Università degli Studi di Verona

Complessità

Riassunto dei principali argomenti

Matteo Danzi, Davide Bianchi

29 marzo 2018

Indice

1 Introduzione

1.1 Cos'è la complessità computazionale

Nella teoria della complessità ci si pone la seguente domanda:

> *Come scalano le risorse necessarie per risolvere un problema all'aumentare delle dimensioni del problema?*

La teoria della *complessità computazionale* è una parte dell'informatica teorica che si occupa principalmente di classificare i problemi in base alla quantità di *risorse computazionali* (come il tempo di calcolo e lo spazio di memoria) che essi richiedono per essere risolti. Tale quantità è detta anche *costo computazionale* del problema.

1.2 Problemi *facili* e *difficili*

Vediamo quattro esempi di problemi che classificheremo come facili o difficili:

1. (**Eulerian Cycle**) Esiste un modo per attraversare ogni arco di un grafo una e una sola volta?

    - Il problema si può vedere anche nella forma più piccola del problema dei *sette ponti di Königsberg*:
      A Königsberg ci sono 7 ponti, esiste un percorso che attraversa tutti i ponti una e una sola volta per poi tornare al punto di partenza?
      Se avessi n ponti e su ogni riva partissero 2 ponti avrei 2^n possibili percorsi.
    - La **soluzione di Eulero** dice che un grafo connesso non orientato ha un percorso che parte e inizia esattamente nello stesso vertice e attraversa ogni arco esattamente una volta se e solo se ogni vertice ha grado dispari (grado = numero di archi uscenti).
      Se ci sono esattamente due vertici v e u, di grado dispari, allora esiste un percorso che parte da u e attraversa ogni arco esattamente una volta e finisce in v.
    - Seguendo quindi la soluzione di Eulero, *quanto costa decidere* se un grafo G ha un tour Euleriano?

      ```
      odd-vertex-num = 0;
      For each vertex v of G
          if (deg(v) is odd)
              increment odd_vertex-num
      If(odd-vertex-num is neither 0 nor 2)
          output no Eulerian tour
      output Eulerian
      ```

      Questo algoritmo ha complessità: $O(|E| + |V|)$
      Il costo e l'algoritmo sono gli stessi se vogliamo *provare* che G *non* ha un tour Euleriano.

2. (**Hamiltonian Cycle**) Esiste un modo per attraversare ogni nodo di un grafo una e una sola volta?

   Esistono diverse soluzioni:

    - Provo tutte le possibilità ogni volta, costo: $O(2^n)$
    - Provo tutte le possibili permutazioni, costo: $O(n!)$
    - La soluzione migliore ad oggi è: $O(1.657^n)$

   Alla domanda: *Quanto costa decidere se un grafo ha un tour hamiltoniano?* Non sappiamo rispondere. Non sappiamo dire se il problema ha una soluzione non esponenziale. Per quanto ne sappiamo meglio di $O(1.657^n)$ non sappiamo fare.

   Non sappiamo nemmeno dire se Hamiltonian Cycle è più difficile di Eulerian Cycle.

3. N è un numero primo?

   Il migliore algoritmo conosciuto per decidere se N è un numero primo impiega $O((\log N)^{6+\epsilon})$

4. Quali sono i fattori primi di un numero?

   Ad oggi non conosciamo una procedure per fattorizzare un numero molto grande nei suoi divisori, che non sia provare tutte le possibilità.

1.3 Risolvere vs Verificare

La seguente tabella riassume in modo generico quanto detto nella sezione precedente riguardo alla difficoltà di risolvere problemi e verificare tali problemi su un istanza.

Tabella 1: Risolvere vs Verificare

Problema	**Risolvere**	**Verificare**
Eulerian Cycle	*facile*	*facile*
Hamiltonian Cycle	*difficile?*	*facile*
N è primo?	*facile*	*facile*
N ha un numero piccolo di fattori?	*difficile?*	*facile*

2 Problema computazionale

Un problema computazionale è una semplice relazione p che mappa l'insieme *infinito* di possibili input (domande o istanze) con un insieme *finito* (non vuoto) di output, cioè di risposte o soluzioni alle istanze.

$$\mathrm{p}: \text{ istanze infinite } \mapsto \text{ soluzioni finite alle istanze}$$

Un problema computazionale non è una singola domanda, ma è una **famiglia di domande**:

- Una domanda per ogni possibile istanza
- Ogni domanda è dello stesso tipo (appartiene alla stessa classe)

Esempio 2.0.1. Il seguente esempio è un problema computazionale:

- Input: Qualsiasi grafo G
- Domanda: Il grafo G contiene un ciclo Euleriano?

Esempio 2.0.2. Il seguente esempio *non* è un problema computazionale:

- Domanda: È vero che il bianco vince sempre a scacchi, sotto l'ipotesi della giocata perfetta?

