DENTRO LA SCATOLA

Rubrica a cura di

Fabio A. Schreiber

Il Consiglio Scientifico della rivista ha pensato di attuare un'iniziativa culturalmente utile presentando in ogni numero di Mondo Digitale un argomento fondante per l'Informatica e le sue applicazioni; in tal modo, anche il lettore curioso, ma frettoloso, potrà rendersi conto di che cosa sta "**dentro la scatola**". È infatti diffusa la sensazione che lo sviluppo formidabile assunto dal settore e di conseguenza il grande numero di persone di diverse estrazioni culturali che - a vario titolo - si occupano dei calcolatori elettronici e del loro mondo, abbiano nascosto dietro una cortina di nebbia i concetti basilari che lo hanno reso possibile.
Il tema scelto per il 2004 è stato: "***Perché gli anglofoni lo chiamano computer***, ovvero: ***introduzione alle aritmetiche digitali***". Per il 2005 il filo conduttore della serie sarà: "***Ma ce la farà veramente?***, ovvero: ***introduzione alla complessità computazionale e alla indecidibilità***" e il suo intento è di guidare il lettore attraverso gli argomenti fondanti dell'Informatica e alle loro implicazioni pratiche e filosofiche. La realizzazione degli articoli è affidata ad autori che uniscono una grande autorevolezza scientifica e professionale a una notevole capacità divulgativa.

Potenza e limiti del calcolo automatico: problemi intrattabili

Pierluigi San Pietro

1. Introduzione

Questo è l'ultimo articolo di una serie di quattro dedicata agli aspetti più concettuali e teorici dell'informatica. I primi due articoli si sono focalizzati sulla nozione di algoritmo come formalizzazione della nozione di calcolo automatico, e sulla possibilità di risolvere problemi in modo algoritmico. Il terzo, che costituisce la prima parte del presente, ha introdotto il concetto di complessità computazionale di un algoritmo, intesa come una misura, indipendente dal calcolatore e dal linguaggio di programmazione utilizzato, del tempo di calcolo e dello spazio di memoria necessari per eseguire l'algoritmo. Questo articolo si sofferma sulle conseguenze più profonde dei concetti di complessità computazionale, con importanti ricadute pratiche in domini come la sicurezza del commercio elettronico o l'ottimizzazione delle risorse.

2. La teoria della complessità computazionale

Un problema computazionale è un problema che ammette una soluzione algoritmica, ossia una soluzione che può essere calcolata, almeno in teoria, da un computer opportunamente programmato. L'algoritmo però deve essere ragionevolmente efficiente, ossia impiegare un tempo non eccessivo per calcolare la risposta, e non deve aver bisogno di una quantità di memoria maggiore rispetto a quella disponibile.
In questo articolo ci soffermiamo sulla cosiddetta *Teoria della Complessità Computazionale* (TCC), il cui oggetto di studio non è tanto l'efficienza di un singolo algoritmo, ma l'ammontare delle risorse computazionali (come tempo di esecuzione, spazio di memoria, dimensione dei circuiti) necessarie per risolvere *problemi* computazionali. La TCC è indipendente dagli algoritmi effettivamente utilizzati per risolvere i problemi, ma si propone di determinare la complessità dei migliori algoritmi possibili (spesso anche senza conoscerli). La TCC è in larga misura indipendente persino dal modello computazionale adottato (macchine di Turing, macchine RAM, calcolatori paralleli,...), a patto che questo sia fisicamente realizzabile.
La TCC costituisce uno dei maggiori successi scientifici degli ultimi quaranta anni, con risultati di portata molto generale. I suoi modelli computazionali astratti classificano le migliaia di

problemi computazionali del mondo reale in poche classi di equivalenza, all'interno delle quali i problemi ammettono soluzioni algoritmiche pressappoco della stessa efficienza. Quindi, studiando un nuovo problema X, è sufficiente mostrare che X ricade in una certa classe per avere subito un'idea approssimativa della complessità computazionale delle sue soluzioni, anche senza avere ancora sviluppato direttamente alcun algoritmo per X. Inoltre, nota la classe di complessità, la progettazione di un algoritmo è spesso molto facilitata, in quanto esistono molte tecniche che aiutano nello sviluppo di algoritmi efficienti per i problemi di una stessa classe.

Il contributo più importante della TCC consiste forse nell'avere riconosciuto che esistono problemi di grande rilevanza pratica, la cui risoluzione esatta è (o, spesso, sembra) intrinsecamente difficile qualunque sia lo strumento di calcolo utilizzato, e che si possono quindi risolvere solo in casi particolari.

