ROBERT SEDGEWICK

Cẩm nang Thuật Toán

NHÀ XUẤT BẢN KHOA HỌC VÀ KỸ THUẬT

ROBERT SEDGEWICK

CẨM NANG
THUẬT TOÁN

Tập 2 : CÁC THUẬT TOÁN CHUYÊN DỤNG

(In lần thứ 5)

Người dịch : TRẦN ĐAN THƯ
BÙI THỊ NGỌC NGA
NGUYỄN HIỆP ĐOÀN
ĐẶNG ĐỨC TRỌNG
TRẦN HẠNH NHI

Hiệu đính : GS.TS. HOÀNG KIẾM

NHÀ XUẤT BẢN KHOA HỌC KỸ THUẬT

Algorithms

Robert Sedgewick, Princeton University (USA)

2nd Edition

Addison-Wesley Publishing Co.

LỜI NÓI ĐẦU

Trước khi viết một chương trình máy tính, cho dù đơn giản nhất, bất cứ ai, dù ở trình độ nào, cũng đều phải suy tư ít nhiều về THUẬT TOÁN. Trong tổng thể kiến thức về TIN HỌC, các Thuật toán (Algorithms) cùng với Cấu trúc Dữ liệu (Data Structure) được xem là những tri thức quan trọng hàng đầu và không thể thiếu đối với bất kỳ người lập trình (ứng dụng hay hệ thống) nào muốn đạt mục đích với hiệu quả cao nhất.

Tuy nhiên, cho đến nay, trong hầu hết các ấn phẩm và giáo trình Tin học, các thuật toán đều trình bày ở dạng quá "nôm na" hay rườm rà qua vài ví dụ bằng lời hay các lưu đồ (flowcharts); hoặc lại quá trừu tượng khi dùng đến các khái niệm của lý thuyết và độ phức tạp thuật toán. Do đó, người đọc cũng như người lập trình đều cảm thấy thiếu các căn cứ tin cậy về các thuật toán - những "điểm tựa" vững để tạo ra thế giới các chương trình, phần mềm ... như thể nhà bác học Archimède cổ xưa từng mơ ước dùng cho "đòn bẩy" nâng bổng cả trái đất. Các khiếm khuyết đó đã được khắc phục trọn vẹn trong cuốn sách này. Nó được dịch trọn từ nguyên bản tiếng Anh cuốn "Algorithms" [của Robert Sedgewick, Princeton University (USA), Second Edition], do Addison-Wesley Publishing Co. tái bản nhiều lần; và ngày nay đã trở thành một trong các tư liệu "kinh điển" cả về lý thuyết lẫn thực hành, cho người lập trình trên thế giới.

Mục tiêu chủ chốt của cuốn sách này là tổng hợp có hệ thống các phương pháp cơ bản, từ nhiều lãnh vực ứng dụng riêng biệt, nhằm cung cấp các giải thuật tốt nhất đã được kiểm chứng và công bố, để giải các bài toán cụ thể bằng máy tính.

Cuốn sách gồm 45 chương, chia thành tám phần theo các phạm trù ứng dụng lớn. Các chương được trình bày theo trình tự từ căn

bản đến cao cấp. Tuy vậy, bạn đọc có thể sử dụng từng phần tương đối độc lập, theo nhu cầu riêng của mình.

Cuốn sách được dịch và xuất bản thành hai tập:

- Tập I (đã xuất bản) : Các thuật toán thông dụng, gồm 23 chương, thuộc bốn phần đầu của nguyên bản;
- Tập II (tập này) : Các thuật toán chuyên dụng, gồm 20 chương trong bốn phần cuối của nguyên bản [các chương 44 và 45 đã kê ở mục lục của Tập I chúng tôi không đưa vào Tập II vì xét thấy không thiết yếu cho đông đảo bạn đọc - *Nhóm biên dịch*].

Cuốn sách được dùng cho các sinh viên ngành Tin học hoặc các đối tượng khác đã làm quen với máy tính và có kỹ năng nhất định về lập trình. Họ cần biết vận dụng một ngữ trình nào đó, ít nhất là Pascal chuẩn, vì các thuật toán ở đây được trình bày dưới dạng "tựa Pascal"). Sách cũng hữu ích cho các thày và trò các trường Trung học hệ đào tạo chuyên Tin.

Với tư cách một tài liệu tham khảo nghiêm túc, cuốn sách cũng hữu ích cho các cán bộ nghiên cứu ngành khác, hay cho những ai có nhu cầu phát triển các phần mềm hệ thống hoặc các trình ứng dụng; bởi lẽ, ngoài nội dung thuật toán nó còn cung cấp chi tiết các thông tin hữu dụng về cài đặt và hiệu quả sử dụng chúng.

Mặc dù nhóm dịch thuật đã cố gắng bám sát nguyên bản, song không khỏi còn thiếu sót (nhất là về các thuật ngữ tiếng Việt còn nhiều bàn cãi); song chúng tôi và Nhà Xuất Bản vẫn hy vọng cuốn sách sẽ đáp ứng đúng nhu cầu và được đông đảo bạn đọc quan tâm đến Tin Học đón nhận, như một cuốn Cẩm Nang tra cứu.

TP.HCM, 30/4/1995
Gs.Ts. HOÀNG KIẾM
Cv.Ks. NGUYỄN PHÚC TRƯỜNG SINH

24

CÁC PHƯƠNG PHÁP HÌNH HỌC CƠ BẢN

Máy vi tính ngày càng được sử dụng nhiều hơn để giải quyết các vấn đề có quy mô lớn về hình học. Các đối tượng hình học như điểm, đường thẳng, đa giác là nguồn gốc của một tập đáng kể các bài toán và các thuật toán.

Các thuật toán hình học rất quan trọng trong các hệ thống phân tích và thiết kế kiểu đối tượng vật lý. Một thiết kế viên làm việc với các đối tượng vật lý có trực giác hình học khó có thể đáp ứng trong máy vi tính. Nhiều áp dụng khác liên quan trực tiếp đến việc xử lý dữ liệu hình học. Nhiều ứng dụng khác trong toán và thống kê, về các lĩnh vực mà trong đó có nhiều loại vấn đề có thể đưa ra một cách tự nhiên bằng hình tượng hình học.

Hầu hết các thuật toán mà chúng ta đã nghiên cứu đều tập trung vào văn bản và các con số, chúng được thiết kế và xử lý sẵn trong phần lớn các môi trường lập trình Thực vậy, các phép toán cơ bản cần thiết đều được cài đặt trong phần cứng của đa số các hệ máy tính. Chúng ta sẽ thấy đối với các vấn đề hình học thì tình huống khác hẳn, ngay cả các phép toán sơ cấp trên điểm và đoạn thẳng cũng có thể là một thách thức về tính toán.

Các vấn đề hình học thì dễ hình dung, nhưng chính điều đó lại có thể là một trở ngại. Nhiều bài toán có thể giải quyết ngay lập tức bằng cách nhìn vào một mảnh giấy (ví dụ như: một điểm có nằm trong một đa giác hay không) lại đòi hỏi những chương trình không đơn giản. Đối với các vấn đề phức tạp hơn, trong nhiều ứng dụng khác nhau, phương pháp dùng trên máy vi tính có thể hoàn toàn khác với phương pháp sử dụng bởi con người.