Non è un problema computazionale perché non ho un insieme infinito di possibili partite in input.

2.1 Risolvere un problema computazionale

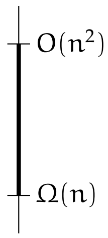
Risolvere un problema computazionale significa trovare un **algoritmo**, cioè una procedura che risolve il problema matematico in un numero finito di passi (di computazione elementare), che solitamente include la ripetizione di un operazione. È un procedimento deterministico che mappa l'input sull'output.

> Un algoritmo è una procedura *finita*, *definita*, *efficace* e con un input e un output.
>
> *Donald Knuth – The Art of Computer Programming*

2.2 Complessità di un problema computazionale

Misura della complessità. Come misuro la complessità di un problema computazionale? Come faccio a dire quanto è facile rispetto ad altri problemi?

- Do un **upper bound**: trovo un algoritmo qualsiasi che risolve il problema in modo da calcolare qual è il suo costo.
- Do un **lower bound**: trovo la minima quantità di risorse che ogni algoritmo utilizza per risolvere il problema. Tutti gli algoritmi sono *al minimo* complessi come il limite inferiore che abbiamo stabilito. Nessuno può fare di meglio.



2.3 Trattabilità di un problema.

La crescita della complessità di un problema è riducibile a 2 categorie fondamentali.

Crescita polinomiale. Un problema ha crescita polinomiale quando le risorse necessarie alla sua risoluzione sono limitate ad n^k, per qualche k. Se la taglia del problema aumenta, la sua complessità aumenta di un qualche fattore costante. Infatti, se la taglia dell'input va da n a $2n$ allora la complessità del problema si modifica in $(2n)^k = 2^k n^k$, ovvero aumenta di un fattore 2^k (costante). Raggruppiamo nella classe **P** i problemi di questo tipo.

Crescita esponenziale. Un problema ha crescita esponenziale la necessità di risorse necessarie alla sua risoluzione è proporzionale a c^n, per qualche costante $c > 1$. Se la taglia dell'input va da n a $2n$ c^n allora la richiesta di risorse si diventa $c^{2n} = c^n * c^n$, aumentando quindi di un fattore che cresce con l'aumentare di n. Raggruppiamo nella classe **Exp** i problemi di questo tipo.

3 Le classi di problemi computazionali

Notazione e idee di base. Formalmente definiamo un problema come un elemento $\mathbb{A}$ di una relazione

$$\mathcal{R} \subseteq \mathcal{I}(\mathbb{A}) \times \mathtt{Sol}$$

dove:

- $\mathcal{I}(\mathbb{A})$ è l'insieme delle istanze del problema $\mathbb{A}$
- $\mathtt{Sol}$ è l'insieme delle soluzioni delle istanze di $\mathbb{A}$

Si può quindi dire che

$$\forall x \in \mathcal{I}(\mathbb{A}),\ \mathtt{Sol}(x) = \{\text{Soluzioni di } x\}$$

Non è restrittivo restringersi ai **problemi di tipo decisionale**, ovvero quei problemi che hanno come soluzione una risposta del tipo *si* o *no*, quindi i problemi del tipo

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

L'algoritmo $\mathcal{A}$ per un problema $\mathbb{A}$ è un algoritmo che dato il problema, $\forall x \in \mathcal{I}(\mathbb{A})$, $\mathcal{A}(x) = \mathbb{A}(x)$. Inoltre, dato un algoritmo $\mathcal{A}$, definiamo $T_{\mathcal{A}}(|x|)$ la sua **complessità**, cioè il *tempo che impiega* $\mathcal{A}$ sull'istanza di taglia $|x|$. Notare che $|x|$ è la taglia dell'istanza x.

3.1 Classe P

Intuitivamente la classe **P** è definita come la classe di problemi di **complessità polinomiale**. Introduciamo qui la definizione formale.

Definizione 3.1.1 (Classe P). Definiamo la classe di problemi **P** come l'insieme dei problemi di complessità polinomiale, ovvero

$$\mathbf{P} = \left\{ \mathbb{A} \mid \exists \mathcal{A} \text{ t.c. } \exists c \text{ costante e } \forall x \in \mathcal{I}(\mathbb{A}),\ \ \mathcal{A}(x) = \mathbb{A}(x) \ \text{ e } \ T_{\mathcal{A}}(|x|) \leqslant |x|^c \right\}$$

Esempio 3.1.1 (Eulerian Cycle). Un semplice esempio di problema appartenente alla classe **P** è il problema del tour euleriano. Per questo problema infatti abbiamo che è un problema computazionale di decisione:

- Input: grafo G
- Output: $yes \Leftrightarrow \exists$ Eulerian Cycle in G.

