Fondamenti di Algoritmi, Complessità e Problem Solving

Corso tenuto dal Professor Ferdinando Cicalese

Università di Verona

Alessio Gjergji

Indice

Capitolo 1

Introduzione

1.1 Cos'è la complessità computazionale?

La Teoria della Complessità pone domande fondamentali sullo studio dei problemi computazionali, cercando di comprendere la natura e le sfide che questi rappresentano. Le questioni centrali che guida la ricerca in questo campo includono:

- Come le risorse necessarie per risolvere un problema si scalano con una misura della dimensione del problema?
- Perché alcuni problemi sono difficili e altri facili?
- Cosa rende i problemi difficili, "difficili"?
- Perché tutto ciò è importante per noi?

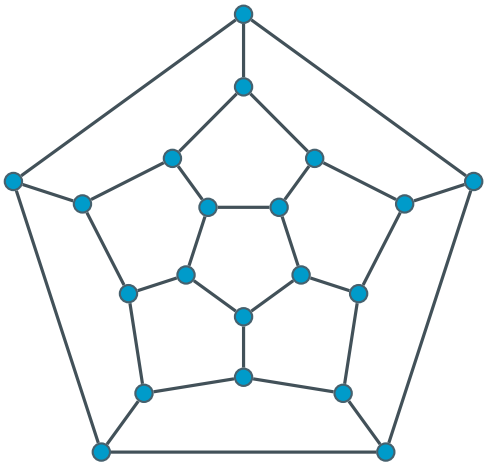
Queste domande ci aiutano a capire non solo la natura dei problemi computazionali ma anche come e perché alcune questioni sono intrinsecamente più complesse di altre. Esplorando queste domande, la Teoria della Complessità ci fornisce strumenti e metriche per valutare e confrontare problemi computazionali, gettando luce sui limiti del calcolo e sull'efficacia degli algoritmi.

Definizione di Complessità Computazionale

La Complessità Computazionale è lo studio delle risorse necessarie per risolvere problemi computazionali.

1.1.1 Primi problemi computazionali

Supponiamo di di avere il seguente grafo:



Supponiamo di voler risolvere questi due problemi:

- C'è un cammino del grafo $\mathcal{G}$ che tocca tutti gli archi esattamente una volta?
- C'è un cammino del grafo $\mathcal{G}$ che tocca tutti i nodi esattamente una volta?

Il primo problema è noto come **Problema del Ciclo Euleriano**, mentre il secondo è noto come **Problema del Cammino Hamiltoniano**.

1.1.2 Il Ciclo Euleriano

Il problema del Ciclo Euleriano è stato formulato da Leonhard Euler nel 1736, quando risolse il problema dei ponti di Königsberg. Il problema consisteva nel trovare un percorso che attraversasse ogni ponte della città esattamente una volta senza ripercorrere lo stesso ponte.

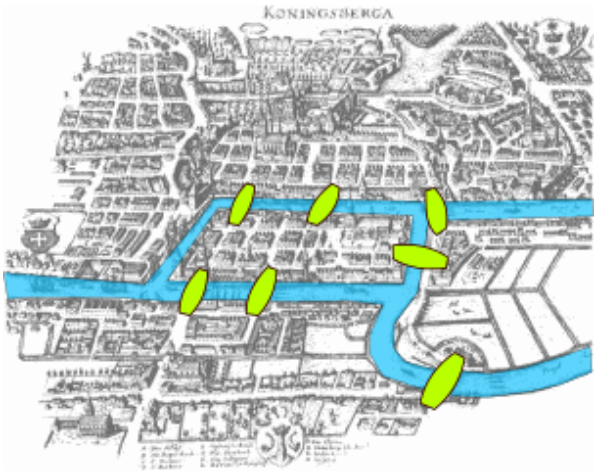


Figura 1.1.1: I ponti di Königsberg

Euler riuscì a trasformare questo problema pratico in un quesito astratto di teoria dei grafi. Sostanzialmente, il problema chiede se esiste un ciclo in un grafo che attraversi tutti gli archi

esattamente una volta. Euler formulò un teorema, che descrive le condizioni necessarie affinché esista tale ciclo.

Teorema di Euler

Un grafo connesso e non orientato possiede un ciclo che inizia e termina sullo stesso vertice e attraversa ogni arco esattamente una volta se e solo se ogni vertice ha grado pari. Se ci sono esattamente due vertici di grado dispari, allora esiste un percorso che inizia in un vertice, attraversa ogni arco esattamente una volta, e termina nell'altro vertice.

Euler dimostrò che un grafo ha un ciclo euleriano se e solo se è connesso e ogni nodo ha un grado pari. Se il grafo non è connesso, o se ha più di due nodi con un grado dispari, allora non può avere un ciclo euleriano. Questo risultato non solo ha risolto il problema dei ponti di Königsberg ma ha anche gettato le basi per il campo della teoria dei grafi.

Il costo computazionale per **decidere** se un grafo ha un ciclo euleriano è $O(|V| + |E|)$, dove $|V|$ è il numero di nodi e $|E|$ è il numero di archi.

Algorithm 1: Determinazione di un ciclo euleriano in un grafo $\mathcal{G}$

Input : Un grafo connesso e non orientato $\mathcal{G}$
Output: Booleano che indica se il grafo $\mathcal{G}$ ha un ciclo euleriano

```
1  odd-vertex-num ← 0
2  foreach v ∈ G.V do
3  |  if degree(v) mod 2 ≠ 0 then
4  |  |  odd-vertex-num ← odd-vertex-num + 1
5  |  end
6  end
7  if odd-vertex-num = 0 ∨ odd-vertex-num = 2 then
8  |  return true
9  end
10 return false
```

Ci chiediamo ora quanto costa **certificare** che il grafo $\mathcal{G}$ ha un ciclo euleriano. La complessità computazionale per certificare che un grafo ha un ciclo euleriano è $O(|V| + |E|)$, dove $|V|$ è il numero di nodi e $|E|$ è il numero di archi. Questo perché ci basterebbe scorrere il grafo e contare il grado di ogni nodo.