3. Un semplice esempio

Un commesso viaggiatore deve visitare in automobile N città. Desidera stabilire se esiste un percorso che visita una sola volta tutte le città e la cui lunghezza complessiva sia inferiore a un valore K, oltre il quale la sua attività non è più conveniente. A tale fine, ha a disposizione una cartina stradale, da cui si ricava facilmente l'esistenza e la lunghezza delle strade dirette di collegamento da una città a un'altra.

Si rivolge quindi a un programmatore, grande esperto di sistemi e linguaggi informatici, ma assai digiuno di TCC, che prepara il seguente algoritmo:

1. disponi le N città in una permutazione qualsiasi C_1, C_2, ..., C_N;

2. se esiste un percorso che parte da C_1 passa per C_2, ..., per arrivare infine a C_N, calcolane la lunghezza complessiva L; se L è minore di K, stampa il percorso e termina con successo;

3. se non ci sono più permutazioni disponibili, termina con un fallimento, altrimenti scegli una nuova permutazione e riparti da 2.

Supponiamo di dovere visitare 4 città, chiamate A, B, C e D. L'algoritmo per esempio può calcolare dapprima la lunghezza del percorso A B C D (cioè il percorso che parte da A, poi passa per B, poi per C e infine per D). Se vi sono strade di collegamento fra A e B, B e C, C e D, l'algoritmo calcola la lunghezza complessiva del percorso e, se questa è minore di K, termina l'esecuzione. Altrimenti prova un'altra sequenza, per esempio A C B D, e così via fino a trovare una soluzione o esaurire tutti i percorsi possibili.

Quanto tempo ci vuole per ottenere una risposta?

Il caso migliore (il più favorevole per l'algoritmo) è quello in cui proprio il primo percorso scelto ha lunghezza inferiore a K. Il caso peggiore, invece, corrisponde alla situazione in cui si provano tutte le sequenze del tipo A B C D, A B D C, A C B D, ..., per concludere magari che nessuna ha lunghezza inferiore a K. Calcoliamo il numero di queste sequenze (o *permutazioni* di N elementi). La città da visitare per prima può essere una qualunque delle quattro, mentre in seconda posizione ci può essere una qualunque delle tre rimanenti, seguita in terza posizione da una qualunque delle due ancora da visitare, seguita infine dall'ultima città non ancora visitata. Ci sono quindi $4 \cdot 3 \cdot 2 \cdot 1 = 24$ possibili percorsi, che anche un computer scalcinato può enumerare in un batter d'occhio.

Cosa succede all'aumentare del numero delle città? Se al posto di 4 città ce ne fossero 10? O 30?

Il ragionamento precedente può essere generalizzato alla visita di N città, ricavando che nel caso peggiore l'algoritmo deve verificare $N! = N\,(N-1)\,(N-2)\ldots 1$ possibili percorsi. Per visitare 10 città, l'algoritmo dovrebbe provare 10! permutazioni, cioè quasi quattro milioni di casi: questo richiederebbe più tempo, ma basterebbe acquistare un calcolatore più potente per cavarsela in una manciata di secondi. E per visitare 30 città? È sufficiente comprare un calcolatore ancora più potente?

Le permutazioni da provare sono 30!, circa $2{,}6 \cdot 10^{32}$. Se un'antica specie aliena avesse costruito, al momento della nascita dell'universo (circa 13 miliardi di anni fa), un supercalcolatore in grado di esaminare mille miliardi di percorsi al secondo, e avesse cominciato subito a fare eseguire l'algoritmo, sarebbero necessari ulteriori 8000 miliardi di anni per completare l'analisi. Il commesso viaggiatore non sarà certo molto soddisfatto del software acquistato, e si domanderà (correttamente) quali studi abbia compiuto il programmatore.

4. Trattabilità e Intrattabilità. La classe P

In termini più precisi, l'algoritmo precedente ha complessità asintotica $\Theta(N!)$, dove N è la dimensione dei dati in ingresso. Un algoritmo di questo tipo è del tutto inutile, salvo che per casi semplicissimi. Gli algoritmi che utilizzano, al crescere delle dimensioni dell'input una quantità di risorse "irragionevolmente alta" sono chiamati *intrattabili*, altrimenti sono *trattabili*. La definizione di (in)trattabilità dipende crucialmente da cosa si intenda con la locuzione "una quantità di risorse irragionevolmente alta". Ci limitiamo per ora allo studio del solo tempo di esecuzione, rimandando al paragrafo 8 per considerazioni sulle altre risorse.