Bản chất thiết kế của hình học cổ điển và các ứng dụng hữu ích đã rất phổ biến làm cho người ta có thể hoài nghi rằng các thuật toán hình học phải có một lịch sử lâu dài. Nhưng thật sự hầu hết các công việc trong lãnh vực này lại hoàn toàn mới. Tuy nhiên, công trình của các nhà toán học cổ điển lại thường rất hữu ích trong việc xây dựng các thuật toán cho máy tính hiện đại. Các giải thuật hình học là một lãnh vực rất đáng nghiên cứu bởi vì ngữ cảnh lịch sử mạnh mẽ của nó, vì các thuật toán căn bản vẫn còn đang được phát triển và vì nhiều ứng dụng quy mô lớn, quan trọng cần các thuật toán này.

ĐIỂM, ĐOẠN THẲNG VÀ ĐA GIÁC

Hầu hết các chương trình chúng ta sẽ nghiên cứu thực hiện trên các đối tượng hình học đơn giản trong không gian hai chiều, tuy nhiên chúng ta cũng sẽ xem xét một số thuật toán khác trong không gian nhiều chiều. Đối tượng cơ sở là một điểm, mà chúng ta sẽ xét bằng một cặp số nguyên - tọa độ của điểm đó trong hệ trục tọa độ Descart thường dùng. Một đoạn thẳng là một cặp điểm được nối với nhau bởi một phần của đường thẳng. Một đa giác là một danh sách các điểm, với hai điểm cạnh nhau được nối bởi một đoạn thẳng, và điểm đầu nối với điểm cuối tạo thành một hình đóng.

Làm việc với các đối tượng hình học này, chúng ta cần phải quyết định thể hiện chúng như thế nào. Thông thường, ta dùng một mảng để biểu diễn một đa giác, dù rằng trong một số trường hợp chúng ta

có thể dùng danh sách liên kết hay các kiểu khác. Hầu hết các chương trình của chúng ta ở đây sẽ dùng kiểu lưu trữ đơn giản, dễ hiểu như sau:

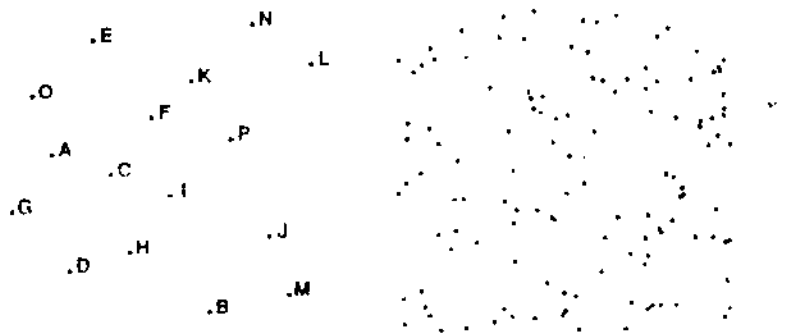
```
type point = record x , y : integer end;
   line = record p1, p2 : point end;
var polygon : array [0..Nmax] of point;
```

Chú ý rằng các điểm được giới hạn là có tọa độ nguyên. Cũng có thể sử dụng kiểu real (số thực) nhưng dùng tọa độ nguyên làm cho các thuật toán hiệu quả và đơn giản hơn, mà chúng cũng không đến nỗi hạn chế như ta tưởng. Như đã đề cập ở chương 2, làm việc với các số nguyên khi có thể chính là sự tiết kiệm thời gian đáng kể trong nhiều môi trường tính toán, bởi vì các phép tính số nguyên thường hiệu quả hơn các phép tính trên kiểu chấm động. Do đó, chúng ta sẽ sử dụng các số nguyên khi có thể để không phải làm phức tạp hoá vấn đề.

Nhiều đối tượng hình học phức tạp sẽ được biểu diễn dựa trên các thành phần cơ sở này. Ví dụ như đa giác là một mảng các điểm. Để ý rằng nếu dùng một mảng các đoạn thẳng sẽ làm cho mỗi đỉnh của đa giác được lưu trữ hai lần (dù rằng cách này có thể làm biểu diễn tự nhiên hơn trong một số thuật toán). Trong một vài ứng dụng, để tiện lợi hơn, ta cũng có thể thêm thông tin cho mỗi điểm hay đoạn thẳng bằng cách thêm vào trường *info* trong record.

Chúng ta sẽ dùng tập hợp các điểm trong hình 24.1 trang sau để minh họa cho thao tác của nhiều thuật giải hình học.

Trong đó 16 điểm bên trái được đánh dấu bằng các ký tự đơn để giải thích các ví dụ, chúng có các tọa độ nguyên trong hình 24.2 (các ký tự gán cho các điểm theo thứ tự khi nhập điểm vào). Chương trình thường không tham chiếu các điểm bằng tên, chúng chỉ đơn giản lưu trữ chúng trong một mảng và tham chiếu đến bằng chỉ số



Hình 24.1 *Các tập điểm mẫu cho những thuật toán hình học*

tương ứng. Thứ tự các điểm trong mảng có thể là quan trọng trong một số chương trình, việc sắp thứ tự các điểm theo một kiểu nào đó cũng là mục tiêu của một số thuật giải hình học. Bên phải hình 24.1 là 128 điểm, được tạo ra một cách ngẫu nhiên với toạ độ nguyên trong khoảng từ 0 đến 1000.

Một chương trình tiêu biểu dùng một mảng *p[1..N]* các điểm, và đơn giản nhập vào *N* cặp số nguyên, gán cặp thứ nhất cho x và y của p[1], cặp thứ hai cho *p[2]*,... Khi p biểu diễn một đa giác, để thuận lợi hơn thì thường dùng thêm các giá trị "lính canh": *p[0]=p[N]* và *p[N+1]=p[1]*

GIAO CÁC ĐOẠN THẲNG

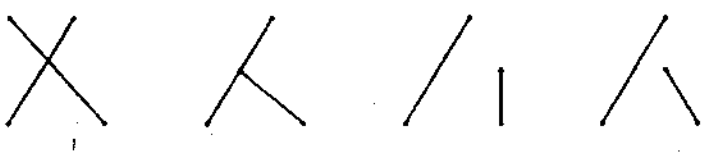
Trong bài toán hình học sơ cấp đầu tiên, chúng ta sẽ xét xem 2 đoạn thẳng có giao nhau hay không. Hình 24.3 minh họa một số tình huống có thể có.

Trong trường hợp đầu tiên, các đoạn thẳng giao nhau. Trường hợp 2 thì một điểm đầu của đoạn thẳng nằm trên đoạn thẳng kia, chúng ta sẽ xem như là giao nhau nếu xét đoạn thẳng là "đóng" (nghĩa là 2 điểm đầu cũng là một phần của đoạn thẳng). Khi đó các

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
x	3	11	6	4	5	8	1	7	9	14	10	16	15	13	3	12
y	9	1	8	3	15	11	6	4	7	5	13	14	2	16	12	10

Hình 24.2 *Tọa độ các điểm mẫu lấy theo bên trái của hình 24.1*

đoạn thẳng có một điểm đầu chung là giao nhau. Trong cả 2 trường hợp cuối, các đoạn thẳng không giao nhau, nhưng chúng khác nhau khi chúng ta xét đến giao điểm của các đường thẳng do các đoạn thẳng xác định. Trong trường hợp 4, giao điểm này rơi trên một đoạn thẳng, còn trường hợp 3 thì không. Ngoài ra, các đường thẳng còn có thể song song nhau (một trường hợp đặc biệt thường xảy ra là khi 1 hay cả 2 đoạn thẳng chỉ là 1 điểm).



