

Elementi di complessità computazionale

Prof. Paolo Brandimarte

Dip. di Scienze Matematiche – Politecnico di Torino

e-mail: paolo.brandimarte@polito.it

URL: staff.polito.it/paolo.brandimarte

Questa versione: 6 agosto 2024

NOTA: A uso didattico interno per il corso di laurea in Ingegneria Matematica PoliTO. Da non postare o ridistribuire.

Contenuto

Le slide seguenti sono tratte dal capitolo 9 di: P. Brandimarte, *Ottimizzazione per la Ricerca Operativa*, CLUT 2022.

- Complessità di problemi e algoritmi.
- Teoria della NP-completezza: classi \mathcal{P} , $\mathcal{NP}\mathcal{C}$ e $\mathcal{NP}\mathcal{H}$.

Complessità di problemi e algoritmi

Si vuole caratterizzare la complessità in funzione della dimensione di un problema. Occorre distinguere la complessità degli algoritmi da quella dei problemi.

Nel caso di un algoritmo caratterizzato da un numero finito di passi, possiamo valutare (eventualmente nel caso peggiore) il numero di operazioni elementari in funzione della dimensione n del problema.

NB. Non saremo molto precisi nel caratterizzare la dimensione del problema (spazio di memoria richiesto) e il numero di passi (su macchina di Turing). Inoltre ci limitiamo a trattare la complessità nel caso peggiore (non quello medio).

Per esempio, un algoritmo che consideri tutte le permutazioni di n oggetti ha complessità $O(n!)$ ed è certamente non pratico per n grandi. Allo stesso modo, un algoritmo che considera tutti gli assegnamenti possibili di n variabili binarie ha complessità esponenziale $O(2^n)$.

Al contrario, un algoritmo di ordinamento di n oggetti ha complessità polinomiale. Gli algoritmi più semplici hanno complessità $O(n^2)$, altri hanno complessità $O(n \log_2 n)$.

Nel caso di algoritmi iterativi che generano una sequenza di soluzioni, si può cercare di caratterizzare la velocità di convergenza (es., lineare o quadratica).

Una questione più sottile si pone quando si vuole caratterizzare la complessità intrinseca di un *problema*.

Per comprendere la natura della questione, consideriamo un semplice problema di scheduling di n job su macchina singola. Indichiamo con p_j , il tempo necessario per il job $j = 1, \dots, n$, e con d_j , $j = 1, \dots, n$ la sua data di consegna (due date).

La soluzione è una sequenza di job, ovvero una permutazione σ in cui $\sigma(k)$ è l'indice del job in posizione k . I tempi di completamento sono

$$\begin{aligned} C_{\sigma(1)} &= p_{\sigma(1)}, \\ C_{\sigma(k)} &= C_{\sigma(k-1)} + p_{\sigma(k)}, \quad k = 2, \dots, n. \end{aligned}$$

Si vuole minimizzare la massima lateness,

$$L_{\max} \doteq \max_{j \in [n]} L_j,$$

dove $L_j \doteq C_j - d_j$. Tale problema viene indicato con la stringa $1//L_{\max}$.

Teorema (regola EDD - Earliest Due Date) Per il problema $1//L_{\max}$ esiste una soluzione ottima in cui $d_{\sigma(k)} \leq d_{\sigma(k+1)}$.

Dimostrazione. Supponiamo che esista una soluzione ottima in cui, per due job consecutivi in sequenza, prima $i = \sigma(k)$ e poi $j = \sigma(k+1)$, si abbia $d_i > d_j$. Indichiamo con C_i e $C_j = C_i + p_j$ i due tempi di completamento nella soluzione corrente, per la quale abbiamo due valori di lateness $L_i = C_i - d_i$ e $L_j = C_j - d_j$.

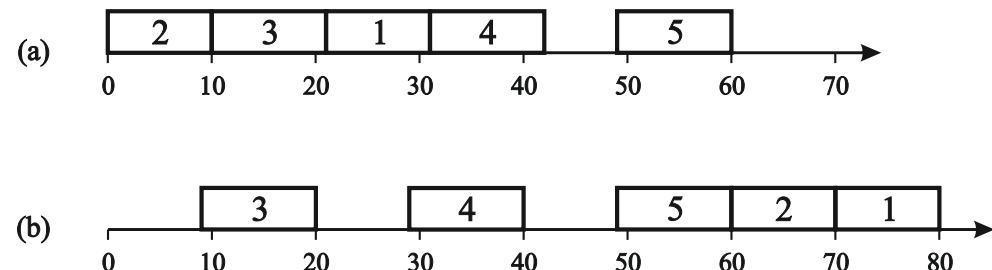
Se scambiano i due job, avremo C'_j e $C'_i = C_j$. Per il job j , che anticipiamo, abbiamo $C'_j < C_j$ e $L'_j < L_j$, e quindi la lateness del job j non può che migliorare nella nuova soluzione. Per il job i , che viene spostato in avanti nella sequenza, abbiamo

