Capitolo settimo

Problemi «facili» e «difficili»

Nel capitolo 5 abbiamo esaminato due algoritmi per ordinare n elementi contenuti in un vettore $C[\,0 : n\text{-}1]$: sono INSERTION_SORT e MERGE_SORT e appartengono all'Olimpo degli algoritmi classici. Il primo, costruito secondo una strategia semplice, ha complessità in tempo di tipo n^2; il secondo, più sofisticato, ha complessità in tempo di tipo $n \log_2 n$ ed è quindi più efficiente del primo.[1]

Poniamo ora che qualcuno ci metta a disposizione un semplice algoritmo per generare tutte le permutazioni di un insieme di n elementi (lo faremo noi nell'ultimo capitolo, per motivi differenti). Una persona molto pigra potrebbe allora utilizzarlo per costruire un nuovo algoritmo di ordinamento in modo semplicissimo: generare una a una, nel vettore C, le permutazioni dei suoi elementi e, per ciascuna di esse, eseguire una scansione di C fino a trovare la permutazione in cui ogni elemento è minore o uguale al sucessivo (ovvero $C[i] \leq C[\,i+1]$ per $0 \leq i \leq n-2$), nel qual caso il vettore risulta ordinato. Algoritmo graziosissimo e orrendo allo stesso tempo perché, come abbiamo visto nel capitolo 6, il numero di permutazioni è $n!$ e questa grandezza cresce in modo esponenziale con n. Dunque l'intero algoritmo ha complessità in tempo esponenziale: lo chiameremo FOOLISH_SORT.

Dal confronto tra i tre algoritmi possiamo trarre alcune consi-

[1] Ricordiamo che la complessità in tempo si riferisce sempre «al caso pessimo» per quanto riguarda la distribuzione dei dati di ingresso. Caso che per INSERTION_SORT si verifica se i dati appaiono in ordine decrescente, mentre per MERGE_SORT qualunque distribuzione iniziale dei dati conduce alla stessa complessità.

derazioni. INSERTION_SORT è meno efficiente di MERGE_-SORT, ma in fondo sono entrambi accettabili se il numero n di dati da ordinare non è molto grande. Ricordiamo infatti che le loro complessità in tempo sono state determinate «in ordine di grandezza», cioè tengono conto del tipo di crescita delle due funzioni ma non di eventuali «coefficienti moltiplicativi». Così per esempio se gli esatti numeri di operazioni eseguite dai due algoritmi fossero rispettivamente $10\,n^2$ e $100\,n\log_2 n$, INSERTION_SORT sarebbe più rapido di MERGE_SORT per $n \leq 32$ (infatti si ha $10 \times 32^2 = 10.240$ e $100 \times 32\log_2 32 = 16.000$). In ogni caso i due algoritmi sono entrambi accettabili perchè al crescere di n la loro complessità in tempo è limitata superiormente da un polinomio in n: ciò è ovvio per la funzione n^2 che è già espressa come un polinomio, ma vale anche per la funzione $n\log_2 n$ perché il logaritmo cresce così lentamente che, per $n \to \infty$, si ha $n\log_2 n < n^{1+\epsilon}$ per un valore di ϵ piccolo quanto si vuole.

Il terzo algoritmo, FOOLISH_SORT, è invece assolutamente inaccettabile benché possa essere costruito senza alcuno sforzo una volta disponibile un programma che generi le permutazioni: infatti esso ha complessità in tempo esponenziale e richiederebbe secoli di calcolo anche per valori di n molto modesti. Non si dimentichi che *la facilità di costruzione di un algoritmo non è in alcun modo correlata alla sua efficienza*, e al miglioramento di quest'ultima deve tendere il progetto algoritmico.

Per concludere si possono in genere individuare diversi algoritmi per risolvere lo stesso problema, di formulazione più o meno semplice ma di diversissima complessità in tempo. Si tratta di cercarne uno ragionevolmente efficiente e auspicabilmente ottimo, ma come vedremo nel seguito non sempre questo è possibile. Gli studi di algoritmica hanno infatti stabilito una fondamentale dicotomia tra problemi:

- *Sono facili (o trattabili) i problemi per cui è noto un algoritmo di complessità in tempo polinomiale nella dimensione dei dati d'ingresso.*

- *Sono difficili (o intrattabili) i problemi per cui non sono noti algoritmi di complessità in tempo polinomiale nella dimensione dei dati d'ingresso.*

Quindi un problema è detto facile o difficile in relazione al tempo impiegato dal miglior algoritmo noto per risolverlo, non al lavoro mentale necessario a scovare un algoritmo di soluzione. L'ordinamento è un problema facile perché può essere risolto in tempo polinomiale, per esempio mediante INSERTION_SORT o MERGE _SORT. La costruzione di tutte le permutazioni di un insieme è banalmente un problema difficile perché l'output ha una dimensione esponenziale e richiede quindi tempo esponenziale per essere generato: ma l'argomento è molto più complicato di così. Ritorneremo su di esso alla fine di questo capitolo, discutendo in modo più semplice che si possa alcuni sviluppi di massima importanza relativi agli studi sulla complessità in tempo degli algoritmi, indicata d'ora in avanti semplicemente come «complessità».

