hamiltoniano sse $R(G)$ è soddisfacibile (correttezza della riduzione)

Riduzione da Hamilton Path a SAT– I

Complessità della riduzione: appartenenza a **SPACE**(log n)

- La trasformazione R è corretta ma non abbiamo ancora dimostrato che sia una riduzione
  - dobbiamo provare che può essere calcolata in spazio logaritmico
- A questo scopo definiamo una apposita MdT M
  - solo cenni, i dettagli sono pesanti

Riduzione da Hamilton Path a SAT– II

Complessità della riduzione: appartenenza a **SPACE**(log n)

- L'idea però è semplice: per generare le clausole ci servono dei for da 1 a n, annidati al massimo 3 volte, ad es.

  for i=1 to n
  for j=1 to n
  for k=1 to n
  if i $\neq$ k then append "$\wedge(\neg x_{ij} \vee \neg x_{kj})$" to the output

- tutti i contatori vanno da 1 a n quindi possono essere contenuti in spazio log n (rappresentandoli in binario)
  - non essendo necessaria altra memoria ausiliaria, il calcolo di R è in **SPACE**(log n)
  - (ricordate che input e output non vengono considerati nella misura di spazio)

Riduzione da Hamilton Path a SAT– III

Complessità della riduzione: appartenenza a **SPACE**(log n)

- la MdT M opera così:

1. scandisce l'input contando i nodi e salvando il loro numero (n) in binario su di un primo nastro ausiliario
2. genera le clausole che non dipendono dagli archi del grafo usando 3 nastri ausiliari
   - l'analogo dei contatori i, j, k nello pseudocodice precedente
3. per l'ultimo gruppo di clausole procede cosìı:
   - usa i 3 nastri ausiliari analoghi a i, j, k per generare *tutte* le possibili clausole $\neg x_{ki} \vee \neg x_{k+1,j}$; per ciascuna di esse:
   - salva la clausola su di un quinto nastro ausiliario
   - verifica se l'arco (i, j) appartiene al grafo (scandendo l'input)
   - se no, allora copia la clausola sul nastro di output

Riduzione da Hamilton Path a SAT– IV

Complessità della riduzione: appartenenza a **SPACE**($\log n$)

- Valutiamo lo spazio occupato:
  - ciascun contatore occupa spazio $\log n$
  - ciascuna clausola $\neg x_{ki} \vee \neg x_{k+1,j}$ occupa lo spazio necessario a 4 indici $(k,\ i,\ k+1,\ j)$ più una quantità costante di simboli
    - complessivamente $O(\log n)$
  - pertanto lo spazio totale richiesto è $O(\log n)$
- Dalla correttezza di R e dalla sua calcolabilità in spazio $\log n$ deriva che R è una riduzione da Hamilton Path a SAT

Composizione di riduzioni

Composizione di riduzioni: Introduzione – I

- Sembra ragionevole dire che la composizione $R \cdot R'$ [1] di due riduzioni R ed R' sia ancora una riduzione
  - ipotesi correlata all'uso delle riduzioni:
  - se un problema A è almeno tanto difficile quanto B, e B è almeno tanto difficile quanto C, allora A dovrebbe essere almeno tanto difficile quanto C
- Proprietà transitiva molto utile, perchè permette di derivare facilmente nuove riduzioni da quelle già note
  - e abbiamo visto che trovare nuove riduzioni "da zero" è faticoso... (si tratta di dimostrare un teorema)
  - la chiusura delle riduzioni risp. alla composizione garantisce la prop. transitiva
  - conferma ragionevolezza formalizzazione complessità relativa

[1] Ricordate che $(R \cdot R')(x) = R'(R(x))$

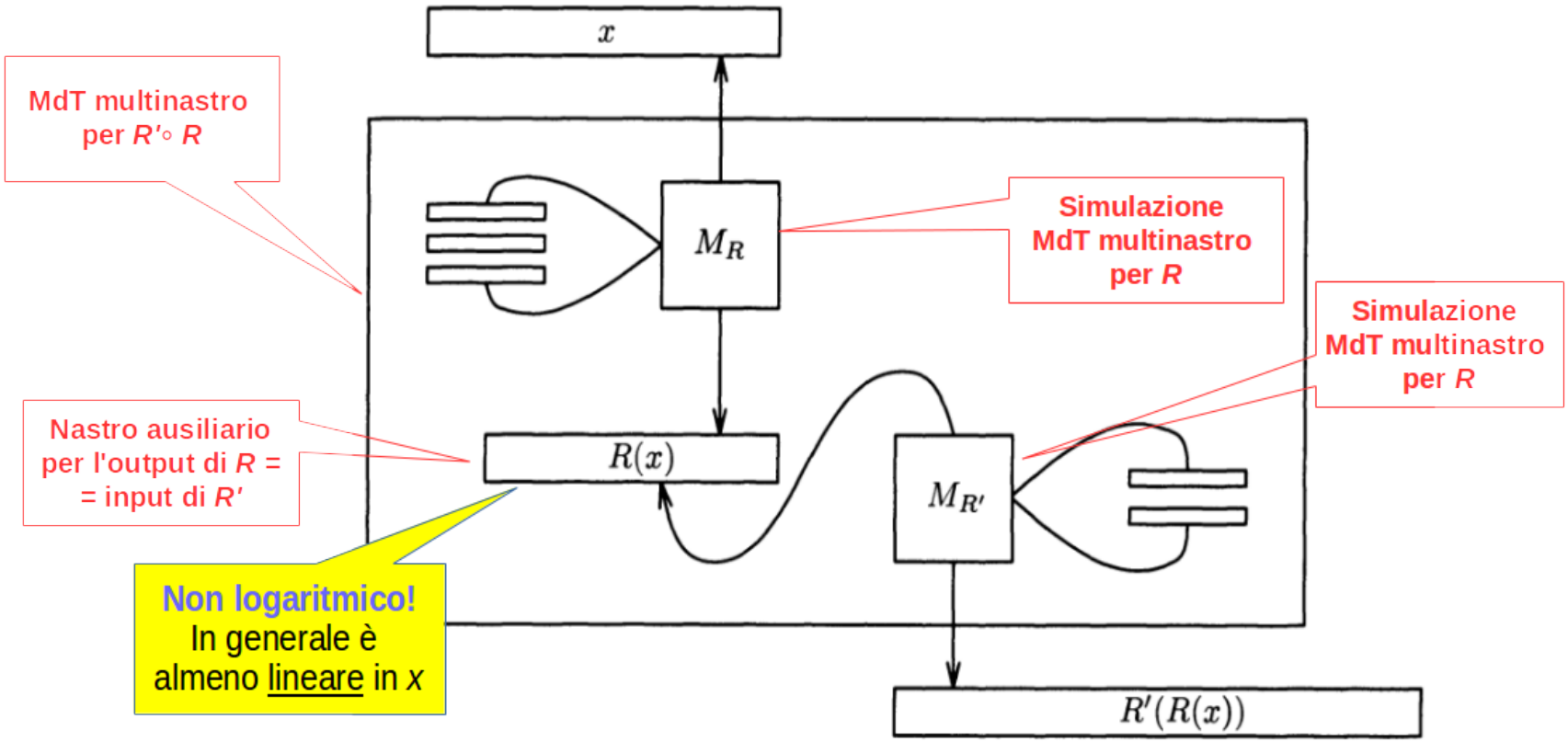
Composizione di riduzioni: Introduzione – I