1.1.3 Il Cammino Hamiltoniano e il Ciclo Hamiltoniano

Proprio come il problema dei ponti di Königsberg ha portato alla definizione del ciclo euleriano, la ricerca di un ciclo che visiti ogni vertice di un grafo esattamente una volta ha portato alla definizione di un ciclo hamiltoniano. Questo problema prende il nome dal matematico William Rowan Hamilton che lo studiò nel XIX secolo.

A differenza dei cicli euleriani, dove un ciclo deve attraversare tutti gli archi esattamente una volta, un ciclo hamiltoniano deve passare per tutti i vertici una sola volta e tornare al vertice di partenza. Il problema del ciclo hamiltoniano chiede se tale ciclo esista in un dato grafo.

Problema del Ciclo Hamiltoniano

Un ciclo hamiltoniano esiste in un grafo $\mathcal{G}$ se e solo se esiste una sequenza chiusa che visita ogni vertice una volta prima di ritornare al vertice iniziale. Tuttavia, a differenza del problema euleriano, non esiste un criterio semplice per verificare l'esistenza di un ciclo hamiltoniano in un grafo generale.

Il problema del ciclo hamiltoniano è `NP-completo`, il che significa che non è noto alcun algoritmo efficiente che lo risolva per ogni grafo in tempo polinomiale. È interessante notare che, mentre per i cicli euleriani è relativamente semplice determinarne l'esistenza e costruirli, per i cicli hamiltoniani anche solo la verifica della loro esistenza può essere computazionalmente impegnativa.

Quanto costa **decidere** se un grafo $\mathcal{G}$ è hamiltoniano? Non lo sappiamo con certezza! Forse richiede tempo esponenziale? Quanto costa **verificare** che un grafo G sia o non sia hamiltoniano? Per mostrare che lo è, basta fornire un cammino hamiltoniano.

1.1.4 Il Problema della partizione massima

Consideriamo i seguenti 38 numeri. La loro somma è 2 000 000:

```
14175 15055 16616 17495 18072 19390 19731 22161 23320 23717
26343 28725 29127 32257 40020 41867 43155 46298 56734 57176
58306 61848 65825 66042 68634 69189 72936 74287 74537 81942
82027 82623 82802 82988 90467 97042 97507 99564
```

Per verificare una potenziale soluzione a questo problema, si potrebbe pensare di provare tutte le possibili combinazioni dei 38 numeri presi 19 alla volta. Questo approccio, tuttavia, comporterebbe l'esplorazione di circa 35×10^9 configurazioni diverse, un compito computazionalmente oneroso.

Se, invece, vi fosse data una specifica partizione dei numeri, il costo per verificarne la validità sarebbe notevolmente inferiore. Per confermare che una partizione proposta è una soluzione valida, sarebbe sufficiente:

1. Assicurarsi che ciascun gruppo sia composto esattamente da 19 numeri.
2. Sommare i numeri in uno dei gruppi per verificare che la loro somma sia pari a 1000000.

Questa metodologia di verifica fornisce un modo efficiente per confermare la correttezza di una soluzione proposta senza la necessità di esaminare tutte le possibili combinazioni.

1.1.5 La Primalità di un Numero e la Ricerca di Fattori Piccoli

Un problema fondamentale nell'aritmetica e nella crittografia è determinare se un numero intero N sia primo o meno. Inoltre, si può voler sapere se N ha un fattore piccolo, inferiore a un certo limite q.

Un esempio storico notevole fu presentato da Frank Cole nel 1903, quando dimostrò che:

$$N = 193707721 \times 761838257287$$

scomponendo così il numero in due fattori primi, senza l'ausilio di computer o calcolatrici elettroniche, ma probabilmente mediante metodi sistematici e molta pazienza.

Algoritmi per la Primalità

L'algoritmo più efficiente conosciuto per decidere se un numero intero N sia primo ha una complessità temporale di $O((\log N)^{6+\epsilon})$, dove ϵ è un piccolo numero positivo. Questo dimostra che, sebbene non sia immediato, il problema della primalità può essere risolto in tempo polinomiale.

Fattorizzazione:

D'altra parte, non conosciamo una procedura efficiente per fattorizzare un grande intero nei suoi divisori, al di là del tentativo di tutte le possibilità. Questo rende la fattorizzazione di grandi numeri una sfida significativa, specialmente per i numeri che hanno solo fattori grandi.

Costo della Certificazione di un Fattore Piccolo

La certificazione o verifica di un fattore piccolo di N può essere relativamente semplice e rapida. Se ci viene dato un fattore $f < q$, possiamo semplicemente dividere N per f e verificare se il risultato è un numero intero senza resto. Questo processo ha un costo computazionale di $O(\log N)$, rendendolo efficiente anche per numeri molto grandi.

Questa discussione evidenzia il contrasto tra la relativa facilità di verificare la primalità o la presenza di fattori piccoli e la difficoltà significativa di fattorizzare numeri grandi. Questo contrasto è al cuore di molti sistemi di crittografia moderni, che si affidano alla difficoltà di fattorizzare come garanzia di sicurezza.

1.1.6 Il problema degli scacchi

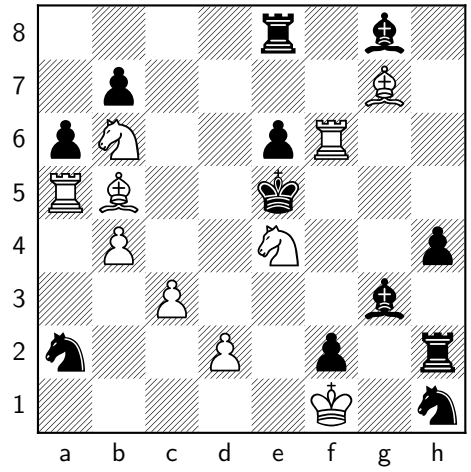


Figura 1.1.2: Scacco Matto in 3 mosse

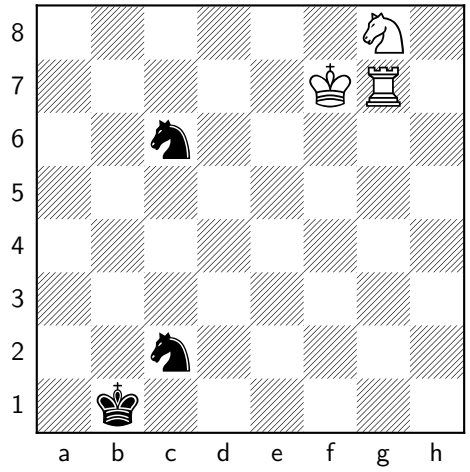


Figura 1.1.3: Scacco Matto in 262 mosse

Un aspetto interessante degli scacchi è la verifica della presenza di uno scacco matto in una data posizione. La complessità di tale verifica aumenta con il numero di mosse necessarie per raggiungere lo scacco matto dallo stato attuale della scacchiera. Consideriamo il seguente ragionamento logico per dimostrare uno scacco matto in n mosse:

"Esiste una mossa w_1 tale che, per qualsiasi risposta b_1, esiste una mossa w_2 tale che, per qualsiasi risposta b_2, ..., fino a che per qualsiasi risposta b_{n-1}, esiste una mossa w_n che produce uno scacco matto."

