Lezione di Informatica Teorica: Classi di Complessità e Riduzioni

Appunti da Trascrizione Automatica

30 giugno 2025

Indice

1 Introduzione alle Classi di Complessità

Abbiamo iniziato a definire classi di complessità che sono sottoclassi della classe **R** (Problemi Decidibili). Ci chiediamo quali problemi siano facili e quali difficili.

Abbiamo introdotto i concetti di *computation time* e *running time* per Macchine di Turing (MT) e le funzioni di tempo. Da ciò, abbiamo definito le classi:

- **Time(**$f(n)$**)**: L'insieme di tutti i linguaggi decidibili da una Macchina di Turing deterministica in tempo $O(f(n))$.
- **NTime(**$f(n)$**)**: L'insieme di tutti i linguaggi decidibili da una Macchina di Turing non deterministica in tempo $O(f(n))$.

Queste classi ci permettono di definire il concetto di risolvibilità in tempo polinomiale, esponenziale, ecc., sia in modalità deterministica che non deterministica.

1.1 Classi P e NP

Definizione 1 (Classe P)**.** *La classe **P** è l'insieme di tutti i problemi di decisione decidibili da una Macchina di Turing deterministica in tempo polinomiale. Formalmente:* $\mathbf{P} = \bigcup_{k \geq 1} Time(n^k)$.

Definizione 2 (Classe NP)**.** *La classe **NP** è l'insieme di tutti i problemi di decisione decidibili da una Macchina di Turing non deterministica in tempo polinomiale. Formalmente:* $\mathbf{NP} = \bigcup_{k \geq 1} NTime(n^k)$.

Nota Importante: NP sta per *Non-deterministic Polynomial*, non per *Non-Polynomial*.

1.1.1 Relazione tra P e NP: Il Problema P vs NP

È intuitivo che $\mathbf{P} \subseteq \mathbf{NP}$. Tutto ciò che può essere deciso in tempo polinomiale deterministico può anche essere deciso in tempo polinomiale non deterministico (una MT deterministica è un caso particolare di MT non deterministica).

La relazione inversa, ovvero se $\mathbf{NP} \subseteq \mathbf{P}$ (il che implicherebbe $\mathbf{P} = \mathbf{NP}$), è un problema tuttora aperto e uno dei più importanti problemi non risolti dell'informatica teorica (e uno dei problemi del millennio).

Attualmente, sappiamo simulare una Macchina di Turing non deterministica con una deterministica con un overhead esponenziale. Questo significa che un linguaggio in **NP** può essere deciso da una MT deterministica in tempo esponenziale. Questo non lo colloca automaticamente in **P**. Non è stato ancora dimostrato che non esista un metodo più efficiente (polinomiale) per tale simulazione. La congettura comune è che $\mathbf{P} \neq \mathbf{NP}$.

1.2 Riduzione Polinomiale

Il concetto di riduzione che abbiamo già visto viene specificato per il contesto delle classi di complessità.

Definizione 3 (Riduzione Polinomiale)**.** *Siano A e B due linguaggi. Una **riduzione polinomiale** da A a B è una funzione* $f : \Sigma^* \to \Sigma^*$ *tale che:*

1. *f è calcolabile da una Macchina di Turing deterministica in tempo polinomiale.*
2. *Per ogni stringa* $w \in \Sigma^*$, $w \in A \iff f(w) \in B$.

La riduzione è denotata con $A \leq_p B$.

Il vincolo cruciale è che la trasformazione f (implementata da un trasduttore) deve avere un tempo di esecuzione polinomiale rispetto alla taglia dell'input.

2 NP-Hardness e NP-Completezza

Questi concetti sono fondamentali per classificare la difficoltà dei problemi all'interno e al di fuori di NP.

Definizione 4 (NP-Hardness)**.** *Un linguaggio L è **NP-hard** se per ogni linguaggio $L' \in \mathbf{NP}$, esiste una riduzione polinomiale da L' a L ($L' \leq_p L$).*

Intuitivamente, un linguaggio NP-hard è *almeno altrettanto difficile* quanto qualsiasi problema in NP.