Una funzione di complessità fattoriale o esponenziale porta a una rapida esplosione dei tempi di esecuzione in funzione delle dimensioni n dell'ingresso, come si vede nella tabella 1, in cui si ipotizza di eseguire gli algoritmi su un calcolatore in grado di svolgere un milione di operazioni elementari al secondo.

Per esempio, se un algoritmo per il problema del commesso viaggiatore avesse complessità $\Theta(2^n)$ potrebbe forse arrivare a trattare casi con una quarantina di città, ma certo non potrà mai risolvere problemi con 100 città o più. Gli algoritmi di complessità $\Theta(n \log(n))$, come per esempio i migliori algoritmi di ordinamento visti nel terzo articolo di questa serie, possono in pratica risolvere problemi di dimensione qualunque, ma anche algoritmi in $\Theta(n^2)$ o in $\Theta(n^3)$ sono ragionevolmente efficienti in molte situazioni pratiche. La distinzione fondamentale sembra quindi fra gli algoritmi che hanno complessità esponenziale (o più alta) e quelli di complessità *polinomiale*. Un algoritmo ha complessità polinomiale se la sua complessità non cresce più rapidamente di un polinomio, ossia se è $\Theta(n^k)$ per qualche k. Per esempio, $n \log(n)$ ha complessità polinomiale (perché cresce più lentamente del polinomio n^2, mentre 2^n cresce più rapidamente di qualunque polinomio). Secondo gran parte della comunità scientifica, questa considerazione è generale: gli algoritmi trattabili sono solo quelli di complessità polinomiale.

Questa definizione di trattabilità può tuttavia sconcertare. Infatti, mentre può essere chiaro che funzioni polinomiali dell'ordine di n^3 sono ragionevolmente efficienti, appare alquanto singolare mettere in questa classe polinomi del tipo $n^{10^{300}}$. Tuttavia, la scelta è basata su criteri empirici: non si conoscono problemi di reale importanza pratica con complessità polinomiale così elevata, ma anzi per ogni problema interessante o si è trovato un algoritmo ragionevole, ad esempio in $\Theta(n^4)$, oppure si sa, o si pensa, che il problema abbia complessità più che polinomiale (tipicamente esponenziale). Se tuttavia un giorno si trovasse un problema importante per cui i migliori algoritmi funzionano in tempo $n^{10^{300}}$, la scelta "efficiente = polinomiale" andrebbe rivista.

A questo punto, il commesso viaggiatore dell'esempio potrebbe pensare che basti assumere un programmatore più bravo che, con un po' di tempo e d'ingegno, realizzerà certo un algo-

Dimensione del problema	Tempo di esecuzione in secondi in funzione della dimensione				
n	$n \log(n)$	n^2	n^3	2^n	$n!$
10	0,00003	0,0001	0,001	0,001	3,6288
20	0,00008	0,0004	0,008	1,05	$2,4 \cdot 10^{12}$
30	0,0001	0,0009	0,027	1073,742	$2,6 \cdot 10^{26}$
50	0,0003	0,0025	0,125	$1,1 \cdot 1\,0^9$	$3 \cdot 10^{58}$
100	0,0007	0,01	1	$1,3 \cdot 10^{24}$	$9,3 \cdot 10^{151}$
1000	0,001	1	1000	$1,1 \cdot 10^{295}$	$\approx 10^{3000}$
10^6	19,93	10^6	10^{12}	$\approx 10^{300000}$	$\approx 10^{6000000}$

TABELLA 1

Tempi di esecuzione in secondi per algoritmi di varia complessità. Un "passo" elementare = un microsecondo

ritmo che sia stavolta polinomiale anche nel caso peggiore.
La TCC però ci insegna che ciò è assai improbabile, pur non potendo escluderlo del tutto. Come è possibile?
La TCC si propone non tanto di trovare algoritmi per risolvere i problemi, ma di dimostrare che per certi problemi non vi è *nessun algoritmo trattabile* in tutti i casi. Si è visto, nel primo e nel secondo articolo di questa serie, che è possibile dimostrare che per certi problemi non esiste alcun algoritmo. Invece ora interessa sapere se esiste un algoritmo che sia anche efficiente. La TCC studia la complessità dei *problemi*, ossia la complessità del migliore algoritmo possibile (noto o ignoto che sia) che risolve un certo problema.
In generale, i problemi per i quali esiste un algoritmo trattabile sono anch'essi detti trattabili, altrimenti sono intrattabili. Quindi la classe dei problemi trattabili coincide con la classe dei problemi che ammettono l'esistenza di algoritmi polinomiali per la loro soluzione (su una Macchina di Turing). Tale classe viene chiamata **P** (Polinomiale).