1. *Camminare per una città*

Königsberg, «la montagna dei re», era il nome di una delle più importanti città europee. Capitale della Prussia Orientale era bagnata dal fiume Pregel che, tra due anse, formava l'isola di Kneiphof, centro cittadino. Qui sorgevano la splendida Cattedrale e la famosa *Universitas Albertina* ove insegnarono molti importantissimi studiosi, da Immanuel Kant a David Hilbert.[2]

Riferendosi alla conformazione di questa città Leonhard Euler (Eulero, in italiano) pubblicò nel 1736 un articolo destinato a costituire il fondamento della futura *teoria dei grafi* e a prefigurare la nascita della *topologia*. Esprimendosi in latino che era la lingua della scienza a quell'epoca, Eulero usa la nuova locuzione di *geometria situs*, geometria del luogo: si parla, è vero, di enti geometrici, ma non interessa la forma o la distanza tra essi, solo come sono collegati tra loro. Ed ecco il problema, per altro molto semplice, da cui era partito.

[2] Alla fine della Seconda Guerra Mondiale la città fu rasa al suolo dall'Armata Sovietica e la popolazione tedesca deportata nella DDR (a noi più nota come Germania Est). Oggi con il nome di Kaliningrad fa parte della Russia. Anche il fiume ha cambiato nome. Riportiamo solo in nota questi tristissimi fatti perché vogliamo continuare a parlare della Königsberg di un tempo, più precisamente del XVIII secolo.

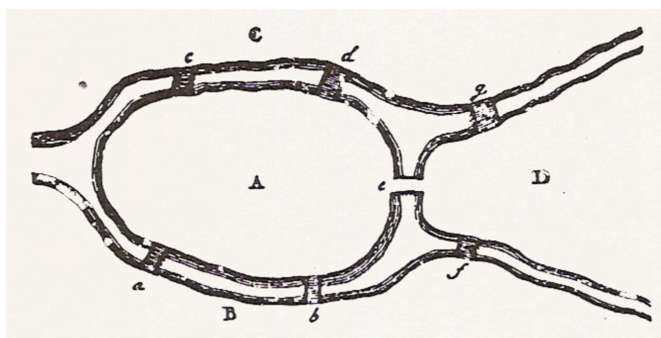


Fig. 7.1. *Schizzo della città di Königsberg con l'isola di Kneiphof (A) e i sette ponti sul Pregel (da un famoso articolo di Eulero del 1736).*

Si trattava sostanzialmente di un gioco che circolava tra la gente. La città di Königsberg ha la conformazione indicata nella figura 7.1, immagine destinata ad apparire poi in tutti i testi di storia della Matematica. Nella città vi sono quattro distretti indicati con A, B, C, D (A è Kneiphof) e sette ponti sul Pregel indicati con a, b, c, d, e, f, g. Ci si chiede se sia possibile fare una passeggiata per la città attraversando tutti i ponti esattamente una volta e tornando al punto di partenza. Eulero affermò che davanti al gioco egli si pose il problema generale di come stabilire se un tale percorso esiste in una città di conformazione arbitraria, con un arbitrario numero di fiumi, isole e ponti. Il teorema che ne segue indica un semplice criterio per decidere per il sì o per il no: a noi interessa anzitutto mostrare che utilizzando questo teorema il problema risulta *semplice*. E a tale scopo descriviamo la città nella Figura 7.2(a) come un *grafo*, cioè come un insieme di nodi (nel caso presente i distretti A, B, C, D indicati con piccoli cerchi) e un insieme di archi che li congiungono (i ponti a, b, c, d, e, f, g, indicati con segmenti di forma qualsiasi). Un percorso che partendo da un nodo torna su se stesso percorrendo tutti gli archi esattamente una volta si chiama *Ciclo Euleriano* del grafo: come vedremo, nel grafo di Königsberg tale ciclo non esiste e quindi la passeggiata è impossibile.

Il risultato generale dimostrato da Eulero può esprimersi in

termini di teoria dei grafi sotto due semplici condizioni che sono implicite trattando dell'assetto di una città:

- Non esistono archi che congiungono un nodo con se stesso.
- Il grafo è *connesso*, ovvero per ogni coppia di nodi esiste sempre una successione di archi che li connette.