Come abbiamo già visto quindi:

$$\exists \mathcal{A} \text{ t.c. } T_{\mathcal{A}}(|G|) = O(|E| + |V|) = O(|G|)$$

Eulerian Cycle $\in$ **P** perché $\exists \mathcal{A}$ che impiega un tempo che è nell'ordine della taglia di G, in particolare $\exists c$ costante dove $c = 1$.

Esempio 3.1.2 (Hamiltonian Cycle). Ci chiediamo allora se anche Hamiltonian Cycle $\in$ **P** ? La risposta è che non lo sappiamo dire. Quello che sappiamo per questo problema è che:

$$\exists \mathcal{A} \text{ t.c. } T_{\mathcal{A}}(|G|) = O(a^{|G|})$$

dove a è costante.

3.2 Classe Exp

Dal momento che non sappiamo se alcuni problemi stiano oppure no nella classe **P** (dal momento che non si conosce un algoritmo che li risolva in tempo polinomiale), si definisce la classe **Exp**, che racchiude tutte le istanze di questa tipologia di problemi di **complessità esponenziale**.

Definizione 3.2.1 (Classe Exp). Definiamo la classe di problemi **Exp** come la classe di problemi di complessità esponenziale, ovvero

$$\mathbf{Exp} = \left\{ \mathbb{A} \mid \exists \mathcal{A} \text{ t.c. } \forall x \in \mathcal{I}(\mathbb{A}),\ \mathcal{A}(x) = \mathbb{A}(x) \ \text{ e } \ T_{\mathcal{A}}(|x|) \leqslant 2^{|x|^c} \right\}$$

Esempio 3.2.1 (Hamiltonian Cycle). Ci chiediamo se Hamiltonian Cycle $\in$ **Exp** ? Se prendiamo l'algoritmo che prova tutte le combinazioni di archi cioè $\binom{|E|}{n}$ per vedere se formano un ciclo hamiltoniano. La complessità di quest'algoritmo è al massimo $2^{|E|^2}$.

Se invece prendiamo l'algoritmo che considera tutte le possibili permutazioni dei vertici del grafo abbiamo che la complessità è $n!$. Quindi il problema Hamiltonian Cycle $\notin$ **Exp**

Relazione tra P ed Exp. La domanda che sorge spontanea è **P** $\subseteq$ **Exp** ?

La risposta alla domanda è banalmente **si**, in quanto, dato un algoritmo $\mathcal{B}$ con complessità $T_{\mathcal{B}}(|x|)$, possiamo dire che

$$T_{\mathcal{B}}(|x|) = O(|x|^c) = O(2^{|x|^c}) \Rightarrow \mathbb{A} \in \mathbf{Exp}$$

Problema K-Graph-Colouring. Analizziamo ora il problema della K-colorabilità di un grafo G:

- Input: G non orientato.
- Output: $yes \Leftrightarrow \exists$ colorazione *propria* dei vertici di G ovvero:

$$\exists f :\ v \mapsto \{0, \dots, k-1\} \quad \text{t.c.} \quad \forall (u, v) \in E(G) \quad f(u) \neq f(v)$$

(a) Grafo con colorazione *non* propria (b) Grafo con colorazione propria

Problema 2-Graph-Colouring. Consiste nel trovare se esiste una 2 colorazione del grafo dato in input in modo tale che un arco non si trovi tra due vertici dello stesso colore. Questo problema corrisponde a dire se il grafo è **bipartito**, cioè se *posso suddividere il grafo in due classi diverse*.
Per vedere se è bipartito si effettua una **BFS**, cioè una visita in ampiezza, e si controlla se c'è un ciclo dispari. Se c'è allora non è bipartito e quindi nemmeno 2-colorabile.

È 2-colorabile $\Leftrightarrow$ è Bipartito $\Leftrightarrow$ non contiene un ciclo dispari. La visita BFS ha una complessità pari a $O(|E| + |V|)$, perciò il problema è risolvibile in tempo polinomiale, perciò possiamo concludere che 2-Graph-Colouring $\in$ **P**.

Problema 3-Graph Colouring Il problema 3-Graph Colouring $\in$ **P**? Non sappiamo rispondere a questa domanda, poiché non sappiamo se esiste un algoritmo che lo svolga in tempo polinomiale. Il problema 3-Graph Colouring $\in$ **Exp**? Se consideriamo l'algoritmo che prova tutte le possibili colorazioni abbiamo che:

$$3^n \text{ sono le colorazioni dei vertici, dove } n = |V(G)|$$

Bisogna vedere se ci sono archi monocolore e quindi la complessità diventa:

$$O(3^n \cdot |E|) = O(3^{2n}) = O((2^{\log_2 3})^{2n}) = O(2^{2n \log_2 3})$$

Perciò possiamo concludere che il problema 3-Graph Colouring $\in$ **Exp**.