5. La classe NP e i problemi NP-completi

Abbiamo già visto nella terza parte di questa serie come a volte sia possibile trovare un limite inferiore alla complessità di un problema, ossia una soglia di efficienza algoritmica invalicabile: una volta trovato un algoritmo che appartenga alla classe di complessità asintotica di questa "soglia", per quanti sforzi si facciano per ideare nuovi algoritmi, non si riesce a oltrepassare questa barriera, perché i nuovi algoritmi portano soltanto miglioramenti marginali (per esempio, abbassano le costanti moltiplicative, oppure migliorano le prestazioni per valori limitati delle dimensioni dei dati o in casi particolari).
Trovare limiti inferiori alla complessità di un problema è difficile: si deve prescindere da ogni particolare algoritmo risolutivo, ma la dimostrazione deve essere valida per tutti i possibili algoritmi (inclusi anche quelli non ancora inventati). La cosa sembra ancora più complicata in quanto apparentemente questo procedimento di dimostrazione deve essere ripetuto per ogni diverso problema. Poiché nella pratica vi sono migliaia e migliaia di problemi da risolvere con il calcolatore sembra che il procedimento abbia poca speranza. Si è però scoperto che moltissimi problemi interessanti, in apparenza assai diversi fra loro, sono in un certo senso "riducibili" a pochi problemi fondamentali. In particolare, esiste una grande varietà di problemi di notevole interesse pratico, detti problemi *NP-completi*, per cui esistono forti motivi per pensare che siano tutti intrattabili, cioè non appartengano alla classe P, senza però poterlo asserire con certezza.
Nessuno è mai riuscito a escogitare un algoritmo polinomiale per calcolare una soluzione del problema del commesso viaggiatore. Tuttavia, è molto facile trovare un algoritmo polinomiale (addirittura in tempo lineare $\Theta(n)$) per *verificare se un percorso dato* sia una soluzione: basta calcolare la lunghezza del percorso e confrontare il risultato con K. Lo stesso algoritmo "stupido" che prova tutti i percorsi è strutturato in una parte che genera un percorso e in un'altra che lo verifica. Si è constatato che la facilità nel verificare una soluzione piuttosto che nel calcolarla è un fatto di validità generale, tanto da meritare addirittura l'introduzione di una classe di complessità *ad hoc*. La classe dei problemi per cui esiste un algoritmo polinomiale nelle dimensioni dell'input (per una Macchina di Turing) per controllare che una soluzione data sia corretta si chiama **NP** (Nondeterministico Polinomiale)[1].
Tutti i problemi della classe P sono ovviamente anche in NP. Esistono problemi in NP che non sono in P?
Uno degli sviluppi più sorprendenti e importanti della teoria della complessità è stato scoprire che un gran numero di problemi pratici interessanti sono contenuti nella classe NP, definita inizialmente a partire da modelli matematici puramente teorici, ma senza potere

[1] **NP** *non* significa Non Polinomiale, come talvolta erroneamente si ritiene. Il termine "Nondeterministico" deriva invece da un particolare modello di macchina di Turing (TM), la TM nondeterministica, in grado di risolvere un problema "indovinandone" una soluzione finita e poi verificandola. Anche se i modelli nondetermistici non sono fisicamente realizzabili come tali, ogni funzione calcolabile da una TM nondeterministica può essere calcolata anche da una TM (ma, almeno apparentemente, in un tempo maggiore).

stabilire se essi si trovino anche in P. Quindi, per essi si può determinare efficientemente se una *soluzione data* è accettabile, ma non è noto se sia possibile anche *calcolare* efficientemente una soluzione. Quasi tutti questi problemi, come quello del commesso viaggiatore, costituiscono una sottoclasse particolare di NP, quella dei problemi ***NP-completi*** (riquadro a piè pagina). I problemi NP-completi hanno un ruolo davvero speciale nella classe NP: basterebbe trovare un algoritmo che risolva in tempo polinomiale un solo problema NP-completo per consentire di risolvere in tempo polinomiale qualunque altro problema della classe NP (e pertanto in tal caso si sarebbe dimostrato che P = NP).