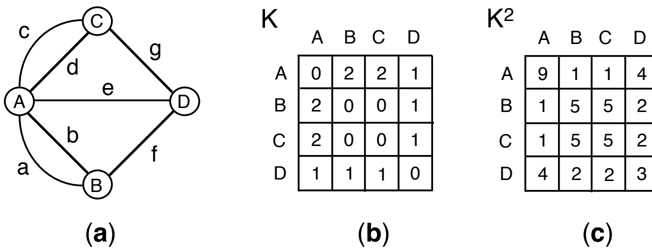
Detto *grado di un nodo* il numero di archi incidenti a esso (per esempio nel grafo di Königsberg il nodo A ha grado 5) abbiamo:

Teorema (Eulero). *Un grafo ammette un Ciclo Euleriano se e solo se tutti i nodi hanno grado pari.*

La condizione necessaria (*solo se*) si dimostra immediatamente, perché se un nodo X ha grado $g = 1$ si può entrare in esso attraverso l'unico arco incidente ma non si può più uscire senza traversare un'altra volta lo stesso arco; e se X ha grado $g > 1$ dispari vi si può entrare e uscire percorrendo $g - 1$ archi diversi, ma quando si entra per l'ultima volta dall'arco rimanente non si può più uscire se non attraverso un arco già percorso. Pertanto il grafo di Königsberg non ammette alcun Ciclo Euleriano perché addirittura tutti i suoi nodi hanno grado dispari (e ne sarebbe bastato uno).

Dimostrare la condizione sufficiente (*se* tutti i nodi hanno grado pari esiste almeno un Ciclo Euleriano) è assai più difficile e in questo risiede il vero interesse del teorema: lo dimostreremo costruttivamente progettando un algoritmo con cui generare un tale ciclo. A tale scopo dobbiamo anzitutto stabilire come i grafi possano essere rappresentati in un calcolatore: un metodo standard fa uso di matrici, di cui ricordiamo anzitutto la definizione.

Sia in Matematica che in Informatica una *matrice a due dimensioni* è un insieme di $n_1 \times n_2$ elementi, ove gli interi n_1, n_2 indicano rispettivamente l'ampiezza degli intervalli in cui possono variare due indici i, j, e un elemento della matrice è individuato con una coppia di valori di tali indici. In Matematica si fa variare l'indice i da 1 a n_1 e l'indice j da 1 a n_2; nel coding si usano in genere gli intervalli da 0 a $n_1 - 1$ e da 0 a $n_2 - 1$, ma ovviamente nulla in sostanza cambia tra i due casi. Una matrice M è dunque indicata nel coding come $M[\,0 : n_1 - 1,\ 0 : n_2 - 1\,]$



K

	A	B	C	D
A	0	2	2	1
B	2	0	0	1
C	2	0	0	1
D	1	1	1	0

K^2

	A	B	C	D
A	9	1	1	4
B	1	5	5	2
C	1	5	5	2
D	4	2	2	3

FIG. 7.2. *(a) La città di Königsberg rappresentata come un grafo. (b) La matrice di adiacenza K che indica, per ogni coppia di nodi, il numero di archi che li connettono. (c) La matrice K^2 che indica, per ogni coppia di nodi, il numero di percorsi di due archi che li connettono.*

e un suo elemento come $M[i, j]$. I vettori, che ci sono già noti, sono matrici a una sola dimensione. E mentre i vettori sono rappresentati in modo naturale con una sequenza di elementi, le matrici a due dimensioni sono rappresentate graficamente con una tabella di n_1 righe e n_2 colonne, anche se nella memoria di un calcolatore queste righe sono allocate in sequenza, una dopo l'altra. Se $n_1 = n_2$ la matrice è detta *quadrata*.

Un metodo molto pratico per rappresentare un grafo di n nodi in un calcolatore, anche se non è il più efficiente in alcune applicazioni, è quello di impegare una matrice quadrata $n \times n$ detta *matrice di adiacenza*, in cui sia le righe che le colonne corrispondono ai nodi e nella cella all'incrocio tra la riga i e la colonna j è riportato il numero di archi che connettono i nodi i e j. Si veda la matrice di adiacenza K per i ponti di Königsberg nella figura 7.2(b) ove per comodità i valori degli indici i e j sono sostituiti dai nomi dei nodi. Per esempio si ha $K[A, B] = 2$ perché i nodi A e B sono connessi con due archi. Notiamo subito che la matrice è simmetrica rispetto alla diagonale principale perché se vi sono k archi che connettono i e j il numero k appare sia in riga i-colonna j che in riga j-colonna i. Quindi si potrebbe conservare nella memoria solo metà della matrice a costo di qualche accorgimento nei programmi. Inoltre la non esistenza di archi che connettono un nodo con se stesso implica che la diagonale principale della matrice contenga solo zeri.

Un'altra osservazione è che nell'accezione corrente un grafo stabilisce l'esistenza o meno di una relazione tra ogni coppia di nodi (per esempio il grafo delle amicizie su Facebook), quindi tra i nodi i e j solo un arco può esistere o non esistere e la matrice contiene solo uni e zeri. In effetti un grafo come quello di Königsberg che contiene più archi tra due nodi è detto in genere *multigrafo* e include il grafo come caso particolare. Nulla cambia nella nostra trattazione se ci si riferisce a uno o all'altro tipo.