3.3 Classe Time(n)

Definizione 3.3.1 (Classe Time(n))**.** Definiamo la classe **Time**(**n**) come l'insieme dei problemi di complessità lineare, ovvero

$$\textbf{Time}(\textbf{n}) = \left\{ \mathbb{A} \mid \exists \mathcal{B} \text{ per } \mathbb{A} \quad \text{t.c.} \quad \forall x \in \mathcal{I}(\mathbb{A}) \quad T_{\mathcal{B}}(|x|) = O(n) = O(f(|x|)) \right\}$$

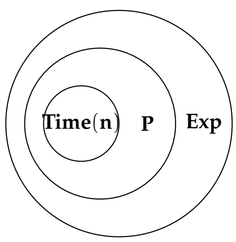
Teorema 3.3.1. $\forall \mathcal{B} \quad \text{t.c.} \quad \mathcal{B}(x) = \mathbb{A}(x) \quad T_{\mathcal{B}}(|x|) > |x|^c \quad \forall c$ *costante*

Teorema 3.3.2. *Qualsiasi **algoritmo di ordinamento** che usa confronti su n elementi ha tempo di esecuzione pari a*

$$\Omega(n \log n)$$

Possiamo dire quindi che:

- **Eulerian Cycle** $\in$ **Time**(**n**) perché esiste un problema che lo risolve in tempo lineare.
- **Sorting** $\notin$ **Time**(**n**) per il teorema 3.3.2.



Possiamo riassumere quindi che:

- **Eulerian Cycle** $\in$ **P**, **Eulerian Cycle** $\in$ **Time**$(\mathbf{n})$.
- **Hamiltonian Cycle** $\in$ **Exp**
- **Hamiltonian Cycle** $\in$ **P** ? non lo sappiamo dire.
- **K-Colouring** $\in$ **Exp**
- **K-Colouring** $\in$ **P**?
  per $k \geqslant 3$ non lo sappiamo dire
  per $k = 2$ sì.

Inoltre, con la definizione della classe **Time**$(\mathbf{n})$ si può dire che:

$$\mathbf{P} = \bigcup_{k \geqslant 0} \mathbf{Time}(\mathbf{n}^k)$$

$$\mathbf{Exp} = \bigcup_{k \geqslant 0} \mathbf{Time}(\mathbf{2}^{\mathbf{n}^k})$$

3.4 Classe NP

La classe **NP** (*non deterministic polinomial time*) è la classe di problemi tali che se la soluzione per un'istanza del problema è *yes*, allora è facile verificarlo.

Definizione 3.4.1. (Classe NP)

$$\begin{aligned}\mathbf{NP} = \{\mathbb{A} \quad | \quad & \exists \mathcal{B}(\overset{x}{\cdot}, \overset{w}{\cdot}) \quad \text{t.c.} \quad T_{\mathcal{B}}(|x| + |w|) = O((|x| + |w|)^c) \\ & \forall x \in \mathcal{I}(\mathbb{A}) \quad \mathbb{A}(x) = yes \Leftrightarrow \exists w \text{ t.c. } \quad |w| = O(|x|^d) \text{ e } \mathcal{B}(x, w) = yes\}\end{aligned}$$

dove:

- $\mathcal{B}(\overset{x}{\cdot}, \overset{w}{\cdot})$ è detto **verificatore** per $\mathbb{A}$. Se la risposta di $\mathbb{A}$ esiste, allora $\mathcal{B}$ dice *yes*. Il verificatore impiega **tempo polinomiale** nella taglia dell'istanza per rispondere.
- x è l'istanza
- w è il certificato.

Hamiltonian Cycle $\in$ **NP ?** Per vedere se il problema Hamiltonian cycle appartiene alla classe **NP** dobbiamo costruire un verificatore $\mathcal{B}$ che agisca in tempo polinomiale.

```
VerifyHamCycle(G = (V,E), C=x1,...,xn)
    if r != |V|: return no                      } O(|w|)
    for each v∈ V                               }
        if v non appare in C: return no         } O(|V| · |C|)
    for i=1 to n-1                              }
        if (xi, xi+1) ∉ E: return no            } O(|C|)
    if (x1, xn) ∉ E: return no                  } O(1)
    return yes
```

Il tempo di esecuzione del verificatore è polinomiale e quindi posso dire che Hamiltonian Cycle $\in$ **NP** .

K-Colouring $\in$ NP ? Per vederlo costruisco il verificatore:

```
VerifyK-Colouring(G = (V,E), f(v_1),...,f(v_n))
    for each E(u,v)                              } O(|E|)
        if f(u) = f(v): return no
    for i=1 to n                                 } O(|V|)
        if f(v_i) ≥ K: return no
    return yes
```

Il tempo di esecuzione del verificatore è polinomiale e quindi posso dire che K-Colouring $\in$ **NP** .