Nonostante gli sforzi trentennali di moltissimi ricercatori, con tecniche matematiche e algoritmiche molto sofisticate, nessuno ha mai trovato un algoritmo polinomiale per un problema NP-completo (ad esempio, il migliore algoritmo noto per il commesso viaggiatore funziona in tempo $\Theta(n^2 2^n)$), ma nessuno ha mai nemmeno dimostrato che tale algoritmo non esiste. È tuttavia assai più facile, per un certo problema, scrivere un algoritmo e valutarne la complessità che provare che non esiste nessun algoritmo di un certa complessità. Quindi la maggior parte dei ricercatori ritiene che effettivamente P sia diverso da NP: se così non fosse, qualcuno probabilmente avrebbe ormai trovato un algoritmo polinomiale per un problema NP-completo, mostrando così che tutti i problemi in NP sono anche in P.

6. Come affrontare in pratica problemi NP-completi

Nella pratica, di solito i problemi NP-completi sono considerati intrattabili "per ignoranza". Quando infatti si trova che un problema interessante è NP-completo, si può tranquillamente interrompere la ricerca di un algoritmo polinomiale, in quanto è assai improbabile riuscire a trovarlo, perché questo equivarrebbe a risolvere così la questione P = NP che ha impegnato a lungo e senza successo menti fra le più brillanti dei nostri tempi.

Questo però non significa che, poiché un problema intrattabile è assolutamente senza speranza di risoluzione algoritmica efficiente in generale, allora possiamo evitare anche di provare a risolverlo. Si possono infatti applicare tecniche algoritmiche ben note, che permettono di risolvere il problema in modo efficiente *non in generale* ma in casi particolari, o a costo di accettare soluzioni anche un po' imprecise.

Ad esempio, sono stati trovati algoritmi ben più efficienti di quello banale per risolvere il problema del commesso viaggiatore. Nel maggio 2004, uno di questi algoritmi ottimizzati [6], implementato su un cluster di un centinaio di workstation, ha risolto il problema della visita ottima di tutte le 24,978 città della Svezia, im-

Riduzione e NP-completezza

La tecnica utilizzata per definire la classe dei problemi NP-completi va sotto il nome di *riduzione polinomiale*. L'idea di riduzione è già stata introdotta nei primi articoli di questa serie, e viene qua estesa alla riduzione polinomiale. Per semplicità, ci limitiamo a problemi di decisione, ossia a problemi p la cui risposta su un dato d sia $p(d)$ = SI o $p(d)$ = NO. Per esempio, il problema del commesso viaggiatore è un problema di decisione: dati il numero delle città, le loro distanze e la lunghezza massima K, esiste un percorso che passa per tutte le città una sola volta ed è di lunghezza inferiore a K? In generale, per risolvere un problema di decisione p sui dati d, al posto di definire direttamente un algoritmo per p si può utilizzare la nostra capacità di risolvere un altro problema di decisione p', diverso da p. Ci basta trovare un algoritmo R che, preso il nostro problema p e i dati d, genera dati d' per il problema p' in modo che la risposta a p' sui dati d' sia la stessa di quella di p per i dati originali d, ossia $p(d) = p'(d')$. Per esempio, se $p(d)$ è SI, anche $p'(d')$ è SI, e viceversa. L'algoritmo R è detto *algoritmo di riduzione*. Se R ha complessità polinomiale si dice che p è *riducibile polinomialmente a* p'. Se la soluzione di p' fosse in tempo polinomiale, allora avremmo trovato un algoritmo polinomiale anche per p: basta applicare la riduzione (tempo polinomiale) e poi risolvere il problema p' (in tempo polinomiale, su dati d' la cui dimensione non può che essere polinomiale in d). La tecnica di riduzione è spesso applicata anche in pratica per costruire algoritmi, ma è fondamentale per mostrare che un problema è NP-completo. Formalmente, un problema p' è NP- completo se ogni problema p della classe NP è riducibile polinomialmente a p'. Quindi, se fosse noto un algoritmo polinomiale per risolvere anche un solo un problema NP-completo, come quello del commesso viaggiatore, si saprebbe automaticamente risolvere in tempo polinomiale ogni altro problema della classe NP: le due classi di complessità coinciderebbero (P = NP). Se invece P ≠ NP, ci sarebbero problemi in NP, fra cui tutti i problemi NP-completi, non risolubili in tempo polinomiale. Per mostrare che un problema p in NP è NP-completo, occorrerebbe dimostrare che tutti i problemi in NP sono riducibili polinomialmente ad esso, il che è ovviamente assai difficile. Tuttavia, una volta trovato il primo problema NP-completo (per esempio, il cosiddetto problema della soddisfacibilità di formule booleane, dimostrato NP-completo da S. Cook nel 1971), è poi sufficiente mostrare che questo problema è riducibile polinomialmente a p.

piegando "solo" l'equivalente di 80 anni di calcolo per un singolo calcolatore. La caratteristica di questo algoritmo (e altri simili) è quello di avere ancora complessità esponenziale nel caso peggiore, ma di essere polinomiale in vari casi particolari, spesso di notevole rilevanza pratica. Tuttavia è sempre possibile costruire esempi, anche piccoli, in cui anche questi algoritmi ottimizzati non riescono a trovare la soluzione ottima in tempi ragionevoli.