Poiché vi sono in genere coppie di nodi non connessi direttamente da un arco, per spostarci da un nodo all'altro consideriamo in genere dei *percorsi* costituiti da uno o più archi in sequenza, ove la lunghezza di un percorso si misura come numero di archi che contiene. Nel grafo di Königsberg non vi è arco tra B e C, ma vi sono ben cinque percorsi di lunghezza due che li connettono: ac, ad, bc, bd, fg. Ma come si calcola questo numero? Esaminando la matrice K vediamo che si può uscire da B con due archi verso A e un arco verso D come indicano i numeri 2 e 1 contenuti nelle celle $K[B, A]$ e $K[B, D]$. Arrivati in A, si va direttamente in C con altri due possibili archi poiché $K[A, C] = 2$: dunque seguendo in alternativa questi archi per ciascuno di quelli che hanno portato da B ad A si ottengono i primi $2 \times 2 = 4$ percorsi di lunghezza 2 elencati sopra. Seguendo invece l'arco che conduce da B a D si può raggiungere C lungo un solo arco perché $K[D, C] = 1$, e si ottiene così il quinto percorso indicato. Generalizziamo ora questo ragionamento in termini matematici impiegando una trattazione informatica per le matrici.

Anzitutto notiamo che matrice del grafo contiene lo stesso numero di righe e colonne ma in algebra questi due parametri possono essere fissati indipendentemente. Prese due matrici $A[0 : m-1, 0 : n-1]$ e $B[0 : n-1, 0 : p]$ con m righe e n colonne la prima, n righe e p colonne la seconda, si definisce il *prodotto* $A \times B$ come una matrice $C[0 : m-1, 0 : p-1]$ i cui elementi sono dati dalla relazione:

$$C[i, j] = \sum_{k=0}^{n-1}(A[i, k] \times B[k, j]) \qquad (1)$$

Cioè ogni elemento $C[i, j]$ della matrice prodotto è ottenuto «combinando» l'intera riga i di A con l'intera colonna j di B come somma dei prodotti tra l'elemento k-esimo della riga i e l'elemento

k-esimo della colonna j, per $0 \leq k \leq n-1$. Questa somma di prodotti prende il nome di *prodotto scalare* tra il vettore riga i e il vettore colonna j. Si noti che il prodotto $A \times B$ è definito solo se il numero di colonne di A è uguale al numero di righe di B, entrambi n nel nostro esempio.

La formula (1) viene messa direttamente in opera nel semplicissimo programma di figura 7.3 per calcolare la matrice C, il cui funzionamento non dovrebbe aver bisogno di spiegazioni. Per quanto riguarda la sua complessità notiamo che ogni prodotto scalare tra due vettori richiede tempo proporzionale a n (ciclo **for** sull'indice k) e che si eseguono $m \times p$ moltiplicazioni (due cicli **for** «esterni»), quindi la complessità totale è proporzionale al prodotto $m \times n \times p$, ovvero è cubica se questi tre valori sono paragonabili tra loro.

```
programma PRODOTTO_MATRICI (A, B, C)
// input: matrici A e B di dimensioni m × n e n × p
// output: matrice C = A × B di dimensioni m × p
    for i = 0 to m − 1
        for j = 0 to p − 1
            S ← 0;
            for k = 0 to n − 1
                S ← S + A[i, k] × B[k, j];
            C[i, j] ← S;
```

FIG. 7.3. *Programma per calcolare il prodotto di due matrici.*

Per operare su un grafo di n nodi indichiamo questi con gli interi da 0 a $n-1$ e rappresentiamo il grafo con la sua matrice di adiacenza $M[0:n-1, 0:n-1]$, ove il nodo k corrisponde sia alla riga che alla colonna k, $0 \leq k \leq n-1$, e ogni cella $M[i,j]$ contiene il numero di archi che connettono i nodi i e j. Notiamo ora che il quadrato M^2 della matrice M si costruisce come caso particolare del prodotto di matrici, $M^2 = M \times M$, ove i numeri di righe e colonne sono tutti uguali. Per cui anche la matrice $M^2[0:n-1, 0:n-1]$ ha dimensioni $n \times n$ e ogni suo elemento è ottenuto come prodotto scalare tra la riga di M

corrispondente al nodo i e la colonna di M corrispondente al nodo j.

Nella Figura 7.2(c) è riportato il quadrato K^2 della matrice di Königsberg ove righe e colonne sono di nuovo indicate con i nomi dei nodi. Per esempio il prodotto scalare tra la riga B e la colonna C di K produce il risultato $2\times2+0\times0+0\times0+1\times1 = 5$ che appare in $K^2[B, C]$. È esattamente il numero di percorsi di lunghezza 2 tra B e C trovato sopra con un ragionamento sul grafo di figura 7.2(a): il lettore potrà immediatamente verificare che tale ragionamento è interpretabile matematicamente come l'esecuzione del prodotto scalare tra la riga B e la colonna C. La matrice K^2 indica dunque il numero dei cammini lunghi due per ogni coppia di nodi di K, ed è ovviamente anch'essa simmetrica rispetto alla diagonale principale i cui elementi, in questo caso, possono essere diversi da zero perché da un nodo si può tornare sullo stesso seguendo più archi. Per esempio il valore 9 nella prima cella indica che ci sono nove percorsi di due archi che iniziano e terminano in A: essi sono *aa*, *bb*, *ab*, *ba*, *cc*, *dd*, *cd*, *dc*, *ee*. Un'estensione di questo ragionamento, che invitiamo il lettore a intraprendere da solo, ci permette di enunciare una proprietà generale che risulterà particolarmente significativa nel prossimo capitolo in cui parleremo di motori di ricerca:

Proprietà 1. *La matrice ottenuta come ℓ-esima potenza della matrice di adiacenza di un grafo, per $\ell \geq 1$, contiene i numeri dei percorsi lunghi ℓ tra tutte le coppie di nodi del grafo.*

In particolare per $\ell = 1$ si ha la matrice di partenza relativa ai singoli archi, cioè ai percorsi lunghi 1.