- Sembra ragionevole dire che la composizione $R \cdot R'$ [1] di due riduzioni R ed R' sia ancora una riduzione
  - ipotesi correlata all'uso delle riduzioni:
  - se un problema A è almeno tanto difficile quanto B, e B è almeno tanto difficile quanto C, allora A dovrebbe essere almeno tanto difficile quanto C
- Proprietà transitiva molto utile, perchè permette di derivare facilmente nuove riduzioni da quelle già note
  - e abbiamo visto che trovare nuove riduzioni "da zero" è faticoso... (si tratta di dimostrare un teorema)
  - la chiusura delle riduzioni risp. alla composizione garantisce la prop. transitiva
  - conferma ragionevolezza formalizzazione complessità relativa

[1] Ricordate che $(R \cdot R')(x) = R'(R(x))$

Composizione di riduzioni: Introduzione – II

- Dimostrare che $R \cdot R'$ è una riduzione non è banale: provare che $R \cdot R' \in \mathbf{SPACE}(\log n)$ non è immediato...
- Ecco un esempio di come non fare:



Composizione di riduzioni: Il teorema

Teorema [Prop. 8.2 nel Papadimitriou]

Se R è una riduzione dal linguaggio L_1 al linguaggio L_2 e R' è una riduzione dal linguaggio L_2 al linguaggio L_3, allora $R \cdot R'$ è una riduzione da L_1 a L_3

Composizione di riduzioni: Il teorema – I

Dimostrazione

- Chiaramente, siccome R e R' sono riduzioni, abbiamo

$$x \in L_1 \Leftrightarrow R(x) \in L_2 \Leftrightarrow R'(R(x)) \in L_3$$

  quindi $R \cdot R'$ è *corretta*.
- Ci resta da dimostrare che si può calcolare in spazio logaritmico
- Idea: memorizzare un solo carattere di $R(x)$ alla volta, generandolo *on demand*

Composizione di riduzioni: II teorema – II

Dimostrazione

- Si comincia simulando $M_{R'}$
- $M_{R'}$ viene simulata mantenendo
  - un indice i (su apposito nastro) con la posizione della testina sull'input $R(x)$
  - il simbolo corrente σ (su apposito nastro)
- Inizialmente $i = 1$ e $\sigma = \triangleright$
- quando i viene incrementato, viene simulata M_R (a partire dall'ultimo passo eseguito) finchè non produce il prossimo simbolo sul nastro di output
  - che viene salvato invece in σ, sopra il vecchio valore
  - incrementando i
- poi si continua la simulazione di $M_{R'}$

Composizione di riduzioni: Il teorema – III

Dimostrazione

- Se invece la testina di $M_{R'}$ deve spostarsi a sinistra
  - i viene decrementato
  - M_R viene simulata di nuovo dall'inizio
  - finchè non produce l'i-esimo simbolo dell'output
  - che viene salvato nel nastro di σ sopra il vecchio valore
  - Domanda: perchè questo non è necessario finchè i aumenta?
- Poi continua la simulazione di $M_{R'}$

Composizione di riduzioni: Il teorema – IV

Dimostrazione

- Valutazione dello spazio:
  - i nastri ausiliari di M_R e $M_{R'}$ sono lunghi $O(\log |x|)$, perchè per ipotesi R e R' sono riduzioni
  - il nastro per i è lungo $O(\log |R(x)|) = O(\log |x|)$
    - Domanda: perchè?
  - il nastro per σ contiene sempre 1 simbolo ($O(1)$)

- Quindi in tutto lo spazio occupato da $R \cdot R'$ è $O(\log n)$
  $\Rightarrow$ $R \cdot R'$ è una riduzione

QED

Completezza
(rispetto a una classe di complessità $\mathcal{C}$)

Completezza = massima difficoltà

- Consideriamo i problemi in una classe di complessità $\mathcal{C}$
- La riducibilità (che gode della proprietà transitiva) induce un ordinamento parziale sui problemi in $\mathcal{C}$
  - che riflette la complessità relativa dei problemi
  - salendo nell'ordinamento si trovano problemi via via più difficili
- Noi siamo interessati ai problemi più difficili di $\mathcal{C}$
  - ad esempio i più difficili in **NP**
  - di cui non sappiamo se appartengono a $\mathbf{P} \subseteq \mathbf{NP}$
  - per i quali il nondeterminismo è davvero *l'unico modo* che conosciamo per risolvere il problema in tempo polinomiale
- Chiamiamo questi problemi *$\mathcal{C}$-completi*

Completezza = massima difficoltà

- Consideriamo i problemi in una classe di complessità $\mathcal{C}$
- La riducibilità (che gode della proprietà transitiva) induce un ordinamento parziale sui problemi in $\mathcal{C}$
  - che riflette la complessità relativa dei problemi
  - salendo nell'ordinamento si trovano problemi via via più difficili
- Noi siamo interessati ai problemi più difficili di $\mathcal{C}$
  - ad esempio i più difficili in **NP**
  - di cui non sappiamo se appartengono a $\mathbf{P} \subseteq \mathbf{NP}$
  - per i quali il nondeterminismo è davvero *l'unico modo* che conosciamo per risolvere il problema in tempo polinomiale
- Chiamiamo questi problemi *$\mathcal{C}$-completi*

Completezza

Definizione formale

Definizione di $\mathcal{C}$-Completezza

Dati una classe di complessità $\mathcal{C}$ e un linguaggio L
diciamo che L è $\mathcal{C}$-completo sse
$L \in \mathcal{C}$ e ogni linguaggio $L' \in \mathcal{C}$ può essere ridotto a L.

- In altre parole, L è almeno difficile quanto qualunque altro linguaggio in $\mathcal{C}$
- Non è detto che ogni classe $\mathcal{C}$ contenga problemi $\mathcal{C}$-completi
- ma di fatto è così per le principali classi di complessità

Completezza

Definizione formale

Definizione di $\mathcal{C}$-Completezza

Dati una classe di complessità $\mathcal{C}$ e un linguaggio L
diciamo che L è $\mathcal{C}$-completo sse
$L \in \mathcal{C}$ e ogni linguaggio $L' \in \mathcal{C}$ può essere ridotto a L.

- In altre parole, L è almeno difficile quanto qualunque altro linguaggio in $\mathcal{C}$
- Non è detto che ogni classe $\mathcal{C}$ contenga problemi $\mathcal{C}$-completi
- ma di fatto è così per le principali classi di complessità

Completezza = appartenenza + *hardness*

Definizione di $\mathcal{C}$-hardness

Dati una classe di complessità $\mathcal{C}$ e un linguaggio L
diciamo che L è $\mathcal{C}$-hard sse
ogni linguaggio $L' \in \mathcal{C}$ può essere ridotto a L.