Esistono varie tecniche per risolvere numerosi (ma non tutti!) problemi intrattabili d'importanza pratica. Per esempio, si possono "scegliere bene" quali alternative considerare per prime, per eliminare precocemente quelle che sicuramente non sono accettabili per la soluzione. Nel problema del commesso viaggiatore, ad esempio, al posto di calcolare la lunghezza di un intero percorso A B C D, possiamo calcolare dapprima la distanza da A a B: se questa supera già il limite K, si può fare a meno di prendere in esame tutti i percorsi che cominciano con A B. Questo semplice accorgimento può ridurre notevolmente il numero di passi dell'algoritmo in molti casi pratici, anche se il caso peggiore resta sempre intrattabile.

Spesso gli algoritmi sono basati su strategie empiriche (dette "euristiche") che cercano di arrivare velocemente a una soluzione corretta in molti casi particolari. Per esempio, si potrebbe costruire un percorso scegliendo ogni volta la città più vicina fra quelle non esplorate: se la città più vicina ad A fosse D, e poi la più vicina a D (fra B e C) fosse C, allora l'algoritmo potrebbe esaminare solo la lunghezza del percorso A D C B. Questa tecnica, che può anche essere combinata con la strategia precedente, si comporta bene in vari casi speciali, ma in generale non può garantire di trovare un risultato corretto in modo efficiente.

Altre tecniche euristiche più sofisticate rinunciano a produrre una soluzione esatta, ma si accontentano di garantire un limite all'errore. Per esempio, un algoritmo approssimato potrebbe per esempio, scorrettamente rispondere "NO" quando invece esiste un percorso di lunghezza inferiore a K, ma che non si scosta da K di più, ad esempio, del 5%. In questo modo, molti problemi intrattabili, ma non tutti, diventano risolubili in modo efficiente.

Infine, esistono algoritmi esponenziali che in pratica si comportano sempre molto bene, in quanto l'esponenzialità corrisponde a casi patologici che di fatto non accadono mai. Un esempio in tal senso è il famoso algoritmo del simplesso, usato per la soluzione di problemi di programmazione lineare, che in pratica è sempre polinomiale anche se il caso peggiore è esponenziale. Si noti però che il problema della programmazione lineare è comunque nella classe P: i "casi patologici" esponenziali pertanto non corrispondono a un problema intrattabile, ma solo a un particolare algoritmo che riesce a essere particolarmente efficiente in pratica proprio "rischiando" ogni tanto di non riuscire di fatto a terminare la computazione.

7. La tesi estesa di Church-Turing

Un dubbio naturale che potrebbe sorgere riguarda proprio le fondamenta della TCC: la complessità di un problema può dipendere dal modello di calcolo? Cosa succederebbe se al posto di una macchina di Turing si usasse un altro modello?

Per esempio, abbiamo definito P usando un modello di calcolo *sequenziale* (la macchina di Turing), che esegue cioè una sola operazione per volta. I modelli di calcolo paralleli, come un sistema multiprocessore, possono invece eseguire molte operazioni contemporaneamente. Per esempio, il problema del commesso viaggiatore potrebbe essere risolto molto fretta, per un qualunque numero N di città, a patto di avere un calcolatore con N! processori: ogni processore potrebbe essere dedicato al calcolo della lunghezza di un singolo percorso. Di fatto, però questo modello parallelo è impossibile da realizzare: per risolvere in questo modo il problema per N = 100 occorrerebbero 10^{157} processori, ossia un numero probabilmente molto superiore agli atomi nell'universo. La classe P può tuttavia essere definita in modo del tutto equivalente utilizzando modelli paralleli, a patto di considerare come "tempo" il lavoro totale svolto dai vari processori. La TCC ha verificato che ogni altro modello "ragionevole" studiato finora definisce esattamente le stesse classi di complessità.