Siamo ora finalmente in grado di ragionare in modo algoritmico sul teorema di Eulero e quindi sull'esistenza di un Ciclo Euleriano in un grafo. Anzitutto controlliamo che il grafo sia connesso (altrimenti il Ciclo Euleriano non può esistere) attraverso il programma CONNESSIONE di figura 7.4 che inserisce in un vettore $R[0 : n\text{-}1]$ i nodi via via raggiungibili dal nodo 0 e controlla che alla fine R contenga tutti i nodi. L'algoritmo, che in gergo realizza una visita *breadth first* («in ampiezza») del grafo, è descritto con qualche licenza nello pseudocodice per renderlo più comprensibile: in particolare introduciamo i costrutti **foreach** («per ogni») e **not-in** («non contenuto») che richie-

dono l'esame di tutti gli elementi di un insieme e, nel caso di **not-in**, anche del controllo di appartenenza all'insieme. Tali costrutti non sono in genere direttamente disponibili nei linguaggi di programmazione ma possono essere semplicemente realizzati con un ciclo **for** o **while**, per esempio trasformando in Python il programma CONNESSIONE.

```
programma CONNESSIONE (M)
// M: matrice di adiacenza M per un grafo di n nodi
// R: vettore di n celle per registrare i nodi raggiungibili da 0
1.     R[0] ← 0; s ← 0; f ← 0;
2.     while s ≤ f     // f indica l'ultima cella occupata di R
3.     i ← R[s];                    // i indica il nodo esaminato
4.         foreach (M[i,j] > 0, 1 ≤ j ≤ n − 1)
             // cioè per ogni elemento > 0 nella riga i
5.             if j not-in R[0 : f]
               // se il nodo j non è tra quelli già inseriti in R
6.                 f ← f + 1; R[f] ← j;
7.                 if f = n − 1
8.                     print "il grafo è connesso";
9.                     stop;
10.         s ← s + 1;
11.    print "il grafo non è connesso";
```

FIG. 7.4. *Programma per decidere se un grafo di matrice $M[0 : n-1, 0 : n-1]$ è connesso, attraverso una sua visita in ampiezza.*

Per comprendere il programma si noti che si parte dal nodo 0 posto in $R[0]$ e si utilizzano due indici s e f che puntano a celle di R: in particolare s punta alla cella contenente il nodo corrente da cui si cercano i nodi raggiungibili con un arco, e f punta all'ultima cella di R in cui è stato inserito un nodo. A partire dal valore dell'indice $s = 0$, e per tale valore via via crescente nel ciclo di **while** delle righe 2-10, si esaminano con il costrutto **foreach** le celle corrispondenti ai nodi connessi con uno o più archi

al nodo i della matrice M (ove $i = R[s]$): cioè i nodi j per cui $M[i, j] > 0$. Ciascuno di tali nodi non già presente in R (controllo alla riga 5 del programma) vi è inserito (riga 6). Se con questa operazione si raggiunge la cella $R[n-1]$ l'algoritmo termina con successo (righe 7-9) poiché n (cioè tutti i) nodi sono stati inseriti in R e quindi raggiunti nelle diverse visite. Se ciò non accade l'indice s del **while** viene incrementato di 1 (riga 10): se risulta $s \leq f$ il ciclo viene ripetuto per il nuovo nodo $R[s]$; se invece s supera il valore di f, che indica l'ultima cella di R in cui è stato inserito un nodo, il **while** si esaurisce e l'algoritmo termina con insuccesso (riga 11) perché non si possono raggiungere altri nodi e quelli raggiunti sono meno di n.

Una limitazione superiore alla complessità dell'algoritmo si calcola facilmente notando che, se il calcolo termina alle righe 8-9, tutte le righe i della matrice M vengono esaminate nel ciclo **foreach** (riga 4) e questo è ripetuto n volte nel ciclo di **while** (riga 2), per un numero totale di n^2 operazioni; inoltre l'esame dell'elemento j contenuto in ciascuna cella di M può richiedere fino a n operazioni nel ciclo implicito nel costrutto **not-in** (riga 5). Dunque le operazioni complessive sono al più n^3. La complessità potrebbe essere valutata più esattamente e lo stesso algoritmo potrebbe essere migliorato, ma al momento ci interessa mostrare che il controllo di connessione del grafo può essere eseguito in tempo polinomiale.