Definizione 5 (NP-Completezza)**.** *Un linguaggio L è **NP-complete** se:*

1. $L \in \mathbf{NP}$
2. *L è NP-hard.*

In altre parole, un linguaggio NP-complete è un problema che appartiene a NP ed è tra i più difficili problemi in NP.

2.0.1 Differenza tra NP-Hard e NP-Complete

La distinzione è cruciale:

- Un problema NP-hard potrebbe non appartenere a NP. Ad esempio, il Linguaggio Universale (A_{TM}) è NP-hard (è difficile almeno quanto ogni problema in NP), ma non è NP-complete perché non è decidibile (e quindi non appartiene a NP).
- Un problema NP-complete è necessariamente in NP.

2.1 Teorema: Relazione tra $L \in$ NP-Complete e P vs NP

Teorema 1. *Sia L un linguaggio NP-complete. Allora, $L \in \mathbf{P}$ se e solo se $\mathbf{P} = \mathbf{NP}$.*

Dimostrazione. Dobbiamo dimostrare la doppia implicazione.

Direzione 1: Se $\mathbf{P} = \mathbf{NP}$, allora $L \in \mathbf{P}$. Poiché L è NP-complete, per definizione $L \in \mathbf{NP}$. Se $\mathbf{P} = \mathbf{NP}$, allora è immediato che $L \in \mathbf{P}$.

Direzione 2: Se $L \in \mathbf{P}$, allora $\mathbf{P} = \mathbf{NP}$. Sappiamo già che $\mathbf{P} \subseteq \mathbf{NP}$. Per dimostrare che $\mathbf{P} = \mathbf{NP}$, ci resta da dimostrare che $\mathbf{NP} \subseteq \mathbf{P}$. Assumiamo $L \in \mathbf{P}$. Poiché L è NP-complete, per definizione L è NP-hard. Questo significa che per ogni linguaggio $L' \in \mathbf{NP}$, esiste una riduzione polinomiale f da L' a L ($L' \leq_p L$). Consideriamo una Macchina di Turing $M_{L'}$ che decide L' utilizzando la riduzione a L. $M_{L'}$ è costruita come segue:

1. Input: una stringa w.
2. Calcola $y = f(w)$. Sia M_f la MT che calcola f.

3. Simula la Macchina di Turing M_L (che decide L) con input y.
4. $M_{L'}$ accetta se M_L accetta, e $M_{L'}$ rifiuta se M_L rifiuta.

Analizziamo il tempo di esecuzione di $M_{L'}$:

1. **Calcolo di $y = f(w)$:** Poiché f è una riduzione polinomiale, M_f opera in tempo $O(|w|^c)$ per una costante $c \geq 1$.
2. **Dimensione di y:** Poiché M_f opera in tempo $O(|w|^c)$, la dimensione dell'output y non può essere maggiore di $O(|w|^c)$. Ovvero, $|y| \leq O(|w|^c)$. (Una macchina non può scrivere più simboli di quanti passi compie).
3. **Simulazione di M_L su y:** Poiché abbiamo assunto $L \in \mathbf{P}$, M_L opera in tempo polinomiale. Sia $O(|input|^d)$ il suo tempo di esecuzione, per una costante $d \geq 1$. Quindi, M_L opera in tempo $O(|y|^d) = O((|w|^c)^d) = O(|w|^{c \cdot d})$.

Il tempo totale di esecuzione di $M_{L'}$ è $O(|w|^c) + O(|w|^{c \cdot d})$. Poiché c e d sono costanti, $c \cdot d$ è anch'essa una costante. Quindi, il tempo totale è polinomiale rispetto a $|w|$. Questo significa che per ogni linguaggio $L' \in \mathbf{NP}$, possiamo costruire una Macchina di Turing deterministica che lo decide in tempo polinomiale. Pertanto, $\mathbf{NP} \subseteq \mathbf{P}$. Combinando con $\mathbf{P} \subseteq \mathbf{NP}$, otteniamo $\mathbf{P} = \mathbf{NP}$. □

Questo teorema implica che per risolvere il problema P vs NP, è sufficiente trovare un algoritmo polinomiale per un singolo problema NP-complete, oppure dimostrare che tale algoritmo non esiste.