Le classi di complessità, pur essendo definite in base a modelli di calcolo, che apparentemente poco o nulla hanno a che vedere con i problemi concreti, consentono di classificare la grande

maggioranza delle migliaia e migliaia di problemi computazionali che sorgono nella pratica. Questo "irragionevole successo" dei nostri modelli matematici, e la sostanziale indipendenza dei risultati della teoria dal modello computazionale scelto, ha portato, su basi empiriche, a ritenere che le classi di complessità come P catturino aspetti importanti del mondo reale. Da qui, la cosiddetta *tesi estesa* di Church-Turing: *La classe di complessità P cattura esattamente la nozione di algoritmo calcolabile in tempo polinomiale, qualunque sia il modello di calcolo fisicamente realizzabile usato per definirla*.
Poiché si ritiene che "efficiente" sia sinonimo di "risolubile in tempo polinomiale", la tesi implica che la classe P cattura tutti i problemi pratici la cui risoluzione può essere considerata "efficiente". La tesi è l'analogo per la complessità computazionale della tesi di Church-Turing, illustrata nel primo articolo di questa serie, che afferma che i problemi computabili sono esattamente quelli risolubili con le macchine di Turing.
Di recente, una sfida alla tesi estesa di Church-Turing è arrivata da un modello di computazione fondato sulle proprietà della meccanica quantistica, che introduce una forma di parallelismo illimitato, basato sui cosiddetti bit quantistici (qubit). Al momento, i calcolatori quantistici realizzati secondo questo modello sono di dimensioni molto ridotte (per esempio, sette bit quantistici, equivalenti a $2^7 = 128$ bit tradizionali, quando invece le memorie dei calcolatori sono misurate in miliardi di bit). È ancora argomento di accesa discussione fra i fisici se calcolatori quantistici di dimensioni utili siano fisicamente realizzabili oppure no. Cosa succederebbe se lo fossero? La tesi originale di Church-Turing è ancora verificata dai modelli quantistici: ciò che è calcolabile da un calcolatore quantistico è calcolabile anche da una macchina di Turing. Tuttavia, ci sono alcuni dubbi sulla validità della tesi nella sua forma estesa. Si è infatti trovato almeno un caso di problema interessante in NP, per il quale non si conoscono algoritmi polinomiali, ma che è invece risolubile in tempo polinomiale da un modello quantistico. Il problema è la scomposizione di un numero in fattori primi, di grande importanza ad esempio nella crittografia. Shor [4] ha trovato un algoritmo polinomiale per risolvere il problema con un modello quantistico. Quindi, se effettivamente il problema non fosse in P e se i "calcolatori quantistici" fossero fisicamente realizzabili, la tesi estesa di Church-Turing non sarebbe valida. Tuttavia, sembra che l'impatto di una simile innovazione sulla TCC sarebbe limitato: non sono mai stati trovati altri problemi intrattabili interessanti per cui la computazione basata su bit quantistici potrebbe portare a soluzioni sostanzialmente più efficienti. In particolare, ma nessuno l'ha dimostrato, si sospetta che i modelli basati sui bit quantistici non siano in grado di risolvere in tempo polinomiale i problemi NP-completi, ma solo alcuni problemi in NP che si ritiene non siano in P. L'unico altro caso interessante trovato finora in cui i modelli quantistici migliorano notevolmente l'efficienza rispetto a quelli tradizionali è nel caso della ricerca in un database non ordinato (che comunque è già polinomiale).
Esistono infine numerosi modelli teorici in cui si possono risolvere in tempo polinomiale problemi intrattabili, e altri che possono addirittura risolvere problemi indecidibili per le Macchine di Turing. Tuttavia, l'esistenza di un modello teorico "Super-Turing" non implica che questo sia anche fisicamente realizzabile, e le attuali proposte sembrano assai lontane dalla realtà fisica (ad esempio, alcuni modelli effettuano calcoli con precisione infinita, o usano improbabili operazioni quantistiche non lineari).

8. Complessità spaziale e circuitale

Ci siamo concentrati finora sui limiti legati al tempo di esecuzione degli algoritmi. Simili considerazioni valgono anche considerando lo spazio di memoria o più in generale la dimensione dei circuiti che devono essere costruiti per calcolare un certo algoritmo, come illustrato dal seguente esempio, dovuto a Stockmeyer, di problema intrattabile.
Consideriamo il problema della progettazione di protocolli di comunicazione. I protocolli possono essere inizialmente definiti da un'opportuna formula, utilizzando una notazione (linguaggio) tratta dalla logica matematica. La formula descrive quali sono le sequenze di scambio accettabili fra i partecipanti al protocollo. In base alla formula, si può progettare un circuito digitale che realizza il protocollo dato. Un circuito digitale è definito combinando opportune porte logiche. Una porta logica è un elemento che svolge un'opera-