Stabilita la connessione del grafo possiamo rispondere alla domanda di esistenza o meno di un Ciclo Euleriano controllando che tutti i nodi abbiano grado pari: operazione eseguita con il programma GRADI di figura 7.5 che restituisce in output anche un vettore $G[0 : n-1]$ ove vengono inseriti i gradi dei nodi se questi sono tutti pari.

Come si vede facilmente il programma scandisce la matrice di adiacenza per righe (ciclo **for** sull'indice i) sommando tutti i numeri di archi specificati sulla riga i (ciclo **for** sull'indice j) per calcolare il grado g del nodo i. Se g è dispari per un qualsiasi nodo i il programma si arresta segnalando il fatto; se ciò non accade il programma controlla l'intera matrice, segnala quindi che tutti i nodi hanno grado pari e li memorizza nel vettore G ove $G[i]$ è il grado del nodo i. L'algoritmo esegue il massimo numero di operazioni quando viene esaminata l'intera matrice, quindi la sua complessità è al più proporzionale a n^2 perché tante sono le

```
programma GRADI (M)
// M è la matrice di adiacenza di un grafo di n nodi
// G è un vettore di output che contiene i gradi dei nodi
    for i = 0 to n − 1
        g ← 0;                  // g indicherà il grado del nodo i
        for j = 0 to n − 1
            g ← g + M[i, j]; // calcolo del grado g per il nodo i
        if g è dispari
            print "Il grafo contiene un nodo di grado dispari";
            stop ;
        else G[i] ← g;
    print "Tutti i nodi del grafo hanno grado pari";
```

FIG. 7.5. *Controllo della parità del grado di tutti i nodi di un grafo.*

celle esaminate e ciascun esame richiede tempo costante.

Concludendo possiamo affermare, sulla base del teorema di Eulero, che il problema di stabilire se un grafo contiene un Ciclo Euleriano può essere risolto eseguendo i due programmi CONNESSIONE e GRADI in tempo proporzionale a n^3 nel caso più sfavorevole (si noti che prevale la complessità di CONNESSIONE). Quindi questo è un problema *facile*, ma i due programmi non ci mostrano quale sia un tale ciclo. Per questo scopo dobbiamo progettare un algoritmo per *costruire* un Ciclo Euleriano, che sia valido per qualsiasi grafo connesso con tutti i nodi di grado pari. L'algoritmo costituirà una dimostrazione della condizione di sufficienza specificata nella parte «se» del Teorema di Eulero: un tipo di dimostrazione diversa da quelle a cui siamo abituati perché ha un carattere «costruttivo».

Dell'algoritmo che proponiamo, di nome CICLO_EULERIANO, riportiamo solo una descrizione a parole perchè il programma non è semplice. I lettori più interessati all'argomento potranno redigerlo e analizzarlo da soli tenendo conto dei punti seguenti.

- Indichiamo con n e m il numero di nodi e di archi del grafo.

Notiamo a questo proposito che nei grafi che ammettono al più un arco tra ogni coppia di nodi risulta $m \leq n\,(n-1)/2$ e che l'eguaglianza si verifica se il grafo è *completo*, cioè vi è un arco per qualunque coppia di nodi: infatti il termine $n\,(n-1)$ dipende dal fatto che ognuno degli n nodi è connesso agli altri $n-1$ con un arco, e la divisione per due dipende dal fatto che ogni arco viene contato due volte per ogni coppia. Ma in multigrafi come quello di Königsberg ove possono esistere più archi che connettono la stessa coppia di nodi, m è indipendente da n e può essere molto maggiore di esso. Mentre la complessità degli algoritmi precedenti dipendeva solo da n per qualsiasi grafo, ora dipenderà crucialmente da m perché tutti gli archi devono essere inseriti nel ciclo. Poiché ogni nodo ha grado pari il numero di archi incidenti a tutti i nodi è almeno $2n$ e si ha $m \geq 2n/2 = n$ perché ogni arco è contato due volte per ogni coppia di nodi connessi. Dunque la complessità sarà ora espressa in funzione di m.

- Un ciclo in un grafo, ovvero un percorso che inizia e termina nello stesso nodo (anche se non contiene tutti i nodi del grafo), sarà espresso come la sequenza ordinata dei nodi incontrati, con l'intendimento che l'ultimo è connesso al primo con un arco che fa parte del ciclo. Un Ciclo Euleriano potrà contenere più volte uno stesso nodo N se il grado di N è almeno quattro ma non può percorrere due volte lo stesso arco. Per esempio nel grafo di figura 7.6 la sequenza di nodi $E\,A\,C$ indica il ciclo che inizia e termina in E percorrendo gli archi f, b, g; la sequenza $A\,C\,E\,A\,B\,C\,D$ indica il ciclo che inizia e termina in A percorrendo gli archi b, g, f, c, d, e, a, ove i nodi A e C di grado quattro vi compaiono due volte: in particolare questo è un Ciclo Euleriano perché contiene tutti gli archi.