Hình 24.3 *Bốn trường hợp giao nhau của hai đoạn thẳng*

Một phương pháp dễ hiểu để giải quyết bài toán này là tìm giao điểm của các đường thẳng xác định bởi các đoạn thẳng đó rồi kiểm xem nó có nằm giữa 2 điểm đầu của cả 2 đoạn thẳng đó hay không. Một cách dễ dàng khác có cơ sở dựa trên một số công cụ mà sẽ rất hữu ích sau này, nên chúng ta sẽ xem xét chi tiết hơn. Cho 3 điểm, chúng ta muốn biết rằng nếu đi từ điểm thứ nhất sang điểm thứ hai, rồi điểm thứ ba thì ngược hay thuận chiều kim đồng hồ. Ví dụ xét hình 24.1, nếu 3 điểm là A, B và C thì câu trả lời là ngược, còn A,B,D thì thuận. Hàm này rất dễ tính từ các phương trình đường thẳng như sau:

```
function ccw (p0, p1, p2 : point) : integer;
  var dx1, dx2, dy1, dy2 : integer;
  begin   dx1:=p1.x;      dy1:=p1.y-p0.y;
          dx2:=p2.x;      dy2:=p2.y-p0.y;
          if dx1 * dy2 > dy1 * dx2 then ccw:=1;
          if dx1 * dy2 < dy1 * dx2 then ccw:=-1;
          if dx1 * dy2 = dy1 * dx2 then
          begin
          if (dx1 * dx2 < 0) or (dy1 * dy2 < 0) then ccw:=-1
          else
          if (dx1*dx1 + dy1*dy1) >= (dx2*dx2 + dy2*dy2)
          then ccw:=0 else ccw:=1;
          end;
  end;
```

Để hiểu được chương trình thực hiện như thế nào, đầu tiên ta giả sử tất cả các giá trị *dx1*, *dx2*, *dy1* và *dy2* đều dương. Sau đó nhận xét rằng độ dốc của đường nối *p0* với *p1* là *dy1/dx1*, của đường nối *p0* với *p2* là *dy2/dx2*. Do đó, nếu độ dốc của đường thứ hai lớn hơn của đường thứ nhất thì đi từ *p0* sang *p1* rồi *p2* là ngược chiều kim đồng hồ; ngược lại là thuận chiều kim đồng hồ. So sánh độ dốc trong chương trình hơi bất tiện bởi vì đường thẳng có thể theo phương đứng (*dx1* hay *dx2* = 0), chúng ta tính tích *dx1*dx2* để tránh trường hợp này. Do đó, độ dốc không cần phải dương mới đúng. Tuy nhiên, nếu độ dốc bằng nhau (3 điểm thẳng hàng), mỗi người có thể tự đặt ra nhiều kiểu giá trị trả về cho *ccw*. Ở đây, chúng ta chọn 3 giá trị: thay vì dùng true hay false, chúng ta dùng 1 và -1, để giá trị 0 cho trường hợp khi *p2* ở trên đoạn thẳng nối *p0* và *p1;* nếu 3 điểm thẳng hàng và *p0* ở giữa *p2, p1*, chúng ta cho *ccw* bằng -1; nếu *p2* ở giữa *p0, p1*, chúng ta cho *ccw* bằng 0; và nếu p1 ở giữa p0, p2, chúng ta gán *ccw* bằng 1. Cách lựa chọn này làm đơn giản các chương trình sử dụng *ccw* trong chương này và chương kế.

Chúng ta có thể dùng trực tiếp *ccw* để cài đặt hàm intersect (xét giao nhau). Nếu cả 2 điểm đầu của một đoạn thẳng ở hai "bên" đường kia (nghĩa là có giá trị *ccw* khác nhau), thì chúng phải giao nhau.

```
function intersect (l1, l2 : line) : boolean;
  begin
  intersect:=((ccw(l1.p1,l1.p2,l2.p1)*ccw(l1.p1,l1.p2,l2.p2))<=0)
    and  ((ccw(l2.p1,l2.p2,l1.p1)*ccw(l2.p1,l2.p2,l1.p2))<=0)
  end;
```

Giải pháp này có vẻ đã dùng một số lượng lớn tính toán chỉ để giải quyết một bài toán đơn giản. Người đọc hãy mạnh dạn thử tìm một phương pháp đơn giản hơn, nhưng phải chú ý bảo đảm xét đủ tất cả các trường hợp. Ví dụ nếu cả 4 điểm đầu (của 2 đoạn thẳng) thẳng hàng thì có tất cả 6 trường hợp xảy ra (không kể khi 4 điểm trùng nhau) mà chỉ có 4 trong số đó là giao nhau. các trường hợp đặc biệt như vậy chính là những khó khăn của các thuật toán hình học, chúng ta không thể tránh khỏi nhưng có thể làm giảm tác dụng theo kiểu như trong *ccw*.

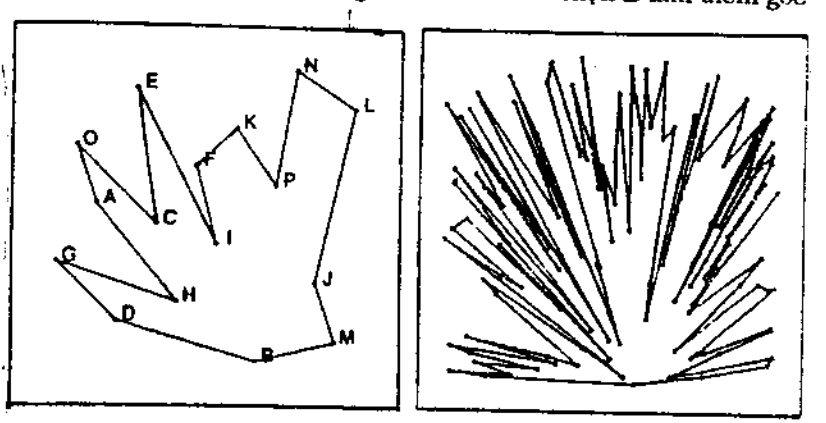
Nếu xét nhiều đường thẳng, tình huống trở nên phức tạp hơn nhiều. Trong chương 27, chúng ta sẽ thấy một thuật giải rắc rối dùng để xác định hai đường thẳng bất kỳ trong tập hợp N đường thẳng có giao nhau hay không.

ĐƯỜNG KHÉP KÍN ĐƠN

Để thấy được đặc điểm riêng của các bài toán ứng với tập hợp các điểm, chúng ta xét bài toán tìm một đường đi, qua một tập hợp N điểm xác định, qua tất cả các điểm, không giao nhau và cuối cùng trở về điểm bắt đầu. Đường đi như trên gọi là đường khép kín đơn. Ta có thể nghĩ ra nhiều ứng dụng cho bài toán này: các điểm có thể

tượng trưng cho những căn nhà, và đường đi tìm được là con đường mà người phát thư có thể dùng để đi đến mỗi căn nhà mà không phải đi qua đường đã đi. Hay là chúng ta có thể đơn giản muốn tìm một cách vẽ các điểm bằng plotter sao cho hợp lý. Bài toán này rất cơ bản, bởi vì nó chỉ đòi hỏi một đường đi khép kín bất kỳ nối các điểm. Bài toán tìm con đường tốt nhất trong những con đường như trên thì lại rất khó, gọi là bài toán *"hành trình của người bán hàng"*, và chúng ta sẽ xem xét nó cụ thể hơn trong vài chương cuối cùng. Trong chương kế, chúng ta sẽ xem bài toán tương tự nhưng dễ hơn nhiều là tìm đường đi ngắn nhất qua N điểm xác định. Trong chương 31, chúng ta sẽ thấy làm thế nào để tìm được con đường ngắn nhất kết nối một tập hợp các điểm.