Questo implica una sequenza di mosse forzate in cui il Bianco, indipendentemente dalle mosse del Nero, può garantire uno scacco matto. Tale sequenza di mosse è spesso indicata come "matto in n mosse". Il processo di verifica di questa affermazione richiede una conoscenza approfondita delle strategie di scacchi e, per posizioni complesse, può richiedere molto tempo.

1.2 Problema computazionale

Problema computazionale

Un problema computazionale descrive un infinito insieme di possibili input, chiamati *istanze*, e rappresenta una relazione che mappa ogni istanza a un insieme non vuoto di possibili output.

Questa definizione sottolinea due componenti fondamentali di un problema computazionale:

- **Insieme di Istanze**: Ogni problema computazionale ammette un'infinità di possibili istanze di input. Queste istanze rappresentano le diverse configurazioni o scenari su cui il problema può operare.

- **Relazione Input-Output**: Per ogni istanza di input, esiste una relazione ben definita che determina l'insieme dei possibili output. Questa relazione è cruciale per definire correttamente il problema e per comprendere la natura delle soluzioni possibili.

Un problema computazionale non è una singola domanda, ma una famiglia di domande, ognuna delle quali corrisponde a una diversa istanza di input.

1.2.1 Misurazione dell'efficienza algoritmica

L'efficienza di un algoritmo nel risolvere un problema computazionale è fondamentale per comprendere sia la sua praticità sia la sua applicabilità a insiemi di dati di grandi dimensioni. La misurazione dell'efficienza algoritmica si basa su diversi fattori chiave:

- **Dimensione dell'istanza (n)**: Rappresenta la quantità di dati in input all'algoritmo. La dimensione dell'istanza è spesso il fattore più diretto che influenza la complessità di un problema.
- **Crescita delle risorse utilizzate ($T(n)$)**: Indica come le risorse necessarie per eseguire l'algoritmo crescono con l'aumentare della dimensione dell'input. $T(n)$ è spesso espressa in termini di tempo (*ad esempio, il numero di operazioni richieste*) o spazio (*ad esempio, la quantità di memoria necessaria*).
- **Caso peggiore**: Analizza la complessità dell'algoritmo nella situazione più sfavorevole possibile. Questo scenario fornisce un limite superiore sulle risorse necessarie per qualsiasi input di dimensione n.

Anche se non trattato qui, è importante notare che, oltre al caso peggiore, si può considerare anche il caso medio, che fornisce un'indicazione delle risorse necessarie in media, dato un insieme rappresentativo di istanze di input.

Questa analisi permette di valutare e confrontare algoritmi in base alla loro efficienza, guidando la scelta dell'algoritmo più adatto per un determinato problema e insieme di dati.

1.2.2 Complessità computazionale

La **complessità computazionale** si riferisce alla quantità di risorse computazionali necessarie per risolvere il problema in questione. La complessità può essere esplorata sotto due principali prospettive: i limiti superiori e i limiti inferiori.

Limiti Superiori

I *limiti superiori* indicano quanto bene possiamo risolvere il problema, cioè la complessità dell'algoritmo più efficiente conosciuto che risolve il problema. Dimostrare un limite superiore significa mostrare che esiste almeno un algoritmo con la complessità affermata per risolvere il problema.

Limiti Inferiori

I *limiti inferiori*, d'altra parte, indicano quanto sia difficile il problema, ossia la quantità minima di risorse che ogni algoritmo deve utilizzare per risolverlo. Per stabilire un limite inferiore, dobbiamo mostrare che tutti gli algoritmi hanno una complessità almeno pari al limite inferiore affermato.

Esempio

Consideriamo il problema della moltiplicazione di due numeri interi di n cifre.

- **Limiti Superiori:** La procedura elementare insegnata nelle scuole richiede $O(n^2)$ moltiplicazioni ed addizioni elementari. Tuttavia, questo non è necessariamente la complessità del problema della moltiplicazione, poiché esistono algoritmi più efficienti.
- **Limiti Inferiori:** Al minimo, dobbiamo leggere tutto l'input, il che implica una complessità di almeno $\Omega(n)$.
- L'*algoritmo di Karatsuba* riduce la complessità a $O(n^{\log_2 3}) = O(n^{1.585})$, dimostrando che la moltiplicazione può essere effettuata più velocemente rispetto alla procedura elementare.
- L'*algoritmo migliore conosciuto* per la moltiplicazione ha una complessità inferiore a $o(n^{1+\epsilon})$ per ogni $\epsilon > 0$, dove la notazione $o(\cdot)$ indica una crescita asintotica più lenta rispetto alla funzione dentro le parentesi.

Questo esempio illustra come l'analisi della complessità fornisca una comprensione profonda sia delle potenzialità sia dei limiti degli algoritmi nel risolvere problemi computazionali.

1.2.3 Trattabilità di un problema

La comprensione della differenza tra **crescita polinomiale** e **crescita esponenziale** delle risorse necessarie da un algoritmo è fondamentale per valutare la sua efficienza e praticabilità. Jack Edmonds, nel 1965, ha sottolineato l'importanza di questa distinzione, che rimane centrale nell'analisi degli algoritmi.