- Quindi L è $\mathcal{C}$-completo sse L è $\mathcal{C}$-hard e appartiene a $\mathcal{C}$
- Se L è $\mathcal{C}$-hard, allora per risolverlo servono almeno le risorse specificate da $\mathcal{C}$

Completezza come categorizzazione di problemi

Caratterizzazione della vera complessità del problema

- Un problema in **P** appartiene anche a **NP**, **PSPACE, EXP** ...
  - si possono sempre scrivere algoritmi inefficienti...
- Quindi la semplice *appartenenza* a una classe non basta a caratterizzarne l'effettiva complessità
- Un problema si ritene completamente compreso quando si dispone di un risultato di completezza rispetto a una classe $\mathcal{C}$
- Perchè la completezza (= max difficoltà) ci dice che quel problema
  - è "il massimo che si può fare" con le risorse di calcolo specificate da $\mathcal{C}$
  - se riduciamo le risorse e rimpiccioliamo la classe $\mathcal{C}$, il problema finisce fuori
  - e allora $\mathcal{C}$ rappresenta davvero la complessità intrinseca del problema (risorse necessarie e sufficienti x la soluzione)

Completezza come categorizzazione di problemi

Caratterizzazione della vera complessità del problema

- Un problema in **P** appartiene anche a **NP**, **PSPACE, EXP** ...
  - si possono sempre scrivere algoritmi inefficienti...
- Quindi la semplice *appartenenza* a una classe non basta a caratterizzarne l'effettiva complessità
- Un problema si ritene completamente compreso quando si dispone di un risultato di completezza rispetto a una classe $\mathcal{C}$
- Perchè la completezza (= max difficoltà) ci dice che quel problema
  - è "il massimo che si può fare" con le risorse di calcolo specificate da $\mathcal{C}$
  - se riduciamo le risorse e rimpiccioliamo la classe $\mathcal{C}$, il problema finisce fuori
  - e allora $\mathcal{C}$ rappresenta davvero la complessità intrinseca del problema (risorse necessarie e sufficienti x la soluzione)

Completezza come categorizzazione di problemi

Caratterizzazione della vera complessità del problema

- Un problema in **P** appartiene anche a **NP**, **PSPACE, EXP** ...
  - si possono sempre scrivere algoritmi inefficienti...
- Quindi la semplice *appartenenza* a una classe non basta a caratterizzarne l'effettiva complessità
- Un problema si ritene completamente compreso quando si dispone di un risultato di completezza rispetto a una classe $\mathcal{C}$
- Perchè la completezza (= max difficoltà) ci dice che quel problema
  - è "il massimo che si può fare" con le risorse di calcolo specificate da $\mathcal{C}$
  - se riduciamo le risorse e rimpiccioliamo la classe $\mathcal{C}$, il problema finisce fuori
  - e allora $\mathcal{C}$ rappresenta davvero la complessità intrinseca del problema (risorse necessarie e sufficienti x la soluzione)

Completezza come categorizzazione di problemi

Caratterizzazione della vera complessità del problema

- Un problema in **P** appartiene anche a **NP**, **PSPACE, EXP** ...
  - si possono sempre scrivere algoritmi inefficienti...
- Quindi la semplice *appartenenza* a una classe non basta a caratterizzarne l'effettiva complessità
- Un problema si ritene completamente compreso quando si dispone di un risultato di completezza rispetto a una classe $\mathcal{C}$
- Perchè la completezza (= max difficoltà) ci dice che quel problema
  - è "il massimo che si può fare" con le risorse di calcolo specificate da $\mathcal{C}$
  - se riduciamo le risorse e rimpiccioliamo la classe $\mathcal{C}$, il problema finisce fuori
  - e allora $\mathcal{C}$ rappresenta davvero la complessità intrinseca del problema (risorse necessarie e sufficienti x la soluzione)

Completezza come categorizzazione di problemi

Caratterizzazione della vera complessità del problema

- Un problema in **P** appartiene anche a **NP**, **PSPACE, EXP** ...
  - si possono sempre scrivere algoritmi inefficienti...
- Quindi la semplice *appartenenza* a una classe non basta a caratterizzarne l'effettiva complessità
- Un problema si ritene completamente compreso quando si dispone di un risultato di completezza rispetto a una classe $\mathcal{C}$
- Perchè la completezza (= max difficoltà) ci dice che quel problema
  - è "il massimo che si può fare" con le risorse di calcolo specificate da $\mathcal{C}$
  - se riduciamo le risorse e rimpiccioliamo la classe $\mathcal{C}$, il problema finisce fuori $\leftarrow$ *Intuitivamente OK, ma siamo sicuri?*
  - e allora $\mathcal{C}$ rappresenta davvero la complessità intrinseca del problema (risorse necessarie e sufficienti x la soluzione)

Chiusura delle classi rispetto alle riduzioni

Definizione di chiusura

Una classe di complessità $\mathcal{C}$ è chiusa rispetto alle riduzioni se contiene tutti i problemi riducibili a qualche $L \in \mathcal{C}$

- cioè se quando contiene L, contiene anche tutti i problemi più semplici di L (o difficili quanto L)

Proposizione 8.3 del Papadimitriou

L, NL, P, NP, coNP, PSPACE, EXP sono chiusi rispetto alle riduzioni

- la lista non è affatto esaustiva...

Proprietà che seguono dalla chiusura - I

Proposizione 8.4 del Papadimitriou

Se due classi $\mathcal{C}_1$ e $\mathcal{C}_2$ sono chiuse rispetto alle riduzioni ed esiste un problema L completo sia per $\mathcal{C}_1$ che per $\mathcal{C}_2$, allora $\mathcal{C}_1 = \mathcal{C}_2$

- Prova:
  - Supponiamo per assurdo che esista $L' \in \mathcal{C}_1 \setminus \mathcal{C}_2$.
  - Per l'ipotesi di $\mathcal{C}_1$-completezza, L' può essere ridotto a L
  - Per l'ipotesi di chiusura di $\mathcal{C}_2$, $L' \in \mathcal{C}_2$ (assurdo)
  - Simmetricamente, se assumiamo che esista $L' \in \mathcal{C}_2 \setminus \mathcal{C}_1$ otteniamo una contraddizione.
  - Ne segue che $\mathcal{C}_1 = \mathcal{C}_2$ QED

Proprietà che seguono dalla chiusura - II

Corollario della Proposizione 8.4

Se $\mathcal{C}_1 \subseteq \mathcal{C}_2$, $\mathcal{C}_1$ è chiusa rispetto alle riduzioni e contiene un problema $\mathcal{C}_2$-completo L, allora $\mathcal{C}_1 = \mathcal{C}_2$

- Prova:
  - Supponiamo per assurdo che esista $L' \in \mathcal{C}_2 \setminus \mathcal{C}_1$.
  - Per l'ipotesi di $\mathcal{C}_2$-completezza, L' può essere ridotto a L, che appartiene anche a $\mathcal{C}_1$
  - Per l'ipotesi di chiusura di $\mathcal{C}_1$, $L' \in \mathcal{C}_1$ (assurdo) QED

Possibile utilizzo della Proprietà 8.4 e Corollario

- Ricordiamo che $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP}$
- Grazie al corollario sappiamo che
  - se un problema **P**-completo appartenesse a **L**, allora $\mathbf{P} = \mathbf{L}$
  - se un problema **P**-completo appartenesse a **NL**, allora $\mathbf{P} = \mathbf{NL}$
  - se un problema **NP**-completo appartenesse a **P** allora $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se un problema è **NP**-completo, allora può essere risolto in tempo polinomiale con un algoritmo "classico" (deterministico) se e solo se $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se non riuscite a trovare una soluzione polinomiale per un problema, e dimostrate che è **NP**-completo (o più difficile ancora), siete autorizzati a rinunciare
- Inoltre, sapendo che $\mathbf{P} \subset \mathbf{EXP}$,
  - **P** *non contiene* problemi **EXP**-completi
- Se siete curiosi sin da ora di vedere i problemi completi per le diverse classi, visitate il sito del *Complexity Zoo*

