Appunti Algoritmi e Strutture Dati

Luca Seggiani

14 Maggio 2024

1 NP-Completezza

Finora abbiamo considerato problemi che richiedevano algoritmi di risoluzione con complessità al massimo polinomiale ($O(n^2)$ se non addirittura $O(n^3)$). Vediamo adesso una categoria più ampia di problemi, i cosiddetti problemi **NP-completi**. Facciamo alcuni esempi:

- **Problema dello zaino**
  Uno zaino può contenere oggetti che hanno **peso** e **valore**. Si vuole determinare il numero di oggetti di ogni tipo che massimizzi il valore minimizzando il peso.

- **Problema del commesso viaggiatore**
  Bisogna trovare il percorso di minore lunghezza che un commesso viaggiatore deve effettuare per visitare tutte le città una e una sola volta per tornare alla città di partenza.

- **Cammini e cicli hamiltoniani**

  - **Cammino hamiltoniano**: Dato un multi-grafo, trovare, se esiste, un cammino che tocca tutti i nodi una sola volta.
  - **Ciclo hamiltoniano**: Dato un multi grafo, trovare, se esiste, un ciclo che tocca tutti i nodi una sola volta (cioè un cammino hamiltoniano ciclico).

- **Ciclo euleriano**
  Dato un multi-grafo, trovare, se esiste, un ciclo che passa per tutti gli archi una e una sola volta. I cicli euleriani entrano in gioco ad esempio nel **problema dei ponti di Konigsberg**: esiste un modo per visitare tutta la città Konigsberg passando dai suoi 7 ponti una e una sola volta? La risposta è no: è possibile definire un ciclo euleriano solamente su grafi che hanno un numero pari di archi uscenti da ogni nodo.

- **Soddisfattibilità di una formula logica (SAT)**
  Data una formula F composta da n variabili, trovare, se esiste, una combinazione di valori booleani che se assegnati ad F la rendono vera. Tutti i problemi NP-completi possono essere ridotti a un problema SAT, il cui algoritmo di risoluzione ha complessità esponenziale. Ad esempio, se le variabili che compaiono in F sono n, provare tutte le possibili combinazioni di quelle variabili significa provare 2^n diverse combinazioni.

Teoria della NP-completezza

La teoria della NP-completezza classifica un'insieme di problemi, i cosiddetti problemi NP-completi. Si definisce un problema NP-completo come un qualsiasi problema che soddisfa le proprietà:

- E' un problema decisionale (si risponde sì o no);
- Quando la risosta è sì, è facile dimostrarlo (con complessità al massimo polinomiale);
- Tutte le possibili risposte sì possono essere trovate con un algoritmo a forza bruta (di complessità esponenziale).

Notiamo che tutti i problemi prima riportati sono di tipo decisionale, e rispettano le proprietà dei problemi NP-completi. In generale, ogni problema può essere formulato come problema decisionale. Il problema decisionale ha complessità minore o uguale del problema non decisionale corrispondente.

Algoritmi nondeterministici

Per studiare la natura dei problemi NP-completi, introduciamo la nozione di algoritmo nondeterminstico: un algoritmo deterministico, come quelli a cui siamo stati abituati finora, controllano tutti i possibili valori del dominio delle soluzione per trovarne uno soddisfacente. Un algoritmo nondeterministico ha invece la *"virtù magica"* di scegliere la strada giusta: più propriamente, un'algoritmo nondeterministico controlla contemporaneamente tutte le possibili soluzioni e si ferma non appena ne trova una valida. Per ogni algoritmo nondeterministico ne esiste uno deterministico che lo simula, esplorando lo spazio delle soluzioni, fino a trovare una soluzione valida. Se le soluzioni sono in numero esponenziale, l'algoritmo deterministico avrà complessità esponenziale.

Algoritmi per la risoluzione del SAT

Vediamo due algoritmi per la risoluzione del problema SAT, uno deterministico (e ricorsivo), e uno nondeterministico.