$$L'_i = C'_i - d_i = C_j - d_i < C_j - d_j = L_j,$$

quindi il job i peggiora la sua lateness, che però non potrà essere peggio della vecchia lateness del job j . La nuova soluzione migliora quella precedente, contraddicendo l'assunzione di ottimalità.

Abbiamo quindi un algoritmo di complessità polinomiale per il problema. Ma cosa accade se complichiamo leggermente il problema, introducendo dei tempi di rilascio (*release time* o *ready time*) r_i , $i \in [n]$, dei job? Per il problema $1/r_i/L_{\max}$ non sono noti algoritmi di complessità polinomiale, ed è facile costruire controesempi alla regola EDD (da adattare comunque alla disponibilità di job).

J_i	J_1	J_2	J_3	J_4	J_5
d_i	80	70	20	40	60
p_i	10	10	11	11	11
r_i	0	0	9	29	49



Esistono algoritmi branch-and-bound per il problema $1/r_i/L_{\max}$ (quindi complessità esponenziale). Non si conosce un algoritmo di complessità polinomiale per questo problema di ottimizzazione combinatoria (e per molti altri), ma neppure è stato dimostrato che esso non possa esistere.

Esiste una classe molto ampia di problemi di ottimizzazione per cui non sono disponibili algoritmi di complessità polinomiale.

Tuttavia, la questione dell'esistenza o meno di un algoritmo di complessità polinomiale per essi rimane aperta.

La teoria della NP-completezza ci permette di dare una risposta parziale alla questione, mostrando come questi problemi siano equivalenti tra di loro, nel senso che un algoritmo di complessità polinomiale per anche uno solo di essi fornirebbe un algoritmo di complessità polinomiale per tutti i problemi di una classe molto ampia.

Il fatto che decenni di ricerca sulla soluzione di tutti questi problemi non abbiano prodotto un algoritmo polinomiale suggerisce che esso in effetti non esiste.

Occorre distinguere **problemi di decisione** e **problemi di ottimizzazione**. Esempi di problema di decisione sono i seguenti:

Problema K_0 (subset sum): Dati $n+1$ numeri interi positivi a_1, a_2, \dots, a_n e b , esiste un sottoinsieme $\mathcal{J} \subseteq [n]$ tale che $\sum_{i \in \mathcal{J}} a_i = b$?

Problema LS_0 : Dato un problema di lot sizing multiprodotto, con tempi e costi di setup, esiste una soluzione ammissibile rispetto al soddisfacimento della domanda e ai vincoli di capacità produttiva?

Data una specifica istanza di un problema di decisione, la risposta è *sì* oppure *no*.

Dal punto di vista teorico, è più agevole trattare problemi di decisione, ma è facile vedere il legame tra problemi di ottimizzazione e problemi di decisione. Dato un problema di ottimizzazione

$$\min_{x \in S} f(x),$$

possiamo definirne una versione decisionale, scegliendo un numero k e chiedendoci se esiste $x \in S$ tale che $f(x) \leq k$, dove k è un numero intero. Indichiamo

- con PO il problema di ottimizzazione,
- e con PD il corrispondente problema di decisione.

Se abbiamo a disposizione un algoritmo efficiente per PO, allora possiamo risolvere in modo efficiente anche PD: basta risolvere il problema di ottimizzazione e verificare se $f(x^*) \leq k$. Questo ci permette di scrivere

$$PD \rightarrow PO,$$

nel senso che il problema di decisione può essere ricondotto al problema di ottimizzazione.

Non è detto che tale trasformazione sia conveniente, ma possiamo escludere che PO sia facile se PD è difficile, perché un ipotetico algoritmo efficiente per PO risolverebbe anche facilmente PD.

Quindi, per dimostrare che un problema di ottimizzazione è difficile, può bastare dimostrare che è difficile il corrispondente problema di decisione.

D'altro canto, un algoritmo di decisione efficiente potrebbe, in certi casi, essere utilizzato per risolvere il problema di ottimizzazione. Se la funzione di costo in PO assume valori interi non negativi, e abbiamo un upper bound U sul costo ottimo, possiamo applicare una procedura di bisezione.