Possibile utilizzo della Proprietà 8.4 e Corollario

- Ricordiamo che $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP}$
- Grazie al corollario sappiamo che
  - se un problema **P**-completo appartenesse a **L**, allora $\mathbf{P} = \mathbf{L}$
  - se un problema **P**-completo appartenesse a **NL**, allora $\mathbf{P} = \mathbf{NL}$
  - se un problema **NP**-completo appartenesse a **P** allora $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se un problema è **NP**-completo, allora può essere risolto in tempo polinomiale con un algoritmo "classico" (deterministico) se e solo se $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se non riuscite a trovare una soluzione polinomiale per un problema, e dimostrate che è **NP**-completo (o più difficile ancora), siete autorizzati a rinunciare
- Inoltre, sapendo che $\mathbf{P} \subset \mathbf{EXP}$,
  - **P** *non contiene* problemi **EXP**-completi
- Se siete curiosi sin da ora di vedere i problemi completi per le diverse classi, visitate il sito del *Complexity Zoo*

Possibile utilizzo della Proprietà 8.4 e Corollario

- Ricordiamo che $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP}$
- Grazie al corollario sappiamo che
  - se un problema **P**-completo appartenesse a **L**, allora $\mathbf{P} = \mathbf{L}$
  - se un problema **P**-completo appartenesse a **NL**, allora $\mathbf{P} = \mathbf{NL}$
  - se un problema **NP**-completo appartenesse a **P** allora $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se un problema è **NP**-completo, allora può essere risolto in tempo polinomiale con un algoritmo "classico" (deterministico) se e solo se $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se non riuscite a trovare una soluzione polinomiale per un problema, e dimostrate che è **NP**-completo (o più difficile ancora), siete autorizzati a rinunciare
- Inoltre, sapendo che $\mathbf{P} \subset \mathbf{EXP}$,
  - **P** *non contiene* problemi **EXP**-completi
- Se siete curiosi sin da ora di vedere i problemi completi per le diverse classi, visitate il sito del *Complexity Zoo*

Possibile utilizzo della Proprietà 8.4 e Corollario

- Ricordiamo che $\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP}$
- Grazie al corollario sappiamo che
  - se un problema **P**-completo appartenesse a **L**, allora $\mathbf{P} = \mathbf{L}$
  - se un problema **P**-completo appartenesse a **NL**, allora $\mathbf{P} = \mathbf{NL}$
  - se un problema **NP**-completo appartenesse a **P** allora $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se un problema è **NP**-completo, allora può essere risolto in tempo polinomiale con un algoritmo "classico" (deterministico) se e solo se $\mathbf{P} = \mathbf{NP}$
  - $\Rightarrow$ Se non riuscite a trovare una soluzione polinomiale per un problema, e dimostrate che è **NP**-completo (o più difficile ancora), siete autorizzati a rinunciare
- Inoltre, sapendo che $\mathbf{P} \subset \mathbf{EXP}$,
  - **P** *non contiene* problemi **EXP**-completi
- Se siete curiosi sin da ora di vedere i problemi completi per le diverse classi, visitate il sito del *Complexity Zoo*

Esempi di problemi completi (e non) per **NP**

SAT è **NP** completo

Teorema di Cook-Levine

- Il teorema consiste di due parti: **NP**-hardness e appartenenza a **NP**
- la hardness vale addirittura per la soddisfacibilità di formule in CNF (conjunctive normal form)

Teorema (CNF SAT è NP-hard)

Se $L \in \mathbf{NP}$ allora L può essere ridotto alla soddisfacibilità di una formula proposizionale in CNF

Prova che SAT è **NP**-hard I

- Siccome $L \in \mathbf{NP}$, esiste una MdT nondeterministica M che accetta L in tempo $T = p(n)$, dove $p(n)$ è un polinomio in n (per def. di **NP**)
- La computazione di M può essere rappresentata con una tabella quadrata con $p(n)$ righe e colonne
  - detta *computation table*
- Questa matrice può essere descritta con formule proposizionali in CNF

Prova che SAT è **NP**-hard II

▷	0_s	1	1	0	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1_{q_0}	1	0	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1_{q_0}	0	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1	0_{q_0}	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1	0	$⊔_{q_0}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1	$0_{q'_0}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1	1_q	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1_q	1	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	$▷_q$	1	1	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	1_s	1	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	1_{q_1}	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	1	$⊔_{q_1}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	$1_{q'_1}$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	$▷_q$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	$⊔_s$	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔
▷	▷	▷	"yes"	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔	⊔

Prova che SAT è **NP**-hard III

- Descriviamo questa tabella con un numero di simboli proposizionali proporzionali al numero di celle nella tabella
  - Siano $\{q_1, \ldots, q_r\}$ gli stati di M, e $\{\sigma_1, \ldots, \sigma_l\}$ il suo alfabeto
  - Per ogni passo t e cella s (coordinate nella tabella) usiamo:
  - $P^i_{s,t}$ (*true* sse la cella contiene σ_i) $1 \leq i \leq l$
  - Q^i_t (*true* sse lo stato al tempo t è q_i) $1 \leq i \leq r$
  - $S_{s,t}$ (*true* sse la testina al tempo t è sulla cella s) $1 \leq s \leq T$
- Poi dobbiamo descrivere le formule in CNF che "obbligano" queste proposizioni a comportarsi come vogliamo

Prova che SAT è **NP**-hard IV

- Ad ogni passo la testina è su esattamente una cella: per ogni $t = 1, \ldots, T$

$$\left(\bigvee_{s=1,\ldots,T} S_{s,t} \right) \wedge \left(\bigwedge_{1 \leq s < i \leq T} \neg S_{s,t} \vee \neg S_{i,t} \right)$$

- Ad ogni passo in ogni cella si trova esattamente un simbolo: per ogni $s, t = 1, \ldots, T$

$$\left(\bigvee_{i=1,\ldots,I} P^i_{s,t} \right) \wedge \left(\bigwedge_{1 \leq i < j \leq T} \neg P^i_{s,t} \vee \neg P^j_{s,t} \right)$$

Prova che SAT è **NP**-hard V

- Ad ogni passo c'è esattamente uno stato corrente: per ogni $t = 1, \ldots, T$

$$\left(\bigvee_{i=1,\ldots,r} Q_t^i \right) \wedge \left(\bigwedge_{1 \leq i < j \leq r} \neg Q_t^i \vee \neg Q_t^j \right)$$

- Le condizioni iniziali sono soddisfatte: se lo stato iniziale è q_1, il blank q_2 e il nastro iniziale $\sigma_{i_1} \cdots \sigma_{i_n}$, (seguito da blanks)

$$Q_1^1 \wedge S_{1,1} \wedge P_{1,1}^{i_1} \wedge P_{2,1}^{i_2} \wedge \cdots \wedge P_{n,1}^{i_n} \wedge P_{n+1,1}^2 \wedge \cdots \wedge P_{T,1}^2$$

Prova che SAT è **NP**-hard VI

- I simboli non possono cambiare magicamente: se non c'è la testina restano uguali. Per ogni $s, t = 1, \ldots, T$ e $i = 1, \ldots, l$

$$\neg S_{s,t} \wedge P^i_{s,t} \rightarrow P^i_{s,t+1}$$

(espressa con la formula in CNF $S_{s,t} \vee \neg P^i_{s,t} \vee P^i_{s,t+1}$)