Một cách dễ dàng để giải quyết ngay bài toán cơ bản trên là như sau: Chọn một trong các điểm làm "điểm gốc". Sau đó tính góc tạo bằng cách vẽ một đường từ mỗi điểm trong tập hợp đến gốc, rồi từ gốc vẽ ra theo phương ngang. Sau đó, sắp thứ tự các điểm theo thứ tự tăng dần của góc tương ứng. cuối cùng nối các điểm cạnh nhau lại. Kết quả chính là *simple closed path* nối các điểm, ví dụ như hình 24.4 là ứng với các điểm trong hình 24.1. Nếu chọn B làm điểm gốc



Hình 24.4 *Đường khép kín đơn*

thì các điểm sẽ được sắp theo thứ tự như sau:

B M J L N P K F I E C O A H G D B

và ta tạo được một đa giác đóng đơn.

Gọi *dx*, *dy* là khoảng cách từ điểm gốc đến một điểm khác theo trục x (hoành) và y (tung), thì góc cần tính trong thuật giải này là $tan^{-1}\ dy/dx$. Mặc dù *arctangent* là một hàm thiết kế sẵn của Pascal (và một số môi trường lập trình khác), nhưng nó có vẻ chậm, và dẫn tới ít nhất hai điều kiện khó chịu khi tính là: *dx* có bằng 0 hay không và điểm đang xét nằm trong góc vuông (phần tư) nào. Bởi vì góc chỉ dùng để sắp thứ tự trong thuật toán này, nên chúng ta có thể sử dụng một hàm khác dễ tính hơn nhưng vẫn có cùng thuộc tính thứ tự như hàm *arctangent* (cho nên khi sắp thứ tự, chúng ta vẫn nhận được cùng một kết quả). Một hàm khá tốt có thuộc tính như trên là *dy/(dy+dx)*. Việc kiểm tra các điều kiện ngoại lệ vẫn cần thiết nhưng đơn giản hơn. Chương trình sau đây trả về giá trị trong đoạn 0 đến 360, không phải là góc tạo bởi *p1* và *p2* so với phương ngang, nhưng có cùng thuộc tính thứ tự như góc đó.

```
function theta (p1, p2 : point) : real;
  var  dx, dy, ax, ay : integer;
  begin   dx:=p2.x - p1.x;    ax:= abs(dx);
          dy:=p2.y - p1.y;    ay:= abs(dy);
          if (dx=0) and (dy=0) then t:=0
                               else  t:=dy/(ax+ay);
          if dx<0 then t:=2-t
          else if dy<0 then t:=4+t;
          theta := t*90.0
  end;
```

Trong một số môi trường lập trình, việc sử dụng các chương trình như vậy thay cho các hàm lượng giác chuẩn có thể không đáng

kể; trong các môi trường khác thí nó có thể tạo ra sự tiết kiệm đáng kể. (Trong một số trường hợp, việc thay đổi cho *theta* có giá trị nguyên, tránh việc phải dùng hoàn toàn các số thực, có thể rất có ý nghĩa).

ĐIỂM NẰM TRONG ĐA GIÁC

Tiếp theo, chúng ta sẽ xét một bài toán rất tự nhiên: cho một điểm và một đa giác biểu diễn bằng một mảng các điểm, xác định xem điểm nằm bên trong hay bên ngoài đa giác. Một giải pháp dễ hiểu cho bài toán này là: vẽ một đoạn thẳng dài bắt đầu từ điểm đó, theo một hướng bất kỳ (dài để đủ cho điểm đầu còn lại phải bảo đảm nằm ngoài đa giác) và đếm số lượng đoạn thẳng tạo được do nó cắt qua đa giác. Nếu là số lẻ, điểm đó nằm trong đa giác, nếu chẵn thì điểm nằm ngoài. Điều này dễ thấy nếu chúng ta theo dõi những gì xảy ra khi đi từ điểm đầu đoạn thẳng bên ngoài đa giác: sau đoạn thẳng thứ nhất, chúng ta ở bên trong đa giác, sau đoạn thẳng thứ hai, ta lại ở ngoài đa giác v.v.. Nếu chúng ta thực hiện trong số lần chẵn, điểm mà chúng ta kết thúc (nghĩa là điểm cần xét) phải ở ngoài đa giác.

Tuy nhiên, không phải chỉ đơn giản như thế, bởi vì một số giao điểm có thể trùng với đỉnh của đa giác. Hình 24.5 đưa ra một số tình huống phải được xử lý. Trường hợp đầu tiên rõ ràng là "bên ngoài"; trường hợp thứ hai thì dễ thấy là "bên trong"; trường hợp thứ 3, đoạn thẳng để kiểm tra kết thúc tại một đỉnh của đa giác (sau khi đi qua 2 đỉnh khác); và trong trường hợp thứ tư, đoạn kiểm tra trùng khớp với một cạnh của đa giác trước khi kết thúc. Trong một số trường hợp khi đoạn kiểm tra giao với một đỉnh của đa giác, nó phải được đếm là một giao điểm với đa giác; nhưng trong các trường hợp khác, lại phải đếm là 0 (hay 2). Người đọc có thể giải trí bằng cách thử tìm ra một cách kiểm tra đơn giản có thể phân biệt được các trường hợp này trước khi xem tiếp tục.

Hình 24.5 *Một số tình huống phải được xử lý bằng thuật toán*

Nhu cầu xử lý các tình huống khi các đỉnh của đa giác rơi trên đoạn thẳng kiểm tra buộc chúng ta phải làm nhiều hơn là chỉ đếm số đoạn thẳng do đa giác giao với đoạn kiểm tra. Thực chất, chúng ta muốn đi vòng quanh đa giác, tăng một biến đếm khi nào ta đi từ một bên của đoạn kiểm tra sang một bên khác. Một cách để thực hiện là đơn giản bỏ qua các điểm rơi trên đoạn kiểm tra, như trong chương trình sau đây:

```
function inside (t:point) : boolean;
  var   count, i, j : integer;
        lt, lp : line;
  begin   count:=0; j:=0; p[0]:=p[N]; p[N+1]:=p[1];
          lt.p1:=t; lt.p2:=t; lt.p2.x:=maxint;
          for i:=1 to N do
          begin   lp.p1:=p[i]; lp.p2:=p[i];
                  if not intersect (lp,lt) then
                  begin   lp.p2:=p[j]; j:=i;
                  if intersect(lp,lt) then count:=count+1;
                  end;
          end;
          inside:=((count mod 2)=1);
  end;
```

Chương trình này dùng đường thẳng kiểm tra theo phương ngang để dễ tính toán (cho rằng hình 24.5 quay 45 độ). Biến j được dùng để lưu trữ chỉ số của điểm cuối cùng của đa giác mà không nằm

trên đoạn kiểm tra. Chương trình giả sử rằng *p[1]* là điểm có tọa độ x nhỏ nhất trong số tất cả các điểm có tọa độ y nhỏ nhất, vì vậy nếu *p[1]* nằm trên đoạn kiểm tra thì *p[0]* không thể. Cùng một đa giác có thể biểu diễn bằng *N* mảng khác nhau, nhưng điều này cho thấy việc áp đặt một quy luật chuẩn cho *p[1]* đôi khi lại tiện lợi (Ví dụ như, với cùng quy luật này, *p[1]* rất thích hợp để làm điểm gốc cho thủ tục nêu trên để tìm một đa giác đóng đơn). Nếu điểm kế tiếp trên đa giác mà không nằm trên đoạn kiểm tra, ở cùng một phía như điểm thứ j đối với đoạn kiểm tra thì chúng ta không cần phải tăng biến đếm giao điểm *(count)*. Ngược lại, chúng ta có được một giao điểm. Người đọc có thể kiểm tra thuật toán hoạt động chính xác trong các trường hợp của hình 24.5.