Indichiamo con \mathcal{P} la classe dei problemi di decisione per cui esiste un algoritmo di complessità polinomiale, in grado di risolvere tutte le possibili istanze del problema.

Con questo vogliamo dire che il numero di passi, e quindi la complessità *temporale* dell'algoritmo è limitata superiormente da una funzione polinomiale dello spazio di memoria necessario per descrivere ogni istanza del problema.

Un tale algoritmo è in grado di fare *due* cose:

1. generare una soluzione;
2. verificarne la correttezza.

Esistono problemi, come \mathcal{K}_0 , per cui la prima parte del compito è difficile, ma la seconda no. Se enumeriamo, mediante un albero di ricerca, tutti i possibili sottinsiemi \mathcal{J} , possiamo verificare se una specifica istanza ha risposta positiva o negativa, ma chiaramente tale algoritmo ha complessità esponenziale.

Tuttavia, se disponessimo di un ipotetico calcolatore **non deterministico**, in grado di eseguire un numero infinito di processi di calcolo in parallelo, saremmo in grado di risolvere il problema in tempo polinomiale.

Definizione (classe \mathcal{NP}) Si definisce classe \mathcal{NP} l'insieme dei problemi di decisione le cui istanze che hanno risposta positiva sono verificabili in tempo polinomiale.

Per definizione, $\mathcal{P} \subseteq \mathcal{NP}$. Una questione meno ovvia è se vale $\mathcal{P} \equiv \mathcal{NP}$ o $\mathcal{P} \subset \mathcal{NP}$ in senso stretto.

È ragionevole, da questo punto di vista, cercare di caratterizzare la sottoclasse dei problemi più difficili in \mathcal{NP} .

Definizione (riduzione in tempo polinomiale) Siano P e Q due problemi di decisione, per cui ogni istanza I_P di P può essere trasformata in tempo polinomiale in un'istanza I_Q di Q tale che

$$I_P \text{ ha risposta positiva se e solo se } I_Q \text{ ha risposta positiva.}$$

Diremo che P è **riducibile in tempo polinomiale** a Q , e useremo la notazione $P \prec Q$. La notazione $P \prec Q$ sottolinea che la complessità di P non è maggiore della complessità di trasformare P in Q e poi risolvere Q ,

$$\text{compl}(P) \leq \text{compl}(Q) + \text{compl}(P \rightarrow Q).$$

Se la trasformazione ha una complessità trascurabile, la riduzione di P a Q mostra che Q non è più facile di P . Se P è difficile e $P \prec Q$, Q non può essere facile. Altrimenti, potremmo trasformare un'istanza di P in una di Q , e poi applicare l'algoritmo efficiente per Q .

Definizione. Un problema di decisione P è detto NP-difficile (NP-hard) se ogni problema nella classe \mathcal{NP} è riducibile a P . Indichiamo con \mathcal{NPH} la classe dei problemi NP-difficili.

Definizione. Un problema di decisione P è detto NP-completo se è in \mathcal{NP} ed è NP-difficile. Indichiamo con \mathcal{NPC} la classe dei problemi NP-completi.

Le implicazioni pratiche di tali definizioni sono:

1. un problema NP-difficile non è più facile di un problema qualsiasi in \mathcal{NP} ;
2. la classe \mathcal{NPC} è la classe dei problemi più difficili in \mathcal{NP} .

Per dimostrare che un problema di decisione P è NP-completo, occorre dimostrare:

1. che P è in \mathcal{NP} ;
2. che un problema NP-completo Q può essere ridotto in tempo polinomiale a P , cioè $Q \prec P$.

Osserviamo che, dato che Q è NP-completo, $P \prec Q$, e se trascuriamo la complessità della trasformazione, questo implica

$$\text{compl}(P) \leq \text{compl}(Q).$$

Ma il secondo passo della dimostrazione di NP-completezza, ovvero dimostrare che $Q \prec P$, implica anche che

$$\text{compl}(Q) \leq \text{compl}(P).$$

Le due diseguaglianze, sempre a meno della complessità della trasformazione, mostrano che