- La computazione accetta l'input: se "*yes*" $= q_k$

$$Q^k_T$$

Prova che SAT è **NP**-hard VII

- Le configurazioni cambiano come specificato dalla relazione di transizione Δ di M. Per ogni

$$(q_i, \sigma_j, q_k, \sigma_m, \leftarrow) \notin \Delta$$

abbiamo per ogni $t = 1, \ldots, T-1$ e $s = 2, \ldots, T$

$$Q_t^i \wedge S_{s,t} \wedge P_{s,t}^j \rightarrow \neg(Q_{t+1}^k \wedge P_{s,t+1}^m \wedge S_{s-1,t+1})$$

equivalente alla CNF

$$\neg Q_t^i \vee \neg S_{s,t} \vee \neg P_{s,t}^j \vee \neg Q_{t+1}^k \vee \neg P_{s,t+1}^m \vee \neg S_{s-1,t+1}$$

- Per esercizio scrivere le formule rimanenti che corrispondono alle mosse $\rightarrow$ e $-$
- La riduzione $R(x)$ è l'*and* di tutte queste formule

Prova che SAT è **NP**-hard VIII

- Ora è facile dimostrare la correttezza della riduzione:
  - se M accetta x, allora la computation table corrispondente ci dice com'è fatto l'assegnamento di verità che soddisfa $R(x)$
  - dato un assegnamento di verità che soddisfa $R(x)$ si può costruire la computation table che accetta x
  - all'esame vi chiedo i dettagli, allo stesso livello della riduzione da Hamilton Path a **NP**, quindi svolgeteli per esercizio

Prova che SAT è **NP**-hard IX

- Resta da dimostrare che R può essere calcolata in spazio $O(\log n)$
- La dimostrazione è analoga a quella per la riduzione di Hamilton Path:
  - con dei contatori che occupano spazio $O(\log p(n))$ genero le formule di $R(x)$ e le copio sul nastro di output
  - poichè $p(n)$ è un polinomio in n, diciamo di grado c, $\log p(n)$ è $O((c+1)\log n) = O(\log n)$

QED

Prova che SAT appartiene a **NP**

Ovvero: quanto manca per dimostrare che SAT è **NP**-completo

- Per risolvere SAT *nondeterministicamente* basta
  1. generare un truth assignment T (con **choose**)
  2. verificare se T soddisfa ϕ
- La generazione deve garantire che ogni possibile T sia generato da almeno una computazione
  - così se ϕ è soddisfatta da almeno un T_0, la MdT ha almeno un run che risponde "yes" (quello che genera T_0)
- Nota: la generazione nondeterministica di T astrae l'esplorazione dello spazio esponenziale di tutti i truth assignments
- nel modello nondeterministico ha un costo pari alla sola scrittura di T; che è *polinomiale* perchè:
- Rappresenteremo T con la lista dei simboli proposizionali P tali che $T(P) = true$
  - tutti gli altri sono implicitamente falsi

Prova che SAT appartiene a **NP**

Ovvero: quanto manca per dimostrare che SAT è **NP**-completo

- Per risolvere SAT *nondeterministicamente* basta
  1. generare un truth assignment T (con **choose**)
  2. verificare se T soddisfa ϕ
- La generazione deve garantire che ogni possibile T sia generato da almeno una computazione
  - così se ϕ è soddisfatta da almeno un T_0, la MdT ha almeno un run che risponde "yes" (quello che genera T_0)
- Nota: la generazione nondeterministica di T astrae l'esplorazione dello spazio esponenziale di tutti i truth assignments
- nel modello nondeterministico ha un costo pari alla sola scrittura di T; che è *polinomiale* perchè:
- Rappresenteremo T con la lista dei simboli proposizionali P tali che $T(P) = true$
  - tutti gli altri sono implicitamente falsi

Prova che SAT appartiene a **NP**

Ovvero: quanto manca per dimostrare che SAT è **NP**-completo

- Per risolvere SAT *nondeterministicamente* basta
  1. generare un truth assignment T (con **choose**)
  2. verificare se T soddisfa ϕ
- La generazione deve garantire che ogni possibile T sia generato da almeno una computazione
  - così se ϕ è soddisfatta da almeno un T_0, la MdT ha almeno un run che risponde "yes" (quello che genera T_0)
- Nota: la generazione nondeterministica di T astrae l'esplorazione dello spazio esponenziale di tutti i truth assignments
- nel modello nondeterministico ha un costo pari alla sola scrittura di T; che è *polinomiale* perchè:
- Rappresenteremo T con la lista dei simboli proposizionali P tali che $T(P) = true$
  - tutti gli altri sono implicitamente falsi

Prova che SAT appartiene a **NP**

Ovvero: quanto manca per dimostrare che SAT è **NP**-completo

- Per risolvere SAT *nondeterministicamente* basta
  1. generare un truth assignment T (con **choose**)
  2. verificare se T soddisfa ϕ
- La generazione deve garantire che ogni possibile T sia generato da almeno una computazione
  - così se ϕ è soddisfatta da almeno un T_0, la MdT ha almeno un run che risponde "yes" (quello che genera T_0)
- Nota: la generazione nondeterministica di T astrae l'esplorazione dello spazio esponenziale di tutti i truth assignments
- nel modello nondeterministico ha un costo pari alla sola scrittura di T; che è *polinomiale* perchè:
- Rappresenteremo T con la lista dei simboli proposizionali P tali che $T(P) = true$
  - tutti gli altri sono implicitamente falsi

Prova che SAT appartiene a **NP**– I

1 - Genera T

- Usiamo una MdT nondeterminstica con 1 nastro di input e 1 nastro ausiliario
  - che conterrà T (lista delle proposizioni vere)

1. Prima scandiamo l'input da sinistra a destra cercando i simboli proposizionali
   - per ognuno scegliamo nondeterministicamente se scriverlo sul secondo nastro o saltarlo

Prova che SAT appartiene a **NP**– I

1 - Genera T

- Usiamo una MdT nondeterminstica con 1 nastro di input e 1 nastro ausiliario
  - che conterrà T (lista delle proposizioni vere)

1. Prima scandiamo l'input da sinistra a destra cercando i simboli proposizionali
   - per ognuno scegliamo nondeterministicamente se scriverlo sul secondo nastro o saltarlo

Prova che SAT appartiene a **NP**– II

2 - T soddisfa ϕ?

2. Scandiamo l'input una seconda volta;
   - per ogni clausola e per ogni suo letterale controlliamo se è vero cercando il suo simbolo proposizionale nel secondo nastro
   - se è vero, saltiamo alla prossima clausola; se è falso, al prossimo letterale
   - se raggiungiamo la fine della clausola, terminiamo con "no";
   - se esauriamo le clausole, terminiamo con "si"

Prova che SAT appartiene a **NP**– II

2 - T soddisfa ϕ?

2. Scandiamo l'input una seconda volta;
   - per ogni clausola e per ogni suo letterale controlliamo se è vero cercando il suo simbolo proposizionale nel secondo nastro
   - se è vero, saltiamo alla prossima clausola; se è falso, al prossimo letterale
   - se raggiungiamo la fine della clausola, terminiamo con "no";
   - se esauriamo le clausole, terminiamo con "si"

Prova che SAT appartiene a **NP**– II

2 - T soddisfa ϕ?

2. Scandiamo l'input una seconda volta;
   - per ogni clausola e per ogni suo letterale controlliamo se è vero cercando il suo simbolo proposizionale nel secondo nastro
   - se è vero, saltiamo alla prossima clausola; se è falso, al prossimo letterale
   - se raggiungiamo la fine della clausola, terminiamo con "no";
   - se esauriamo le clausole, terminiamo con "si"

Prova che SAT appartiene a **NP**– III

2 - T soddisfa ϕ?