- L'algoritmo CICLO_EULERIANO costruisce un percorso sul grafo *marcando* ogni arco attraversato perché non sia riutilizzato in seguito. Gli archi non marcati si dicono *liberi*. Il percorso inizia da un nodo qualsiasi, diciamo N_1, e si sviluppa secondo una sequenza $N_1\,N_2\,N_3 \cdots$ scegliendo da ogni nodo N_i un arco non marcato che conduce a N_{i+1}, marcando questo arco e proseguendo finché si

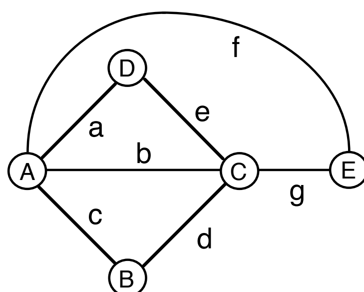


FIG. 7.6. *Un grafo che contiene un Ciclo Euleriano $A\,C\,E\,A\,B\,C\,D$ (con ritorno in A) ma non contiene un Ciclo Hamiltoniano.*

incontra un nodo N_{k+1} che non ha archi liberi incidenti. Come ora dimostriamo questo implica che sia $N_1 = N_{k+1}$, cioè $N_1\,N_2\,\ldots\,N_k$ è un ciclo (non necessariamente euleriano).

- Per dimostrare che $N_1 = N_{k+1}$ notiamo che all'inizio del percorso viene marcato l'arco che conduce da N_1 ad N_2, quindi il numero di archi liberi incidenti in N_1 diventa dispari; per tutti i nodi successivi del percorso vengono marcati sia l'arco entrante che l'arco uscente, quindi il numero di archi liberi incidenti ciascuno di questi nodi diventa zero (se il grado del nodo era 2) e il nodo stesso non sarà più incontrato, o diventa un numero pari (se il grado era 4, 6, ecc.) e il nodo sarà eventualmente incontrato ancora nel percorso diminuendo di due il grado per ogni attraversamento. Anche il nodo N_1 sarà incontrato più volte se il suo grado era almeno 4, ma è l'unico in cui il numero di archi liberi era dispari e può quindi divenire zero quando sarà raggiunto da un altro nodo, dunque dal nodo N_k.

- Notiamo ora che il numero di archi percorsi in un ciclo (cioè quelli marcati) è uguale al numero di nodi della sequenza che lo rappresenta. Dunque se nel ciclo $N_1\,N_2\,\ldots\,N_k$ si ha $k = m$, ossia il numero totale di archi del grafo, si tratta di un Ciclo Euleriano e l'algoritmo termina. Altrimenti deve esistere almeno un nodo N_i nel ciclo che ha an-

cora archi liberi incidenti. Infatti possono presentarsi due casi:

1. se un nodo N' del grafo non appare nel ciclo, almeno un nodo N_i del ciclo deve avere archi liberi incidenti perché N' possa essere raggiunto dai nodi del ciclo, altrimenti il grafo non sarebbe connesso;

2. se tutti i nodi del grafo appaiono nel ciclo, gli $m-k$ archi liberi non appartenenti al ciclo devono essere incidenti a qualche nodo del ciclo.

Si percorre allora il ciclo a partire da N_i anziché da N_1, cioè si riscrive come $N_i\, N_{i+1}\, \ldots\, N_k\, N_1\, \ldots\, N_{i-1}$ e si prosegue il ciclo con uno degli archi liberi incidenti N_i, iterando il procedimento finché tutti gli archi sono marcati.

Nell'esempio di figura 7.6, trovato il ciclo $E\,A\,C$ non si può procedere ulteriormente perché i due archi f, g incidenti a E sono stati marcati. Si riscrive allora il ciclo come $A\,C\,E$ con gli archi b, g, f marcati e si prosegue da A attraverso gli archi liberi c, d, e, a ottenendo il Ciclo Euleriano $A\,C\,E\,A\,B\,C\,D$.

Per valutare la complessità dell'algoritmo notiamo anzitutto che il Ciclo Euleriano risultante è rappresentato da una sequenza lunga m (numero degli archi del grafo) quindi m è un limite inferiore al numero di operazioni richieste. Deve inoltre risultare $m \geq n-1$ (e può risultare $m \gg n$) altrimenti il grafo non sarebbe connesso: quindi m sarà il parametro essenziale di riferimento nella valutazione della complessità. Dalla struttura dell'algoritmo risulta chiaro che, se il primo ciclo trovato $N_1\, N_2\, \ldots\, N_k$ è la soluzione, l'algoritmo ha una complessità proporzionale a m, altrimenti questa dipende in modo cruciale da come si esegue la ristrutturazione del ciclo per prolungarlo e da quante volte questa ristrutturazione viene eseguita.

La soluzione più semplice, anche se non la più efficiente, è scandire ogni ciclo che si blocca su un nodo privo di archi liberi alla ricerca di un nodo che ne abbia ancora, e riscriverlo a partire da questo nodo come indicato sopra: poiché ogni ciclo è lungo al massimo m tale operazione richiede al massimo m passi. E poichè il numero totale di cicli che possono essere via via costruiti per ristrutturazione è anch'esso al massimo m (questo accade se a ogni ristrutturazione si aggiunga solo un nodo) la complessità

dell'algoritmo nel caso peggiore è proporzionale a m^2. Dunque anche il problema di *costruire* un Ciclo Euleriano è facile.