Nếu đa giác chỉ có 3 hay 4 cạnh, thường gặp trong nhiều ứng dụng, thì một chương trình phức tạp như vậy sẽ không được sử dụng, một thủ tục đơn giản, đặt cơ sở trên việc gọi *ccw* sẽ thích hợp hơn. Một trường hợp đặc biệt quan trọng khác là đối với đa giác lồi, được xét trong chương kế, nó có đặc điểm là không có đoạn kiểm tra nào có hơn 2 giao điểm với đa giác. Trong trường hợp này, một thủ tục như tìm nhị phân có thể dùng để xác định trong *O(logN)* bước để biết được điểm có ở bên trong hay không.

BÀN LUẬN

Chỉ từ một vài ví dụ, thì sự khó khăn trong việc giải quyết các bài toán hình học riêng biệt trên máy tính rất dễ bị đánh giá thấp. Còn có nhiều tính toán hình học sơ cấp khác mà chúng ta chưa bàn luận tới. Ví dụ như, chương trình tính diện tích của một đa giác cũng là một bài toán đáng chú ý. Tuy nhiên, các bài toán chúng ta đã xét cung cấp một số công cụ cơ bản mà sẽ rất hữu ích trong những phần sau để giải quyết các bài toán khó hơn.

Một số thuật toán chúng ta sẽ nghiên cứu hàm chứa việc xây

dựng các cấu trúc hình học từ một tập xác định các điểm. "Đa giác đóng đơn" là một ví dụ cơ bản. Chúng ta sẽ cần phải lựa chọn trong các cách biểu diễn thích hợp cho các cấu trúc như vậy, tạo ra các thuật toán để xây dựng chúng, xem xét công dụng của chúng trong các ứng dụng riêng biệt. Thông thường , các vấn đề này lẫn vào nhau. Ví dụ như thuật toán dùng trong thủ tục inside của chương này chủ yếu dựa trên cách biểu diễn đa giác đóng đơn bằng một tập hợp các điểm có thứ tự (hơn là một tập các đoạn thẳng không thứ tự).

Chúng ta sẽ nghiên cứu nhiều thuật toán về truy tìm hình học: chúng ta muốn biết điểm nào trong một tập hợp là gần với một điểm cho trước, hay điểm nào rơi trong một hình chữ nhật, hay điểm nào là gần nhất so với một điểm khác. Nhiều thuật giải tương ứng cho các bài toán truy tìm như vậy liên quan mật thiết với các thuật giải đã nghiên cứu trong các chương 14 - 17. Sự tương ứng khá rõ ràng.

Một vài thuật toán hình học có thể được phân tích chi tiết để đưa đến các khẳng định chính xác về tính năng của chúng. Như chúng ta đã thấy, thời gian thực hiện của một thuật toán hình học có thể phụ thuộc vào nhiều thứ: sự phân bố của các điểm, thứ tự của chúng trong dữ liệu nhập, và các hàm lược giác được sử dụng có tác động đáng kể trên thời gian thực hiện của các thuật toán hình học hay không. Tuy nhiên, thông thường trong các tình trạng như vậy, chúng ta sẽ đưa ra các thuật toán tốt cho các áp dụng riêng biệt. Ngoài ra, nhiều thuật toán xuất phát từ những suy nghĩ phức tạp và được thiết kế cho việc thực hiện tốt những trường hợp xấu nhất.

BÀI TẬP

1. Xác định giá trị của *ccw* cho ba trường hợp khi hai điểm là đồng nhất (và điểm thứ thứ ba thì khác), và cho trường hợp khi tất cả các điểm trùng nhau.

2. Viết thuật giải nhanh để xác định hai đoạn thẳng là song song hay không, không dùng bất kỳ một phép chia nào.

3. Viết thuật giải nhanh để xác định bốn đoạn thẳng có tạo nên một hình vuông không, không dùng bất kỳ một phép chia nào.

4. Cho một mảng các đoạn thẳng, làm thế nào để kiểm tra xem chúng có tạo thành một đa giác đóng đơn hay không ?

5. Vẽ các đa giác đóng đơn có được khi dùng A,C, và D trong hình 24.1 làm điểm gốc trong phương pháp nêu trên.

6. Giả sử rằng chúng ta dùng một điểm tùy ý làm "điểm gốc" cho phương pháp nêu trên để tính một đa giác đóng đơn. Hãy đưa ra các điều kiện mà điểm phải thỏa để phương pháp thực hiện tốt.

7. Hàm *intersect* trả về giá trị gì khi được gọi với hai bản sao của cùng một đoạn thẳng ?

8. *Inside* gọi đỉnh của đa giác là bên trong hay bên ngoài đa giác?

9. Giá trị tối đa có thể đạt được cho *count* là bao nhiêu khi *inside* được thực hiện trên một đa giác có N đỉnh ? Cho một ví dụ thể hiện câu trả lời của bạn.

10. Viết một chương trình hiệu quả để xác định một điểm cho trước có nằm trong một tứ giác hay không?

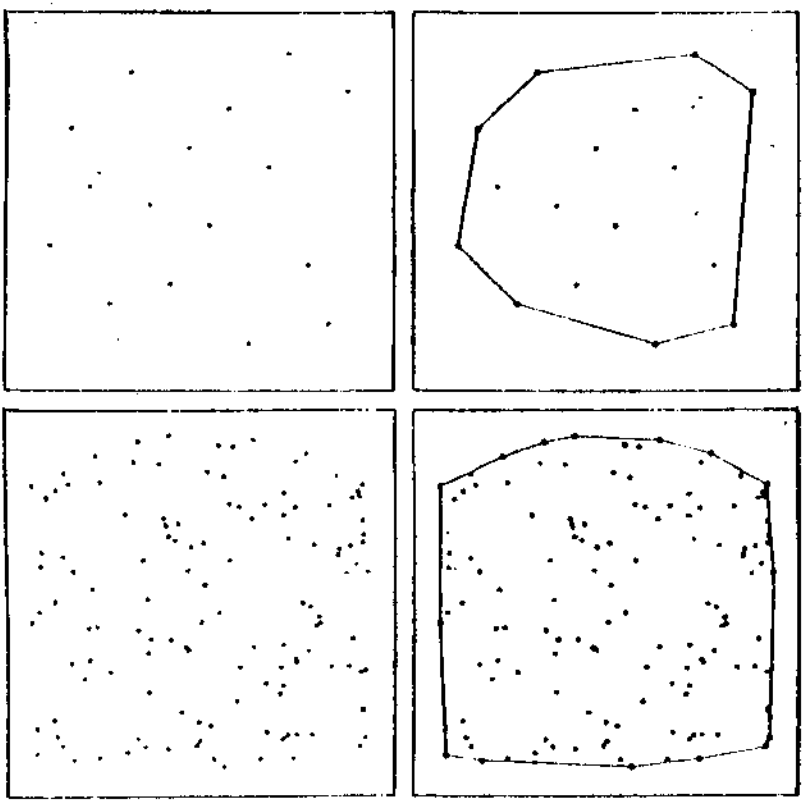
25

TÌM BAO LỒI

Thông thường, khi xử lý một số lớn các điểm, ta chú ý đến biên của tập các điểm này. Đứng trước biểu đồ các điểm được vẽ trong một mặt phẳng, người xem sẽ khó khăn khi phân biệt những điểm "nằm trong" với những điểm trên biên. Đây cũng là thuộc tính cơ sở để phân biệt các tập điểm; trong chương này, ta sẽ xét các thuật toán để lấy ra "biên tự nhiên" của các điểm.

Phương pháp toán học để định ra biên tự nhiên của một tập điểm thì dựa vào một tính chất hình học - tính lồi. Đây là một khái niệm đơn giản ta đã gặp trước đây : một đa giác lồi có tính chất là bất kỳ 2 điểm nào nằm trong đa giác này thì đoạn thẳng nối 2 điểm trên phải nằm hoàn toàn trong đa giác đó. Ví dụ, đa giác đơn khép kín mà ta xét trong chương trước là không lồi; mặt khác, mọi hình tam giác hay hình chữ nhật là lồi.