Analisi di complessità

- Il passo 1 (generazione di T) è lineare
- Il passo 2 (valutazione della formual) è quadratico
  - per ogni letterale nell'input
  - scandisce il secondo nastro per cercarne il valore
  - in tutto il tempo richiesto è $O(n^2)$

QED

Prova che SAT appartiene a **NP**– III

2 - T soddisfa ϕ?

Analisi di complessità

- Il passo 1 (generazione di T) è lineare
- Il passo 2 (valutazione della formual) è quadratico
  - per ogni letterale nell'input
  - scandisce il secondo nastro per cercarne il valore
  - in tutto il tempo richiesto è $O(n^2)$ QED

Prova che SAT appartiene a **NP**– III

2 - T soddisfa ϕ?

Analisi di complessità

- Il passo 1 (generazione di T) è lineare
- Il passo 2 (valutazione della formual) è quadratico
  - per ogni letterale nell'input
  - scandisce il secondo nastro per cercarne il valore
  - in tutto il tempo richiesto è $O(n^2)$ QED

Prova che SAT appartiene a **NP**– III

2 - T soddisfa ϕ?

Analisi di complessità

- Il passo 1 (generazione di T) è lineare $\leftarrow$ costo ricerca rimosso
- Il passo 2 (valutazione della formual) è quadratico
  - per ogni letterale nell'input
  - scandisce il secondo nastro per cercarne il valore
  - in tutto il tempo richiesto è $O(n^2)$ QED

Altre varianti di SAT

Cosa lo rende difficile?

- I tentativi di semplificare SAT abbastanza da portarlo in **P** falliscono finchè il linguaggio non diventa poco espressivo
- Esempio di semplificazione che resta **NP**-completa: 3-SAT

3-SAT è **NP**-completo

Definizione di k-SAT

- istanze: espressioni booleane ϕ in CNF dove le clausole hanno tutte k letterali, ad es. in 3-SAT:

$$(\neg P \vee Q \vee R) \wedge (P \vee Q \vee R)$$

- risposte: "yes" sse ϕ è soddisfacibile

- È immediato che 3-SAT $\in$ **NP** perchè è un caso particolare di SAT che appartiene a **NP**
- Per dimostrare che 3-SAT è **NP**-completo dobbiamo solo mostrare che è **NP**-hard

3-SAT è **NP**-completo

Definizione di k-SAT

- istanze: espressioni booleane ϕ in CNF dove le clausole hanno tutte k letterali, ad es. in 3-SAT:

$$(\neg P \vee Q \vee R) \wedge (P \vee Q \vee R)$$

- risposte: "yes" sse ϕ è soddisfacibile

- È immediato che 3-SAT $\in$ **NP** perchè è un caso particolare di SAT che appartiene a **NP**
- Per dimostrare che 3-SAT è **NP**-completo dobbiamo solo mostrare che è **NP**-hard

3-SAT è **NP**-completo

Definizione di k-SAT

- istanze: espressioni booleane ϕ in CNF dove le clausole hanno tutte k letterali, ad es. in 3-SAT:

$$(\neg P \vee Q \vee R) \wedge (P \vee Q \vee R)$$

- risposte: "yes" sse ϕ è soddisfacibile

- È immediato che 3-SAT $\in$ **NP** perchè è un caso particolare di SAT che appartiene a **NP**
- Per dimostrare che 3-SAT è **NP**-completo dobbiamo solo mostrare che è **NP**-hard

Prova che 3-SAT è **NP**-hard – I

Tecnica generale per dimostrare la hardness

- Basta ridurre un qualunque problema **NP**-hard X a 3-SAT
  - Ogni problema Y in **NP** si può ridurre a X perchè X è **NP**-hard
  - Componendo quella riduzione con quella da X a 3-SAT abbiamo una riduzione da Y a 3-SAT
  - Quindi ogni problema $Y \in$ **NP** si può ridurre a 3-SAT

- Noi useremo $X = \mathrm{SAT}$
  - la similitudine dei problemi facilita la riduzione
  - Nota: per la stessa ragione la conoscenza di molti problemi completi per le classi $\mathcal{C}$ facilita l'identificazione di altri problemi completi

Prova che 3-SAT è **NP**-hard – I

Tecnica generale per dimostrare la hardness

- Basta ridurre un qualunque problema **NP**-hard X a 3-SAT
  - Ogni problema Y in **NP** si può ridurre a X perchè X è **NP**-hard
  - Componendo quella riduzione con quella da X a 3-SAT abbiamo una riduzione da Y a 3-SAT
  - Quindi ogni problema $Y \in$ **NP** si può ridurre a 3-SAT

- Noi useremo $X = \mathrm{SAT}$
  - la similitudine dei problemi facilita la riduzione
  - Nota: per la stessa ragione la conoscenza di molti problemi completi per le classi $\mathcal{C}$ facilita l'identificazione di altri problemi completi

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- La riduzione spezza le clausole troppo lunghe aggiugendo nuovi simboli proposizionali ausiliari $N_1, N_2, \ldots$
  - ad es. la clausola $A \vee B \vee C \vee D$
  - diventa $(A \vee B \vee N_1) \wedge (\neg N_1 \vee C \vee D)$

  Se la clausola ha più di 4 letterali si spezza in 3 o più clausole usando 2 o più N_i
  - ad es. la clausola $A \vee B \vee C \vee D \vee E$
  - diventa $(A \vee B \vee N_1) \wedge (\neg N_1 \vee C \vee N_2) \wedge (\neg N_2 \vee D \vee E)$

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- Questa traduzione preserva la soddisfacibilità
  - consideriamo una ϕ che contiene $A \vee B \vee C \vee D \vee E$ e un T che soddisfa ϕ
  - consideriamo la traduzione

$$(A \vee B \vee N_1) \wedge (\neg N_1 \vee C \vee N_2) \wedge (\neg N_2 \vee D \vee E)$$

  - Se T soddisfa A o B allora soddisfa la prima clausola qui sopra; le altre due le possiamo soddisfare settando $T(N_1) = T(N_2) = \mathit{false}$
    - possiamo farlo perchè nel T originale gli N_i sono indefiniti
  - Se $T(C) = \mathit{true}$ allora la seconda è soddisfatta; le altre due le soddisfiamo settando $T(N_1) = \mathit{true}$ e $T(N_2) = \mathit{false}$
  - Se T soddisfa D o E allora basta settare $T(N_1) = T(N_2) = \mathit{true}$

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- Questa traduzione preserva la soddisfacibilità
  - consideriamo una ϕ che contiene $A \vee B \vee C \vee D \vee E$ e un T che soddisfa ϕ
  - consideriamo la traduzione

$$(A \vee B \vee N_1) \wedge (\neg N_1 \vee C \vee N_2) \wedge (\neg N_2 \vee D \vee E)$$

  - Se T soddisfa A o B allora soddisfa la prima clausola qui sopra; le altre due le possiamo soddisfare settando $T(N_1) = T(N_2) = \mathit{false}$
    - possiamo farlo perchè nel T originale gli N_i sono indefiniti
  - Se $T(C) = \mathit{true}$ allora la seconda è soddisfatta; le altre due le soddisfiamo settando $T(N_1) = \mathit{true}$ e $T(N_2) = \mathit{false}$
  - Se T soddisfa D o E allora basta settare $T(N_1) = T(N_2) = \mathit{true}$

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- Questa traduzione preserva la soddisfacibilità
  - consideriamo una ϕ che contiene $A \vee B \vee C \vee D \vee E$ e un T che soddisfa ϕ
  - consideriamo la traduzione

$$(A \vee B \vee N_1) \wedge (\neg N_1 \vee C \vee N_2) \wedge (\neg N_2 \vee D \vee E)$$