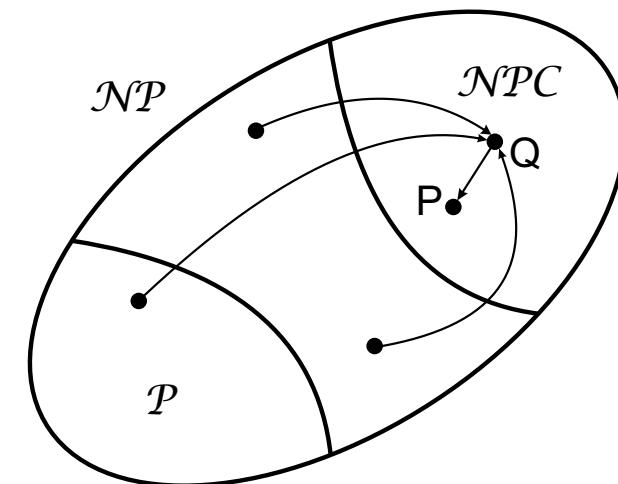
$$\text{compl}(Q) = \text{compl}(P).$$

In altre parole, i problemi della classe \mathcal{NPC} sono equivalenti in termini di complessità computazionale, e un algoritmo polinomiale per uno di essi permetterebbe di risolverli tutti in tempo polinomiale.

Avremmo quindi $\mathcal{P} = \mathcal{NP}$, ipotesi non troppo plausibile a causa dell'equivalenza di una vasta classe di problemi per i quali non è noto un algoritmo di complessità polinomiale, nonostante essi siano stati oggetto di ampio studio nel corso degli anni.

Se accettiamo l'ipotesi $\mathcal{P} \neq \mathcal{NP}$, possiamo rappresentare le relazioni tra le classi \mathcal{P} , \mathcal{NP} e \mathcal{NPC} come in figura. Essa ipotizza una gerarchia per cui la classe \mathcal{P} sarebbe la classe dei problemi più facili in \mathcal{NP} , e \mathcal{NPC} quella dei problemi più difficili.

Tutti i problemi in \mathcal{NP} si possono trasformare nel problema $Q \in \mathcal{NPC}$. Se $P \in \mathcal{NP}$, per dimostrare che $P \in \mathcal{NPC}$, occorre trasformare Q in P .



Se dimostriamo che un problema P è in \mathcal{NPC} , questo può a sua volta essere trasformato in altri problemi, permettendoci di ampliare la classe dei problemi noti in \mathcal{NPC} . Il punto critico, chiaramente, è trovare l'innesto della catena, ovvero il primo problema in \mathcal{NPC} , al quale tutti i problemi in \mathcal{NP} possono essere ricondotti.

Il teorema di Cook dimostra che **problema della soddisfacibilità** soddisfa i requisiti necessari e ci fornisce la soluzione.

Data una formula Booleana in forma canonica disgiuntiva, decidere se esiste un'assegnamento di valori ai suoi elementi che la rende vera.

Per esempio, la formula

$$(A \text{ or } B) \text{ and } (\text{not}(A) \text{ or } C),$$

definita rispetto alle variabili Booleane A , B e C , è soddisfatta se B e C sono entrambe vere. Al contrario,

$$(A \text{ or } B) \text{ and } (\text{not}(A) \text{ or } B) \text{ and } (\text{not}(B))$$

non può essere soddisfatta da alcun assegnamento di verità alle variabili.

A partire dal problema della soddisfacibilità, si può ricavare per riduzioni polinomiali successive una famiglia crescente di problemi NP-completi, compreso il problema \mathcal{K}_0 , che può essere considerato come un cugino in versione decisionale del problema knapsack.

Il fatto che tale problema faccia parte della classe \mathcal{NPC} ci permette di dimostrare il teorema seguente, che risolve la questione da cui siamo partiti.

Teorema. Consideriamo una versione decisionale del problema $1/r_i/L_{\max}$: dati i tempi di rilascio r_i , le date di consegna d_i e i tempi di lavorazione p_i , tutti a valori interi positivi, per n job J_i , $i \in [n]$, esiste una soluzione in cui nessun job è completato in ritardo? Tale problema di decisione è NP-completo.

Dimostrazione. Il problema è chiaramente in \mathcal{NP} , poiché per una data sequenza è facile verificare se i job vengono completati in tempo rispetto alle due date.

Dati gli interi positivi a_j , $j \in [n]$, creiamo n job J_j con parametri

$$r_j = 0, \quad p_j = a_j, \quad d_j = 1 + \sum_{k \in [n]} a_k, \quad j \in [n].$$

Creiamo un ulteriore job J_0 con parametri

$$r_0 = b, \quad p_0 = 1, \quad d_0 = b + 1.$$

Perché tutti i job rispettino la data di consegna, è necessario che il job J_0 inizi la lavorazione al tempo $t = b$. Inoltre, dato che la data di consegna degli altri job è pari alla somma di tutti i tempi di lavorazione, la soluzione non può presentare periodi di tempo in cui la macchina è ferma, prima di avere completato l'intero insieme di job.