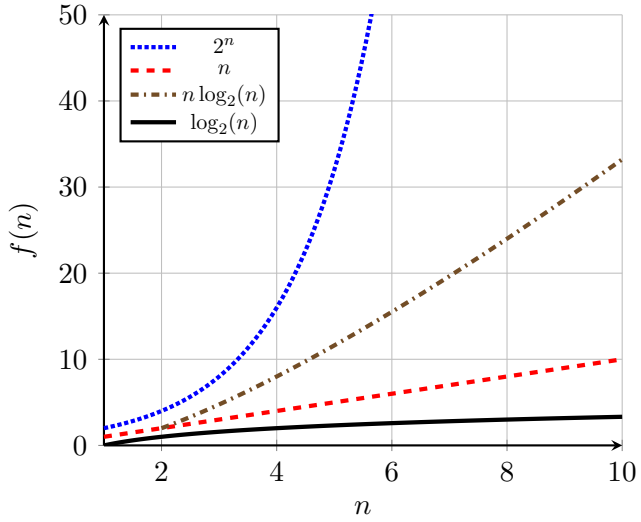
Crescita Polinomiale

La crescita polinomiale si verifica quando il requisito di risorse di un algoritmo è limitato da n^k per qualche costante k. In questo contesto, se l'input raddoppia da n a $2n$, il requisito di risorse aumenta di un fattore costante, da n^k a $2^k n^k$. Questo indica che la crescita delle risorse necessarie è gestibile e prevedibile al crescere delle dimensioni dell'input.

Crescita Esponenziale

Al contrario, la crescita esponenziale descrive una situazione in cui il requisito di risorse cresce proporzionalmente a c^n per qualche costante $c > 1$. Quando l'input raddoppia, il requisito di risorse si quadruplica, passando da c^n a $c^{2n} = (c^n)^2$. Questa rapida escalation rende gli algoritmi con crescita esponenziale impraticabili per input di grandi dimensioni.

Confronto tra crescita polinomiale ed esponenziale



Legge di Moore

La legge di Moore, formulata da Gordon Moore nel 1965, afferma che il numero di transistor in un microprocessore raddoppia approssimativamente ogni 18 mesi.

Ogni due anni la velocità dei computer raddoppia, ma questo non è sempre decisivo. Consideriamo due scenari:

Algoritmi Polinomiali: $O(n^k)$

Con un algoritmo polinomiale, il tempo di esecuzione cresce polinomialmente con la dimensione dell'input n. In un tempo T:

- Se oggi risolviamo istanze di dimensione n_T,
- tra due anni possiamo risolvere istanze di dimensione $n' = n_T 2^{1/k}$.

Questo dimostra che l'aumento di potenza computazionale ci permette di gestire istanze significativamente più grandi in modo più efficiente.

Algoritmi Esponenziali: $\Omega(2^n)$

Con un algoritmo esponenziale, il tempo di esecuzione cresce esponenzialmente con la dimensione dell'input n. In un tempo T:

- Se oggi risolviamo istanze di dimensione n_T,
- tra due anni possiamo risolvere istanze di dimensione $n' = n_T + 1$.

In questo caso, l'aumento di potenza computazionale ci consente di affrontare solo istanze leggermente più grandi, evidenziando i limiti degli algoritmi con crescita esponenziale.

Capitolo 2

NP completezza e co-NP

2.1 Classi di complessità

L'idea fondamentale per dimostrare che determinati problemi sono più difficili dei problemi risolvibili in tempo polinomiale è quella di "ritagliare" una classe di problemi che abbiano una determinata caratteristica, e dimostrare che un problema specifico è almeno tanto difficile quanto i problemi di quella classe, **riducendo** il problema specifico ad un problema della classe.

2.1.1 Tipologie di problemi

- **Problemi decisionali**: problemi che ammettono risposta sì/no. Un problema di decisione è quindi un insieme di istanze, e la risposta è sì se l'istanza appartiene all'insieme, no altrimenti.

  Un problema di decisione è il seguente: dato un grafo $\mathcal{G}$ dire se esiste un cammino euleriano in $\mathcal{G}$.

- **Problemi di ricerca**: problemi di ricerca ammettono una soluzione che può essere trovata in tempo polinomiale.

  Un problema di ricerca è il seguente: dato un grafo $\mathcal{G}$ trovare un cammino euleriano in $\mathcal{G}$.

- **Problemi di ottimizzazione**: problemi di ottimizzazione ammettono una soluzione che può essere trovata in tempo polinomiale.

  Un problema di ottimizzazione è il seguente: dato un grafo $\mathcal{G}$ trovare il cammino euleriano più lungo in $\mathcal{G}$.

Nello studio della complessità considereremo solo problemi decisionali, in quanto tutti i problemi di ricerca e di ottimizzazione possono essere ricondotti a problemi di decisione. Ciò significa che se siamo in grado di dire che un determinato problema di decisione è difficile, allora possiamo dire che anche il problema di ricerca e quello di ottimizzazione sono difficili,

perché se fossimo in grado di risolvere un problema di ricerca o di ottimizzazione, potremmo risolvere anche il problema di decisione in tempo polinomiale.

2.1.2 Formalizzazione di un problema computazionale

In relazione a ciò che è stato detto nella sezione 1.2, un problema computazionale è un insieme infinito di istanze e la loro relazione con la soluzione associata. Matematicamente, possiamo definire un problema computazionale attraverso:

- $\mathbb{A}$ denota il problema computazionale sotto esame.
- $\mathscr{I}(\mathbb{A})$ rappresenta lo spazio delle istanze, ovvero il dominio dei possibili quesiti.
- $\texttt{sol}(\mathbb{A})$ esprime la relazione che associa a ciascuna istanza la sua soluzione o insieme di soluzioni.

Esempio 2.1.1 Consideriamo, ad esempio, il problema del ciclo hamiltoniano. In questo contesto:

- $\mathbb{A}$ corrisponde alla questione di determinare l'esistenza di un ciclo hamiltoniano.
- $\mathscr{I}(\mathbb{A})$ comprende tutti i possibili grafi $\mathcal{G}$ sui quali indaghiamo.
- $\texttt{sol}(\mathbb{A})$ identifica i percorsi che visitano ogni nodo del grafo esattamente una volta, se esistenti.