  - Se T soddisfa A o B allora soddisfa la prima clausola qui sopra; le altre due le possiamo soddisfare settando $T(N_1) = T(N_2) = \mathit{false}$
    - possiamo farlo perchè nel T originale gli N_i sono indefiniti
  - Se $T(C) = \mathit{true}$ allora la seconda è soddisfatta; le altre due le soddisfiamo settando $T(N_1) = \mathit{true}$ e $T(N_2) = \mathit{false}$
  - Se T soddisfa D o E allora basta settare $T(N_1) = T(N_2) = \mathit{true}$

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- Questa traduzione preserva la soddisfacibilità
  - consideriamo una ϕ che contiene $A \vee B \vee C \vee D \vee E$ e un T che soddisfa ϕ
  - consideriamo la traduzione

$$(A \vee B \vee N_1) \wedge (\neg N_1 \vee C \vee N_2) \wedge (\neg N_2 \vee D \vee E)$$

  - Se T soddisfa A o B allora soddisfa la prima clausola qui sopra; le altre due le possiamo soddisfare settando $T(N_1) = T(N_2) = \mathit{false}$
    - possiamo farlo perchè nel T originale gli N_i sono indefiniti
  - Se $T(C) = \mathit{true}$ allora la seconda è soddisfatta; le altre due le soddisfiamo settando $T(N_1) = \mathit{true}$ e $T(N_2) = \mathit{false}$
  - Se T soddisfa D o E allora basta settare $T(N_1) = T(N_2) = \mathit{true}$

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- Fin qui abiamo dimostrato che se ϕ è soddisfacibile, lo è anche la traduzione
- Ora dimostriamo che se la traduzione è soddisfatta da qualche T, allora anche la formula originale ϕ è soddisfatta da T
  - Supponiamo per assurdo che T non soddisfi $A \vee B \vee C \vee D \vee E$
  - Allora l'unico modo per soddisfare le 3 clausole è di soddisfare $N_1, \neg N_1, N_2, \neg N_2$
  - ma questo è impossibile!
  - Quindi T deve soddisfare $A \vee B \vee C \vee D \vee E$

QED (per questo caso particolare)

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- Fin qui abiamo dimostrato che se ϕ è soddisfacibile, lo è anche la traduzione
- Ora dimostriamo che se la traduzione è soddisfatta da qualche T, allora anche la formula originale ϕ è soddisfatta da T
  - Supponiamo per assurdo che T non soddisfi $A \vee B \vee C \vee D \vee E$
  - Allora l'unico modo per soddisfare le 3 clausole è di soddisfare $N_1, \neg N_1, N_2, \neg N_2$
  - ma questo è impossibile!
  - Quindi T deve soddisfare $A \vee B \vee C \vee D \vee E$

QED (per questo caso particolare)

Prova che 3-SAT è **NP**-hard – II

Riduzione da SAT a 3-SAT

- Fin qui abiamo dimostrato che se ϕ è soddisfacibile, lo è anche la traduzione
- Ora dimostriamo che se la traduzione è soddisfatta da qualche T, allora anche la formula originale ϕ è soddisfatta da T
  - Supponiamo per assurdo che T non soddisfi $A \vee B \vee C \vee D \vee E$
  - Allora l'unico modo per soddisfare le 3 clausole è di soddisfare $N_1, \neg N_1, N_2, \neg N_2$
  - ma questo è impossibile!
  - Quindi T deve soddisfare $A \vee B \vee C \vee D \vee E$

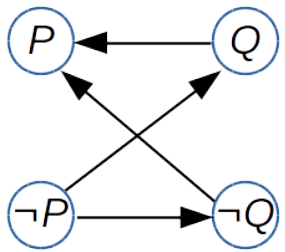
QED (per questo caso particolare)

Esercizio

- Usando queste intuizioni,
  1. scrivere una definizione ricorsiva formale della traduzione di clausole di lunghezza arbitraria
     - caso base: clausole di lunghezza ≤ 3
  2. con questa, scrivere una dimostrazione formale *generale* che la traduzione preserva la soddisfacibilità,
     - Suggerimento: fatela per induzione sulla lunghezza massima delle clausole

Perchè 3-SAT e non 2-SAT?

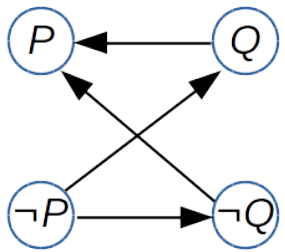
- Una congiunzione di clausole con 2 letterali è come un grafo
- Ogni clausola equivale a due implicazioni, ad es. $(P \vee \neg Q)$ equivale
  - sia a $\neg P \rightarrow Q$
  - sia a $Q \rightarrow P$
- Queste implicazioni sono gli archi del grafo
- Esempio: $\phi = (P \vee Q) \wedge (P \vee \neg Q)$ diventa



- Si può dimostrare che ϕ è soddisfacibile sse nessun ciclo contiene due letterali *complementari* (come P e $\neg P$)

Perchè 3-SAT e non 2-SAT?

- Una congiunzione di clausole con 2 letterali è come un grafo
- Ogni clausola equivale a due implicazioni, ad es. $(P \vee \neg Q)$ equivale
  - sia a $\neg P \rightarrow Q$
  - sia a $Q \rightarrow P$
- Queste implicazioni sono gli archi del grafo
- Esempio: $\phi = (P \vee Q) \wedge (P \vee \neg Q)$ diventa

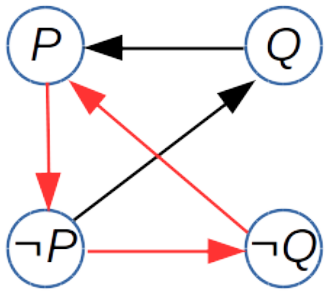


- Si può dimostrare che ϕ è soddisfacibile sse nessun ciclo contiene due letterali *complementari* (come P e $\neg P$)

Perchè 3-SAT e non 2-SAT?

(segue)

- Altro esempio: $(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg P)$ diventa

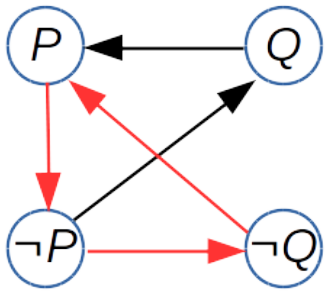


- Il ciclo implica $P \leftrightarrow \neg P$ (una contraddizione)
- Quindi la soddisfacibilità in 2-SAT si riduce alla ricerca di cicli in un grafo di dimensione lineare nell'input
- Questo è un problema in P
  - vedere la "bibbia" *Introduction to Algorithms* di Cormen et al.

Perchè 3-SAT e non 2-SAT?

(segue)

- Altro esempio: $(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg P)$ diventa

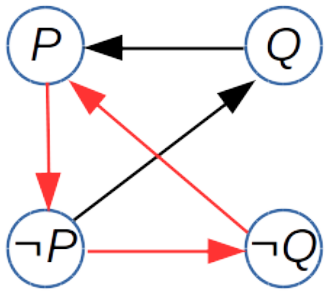


- Il ciclo implica $P \leftrightarrow \neg P$ (una contraddizione)
- Quindi la soddisfacibilità in 2-SAT si riduce alla ricerca di cicli in un grafo di dimensione lineare nell'input
- Questo è un problema in **P**
  - vedere la "bibbia" *Introduction to Algorithms* di Cormen et al.