zione booleana elementare, come per esempio la complementazione di un segnale da "vero" a "falso" o da "falso" a "vero". Le porte logiche sono gli elementi costitutivi di qualunque calcolatore e la loro definizione astratta è indipendente dalla tecnologia con cui sono effettivamente realizzate (attualmente, quella elettronica), che influenza solo il costo, l'ingombro, i consumi e la velocità del circuito, ma non il valore calcolato. È chiaro che, prima di progettare e realizzare fisicamente un circuito che implementa una certa formula, conviene convincersi che la formula stessa sia corretta. Utilizzando uno dei possibili linguaggi logici adatti alla definizione di circuiti digitali, è stato dimostrato che un circuito in grado di verificare la correttezza (più precisamente, la validità) di una formula con al più 616 simboli (che quindi occupa meno di una diecina di righe di questa pagina, dimensione in realtà assai ragionevole per descrivere un protocollo) richiede perlomeno 10^{123} porte logiche: "se anche le porte avessero la dimensione di un protone e fossero connesse da fili infinitamente sottili, il circuito riempirebbe densamente l'intero universo conosciuto" [5].

9. Conclusioni

Lo studio delle limitazioni su quello che può essere realizzato è molto importante in ogni campo dell'ingegneria. Per esempio, la termodinamica è di fondamentale importanza nella costruzione di macchine, poiché stabilisce che non possiamo creare energia dal nulla, che non possiamo realizzare il moto perpetuo e, più in generale, che vi sono limiti invalicabili all'efficienza di una qualunque macchina termica. Allo stesso modo, la teoria della computabilità insegna che alcuni problemi non hanno soluzione algoritmica su una macchina calcolatrice, mentre la teoria della complessità stabilisce che altri problemi non hanno alcuna una soluzione algoritmica *efficiente*.
Questo studio teorico, e soprattutto la questione se P = NP, ha notevoli conseguenze pratiche. Per esempio, quasi tutti i metodi di crittografia a chiave pubblica, usati universalmente per la sicurezza delle transazioni sul World Wide Web, sono basati sull'ipotesi che non esistano algoritmi efficienti per scomporre un numero grande nei suoi fattori primi. Poiché il problema della scomposizione in fattori primi è nella classe NP (per quanto non si sappia se NP-completo), allora se fosse P = NP (per esempio, se esistesse un algoritmo efficiente per risolvere il problema del commesso viaggiatore) esisterebbe anche un algoritmo efficiente per questa scomposizione, e si potrebbero carpire facilmente numeri di carte di credito, conversazioni confidenziali e segreti militari. Come già osservato, per questo problema esiste in realtà un algoritmo efficiente usando modelli di calcolo quantistici, la cui realizzabilità fisica non è tuttavia stata ancora pienamente dimostrata.

Bibliografia

La "bibbia" della NP-completezza, è stata a lungo il volume di Garey e Johnson, ora forse un po' datato. Più recente e generale il testo di Papadimitriou, mentre in italiano si veda Mandrioli e Ghezzi. Il riferimento al sito [6] contiene un esempio di soluzione del problema del commesso viaggiatore. Il libro di Harel [7], infine, è un testo divulgativo sulle limitazioni dei computer, sia per le questioni di indecidibilità che di intrattabilità.

[1] Mandrioli D., Ghezzi C.: *Fondamenti di Informatica Teorica*. UTET, Milano.

[2] Garey M. R., Johnson D. S.: *Computers and Intractability*. W. H. Freeman, San Francisco, 1988 (prima edizione: 1979).

[3] Papadimitriou C.: *Computational complexity*. Addison Wesley Longman, 1994.

[4] Shor P. W.: *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer*. In Proc. 35th Annual Symposium on the Foundations of Computer Science, IEEE Computer Society Press, Los Alamitos, California, 1994, p. 124.

[5] Stockmeyer L.: Classifying the computational complexity of problems. *J. Symbolic logic*, Vol. 52, 1987, p. 1-43.

[6] http://www.tsp.gatech.edu/sweden/index.html

[7] Harel D.: *Computers, Ltd*. Oxford University Press, 2000.

Pierluigi San Pietro è professore straordinario di Ingegneria Informatica presso il Politecnico di Milano ed è docente del corso di Ingegneria del Software. I suoi interessi di ricerca sono rivolti ai metodi e agli strumenti per la specifica e la verifica di sistemi informatici ad alta criticità, e alla teoria degli automi e dei linguaggi formali.
sanpietr@elet.polimi.it