Questo richiede che sia possibile individuare un sottoinsieme \mathcal{J} di job da schedulare prima di J_0 , in modo tale che

$$\sum_{j \in \mathcal{J}} p_j = r_0.$$

Tale insieme risolve il problema subset-sum.

Dai problemi di decisione ai problemi di ottimizzazione

Il teorema dimostra che un problema di decisione legato al problema di ottimizzazione $1/r_i/L_{\max}$ è NP-completo, ma cosa possiamo dire del problema di ottimizzazione stesso?

Per definizione, la classe \mathcal{NPC} contiene solo problemi di decisione.

Possiamo però estendere le classi \mathcal{P} e \mathcal{NPH} , includendo in esse anche problemi di ottimizzazione.

I problemi di ottimizzazione per cui è noto un algoritmo di complessità polinomiale stanno in \mathcal{P} .

Nella classe \mathcal{NPH} possiamo includere problemi di ottimizzazione ai quali possiamo ricondurre un corrispondente problema di decisione.

Nel nostro caso, la versione decisionale di $1/r_i/L_{\max}$ si riduce chiaramente al problema di ottimizzazione.

Inoltre, nella dimostrazione abbiamo assunto che i dati fossero numeri interi. Ma il problema a numeri interi può evidentemente essere ridotto al problema generale. Possiamo quindi affermare che il problema di scheduling $1/r_i/L_{\max}$ è NP-difficile.

L'impatto della codifica di un problema

Nella trattazione ci siamo limitati alle classi fondamentali e siamo stati piuttosto informali e imprecisi.

Non possiamo però fare a meno di considerare l'impatto del modo in cui si codifica un problema.

Sarebbe infatti errato, per esempio, associare a un problema knapsack una dimensione pari al numero di oggetti. La dimensione si riferisce a una codifica binaria che comprende tutti i dati del problema. Per mostrare la rilevanza di ciò, consideriamo un classico algoritmo di programmazione dinamica per la soluzione del problema knapsack:

$$\begin{aligned} \max \quad & \sum_{k=1}^n v_k x_k \\ \text{s.t.} \quad & \sum_{k=1}^n w_k x_k \leq B \\ & x_k \in \{0, 1\} \quad k = 1, \dots, n. \end{aligned}$$

Definiamo la funzione valore

$V_k(s) \doteq$ valore del sottoinsieme ottimale tra gli oggetti $\{k, k+1, \dots, n\}$,
quando la capacità residua è s .

In sostanza, la funzione valore assume che siano già state fatte scelte di inserimento o meno degli oggetti da 1 a $k - 1$; a valle di tale selezione, abbiamo una capacità residua s , e ci chiediamo come utilizzarla al meglio per le scelte rimanenti. Se i dati w_k e B del problema sono interi, lo sarà anche la capacità residua s .

Per risolvere il problema, ovvero trovare il valore di $V_1(B)$, si applica una relazione ricorsiva:

$$V_k(s) = \begin{cases} V_{k+1}(s) & \text{per } 0 \leq s < w_k, \\ \max \{V_{k+1}(s), V_{k+1}(s - w_k) + v_k\} & \text{per } w_k \leq s \leq B. \end{cases} \quad (1)$$

Si tratta di una equazione funzionale con condizione terminale:

$$V_n(s) = \begin{cases} 0 & \text{per } 0 \leq s < w_n, \\ v_n & \text{per } w_n \leq s \leq B. \end{cases} \quad (2)$$

Occorre tabulare tutte le funzioni $V_k(s)$, $k = 1, \dots, n$, per valori interi di s , che assume valori nel range da 0 a B . Pertanto, tale algoritmo ha complessità $O(nB)$.

Questo *non* implica che il problema knapsack abbia complessità polinomiale: per rappresentare il valore B in aritmetica binaria bastano $\lceil \log_2 B \rceil$ bit.

Quindi l'algoritmo, rispetto a tale codifica binaria, ha complessità esponenziale. Se si utilizzasse un computer con una codifica unaria, l'algoritmo che abbiamo considerato avrebbe complessità polinomiale.

Si dice infatti che un algoritmo di questo tipo è **pseudo-polinomiale**.