Perchè 3-SAT e non 2-SAT?

(segue)

- Altro esempio: $(P \vee Q) \wedge (P \vee \neg Q) \wedge (\neg P \vee \neg P)$ diventa



- Il ciclo implica $P \leftrightarrow \neg P$ (una contraddizione)
- Quindi la soddisfacibilità in 2-SAT si riduce alla ricerca di cicli in un grafo di dimensione lineare nell'input
- Questo è un problema in **P**
  - vedere la "bibbia" *Introduction to Algorithms* di Cormen et al.

Clausole positive

- Che succede se invece permettiamo clausole di lunghezza arbitraria ma eliminiamo la negazione?
  - le clausole senza negazione vengono dette *positive*
- Il problema diventa banale:
- L'assegnamento di verità che rende vere tutte le proposizioni ovviamente soddisfa tutte le possibili clausole positive
- Complessità $O(1)$

Clausole positive

- Che succede se invece permettiamo clausole di lunghezza arbitraria ma eliminiamo la negazione?
  - le clausole senza negazione vengono dette *positive*
- Il problema diventa banale:
- L'assegnamento di verità che rende vere tutte le proposizioni ovviamente soddisfa tutte le possibili clausole positive
- Complessità $O(1)$

Clausole di Horn

- Un altro frammento in cui la soddisfacibilità è in **P** sono le *clausole di Horn*:
- Clausole che contengono al massimo 1 letterale positivo
  - quindi hanno disgiunzioni lunghe a piacere e negazione
- Per ora omettiamo la dimostrazione
- Le clausole di Horn corrispondono al frammento proposizionale di Prolog
  - ad es. $P \vee \neg Q \vee \neg R$ corrisponde a P :- Q, R

Un accenno a possibili nozioni di espressività basate sulla complessità computazionale

Complessità ed Espressività– I

- Consideriamo la logica proposizionale come un linguaggio di *rappresentazione della conoscenza*:
- La **NP**-hardness della logica proposizionale garantisce che possiamo rappresentare efficientemente tutti i problemi in **NP** in logica proposizionale
  - e risolverli con i *reasoning engines* per questa logica
- Le formule in CNF hanno la stessa proprietà – quindi la stessa *espressività*
  - possiamo codificare efficientemente gli stessi problemi, perchè CNF-SAT è ancora **NP**-hard
- In questo senso le formule in CNF hanno lo stesso potere espressivo della logica proposizionale non ristretta

Complessità ed Espressività– I

- Consideriamo la logica proposizionale come un linguaggio di *rappresentazione della conoscenza*:
- La **NP**-hardness della logica proposizionale garantisce che possiamo rappresentare efficientemente tutti i problemi in **NP** in logica proposizionale
  - e risolverli con i *reasoning engines* per questa logica
- Le formule in CNF hanno la stessa proprietà – quindi la stessa *espressività*
  - possiamo codificare efficientemente gli stessi problemi, perchè CNF-SAT è ancora **NP**-hard
- In questo senso le formule in CNF hanno lo stesso potere espressivo della logica proposizionale non ristretta

Complessità ed Espressività– I

- Consideriamo la logica proposizionale come un linguaggio di *rappresentazione della conoscenza*:
- La **NP**-hardness della logica proposizionale garantisce che possiamo rappresentare efficientemente tutti i problemi in **NP** in logica proposizionale
  - e risolverli con i *reasoning engines* per questa logica
- Le formule in CNF hanno la stessa proprietà – quindi la stessa *espressività*
  - possiamo codificare efficientemente gli stessi problemi, perchè CNF-SAT è ancora **NP**-hard
- In questo senso le formule in CNF hanno lo stesso potere espressivo della logica proposizionale non ristretta

Complessità ed Espressività– II

- Se invece indeboliamo il linguaggio
  - (limitando le clausole a 2 letterali, o solo a letterali positivi, o solo a clausole di Horn)

  non siamo più in grado di rappresentare efficientemente tutti i problemi in **NP**
  - non conosciamo codifiche in spazio logaritmico, o addirittura in **P**, perchè ignoriamo se $\mathbf{P} = \mathbf{NP}$
  - la traduzione potrebbe essere esponenziale
- Questi frammenti sono quindi *meno* espressivi della logica proposizionale non ristretta
  - a meno che $\mathbf{P} = \mathbf{NP}$
  - a oggi, in pratica, è come se fossero meno espressivi

$\Rightarrow$ La teoria della complessità fornisce un criterio di espressività anche per i casi in cui non sappiamo valutare fino in fondo la potenza dei linguaggi di rappresentazione

Complessità ed Espressività– II

- Se invece indeboliamo il linguaggio
  - (limitando le clausole a 2 letterali, o solo a letterali positivi, o solo a clausole di Horn)

  non siamo più in grado di rappresentare efficientemente tutti i problemi in **NP**
  - non conosciamo codifiche in spazio logaritmico, o addirittura in **P**, perchè ignoriamo se $\mathbf{P} = \mathbf{NP}$
  - la traduzione potrebbe essere esponenziale
- Questi frammenti sono quindi *meno* espressivi della logica proposizionale non ristretta
  - a meno che $\mathbf{P} = \mathbf{NP}$
  - a oggi, in pratica, è come se fossero meno espressivi

$\Rightarrow$ La teoria della complessità fornisce un criterio di espressività anche per i casi in cui non sappiamo valutare fino in fondo la potenza dei linguaggi di rappresentazione

Complessità ed Espressività– II

- Se invece indeboliamo il linguaggio
  - (limitando le clausole a 2 letterali, o solo a letterali positivi, o solo a clausole di Horn)

  non siamo più in grado di rappresentare efficientemente tutti i problemi in **NP**
  - non conosciamo codifiche in spazio logaritmico, o addirittura in **P**, perchè ignoriamo se $\mathbf{P} = \mathbf{NP}$
  - la traduzione potrebbe essere esponenziale
- Questi frammenti sono quindi *meno* espressivi della logica proposizionale non ristretta
  - a meno che $\mathbf{P} = \mathbf{NP}$
  - a oggi, in pratica, è come se fossero meno espressivi

$\Rightarrow$ La teoria della complessità fornisce un criterio di espressività anche per i casi in cui non sappiamo valutare fino in fondo la potenza dei linguaggi di rappresentazione

Nota sui concetti di espressività

- I logici puri dicono che due linguaggi hanno la stessa espressività quando
  - ogni formula nel primo è equivalente a una del secondo e viceversa
  - senza considerare la complessità delle traduzioni (e la loro dimensione)
- In AI invece la complessità conta (ovviamente) e non solo:
- Si distinguono traduzioni modulari e non modulari, per esempio, per questioni di
  - leggibilità
  - efficienza (impatto degli update sulla base di conoscenza)

Nota sui concetti di espressività

- I logici puri dicono che due linguaggi hanno la stessa espressività quando
  - ogni formula nel primo è equivalente a una del secondo e viceversa
  - senza considerare la complessità delle traduzioni (e la loro dimensione)
- In AI invece la complessità conta (ovviamente) e non solo:
- Si distinguono traduzioni modulari e non modulari, per esempio, per questioni di
  - leggibilità
  - efficienza (impatto degli update sulla base di conoscenza)

Capitolo di riferimento

Papadimitriou

- Parte 3, Capitolo 8, paragrafi 8.1 e 8.2
  - solo i risultati e le dimostrazioni riportati esplicitamente nelle slides (saltare i risultati relativi a circuit complexity)
- Articolo di Cook (per la **NP** completezza di SAT)