3 Proprietà delle Riduzioni Polinomiali

3.1 Transitività delle Riduzioni Polinomiali

Teorema 2. *Siano A, B, C tre linguaggi. Se $A \leq_p B$ e $B \leq_p C$, allora $A \leq_p C$.*

Dimostrazione.
- $A \leq_p B$: Esiste una riduzione polinomiale $f : \Sigma^* \to \Sigma^*$ con tempo $O(|w|^c)$ per $c \geq 1$.
- $B \leq_p C$: Esiste una riduzione polinomiale $g : \Sigma^* \to \Sigma^*$ con tempo $O(|w|^d)$ per $d \geq 1$.

Vogliamo dimostrare $A \leq_p C$. Consideriamo la funzione composta $h(w) = g(f(w))$.

1. **Correttezza:** $w \in A \iff f(w) \in B \iff g(f(w)) \in C$. Quindi, $w \in A \iff h(w) \in C$. La correttezza logica è mantenuta.
2. **Tempo di calcolo:**
   - Il calcolo di $f(w)$ richiede $O(|w|^c)$ tempo. La dimensione dell'output $f(w)$ è $O(|w|^c)$.
   - Il calcolo di $g(f(w))$ viene eseguito su input di dimensione $O(|w|^c)$. Poiché g richiede tempo $O(|input|^d)$, il tempo per calcolare $g(f(w))$ è $O((|w|^c)^d) = O(|w|^{c \cdot d})$.

   Il tempo totale per calcolare $h(w)$ è $O(|w|^c) + O(|w|^{c \cdot d})$, che è polinomiale.

Quindi, h è una riduzione polinomiale da A a C, e $A \leq_p C$. □

3.2 Utilizzo della Transitività per Dimostrare NP-Hardness

Teorema 3. *Sia A un linguaggio NP-hard. Se $A \leq_p B$, allora B è NP-hard.*

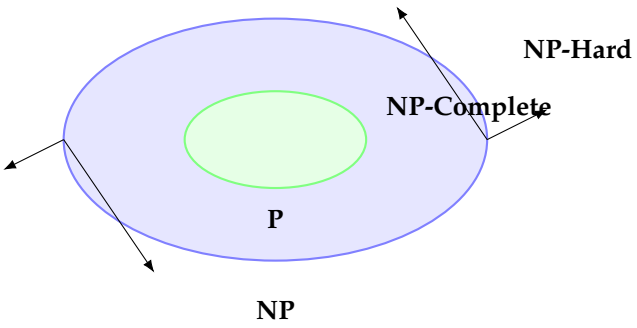
Dimostrazione. Per dimostrare che B è NP-hard, dobbiamo mostrare che per ogni linguaggio $L' \in$ **NP**, $L' \leq_p B$. Sappiamo che A è NP-hard (per ipotesi). Questo significa che per ogni $L' \in$ **NP**, $L' \leq_p A$. Per ipotesi, sappiamo anche che $A \leq_p B$. Combinando queste due riduzioni usando la transitività (Teorema precedente): $L' \leq_p A$ e $A \leq_p B \implies L' \leq_p B$. Poiché L' era un linguaggio generico in NP, abbiamo dimostrato che ogni linguaggio in NP può essere ridotto polinomialmente a B. Quindi B è NP-hard. □

Questo teorema è di fondamentale importanza: una volta dimostrato che un problema è NP-hard (il primo è stato SAT), possiamo dimostrare l'NP-hardness di altri problemi tramite riduzioni "a catena", partendo da un problema NP-hard già noto.

3.3 Illustrazione Grafica delle Classi di Complessità

Immaginiamo una rappresentazione visuale delle classi:

- **P** come un insieme interno (problemi "facili").
- **NP** come un insieme più grande che contiene **P**.
- I problemi **NP**-complete sono i problemi "più difficili" all'interno di **NP**, formando una specie di "bordo" o "frontiera" di **NP**. Se uno di questi fosse in **P**, allora **P** e **NP** collasserebbero.
- I problemi **NP**-hard sono tutti i problemi "almeno difficili quanto" **NP**. Essi includono i problemi **NP**-complete e possono anche includere problemi al di fuori di **NP** (es. problemi indecidibili), che sono ancora più difficili.