Trong toán học, biên tự nhiên của một tập điểm được gọi là bao lồi (convex hull). Bao lồi của một tập điểm được định nghĩa là một đa giác lồi nhỏ nhất chứa toàn bộ tập điểm này. Một cách tương đương, bao lồi là đường ngắn nhất bao quanh một tập điểm. Một tính chất hiển nhiên của bao lồi là các đỉnh của đa giác lồi xác định biên thì thuộc vào tập điểm đã cho. Bài toán ở đây là: cho tập N điểm, tìm tập con các điểm xác định bao lồi của N điểm đã cho. Trong chương này, chúng ta sẽ khảo sát một số thuật toán quan trọng để tìm bao lồi.



Hình 25.1 *Bao lồi của các tập điểm ở hình 24.1*

Hình 25.1 cho thấy các tập điểm và bao lồi của chúng. Có 8 điểm trên biên của tập ít điểm (bên trái) và 15 điểm trên biên của tập nhiều điểm (bên phải). Tổng quát, một bao lồi có thể chứa tối thiểu 3 điểm (xác định một tam giác chứa tất cả các điểm khác) và tối đa là toàn bộ tập điểm (nếu tất cả các điểm của tập này làm thành một đa giác lồi). Số các điểm trên bao lồi của một tập điểm ngẫu nhiên,

nằm đâu đó ở các điểm ngoài cùng, như ta sẽ thấy sau này. Một số thuật toán làm việc tốt khi có nhiều điểm trên bao lồi; một số khác làm việc tốt hơn khi chỉ có ít điểm trên bao lồi.

Một tính chất cơ bản của bao lồi là bất kỳ đường thẳng nào nằm ngoài bao, khi được dời về phía bao, sẽ chạm bao tại một trong các đỉnh của nó. (Đây là một định nghĩa khác về bao lồi: là tập điểm mà mỗi điểm được chạm tới bằng cách dời một đường thẳng từ xa vô hạn vào) Trong thực tế, có thể dễ dàng tìm vài điểm chắc chắn nằm trên bao lồi bằng quy tắc trên với những đường thẳng ngang và dọc: những điểm có toạ độ x hay y lớn nhất và nhỏ nhất đều nằm trên bao lồi. Điều này được dùng làm khởi điểm cho việc xem xét các thuật toán dưới đây.

CÁC QUY LUẬT TÍNH

Đầu vào của thuật toán tìm bao lồi dĩ nhiên là một mảng các điểm; chúng ta có thể dùng kiểu *point* được định nghĩa trong chương trước. Đầu ra là một đa giác, cũng được biểu diễn bằng một mảng các điểm với tính chất là khi lần theo các điểm với trật tự mà chúng xuất hiện trong mảng sẽ đồ lại viền của đa giác trên. Điều này có vẻ đòi hỏi thêm một điều kiện về trật tự khi tìm bao lồi (tại sao không trả về các điểm có trật tự bất kỳ của bao ?), nhưng rõ ràng là đầu ra có sắp xếp sẽ dễ sử dụng hơn, và ta sẽ thấy rằng các tính toán không để ý đến trật tự các điểm cũng không dễ dàng gì hơn. Khi xét các thuật toán, sẽ thuận tiện nếu dùng phép thay thế : mảng dùng cho tập điểm nguồn cũng được dùng để lưu các kết quả. Một cách đơn giản, các thuật toán này sắp xếp lại các điểm trong mảng ban đầu, bao lồi sẽ là M điểm đầu tiên (có thứ tự) trong mảng.

Từ những điều trên, rõ ràng việc tính bao lồi có quan hệ gần gũi với việc sắp xếp. Thực tế, thuật toán tính bao lồi có thể dùng để sắp xếp như sau đây. Cho N số để sắp, ta chuyển chúng thành các điểm

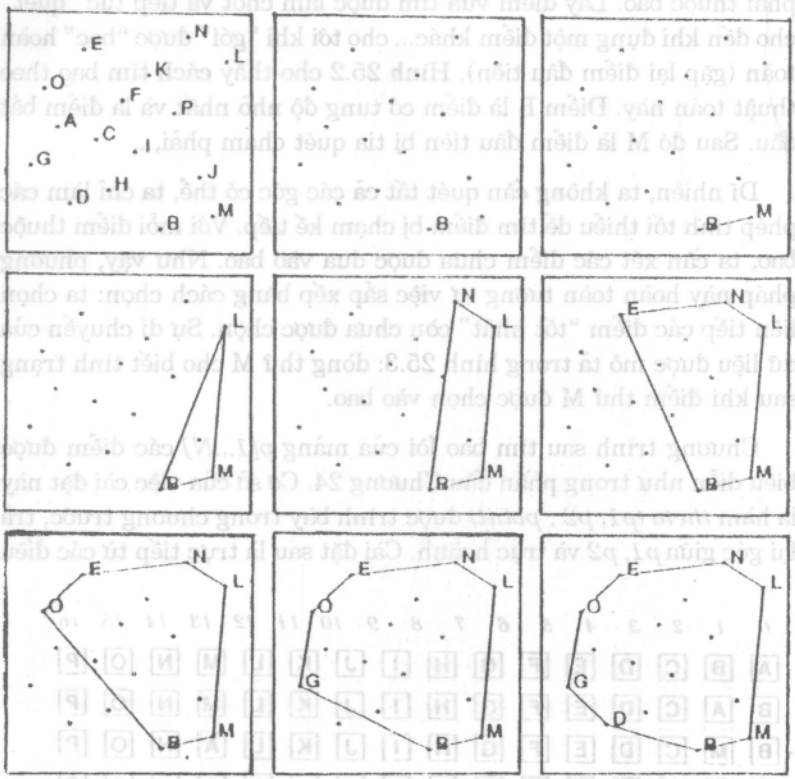
(trong toạ độ cực) bằng cách xem các số như là các góc với một bán kính không đổi. Bao lồi của các điểm này là một đa giác N đỉnh chứa tất cả các điểm. Vì kết quả là được sắp theo thứ tự xuất hiện các đỉnh trên đa giác, nên kết quả này có thể dùng để tìm thứ tự của N số đã cho.

Kết quả trên không phải là cơ sở để cho rằng việc tính bao lồi là dễ dàng hơn việc sắp xếp; ví dụ phải kể đến phí tổn cho các phép tính lượng giác khi chuyển các số thành điểm. So sánh thuật toán tìm bao lồi (chứa các phép tính lượng giác) với thuật toán sắp xếp (chứa các phép so sánh khoá) cũng giống như so sánh trái táo với trái cam. Nhưng ta sẽ thấy rằng có thuật toán tính bao lồi có độ phức tạp cỡ $NlogN$ như thuật toán sắp xếp (dù rằng các phép tính là hoàn toàn khác nhau).

Thực tế, các thuật toán tìm bao lồi không phức tạp hơn việc sắp xếp: có nhiều thuật toán có thời gian thi hành tỉ lệ với $NlogN$ trong trường hợp xấu nhất. Thậm chí, nhiều thuật toán còn đạt tới mức thời gian thi hành có tỉ lệ nhỏ hơn N, vì thời gian thực hiện của các thuật toán này phụ thuộc vào sự phân bố của các điểm và vào số điểm trên bao.