- **Approccio deterministico**
  Un semplice approccio deterministico è dato dalla funzione ricorsiva:

```
bool dsat(Formula f, int* a, int i, int n) {
  if(i > n - 1) return value(f, a);
    else {
      a[i] = 1;
      if(sat(f, a, i+1, n)) return 1;
      else {
        a[i] = 0;
        return(sat(f, a, i+1, n));
      }
    }
}
```

  dove la funzione value() restituisce il valore della formula F all'interpretazione a dei suoi valori. Questo algoritmo, possiamo osservare, ha complessità esponenziale.

- **Approccio nondeterministico**
  Un approccio nondeterministico al problema e quello di scegliere una configurazione a casuale (attraverso la funzione choice(0, 1)), e valutarne la verità:

```
bool nsat(Formula f, int* a, int n) {
  for(int i = 0; i < n; i++)
    a[i] = choice({0, 1});
  if(value(f, a))
       return true;
    else
      return false;
}
```

E così, in complessità $O(n)$, assumendo che l'algoritmo scelga sempre la strada giusta, abbiamo trovato la soluzione ottima. Questo chiaramente non rispecchia la realtà dei fatti.
Possiamo adesso anticipare quella che è la funzione degli algoritmi nondeterministici (e quindi la definizione di NP): l'algoritmo deterministico per il calcolo del SAT richiede complessità esponenziale, mentre la sua controparte nondeterministica richiede complessità polinomiale. Sarà questo a determinare l'appartenza del problema SAT a NP.

P e NP

Definiamo allora P = l'insieme di tutti i problemi decisionali risolvibili in tempo polinomiale con un'algoritmo deterministico. A questo punto NP = l'insieme di tutti i problemi decisionali risolvibili in tempo polinomiale con

un algoritmo nondeterministico. NP è un acronimo: sta proprio per *Nondeterministico Polinomiale*. In altre parole, possiamo dire che NP = l'insieme di tutti i problemi decisionali che ammettono un algoritmo deterministico polinomiale di verifica di una soluzione. Per dimostrare che un problema $R \in$ NP si dimostra che la verifica di una soluzione di R si fa in tempo polinomiale.

Riducibilità

La riducibilità è un metodo per convertire l'istanza di un problema P_1 in un istanza di un problema P_2 e utilizzare la soluzione di quest'ultimo per ottenere la soluzione di P_1. Un problema P_1 si riduce in tempo polinomiale a un problema P_2 se ogni soluzione di P_1 può ottenersi deterministicamente in tempo polinomiale da una soluzione di P_2. Questo si indica come $P_1 \leq P_2$: se P_2 è risolvibile in tempo polinomiale, allora anche P_1 è risolvibile in tempo polinomiale.

Definizione formale di NP-completezza

A questo punto possiamo definire formalmente la NP-completezza. Un problema R è NP-completo se:

- $R \in$ NP,
- $\forall Q \in \text{NP} \Rightarrow Q \leq R$.

Ovvero, un problema è NP-completo se appartiene ad NP ed è complesso almeno quanto ogni problema appartenente a NP.

Teorema di Cook-Levin

Qualsiasi problema R in NP è riducibile al problema del soddisfattibilità della formula logica:

$$\forall R \in \text{NP} : R \leq \text{SAT}$$

Ergo, SAT è più difficile di tutti i problemi in NP (è NP-completo). A questo punto dimostare che un problema è NP-completo significa dire che:

- $R \in$ NP,
- SAT $\leq R$.