Pertanto, la relazione $\mathbb{A}$ si configura come una **connessione** fra lo spazio delle istanze e le corrispondenti soluzioni. Matematicamente, ciò si traduce nella seguente inclusione:

$$\mathbb{A} \subseteq \mathscr{I}(\mathbb{A}) \times \texttt{sol}(\mathbb{A})$$

indicando con $(\mathcal{G}, \mathcal{P})$ l'insieme delle coppie dove $\mathcal{P}$ rappresenta un valido ciclo hamiltoniano nel grafo $\mathcal{G}$.

Per ciascuna istanza x appartenente allo spazio dei problemi computazionali, $\texttt{sol}(x)$ denota l'insieme delle soluzioni y tali che $(x, y) \in \mathbb{A}$, ovvero:

$$\texttt{sol}(x) = \{y \mid (x, y) \in \mathbb{A}\}$$

Problemi di Decisione

Nei problemi di decisione, lo spazio delle soluzioni è binario, $\texttt{sol}(\mathbb{A}) = \{\texttt{yes}, \texttt{no}\}$. Pertanto, A associa a ciascuna istanza x una delle due possibili risposte, determinando se l'istanza soddisfa la proprietà esaminata:

$$A : \mathscr{I}(\mathbb{A}) \to \{\texttt{yes}, \texttt{no}\}$$

Problemi di Ricerca

Per i problemi di ricerca $\mathbb{A}^S$, il problema di decisione associato $\mathbb{A}^D$ verifica l'esistenza di almeno una soluzione per l'istanza x:

$$\mathbb{A}^D(x) = \begin{cases} \texttt{yes} & \text{se esiste almeno una soluzione,} \\ \texttt{no} & \text{altrimenti} \end{cases}$$

2.1.3 La Classe P

Nella teoria della complessità computazionale, la capacità di risolvere un problema computazionale va oltre la semplice identificazione di una soluzione per un'istanza specifica. Ciò richiede l'impiego di un algoritmo deterministico A che, per ogni istanza x nell'insieme $\mathscr{I}(\mathbb{A})$, produce una soluzione y valida, ovvero $y \in \texttt{sol}(x)$:

$$A(x) = y \quad \text{con} \quad y \in \texttt{sol}(x)$$

Il *tempo di esecuzione* $T_A(x)$ dell'algoritmo A su un'istanza x è fondamentale, poiché riflette l'efficienza con cui l'algoritmo raggiunge la soluzione.

Classe P

Un algoritmo A è definito *polinomiale* (poly-time) quando il suo tempo di esecuzione, per ogni istanza $x \in \mathscr{I}(\mathbb{A})$, è limitato superiormente da un polinomio nella dimensione di x:

$$T_A(x) = O(|x|^c) \quad \text{per una certa costante } c \in \mathbb{N}.$$

Un problema computazionale viene considerato *fuori dalla classe P* qualora non esista, per nessuna costante c, un algoritmo capace di risolvere tutte le sue istanze in tempo polinomiale $O(n^c)$.

2.1.4 La Classe EXP

Classe EXP

La classe EXP raggruppa i problemi di decisione risolvibili da un algoritmo deterministico in tempo esponenziale rispetto alla dimensione dell'input. Un problema $\mathbb{A}$ appartiene a EXP se esiste un algoritmo A per cui, data una costante c_A, per ogni istanza $x \in \mathscr{I}(\mathbb{A})$ si ha che:

$$T_A(x) = O(2^{|x|^{c_A}})$$

In altre parole, il tempo di esecuzione di A è limitato superiormente da una funzione esponenziale della dimensione dell'input elevata a una costante. Questa classe include quindi problemi per i quali la risoluzione richiede una quantità di tempo che cresce esponenzialmente con l'aumentare della dimensione dell'input.

Da qui segue che la classe EXP è una generalizzazione della classe P, poiché ogni problema risolvibile in tempo polinomiale è anche risolvibile in tempo esponenziale. In altre parole, P è un sottoinsieme di EXP.

$$\texttt{P} \subseteq \texttt{EXP}$$

Perché se un problema $\mathcal{A} \in \texttt{P}$, allora esiste un algoritmo A, per cui, data una costante c_A, per ogni istanza $x \in \mathscr{I}(\mathbb{A})$ si ha che:

$$T_A(x) = O(|x|^{c_A})$$

Poiché, ogni polinomio è anche una funzione esponenziale, si ha che:

$$T_A(x) = O(2^{|x|^{c_A}})$$

Quindi $\mathbb{A} \in \texttt{EXP}$.

È importante notare che un problema è nella classe EXP, non se gli unici algoritmi conosciuti per risolverlo sono esponenziali, ma se esiste almeno un algoritmo deterministico che lo risolve in tempo esponenziale.

Esempio 2.1.2 Consideriamo il problema del cammino euleriano. Un cammino euleriano è un cammino che attraversa ogni arco del grafo esattamente una volta. In input si ha un grafo $\mathcal{G}$ e si vuole sapere se esiste un cammino euleriano in $\mathcal{G}$, quindi l'output è `yes` se esiste un cammino euleriano e `no` altrimenti. Questo problema è nella classe P, infatti esiste un algoritmo polinomiale per risolverlo, controllando se il grafo è connesso e se ogni nodo ha grado pari.

Esempio 2.1.3 Consideriamo il problema del commesso viaggiatore. In input si ha un grafo completo pesato $\mathcal{G}$ e si vuole sapere se esiste un ciclo hamiltoniano di peso minimo in $\mathcal{G}$, quindi l'output è `yes` se esiste un ciclo hamiltoniano e `no` altrimenti. Non sappiamo se esiste un algoritmo polinomiale per risolvere questo problema, quindi non sappiamo se è nella classe P. Sappiamo che per ogni istanza, se la soluzione è `yes` possiamo certificarla in tempo polinomiale.

Prova Semplice

Una **prova semplice** è un certificato di lunghezza polinomiale che può essere verificato in tempo polinomiale.

Cerchiamo di caratterizzare i problemi che hanno una prova semplice.