Concluso il discorso sulle conseguenze del Teorema di Eulero consideriamo ora un problema apparentemente molto simile ma di diversissima natura computazionale apparso più di un secolo dopo. Nel 1859 William Rowan Hamilton propose un gioco matematico fisicamente realizzato con un dodecaedro di legno ai cui venti vertici erano associati nomi di città diverse, con la richiesta di muoversi lungo gli spigoli del poliedro per attraversare tutte le città esattamente una volta e tornare a quella di origine. Così posto il gioco, come quello originale dei ponti di Könisberg, era associato a un esempio specifico, ma noi lo proporremo in versione generale nei termini della teoria dei grafi poiché i vertici e gli spigoli di un poliedro possono essere rispettivamente associati ai nodi e agli archi di un grafo.

Considerando dunque un grafo arbitrario H ci chiediamo se esiste un *Ciclo Hamiltoniano*, cioè un ciclo che traversi tutti i nodi di H esattamente una volta. Pare in un certo senso il duale del problema di Eulero, ma la sua natura è completamente diversa dal punto di vista del calcolo. Nessuno, dai tempi di Hamilton, ha trovato un criterio generale ragionevolmente semplice per rispondere alla questione, e tantomeno un algoritmo per costruire un tale ciclo se esiste, senza sostanzialmente provare a costruire cicli in tutti i modi possibili. Finché, nel 1971, un risultato matematico fondamentale ha permesso di chiarire i termini della questione come vedremo nel prossimo paragrafo. Prima di addentrarci su questo punto, però, cerchiamo di acquisire qualche dimestichezza col problema.

Un criterio ragionevole per studiare l'esistenza di un Ciclo Hamiltoniano deve ovviamente essere applicabile a qualsiasi grafo. In assenza di tale criterio, per qualche esempio potrebbe essere facile rispondere, in specie se il ciclo esiste, ma per altri sarebbe difficilissimo. Nel grafo di Königsberg di figura 7.2 l'esistenza del ciclo $A\,C\,D\,B$ salta immediatamente all'occhio. Nel grafo di figura 7.6 un tale ciclo non esiste, ma per stabilirlo occorre qualche ragionamento anche se il grafo è molto piccolo.

Come detto si può risolvere il problema con un algoritmo *enumerativo* che prova a costruire il ciclo in tutti i modi possibili su un grafo arbitrario. Lo descriviamo a parole.

- Sia H un grafo arbitrario e siano $N_1, N_2, \ldots, N_n$ i suoi nodi.
- Anzitutto, come nel caso del Ciclo Euleriano, condizione per l'esistenza di un Ciclo Hamiltoniano è che il grafo H sia connesso. E questo, come sappiamo, si decide in tempo polinomiale con l'algoritmo CONNESSIONE.
- Come già ammesso per l'algoritmo FOOLISH_SORT, poniamo di disporre di un algoritmo PERMUTAZIONI che costruisce una a una tutte le permutazioni di un insieme arbitrario di h elementi. Come sappiamo la sua complessità è certamente esponenziale in h poiché tutte le $h!$ permutazioni devono essere generate per poterle controllare.
- Per costruire un ciclo possiamo partire da un nodo qualsiasi, diciamo N_1. Applichiamo l'algoritmo PERMUTAZIONI all'insieme $N_2, N_3, \ldots, N_n$ e concateniamo con N_1, una a una, le permutazioni prodotte dall'algoritmo, ottenendo tutte le permutazioni dei nodi di H che iniziano con N_1. Indichiamo con $N'_1, N'_2, \ldots, N'_n$ una qualunque di queste permutazioni che sono in totale $(n-1)!$ (N'_1 sarà sempre uguale a N_1).
- Scandiamo ciascuna permutazione per controllare via via se esiste un arco che connette N'_i con N'_{i+1}. Se l'arco non esiste si arresta la scansione e si passa alla permutazione successiva. Se esistono gli archi fino a quello che connette N'_{n-1} con N'_n, controlliamo l'esistenza dell'arco tra N'_n e N'_1. Se anche questo esiste la permutazione corrisponde a un Ciclo Hamiltoniano; altrimenti si passa alla permutazione successiva. Se nessuna permutazione genera un ciclo, il grafo non contiene un Ciclo Hamiltoniano.

Ovviamente questo programma ha complessità esponenziale perché, a parte il tempo per scandire ogni permutazione, queste sono $(n-1)!$ in numero.

Si può osservare a questo punto che il metodo proposto risolve un problema più generale di quanto richiesto perché è volto a costruire un Ciclo Hamiltoniano anziché decretarne semplicemente l'esistenza. Ma in realtà non si conosce algoritmo *essenzialmente* più efficiente (cioè polinomiale) per rispondere anche