Ciò che abbiamo appena detto è: se un problema è NP-completo, allora è difficile tanto quanto SAT e può essere usato al suo posto nella dimostrazione di NP-completezza di un altro problema. I problemi NP-completi hanno tutti la stessa difficoltà e sono i più difficili della classe NP. Questi problemi si dicono anche NP-hard (o *NP-ardui, NP-difficili, ...*) Se si trovasse un'algoritmo polinomiale per SAT o per qualsiasi altro problema NP-completo, allora tutti i problemi in NP sarebbero risolvibili in tempo polinomiale, e quindi sarebbe verificato P = NP. Quest'ultima affermazione è ad oggi irrisolta, e

rappresenta uno dei *problemi del millennio* (con annesso premio Clay di un milione di dollari per chiunque dovesse risolverlo).

Dimostrare l'NP-completezza

A questo punto dimostrare che un problema R è NP-completo significa dimostrare che, almeno che non sia vero P = NP, non esiste una soluzione polinomiale di quel problema. Basterà quindi:

- **Dimostrare che R appartiene ad NP**: quindi individuare un'algoritmo polinomiale nondeterministico per risolvere R, o dimostrare che la verifica di una soluzione di R può essere fatta in tempo polinomiale;
- **Dimostrare che esiste un problema NP-completo che si riduce a R**: ovvero scegliere un problema NP-completo noto che sia riducibile a R.

Fattorizzazione di un numero

Facciamo un'esempio: la ricerca dei fattori primi di un numero. La verifica di una possibile fattorizzazione può essere svolta in tempo polinomiale, ergo è NP. Non possiamo però verificare che esso sia anche P: non sono stati trovati algoritmi che possano trovare tutti i fattori in tempo polinomiale. Allo stesso modo, è improbabile che esso sia NP. Questo tipo di problemi viene detto NP-intermediate (se esistono!). Sulla fattorizzazione si basano diversi algoritmi di crittografia a chiave pubblica, che richiedono numeri primi molto grandi: dimostrare che eiste un'algoritmo P per la fattorizzazione non significherebbe dimostrare P = NP (in quanto è improbabile che il problema sia NP-completo), ma significherebbe mettere profondamente in crisi i moderni sistemi di crittografia. L'algoritmo di Shor (1994), che può essere eseguito su computer quantistici, risolve questo problema in tempo polinomiale: è per questo che l'avvento dei computer quantistici potrebbe mettere seriamente in pericolo la sicurezza dei sistemi informatici tradizionali.

Metodologie per affrontare i problemi difficili

Esistono più modi di affrontare questo tipo di problemi (che rappresentano una questione ancora in corso!):

- **Algoritmi di approssimazione**, che approssimano con sempre maggiore accuratezza la soluzione ottimale fino ad un certo grado di incertezza;
- **Algoritmi probabilistici**, che includono componenti stocastiche (euristiche) che cercano di avvicinarsi il più possibile, attraverso la probabilità, alla soluzione ottima.
- **Reti neurali**, che usano approcci basati sull'intelligenza artificiale;

- **Computer quantistici**, come già detto prima nell'esempio dell'algoritmo di Schur.

Problemi fuori da NP

Esistono problemi che sono all'infuori dell'insieme NP: un'esempio può essere il **problema della torre di Hanoi**. Qualsiasi algoritmo nondeterminisitco per la sua risoluzione avrà infatti complessità esponenziale nel caso migliore, non polinomiale. Un'altro esempio è il **calcolo di tutte le permutazioni di un'insieme**, che ha complessità $O(n!)$ fattoriale.

Problemi non risolvibili

Esistono alcuni problemi che non possono essere risolti in generale attraverso un algoritmo. Si fanno alcuni celebri esempi:

- Decidere se un programma termina o meno su un certo input;
- Determinare la soddisfacibilità di una formula nella logica del primo ordine;
- Determinare l'equivalenza di due programmi.