Cũng như nhiều thuật toán hình học khác, ta phải chú ý đến các trường hợp đặc biệt có thể có trong đầu vào. Ví dụ, nếu các điểm nằm trên cùng một đường thẳng thì bao lồi là gì? Tuỳ theo ứng dụng, bao lồi có thể là tất cả các điểm, có thể là hai điểm biên ở 2 đầu, hoặc là một tập điểm bất kỳ có chứa 2 điểm biên. Mặc dù đây có vẻ là một ví dụ cực đoan, việc có nhiều hơn 2 điểm rơi trên cùng một cạnh của bao lồi của tập điểm đã cho thì không phải là hiếm.

Trong các thuật toán dưới đây, chúng tôi không nhấn mạnh vào trường hợp các điểm ở trên cùng một cạnh của bao, vì phải làm nhiều việc hơn (chúng tôi sẽ trình bày khi có dịp thích hợp). Mặt khác, chúng tôi cũng không bỏ qua chúng, vì có thể phân tích tình trạng này khi có yêu cầu.



Hình 25.2 *Thuật Toán Bọc Gói*

THUẬT TOÁN BỌC GÓI

Thuật toán tìm bao lồi tự nhiên nhất, giống như cách một người vẽ bao lồi của tập điểm, là phương pháp "bao quanh" tập điểm đã cho. Bắt đầu từ một điểm chắc chắn nằm trên bao lồi (có thể lấy một trong các điểm có tung độ y nhỏ nhất), dùng một tia quét ngược chiều kim đồng hồ cho đến khi đụng một điểm khác : điểm mới này

phải thuộc bao. Lấy điểm vừa tìm được làm chốt và tiếp tục "quét" cho đến khi đụng một điểm khác... cho tới khi "gói" được "bọc" hoàn toàn (gặp lại điểm đầu tiên). Hình 25.2 cho thấy cách tìm bao theo thuật toán này. Điểm B là điểm có tung độ nhỏ nhất và là điểm bắt đầu. Sau đó M là điểm đầu tiên bị tia quét chạm phải,...

Dĩ nhiên, ta không cần quét tất cả các góc có thể, ta chỉ làm các phép tính tối thiểu để tìm điểm bị chạm kế tiếp. Với mỗi điểm thuộc bao, ta cần xét các điểm chưa được đưa vào bao. Như vậy, phương pháp này hoàn toàn tương tự việc sắp xếp bằng cách chọn: ta chọn liên tiếp các điểm "tốt nhất" còn chưa được chọn. Sự di chuyển của dữ liệu được mô tả trong hình 25.3: dòng thứ M cho biết tình trạng sau khi điểm thứ M được chọn vào bao.

Chương trình sau tìm bao lồi của mảng *p[1...N]* các điểm được biểu diễn như trong phần đầu Chương 24. Cơ sở của việc cài đặt này là hàm *theta (p1, p2 : point)* được trình bày trong chương trước, trả lại góc giữa *p1, p2* và trục hoành. Cài đặt sau là trực tiếp từ các điều

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
B	A	C	D	E	F	G	H	I	J	K	L	M	N	O	P	
B	M	C	D	E	F	G	H	I	J	K	L	A	N	O	P	
B	M	L	D	E	F	G	H	I	J	K	C	A	N	O	P	
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P	
B	M	L	N	E	F	G	H	I	J	K	C	A	D	O	P	
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P	
B	M	L	N	E	O	G	H	I	J	K	C	A	D	F	P	
B	M	L	N	E	O	G	D	I	J	K	C	A	H	F	P	

Hình 25.3 *Dịch chuyển dữ liệu trong thuật toán Bọc-Gói*

vừa được xét. *p[N+1]* được dùng làm biến cầm canh.

```
function   wrap : integer;
  var   i, min, M : integer;
        minangle, v : real; t : point;
  begin
  min:=1;
  for i:=2 to N do if p[i].y<p[min].y then min:=i;
  M:=0; p[N+1]:=p[min]; minangle:=0.0;
  repeat
    M:=M+1; t:=p[M]; p[M]:=p[min]; p[min]:=t;
    min:=N+1; v:=minangle; minangle:=360.0;
    for i:=M+1 to N+1 do
      if theta(p[M],p[i]) > v then
        if theta(p[M],p[i]) < minangle then
          begin min:=i; minangle:=theta(p[M],p[min])
          end;
  until min=N+1;
  wrap:=M;
  end;
```

Đầu tiên, ta tìm điểm có tung độ y nhỏ nhất và lưu trong *p[N+1]* để làm chỗ chấm dứt vòng lặp (sẽ nói sau). Biến M cho biết số điểm thuộc bao, và v là giá trị hiện thời của góc "quét" (là góc giữa trục hoành và đường thẳng nối *p[M-1]* và *p[M]*). Vòng lặp repeat đưa điểm vừa được tìm vào bao bằng cách đổi chỗ điểm này với điểm thứ M, và dùng hàm *theta* (ở chương trước) để tính góc giữa trục hoành và đường thẳng nối điểm trên với mỗi điểm còn lại chưa đưa vào bao, tìm điểm có góc nhỏ nhất trong số các góc lớn hơn v. Vòng lặp kết thúc khi điểm đầu tiên (thực ra là bản sao của điểm đầu tiên được đưa vào *p[N+1]*) được gặp lại.

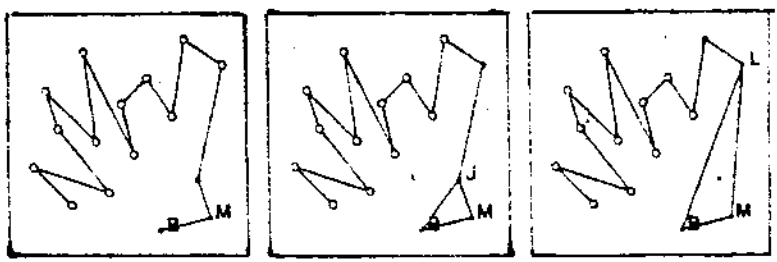
Chương trình này có thể trả lại hoặc không trả lại các điểm rơi trên một cạnh của bao lồi. Tình thế này xảy ra khi có nhiều hơn một

điểm có cùng giá trị *theta* với *p[M]* trong quá trình thực hiện thuật toán; sự cài đặt trên trả lại điểm đầu tiên được gặp trong số các điểm như vậy, mặc dù có thể có những điểm khác gần với *p[M]* hơn. Khi cần tìm các điểm rơi trên một cạnh của bao lồi, ta có thể thay đổi hàm *theta* để lấy khoảng cách giữa 2 điểm đã cho trong đối số của hàm và gán giá trị nhỏ hơn cho điểm gần hơn khi 2 điểm này có cùng một góc.

Nhược điểm chính của thuật toán "bọc gói" là trong trường hợp xấu nhất, khi tất cả các điểm đều thuộc vào bao, thời gian thi hành sẽ tỉ lệ với *N*N* (giống như việc sắp xếp bằng cách chọn). Mặt khác, phương pháp này có thể tổng quát hoá cho trường hợp nhiều chiều hơn. Bao lồi của một tập điểm trong không gian k-chiều là một hình lồi nhỏ nhất chứa tập điểm này (một hình lồi là hình mà mọi đoạn thẳng nối 2 điểm trong hình thì hoàn toàn nằm trong hình đó). Ví dụ bao lồi của tập điểm trong không gian 3 chiều là một vật thể lồi 3 chiều có các mặt bên là phẳng. Bao lồi được tìm bằng cách "quét" một mặt phẳng cho đến khi chạm bao, rồi rào các cạnh bên của các mặt phẳng, chốt các đường biên khác của bao cho đến khi "gói" được "bọc" (có lẽ trong nhiều thuật toán hình học, giải thích điều tổng quát thì dễ hơn là bổ sung !).