2.1.5 La Classe NP

Classe NP

Un problema $\mathbb{A}$ appartiene alla classe NP se esiste un algoritmo deterministico A (*verificatore*) e una funzione polinomiale p tale che, per ogni istanza $x \in \mathscr{I}(\mathbb{A})$, esiste un certificato y di lunghezza polinomiale rispetto alla dimensione di x tale che:

$$B(x,y) = \begin{cases} \texttt{yes} & \text{se } x \in \texttt{sol}(\mathbb{A}), \\ \texttt{no} & \text{altrimenti} \end{cases}$$

Inoltre, esiste una costante c_A tale che il tempo di esecuzione di A è limitato superiormente da una funzione polinomiale della dimensione di x:

$$T_A(x,y) = O(p(|x|^{c_A}))$$

In notazione formale, si ha che:

$$\begin{aligned} \texttt{NP} = \{\mathbb{A} \mid \exists A, c_A, \forall x \in \mathscr{I}(\mathbb{A}) : \mathbb{A}(x) = \texttt{yes} \Rightarrow \\ \exists y, |y| = O((|x|^{c_A})), \text{ t.c. } A(x,y) = \texttt{yes} \wedge T_A(x,y) = O((|x|^{c_A}))\} \end{aligned}$$

È importante notare che il tempo di esecuzione del verificatore A è polinomiale rispetto alla dimensione dell'input x e del certificato y. Ciò significa che, se esiste un certificato y tale che

$A(x, y) = \texttt{yes}$, allora esiste un algoritmo che può verificare la soluzione in tempo polinomiale. Da questo segue che il certificato deve essere di lunghezza polinomiale rispetto alla dimensione dell'input x.

$$|y| = O((|x|^{c_A}))$$

Teorema 2.1.1 La classe P è un sottoinsieme della classe NP.

$$\texttt{P} \subseteq \texttt{NP}.$$

Dimostrazione. Consideriamo il problema $\mathbb{A} \in \texttt{P}$. Allora esiste un algoritmo deterministico A che risolve per ogni istanza $x \in \mathscr{I}(\mathbb{A})$ in tempo polinomiale:

$$T_A(x) = O(|x|^{c_A}).$$

Definiamo un verificatore B per $\mathbb{A}$. Per ogni istanza $x \in \mathscr{I}(\mathbb{A})$ e y, $B(x, y) = A(x)$. Quindi se $\mathbb{A}(x) = \texttt{yes}$, allora per ogni $y \in \{0,1\}^*$, $B(x, y) = \texttt{yes}$ e se $\mathbb{A}(x) = \texttt{no}$, allora per ogni $y \in \{0,1\}^*$, $B(x, y) = \texttt{no}$. □

Teorema 2.1.2 La classe NP è sottoinsieme della classe EXP.

$$\texttt{NP} \subseteq \texttt{EXP}.$$

Dimostrazione. Consideriamo un problema $\mathbb{A} \in \texttt{NP}$. Allora esiste un algoritmo deterministico B e una costante c_B tale che per ogni istanza $x \in \mathscr{I}(\mathbb{A})$, esiste un certificato y di lunghezza polinomiale rispetto alla dimensione di x tale che:

$$B(x, y) = \begin{cases} \texttt{yes} & \text{se } x \in \texttt{sol}(\mathbb{A}), \\ \texttt{no} & \text{altrimenti.} \end{cases}$$

Quindi il tempo di esecuzione di B è limitato superiormente da una funzione polinomiale della dimensione di x:

$$T_B(x, y) = O((|x|^{c_B}))$$

Definiamo un algoritmo deterministico $A^{\texttt{EXP}}$ che per ogni $x \in \mathscr{I}(\mathbb{A})$ cicla su tutti i possibili certificati y della taglia ammissibile per i certificati e usa $B(x, y)$ per verificare se x è una soluzione di $\mathbb{A}$:

Algorithm 2: Algoritmo $A^{\texttt{EXP}}$

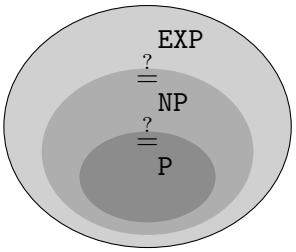
Input: Un'istanza $x \in \mathscr{I}(\mathbb{A})$
Output: yes se $x \in \texttt{sol}(\mathbb{A})$, no altrimenti

```
1 foreach y ∈ {0,1}^{|x|^{c_B}} do
2     if B(x, y) = yes then
3         return yes
4     return no
```

Sappiamo che $A^{\texttt{EXP}}(x) = \texttt{yes}$ se e solo se esiste un certificato $y \in \{0,1\}^{|x|^{c_B}}$ tale che $B(x,y) = \texttt{yes}$, ovvero se $\mathbb{A} = \texttt{yes}$. Il tempo di esecuzione di $A^{\texttt{EXP}}$ è limitato superiormente da una funzione esponenziale della dimensione di x:

$$T_{A^{\texttt{EXP}}}(x) = O(2^{|x|^{c_B}})$$

Quindi $\mathbb{A} \in \texttt{EXP}$, ovvero $\texttt{NP} \subseteq \texttt{EXP}$. □



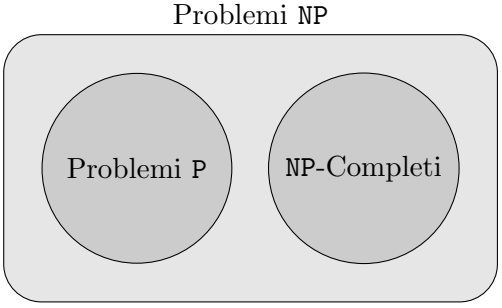
Non sappiamo se le inclusioni sono strette, ma sappiamo che P è incluso strettamente in EXP.

Congettura P ≠ NP

Se $\texttt{P} \neq \texttt{NP}$, quindi $\texttt{P} \subset \texttt{NP}$, allora esistono problemi che possono essere verificati in tempo polinomiale, ma non possono essere risolti in tempo polinomiale.

$$\texttt{P} \subset \texttt{NP}$$
$$\texttt{P} \neq \texttt{NP}$$

Il problema è che non esiste una dimostrazione formale che $\texttt{P} \neq \texttt{NP}$. Siamo convinti di ciò perché crediamo che alcuni problemi NP siano più difficili di altri e se siamo in grado di risolvere tali problemi allora siamo in grado di risolvere tutti i problemi NP. Questa situazione porta alla distinzione tra i problemi NP cosiddetti "difficili" o NP-hard e i problemi NP-completi, per i quali una soluzione efficiente implicherebbe la possibilità di risolvere efficientemente ogni problema nell'insieme NP.