2 Programmazione ad oggetti

Il paradigma della programmazione ad oggetti, come è stato visto nel corso di Fondamenti, consiste nella definizione da parte del programmatore di oggetti astratti, che nascondono (attraverso il meccanismo delle classi o dei prototipi) il loro funzionamento e la loro rappresentazione interna. Le caratterstiche (e i vantaggi) della programmazione ad oggetti sono:

- **Incapsulamento**: l'occultamento del funzionamento interno (*information hiding*) della classe, che permette di trattare la stessa come un'oggetto astratto le cui specifiche d'implementazione sono irrilevanti. Inoltre, l'incapsulamento favorisce la sicurezza (non si ha accesso a parti più "sensibili" del codice) e la manutenzione (si può variare il funzionamento interno della classe senza pregiudicarne la compatibilità con applicativi sviluppati precedentemente, in quanto incapsulata).
- **Decomposizione**: la composizione di più classi in oggetti via via più astratte e modulari: ad esempio, la classe "automobile" potrebbe essere costruita come la composizione di più classi (moduli): ruota, motore, ecc...

- **Riutilizzabilità**: una classe, come oggetto astratto, può essere utile in più applicazioni, e quindi incluso in un programma (attraverso il meccanismo delle librerie, ecc...), senza dover ricorrere alla copiatura diretta del codice.
- **Manutenzione**: come già detto, l'occultamento del funzionamento interno favorisce la manutenzione dell'implementazione della classe.
- **Affidabilità**: la gestione del codice a "compartimenti stagni" permette una maggiore sicurezza su più livelli di astrazione: ci potremo assicurare che la classe è affidabile e ben programmata, a poi sviluppare il resto del programma, o le classi che poggiano su di essa.

Classi ed Oggetti

Una **classe** è una descrizione di un insieme di oggetti con proprietà (attributi), comportamento (operazioni), relazioni e semantica comuni. La classe enfatizza le caratteristiche comuni degli oggetti e sopprime (nasconde) le caratterstiche specifiche degli oggetti. Un'**oggetto** è una manifestazione concreta dell'astrazione (classe), ovvero un'istanza di una classe.

Meta-Programmazione

La meta-programmazione è un meccanismo che permette di appllicare lo stesso codice a diversi tipi di dati, parametrizzando i tipi usati. Si basa su:

- **Indipendenza degli algoritmi dei dati**: ad esempio, si possono definire algoritmi di ordinamento identici su interi, stringhe, ecc...
- **Indipendenza dei contenitori dal loro tipo di contenuto**: ad esempio, una certa struttura dati può contenere interi, altre strutture, ecc..

Funzioni modello

Poniamo di avere la funzione max definita su interi e double:

```
1 int i_max(int a, int b) {
2   return (a > b) ? a : b;
3 }
4
5 double d_max(double a, double b) {
6   return (a > b) ? a : b;
7 }
```

Per ogni tipo su cui vogliamo chiamare la funzione, a meno che non siamo disposti ad accettare conversioni implicite, dobbiamo ridefinire la funzione. Un'altro approccio, che evita la ridefinizione della funzione per ogni tipo di interesse, può essere quello della *templatizzazione* del codice. Si trasforma la funzione in un template:

```
template<class T>
T m_max(T a, T b) {
  return (a > b) ? a : b;
}
```

La parola chiave class (che può essere sostituita da typename) definisce un "tipo" assegnato a T, che viene sostitito da qualsiasi tipo venga usato nella chiamata di funzione:

```
m_max<int>(a, b); m_max<double>(a, b);
```

L'esecuzione della funzione template viene eseguita su qualsiasi tipo gli venga fornito: la definizione dell'implementazione specifica ad ogni tipo viene realizzata dal compilatore a tempo di compilazione. Il meccanismo dei template potrebbe sembrare simile a quello delle macro: le differenze sono la maggiore versatilità e il fatto che i template forniscono il type-checking.

Argomenti impliciti di funzioni modello

Possiamo, senza specificare il tipo su cui la funzione deve lavorare, lasciare al compilatore l'onere di decidere attraverso i tipi degli argomenti forniti. In questo caso valgono le regole di conversione implicita, e il compilatore lancierà un'errore nel caso in cui le chiamate siano ambigue.