4 Problemi NP-Complete: SAT e 3-SAT

Il problema **SAT** (Boolean Satisfiability Problem) è l'insieme delle formule booleane in Forma Normale Congiuntiva (CNF) che sono soddisfacibili. SAT è stato il primo problema dimostrato essere NP-complete (Teorema di Cook-Levin, 1971). Ci fidiamo di questo risultato per ora.

4.1 Definizione di 3-SAT

Definizione 6 (Formula 3-CNF)**.** *Una formula booleana è in **3-CNF** se è in CNF e ogni sua clausola contiene* al più *tre letterali.*

Definizione 7 (Problema 3-SAT)**.** *Il problema **3-SAT** è l'insieme di tutte le formule booleane in 3-CNF che sono soddisfacibili.*

Esempio 1. *Una formula in 3-CNF potrebbe essere:* $F = (x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee x_4) \wedge (x_3 \vee x_5)$ *Ogni clausola ha al più 3 letterali (la seconda ne ha 2, le altre 3).*

Nonostante la restrizione sulla struttura delle clausole, 3-SAT è anch'esso un problema NP-complete.

4.2 Dimostrazione che 3-SAT è NP-Complete

Per dimostrare che 3-SAT è NP-complete, dobbiamo provare due cose:

1. 3-SAT $\in$ **NP**
2. 3-SAT è NP-hard

4.2.1 3-SAT $\in$ NP

Un problema è in NP se può essere deciso da una macchina di Turing non deterministica in tempo polinomiale. Per 3-SAT, una MT non deterministica può:

1. **Guess (indovinare):** Indovinare un'assegnazione di verità per tutte le variabili booleane nella formula. Questo può essere fatto in tempo polinomiale (lineare nel numero di variabili, che è al più lineare nella lunghezza della formula).
2. **Check (verificare):** Verificare se l'assegnazione indovinata soddisfa la formula. Questo comporta la valutazione di ogni clausola. Per una formula in 3-CNF, ogni clausola ha al più 3 letterali, quindi la valutazione di una clausola è costante. Verificare tutte le clausole richiede tempo polinomiale (lineare nel numero di clausole).

Poiché entrambe le fasi (guess e check) richiedono tempo polinomiale, 3-SAT $\in$ **NP**.

4.2.2 3-SAT è NP-hard (mediante riduzione SAT $\leq_p$ 3-SAT)

Per dimostrare che 3-SAT è NP-hard, useremo il teorema precedente e ridurremo SAT a 3-SAT ($SAT \leq_p 3SAT$). Poiché SAT è NP-complete (quindi NP-hard), se $SAT \leq_p 3SAT$, allora 3-SAT è NP-hard.

Obiettivo: Data una formula booleana ϕ in CNF (un'istanza di SAT), costruire una formula ψ in 3-CNF (un'istanza di 3-SAT) tale che ϕ è soddisfacibile se e solo se ψ è soddisfacibile. La trasformazione deve essere polinomiale.

Descrizione della Trasformazione: Sia $\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ una formula in CNF, dove ogni C_i è una clausola. Il processo di trasformazione da ϕ a ψ è iterativo. Costruiamo una serie di formule intermedie $\phi^{(0)}, \phi^{(1)}, \ldots, \phi^{(k)} = \psi$, dove $\phi^{(0)} = \phi$. Ad ogni passo, $\phi^{(j+1)}$ è ottenuta da $\phi^{(j)}$ riscrivendo le clausole che hanno più di tre letterali.

Consideriamo una generica clausola $C_i = (L_1 \vee L_2 \vee \cdots \vee L_k)$, dove L_j sono letterali.

1. **Caso 1:** $k \leq 3$ **(la clausola ha al più 3 letterali).** La clausola C_i viene copiata direttamente in $\phi^{(j+1)}$ senza modifiche.