2.2 Riduzione polinomiale tra problemi

La riduzione polinomiale ci permette di mettere in relazione i problemi tra loro, permettendoci di dimostrare che un problema è almeno tanto difficile quanto un altro problema.

Riduzione polinomiale (Karp)

Siano $\mathbb{A}$ e $\mathbb{B}$ due problemi decisionali. Diciamo che $\mathbb{A}$ si riduce polinomialmente a $\mathbb{B}$, e scriviamo $\mathbb{A} \leq_p \mathbb{B}$, se esiste una funzione calcolabile in tempo polinomiale $f : \{0,1\}^* \to \{0,1\}^*$ tale che per ogni $x \in \{0,1\}^*$:

$$\forall x \in \mathbb{A} \qquad \mathbb{A}(x) = \texttt{yes} \iff \mathbb{B}(f(x)) = \texttt{yes}$$

2.2.1 Colorazione di un grafo

Consideriamo il problema della colorazione di un grafo. Dato un grafo $\mathcal{G} = (V, E)$, dove V è l'insieme dei vertici e E è l'insieme degli archi, il problema consiste nel determinare se è possibile colorare i vertici di $\mathcal{G}$ con k colori in modo che due vertici adiacenti non abbiano lo stesso colore. Solitamente tale problema si associa al problema di allocazione di risorse, dove i vertici rappresentano le risorse e gli archi rappresentano le relazioni di dipendenza tra le risorse.

La coloriazione è **propria** se per ogni arco $(u, v) \in E$ si ha che $c(u) \neq c(v)$, ovvero due vertici adiacenti non possono avere lo stesso colore.

Il problema della `k-colorazione` è un problema `NP` perché possiamo verificare in tempo polinomiale se una colorazione è propria. Quindi una **prova semplice** che dimostra che il problema della `k-colorazione` è `NP` è la seguente:

- Dato un grafo $\mathcal{G}$ e una colorazione $c : V \to \{1, \ldots, k\}$, possiamo verificare in tempo polinomiale se c è una colorazione propria, e tale colorazione si verifica in tempo polinomiale $O(|V|+|E|)$. Immaginiamo che la colorazione sia una funzione che associa ad ogni vertice un numero intero che rappresenta il colore del vertice. Se c è una colorazione propria, allora per ogni arco $(u, v) \in E$ si ha che $c(u) \neq c(v)$.

Ci chiediamo ora se esiste un algoritmo polinomiale per la colorazione di un grafo. La risposta è legata al parametro k che rappresenta la diversa tipologia di problemi che possiamo avere. Proviamo a vedere per i diversi valori di k se esiste un algoritmo polinomiale per la colorazione di un grafo:

- $\mathbf{k = 1}$: se $k = 1$ allora il problema è banale, perché tutti i vertici devono avere lo stesso colore, per farlo basta che il grafo sia massimamente disconnesso, ovvero non ci siano archi tra i vertici. Questo problema è risolvibile in tempo polinomiale.
- $\mathbf{k = 2}$: se $k = 2$ allora il problema è equivalente al problema della `bipartizione` di un grafo, ovvero se è possibile dividere i vertici di un grafo in due insiemi V_1 e V_2 tali che non esistano archi tra vertici dello stesso insieme. Per verificare che un grafo è bipartito

possiamo verificare che non esistano cicli dispari nel grafo. Questo problema è risolvibile in tempo polinomiale, infatti possiamo utilizzare l'algoritmo di BFS, che ha complessità $O(|V| + |E|)$.

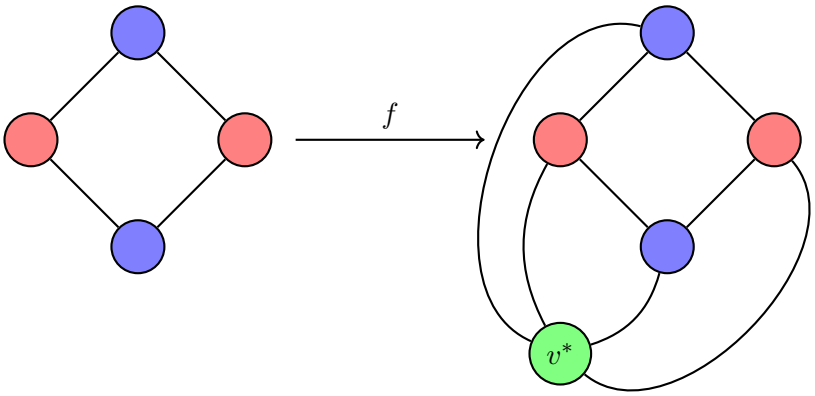
Per $k \geq 3$ non sappiamo se esiste un algoritmo polinomiale per la colorazione di un grafo.

k-colorazione $\leq_p$ (k + 1)-colorazione

Se (k+1)-colorazione è P, allora k-colorazione è P, ovvero (k+1)-colorazione non è più facile di k-colorazione.

$$\texttt{(k+1)-col} \in \texttt{P} \implies \texttt{k-col} \in \texttt{P} \equiv \texttt{k-col} \notin \texttt{P} \implies \texttt{(k+1)-col} \notin \texttt{P}$$

Dimostrazione. Forniamo la funzione calcolabile in tempo polinomiale f, tale che per ogni grafo $\mathcal{G}$, se $\mathcal{G}$ è un grafo k-colorabile, se e solo se $f(\mathcal{G})$ è un grafo (k+1)-colorabile.



L'idea è quella di aggiungere un vertice v^* al grafo $\mathcal{G}$, e collegare v^* a tutti i vertici di $\mathcal{G}$, in modo tale che v^* abbia un colore diverso da tutti gli altri vertici. Per costruire la funzione f possiamo procedere come segue:

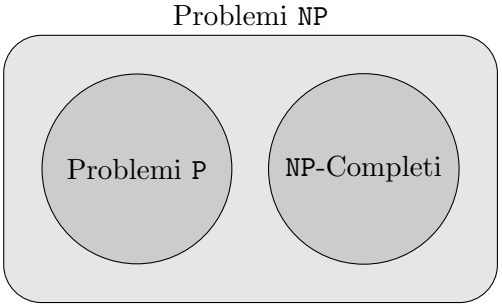
- Dato un grafo $\mathcal{G}$, aggiungiamo un vertice v^* al grafo $\mathcal{G}$.
- Colleghiamo v^* a tutti i vertici di $\mathcal{G}$.
- Assegnamo a v^* un colore diverso da tutti gli altri vertici, ovvero il colore $k + 1$.