2. **Caso 2:** $k > 3$ **(la clausola ha più di 3 letterali).** Questa clausola C_i viene sostituita da un insieme di clausole più piccole, introducendo nuove variabili ausiliarie. Sostituiamo $C_i = (L_1 \vee L_2 \vee L_3 \vee \cdots \vee L_k)$ con due nuove clausole: $C_i' = (L_1 \vee L_2 \vee H_i)$ $C_i'' = (H_i \vee L_3 \vee \cdots \vee L_k)$ dove H_i è una nuova variabile booleana non presente in ϕ o in altre variabili ausiliarie già introdotte. **Attenzione:** La seconda clausola C_i'' ha $k - 2 + 1 = k - 1$ letterali (se $k - 2 \geq 0$). Quindi, C_i'' potrebbe ancora avere più di 3 letterali se $k - 1 > 3$.

Questo processo viene ripetuto. Partiamo da $\phi = \phi^{(0)}$. Per ogni clausola in $\phi^{(j)}$ che ha più di 3 letterali, applichiamo la trasformazione del Caso 2 per ottenere clausole per $\phi^{(j+1)}$. Continuiamo questo processo finché tutte le clausole nella formula risultante hanno al più 3 letterali. Questa formula finale sarà ψ.

Analisi del Tempo di Trasformazione: Ogni passo della trasformazione riduce la lunghezza delle clausole "lunghe". Una clausola con k letterali viene sostituita da una clausola con 3 letterali e una clausola con $k - 1$ letterali. Questo processo garantisce che la dimensione delle clausole si riduca progressivamente. Il numero di iterazioni è al più proporzionale alla lunghezza massima delle clausole in ϕ. Il numero di clausole nella formula finale ψ sarà al più polinomiale nel numero di clausole originali di ϕ (e nella lunghezza di ϕ). Il numero di nuove variabili introdotte è anch'esso polinomiale. Pertanto, la trasformazione è calcolabile in tempo polinomiale.

Correttezza della Trasformazione: ϕ è soddisfacibile $\iff$ ψ è soddisfacibile.

Direzione 1: ϕ è soddisfacibile $\implies$ ψ è soddisfacibile. Supponiamo che ϕ sia soddisfacibile. Esiste quindi un'assegnazione di verità σ per le variabili di ϕ che rende ϕ vera. Vogliamo mostrare che esiste un'assegnazione di verità τ per le variabili di ψ (che include le variabili di ϕ più le variabili ausiliarie H_i) che rende ψ vera. Definiamo τ in modo che valuti le variabili di ϕ esattamente come σ. Per le variabili ausiliarie H_i, le valuteremo in modo opportuno.

Consideriamo una clausola $C_i = (L_1 \vee L_2 \vee \cdots \vee L_k)$ di ϕ. Se $k \leq 3$, C_i è copiata in ψ. Poiché σ soddisfa C_i, anche τ soddisferà C_i (dato che τ replica σ). Se $k > 3$, C_i è sostituita da $C_i' = (L_1 \vee L_2 \vee H_i)$ e $C_i'' = (H_i \vee L_3 \vee \cdots \vee L_k)$. Poiché σ soddisfa C_i, almeno un letterale L_j in C_i è vero sotto σ.

- Se L_1 o L_2 è vero sotto σ: Assegniamo $H_i =$ Falso. Allora C_i' è vera (per L_1 o L_2) e C_i'' diventa (Falso $\vee L_3 \vee \cdots \vee L_k$). Poiché σ soddisfaceva C_i, se L_1 o L_2 erano Falsi, ci doveva essere un altro L_j (con $j \geq 3$) vero. Se questo è il caso, C_i'' sarà vera.

- Se L_1 e L_2 sono Falsi sotto σ: Allora deve esserci un L_j con $j \geq 3$ che è vero sotto σ. Assegniamo $H_i =$ Vero. Allora C_i' diventa (Falso $\vee$ Falso $\vee$ Vero), che è vero. E C_i'' diventa (Vero $\vee L_3 \vee \cdots \vee L_k$), che è vero (dato che H_i è Vero).

In entrambi i casi, possiamo assegnare un valore a H_i in modo che $C_i' \wedge C_i''$ sia vera. Questo processo si applica a cascata per tutte le trasformazioni intermedie. Dunque, se ϕ è soddisfacibile, ψ è soddisfacibile.

Direzione 2: ψ è soddisfacibile $\implies$ ϕ è soddisfacibile. Supponiamo che ψ sia soddisfacibile. Esiste quindi un'assegnazione di verità τ per le variabili di ψ (che include le variabili di ϕ e le ausiliarie) che rende ψ vera. Vogliamo mostrare che l'assegnazione σ ottenuta da τ ignorando le variabili ausiliarie (cioè restringendo τ alle sole variabili di ϕ) rende ϕ vera.

Consideriamo una clausola $C_i = (L_1 \vee L_2 \vee \cdots \vee L_k)$ di ϕ. Se $k \leq 3$, C_i è identica in ϕ e ψ. Poiché C_i è vera sotto τ, lo è anche sotto σ (dato che σ è la restrizione di τ). Se $k > 3$, C_i è sostituita da

$C_i' = (L_1 \vee L_2 \vee H_i)$ e $C_i'' = (H_i \vee L_3 \vee \cdots \vee L_k)$. Poiché τ soddisfa ψ, sia C_i' che C_i'' sono vere sotto τ.

- Se H_i è vera sotto τ: Allora C_i' è vera perché H_i è vera. E C_i'' è vera perché H_i è vera.
- Se H_i è falsa sotto τ: Allora C_i' deve essere vera per L_1 o L_2. E C_i'' deve essere vera per almeno un L_j ($j \geq 3$).

In entrambi i casi, per garantire che $C_i' \wedge C_i''$ sia vera sotto τ, deve essere vero che $(L_1 \vee L_2 \vee \cdots \vee L_k)$ è vero sotto τ (e quindi sotto σ per le sole variabili di ϕ). Infatti, $L_1 \vee L_2 \vee \cdots \vee L_k \equiv (L_1 \vee L_2 \vee H_i) \wedge (H_i \vee L_3 \vee \cdots \vee L_k)$ non è una tautologia. La trasformazione mantiene la soddisfacibilità se e solo se H_i è considerata una variabile. Il punto chiave è che se $C_i' \wedge C_i''$ è vero, allora deve essere vero $(L_1 \vee L_2 \vee \cdots \vee L_k)$. Se $\tau(H_i) = \mathit{Vero}$, allora $L_1 \vee L_2 \vee \mathit{Vero}$ è Vero e $\mathit{Vero} \vee L_3 \vee \cdots \vee L_k$ è Vero. Non ci dice nulla sugli L_j. Se $\tau(H_i) = \mathit{Falso}$, allora $L_1 \vee L_2 \vee \mathit{Falso}$ è Vero (quindi $L_1 \vee L_2$ è Vero) e $\mathit{Falso} \vee L_3 \vee \cdots \vee L_k$ è Vero (quindi $L_3 \vee \cdots \vee L_k$ è Vero). In entrambi i sottocasi, abbiamo che $(L_1 \vee L_2 \vee \cdots \vee L_k)$ è vero. Questo si propaga a ritroso attraverso le iterazioni della trasformazione. Dunque, se ψ è soddisfacibile, anche ϕ è soddisfacibile.

Avendo dimostrato che $SAT \leq_p 3SAT$ e sapendo che SAT è NP-hard, concludiamo che 3-SAT è NP-hard. Poiché 3-SAT $\in$ **NP** e 3-SAT è NP-hard, allora 3-SAT è NP-complete.

4.3 Problema 2-SAT

Si può dimostrare che il problema **2-SAT** (formule in CNF dove ogni clausola ha esattamente due letterali) appartiene alla classe **P**. Nonostante la somiglianza con 3-SAT, 2-SAT è un problema molto più semplice, risolvibile in tempo polinomiale. Questo sottolinea come anche piccole restrizioni sulla struttura del problema possano avere un impatto enorme sulla sua complessità.

4.4 Esercizio: EXACT 3-SAT

Definizione 8 (Problema EXACT 3-SAT)**.** *Il problema **EXACT 3-SAT** è l'insieme di tutte le formule booleane in CNF in cui ogni clausola contiene* esattamente *tre letterali, e la formula è soddisfacibile.*

Esercizio per casa: Dimostrare che EXACT 3-SAT è NP-complete. **Suggerimento:** Ridurre 3-SAT a EXACT 3-SAT.