Se $\mathcal{G}$ è k-colorabile, allora esiste una colorazione c tale che per ogni arco $(u, v) \in E$ si ha $c(u) \neq c(v)$. Se aggiungiamo un vertice v^* al grafo $\mathcal{G}$, e lo colleghiamo a tutti i vertici di $\mathcal{G}$, allora v^* deve avere un colore diverso da tutti gli altri vertici, ovvero il colore $k + 1$. Quindi, se $\mathcal{G}$ è k-colorabile, allora $\mathcal{G}'$ è (k+1)-colorabile.

Se $\mathcal{G}'$ è (k+1)-colorabile senza perdita di generalità diciamo che il colore di v^* è $k+1$. Poiché per ogni v, $c(v) \neq k+1$, $c(v) \in \{1, \ldots, k\}$ e quindi $c(v) \neq c(v^*)$, ovvero tutti i vertici v hanno un colore diverso da v^*. Quindi, se $\mathcal{G}'$ è (k+1)-colorabile, allora $\mathcal{G}$ è k-colorabile. □

2.3 Problemi NP-completi

I problemi NP-completi sono una classe di problemi che sono problemi in NP e se uno di questi problemi è in P, allora tutti i problemi in NP sono in P, ovvero P = NP. Se P ≠ NP, allora nessun problema NP-completo è in P.



Supponiamo che esista un problema $\mathbb{B} \in$ NP tale che per ogni problema $\mathbb{A}$ in NP, esiste una riduzione polinomiale da $\mathbb{A}$ a $\mathbb{B}$. Quindi se $\mathbb{B}$ è in P, allora $\mathbb{A}$ è in P allora per ogni problema $\mathbb{A}$ in NP esiste una riduzione polinomiale da $\mathbb{A}$ a $\mathbb{B}$ e quindi $\mathbb{A}$ è in P. Per contrapposizione, se P ≠ NP, allora $\mathbb{B}$ non è in P e quindi $\mathbb{A}$ non è in P.

Diciamo che $\mathbb{B}$ è NP-completo se $\mathbb{B}$ è in NP e per ogni problema $\mathbb{A}$ in NP, $\mathbb{A}$ è riducibile a $\mathbb{B}$ in tempo polinomiale, ovvero $\mathbb{A} \leq_p \mathbb{B}$.

> Un problema $\mathbb{B}$ è NP-completo se $\mathbb{B}$ è in NP e $\mathbb{B}$ è NP-hard.

Un problema NP-completo deve essere un problema rappresentativo di tutti i problemi in NP, deve essere quindi codificabile in modo tale che tutti i problemi in NP siano riducibili a esso.

2.3.1 Soddisfacibilità booleana

Il problema della soddisfacibilità booleana è il problema di determinare se una formula booleana è soddisfacibile, ovvero se esiste un assegnamento di valori di verità alle variabili della formula che rende la formula vera. L'input al problema è una formula booleana in forma normale congiuntiva (CNF) ϕ e l'output è yes se e solo se esiste un assegnamento di valori di verità alle variabili della formula che rende la formula vera.

Una **formula booleana in forma normale congiuntiva (CNF)** è definita come una congiunzione di clausole. Formalmente, sia $\phi(x_1, \ldots, x_n)$ una formula booleana che dipende dalle variabili booleane $x_1, \ldots, x_n$. Questa formula può essere espressa come segue:

$$\phi(x_1, \ldots, x_n) = C^{(1)} \wedge C^{(2)} \wedge \cdots \wedge C^{(m)}$$

dove m è il numero totale di clausole nella formula, e ogni clausola $C^{(i)}$ è definita come una disgiunzione di uno o più letterali:

$$C^{(i)} = l_1^{(i)} \vee l_2^{(i)} \vee \cdots \vee l_{k_i}^{(i)}$$

In questa definizione, k_i rappresenta il numero di letterali nella i-esima clausola $C^{(i)}$, e ogni letterale $l_j^{(i)}$ può essere una variabile booleana x_t o la sua negazione $\bar{x_t}$, per qualche $t \in \{1, \ldots, n\}$. In altre parole, ogni letterale è definito come:

$$l_j^{(i)} \in \{x_1, \bar{x_1}, x_2, \bar{x_2}, \ldots, x_n, \bar{x_n}\}$$

Questa struttura garantisce che la formula ϕ sia una congiunzione di clausole, dove ogni clausola è una disgiunzione di letterali, e ogni letterale è una variabile booleana o la sua negazione.

Una formula `CNF` è quindi una funzione di n variabili booleane che restituisce un valore booleano. Un assegnamento a di valori di verità alle variabili della formula è una funzione $a : \{a_1, \ldots, a_n\} \to \{0, 1\}$ che assegna un valore di verità a ciascuna variabile della formula. Un assegnamento a è detto **soddisfacente** per la formula ϕ se e solo se $\phi(a_1, a_2, \ldots, a_n) = \texttt{true}$.

Esempio

Consideriamo la seguente formula booleana in forma normale congiuntiva:

$$\phi(x_1, x_2, x_3, x_4) = (x_1 \vee x_2 \vee x_3 \vee \bar{x_4}) \wedge (\bar{x_1} \vee \bar{x_2}) \wedge (x_1 \vee x_2 \vee x_3 \vee \bar{x_4})$$

Un assegnamento $a : \{a_1, a_2, a_3, a_4\} \to \{0, 1\}$ può essere il seguente:

$$a = \{\texttt{true}, \texttt{false}, \texttt{true}, \texttt{true}\}$$

Risolvendo la formula con l'assegnamento a otteniamo:

$$\phi(a_1, a_2, a_3, a_4) = \texttt{true} \wedge \texttt{true} \wedge \texttt{true} = \texttt{true}$$

Quindi l'assegnamento a soddisfa la formula ϕ.

2.3.2 NP-completezza di SAT

Prima di dimostrare che `SAT` è un problema NP-completo, dimostriamo che `SAT` si riduce a `k-col`.

Teorema 2.3.1 `K-col` $\leq_p$ `SAT`