PHƯƠNG PHÁP QUÉT GRAHAM

Phương pháp sau của R.L.Graham (1972) đáng chú ý là vì hầu hết các tính toán dành cho việc sắp xếp: thuật toán có chứa một sắp xếp không tốn kém lắm. Thuật toán bắt đầu bằng việc xây dựng một đa giác đơn khép kín từ các điểm đã cho. Việc xây dựng này dùng một hàm của chương trước: sắp các điểm theo khoá là giá trị của hàm *theta* tương ứng với góc giữa trục hoành và đường thẳng nối mỗi điểm với điểm chốt *p[1]* (điểm có tung độ y nhỏ nhất), và khi lần theo *p[1],p[2],...,p[N], p[1]* ta sẽ được một đa giác khép kín.



Hình 25.4 *Vài bước đầu tiên của phép quét Graham*

Bao lồi được tìm bằng cách đi vòng: thử đặt một điểm vào bao và kiểm tra những điểm trước đấy có còn thuộc bao hay không. Ví dụ, ta xét các điểm theo thứ tự B M J L N P K F I E C O A H G D; vài bước đầu tiên được chỉ trên hình 25.4. Khi bắt đầu, ta biết B, M là thuộc bao. Khi xét J, nó được đưa vào bao và tạo thành một hình tam giác. Sau đó, khi xét L, ta nhận thấy rằng J không thể thuộc vào bao (vì nó lọt trong tam giác BML).

Nói chung, không có khó khăn trong việc kiểm tra có loại một điểm khỏi bao hay không. Sau khi thêm vào một điểm, ta sẽ lần quanh các điểm đã đưa vào bao và xét xem có loại điểm nào hay không. Khi lần quanh bao, ta chờ sự quẹo trái ở mỗi đỉnh của bao. Tại điểm mới thêm, nếu có quẹo phải, thì ta loại điểm này vì đa giác lồi đang có đã chứa trong điểm vừa thêm. Chi tiết hơn, ta sẽ dùng thủ tục ccw trong chương trước để kiểm tra việc loại một điểm. Giả sử sau khi xét các điểm *p[1...i-1]*, ta đã xác định được *p[1...M]* là bao. Khi xét điểm mới *p[i]*, nếu *ccw(p[M], p[M-1], p[i])* là không âm, ta loại *p[M]* ra khỏi bao. Nếu không, *p[M]* vẫn thuộc bao.

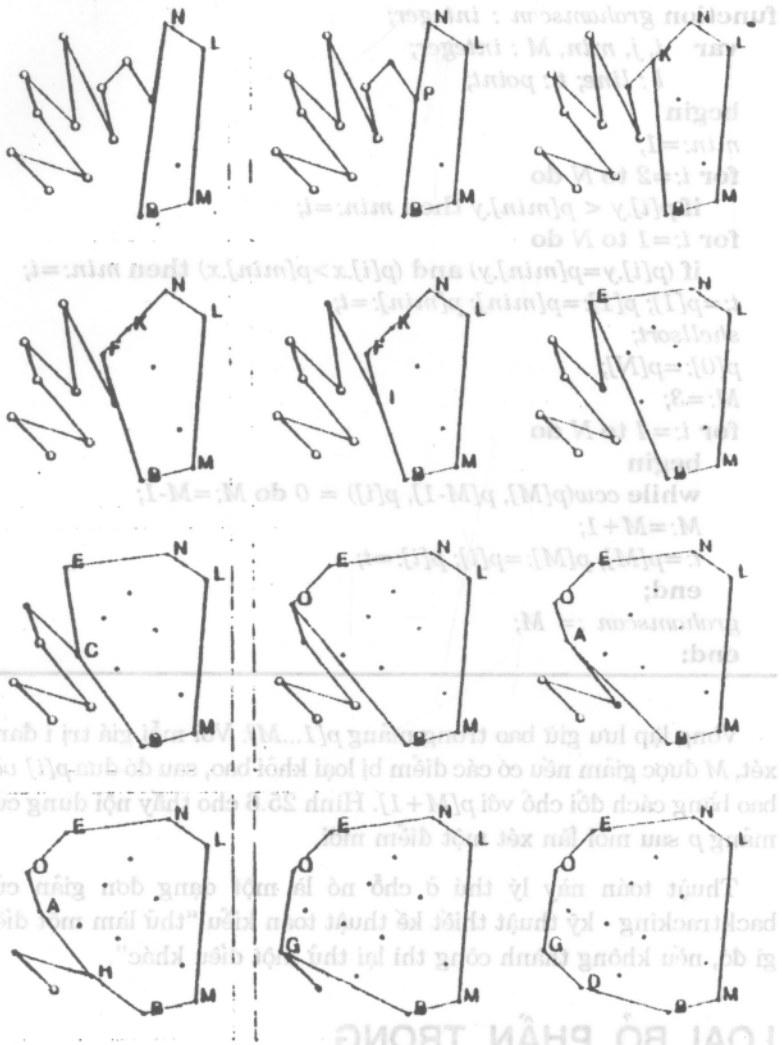
Hình 25.5 cho thấy quá trình thực hiện cách xử lý này trên tập

điểm. Mỗi điểm mới được xử lý như sau: điểm mới này được đưa vào bao và dùng làm "nhân chứng" cho việc loại bỏ các điểm có sẵn trên bao. Sau khi L, N và P được thêm vào bao, P bị loại bỏ vì khi xét tới K (vì NPK là một quẹo phải), rồi F và I được thêm vào. Khi xét E, điểm I bị loại vì FIE là một quẹo phải; rồi F và K bị loại vì KFE và NKE là các quẹo phải. Như vậy, có nhiều hơn một điểm bị loại khi xét một điểm mới. Tiếp tục theo cách này, sau cùng thuật toán quay về điểm B.

Sự sắp xếp ban đầu bảo đảm mỗi điểm được xét lần lượt đều có khả năng thuộc bao, vì mọi điểm được xét trước đó có giá trị *theta* nhỏ hơn. Mỗi đường thẳng nối *p[1]* và *p[i]*, tồn tại sau các phép loại trừ, có tính chất là mọi điểm được xét trước đó đều nằm về cùng một phía của đường thẳng này. Vì vậy khi xét tới điểm *p[N]* (cũng phải thuộc bao do thứ tự sắp xếp), ta đã tìm đủ các điểm trên bao.

Như phương pháp bọc-gói, các điểm trên một cạnh của bao có thể được hoặc không được kể đến dù rằng có 2 trường hợp đối với các điểm cộng tuyến. Trước tiên, nếu có 2 điểm cộng tuyến với *p[1]* thì như đã nêu, việc sắp xếp dùng hàm *theta* có thể hoặc không thể lấy chúng theo thứ tự của chúng trên đường thẳng chung. Các điểm không đúng trật tự sẽ bị loại trong quá trình quét. Thứ hai, có thể xuất hiện các điểm cộng tuyến trên các bao thử nghiệm (và không thể khử các điểm này).

Việc cài đặt không có gì phức tạp, nhưng cần chú ý một số chi tiết. Đầu tiên, điểm có giá trị x lớn nhất trong số các điểm có y nhỏ nhất được đổi chỗ với p[1]. Sau đó *shellsort* được dùng để sắp lại các điểm (hay bất kỳ phép sắp xếp nào cũng được), bằng cách dùng giá trị *theta* của các điểm này với *p[1]*. Sau khi sắp, *p[N]* được chép vào p[0] để làm biến cầm canh trong trường hợp *p[3]* không thuộc bao. Sau cùng, tiến hành việc quét như đã nói trên. Chương trình sau tìm bao lồi của tập điểm *p[1...N]*:



Hình 25.5 *Quá trình thực hiện cách xử lý của phép quét Graham*

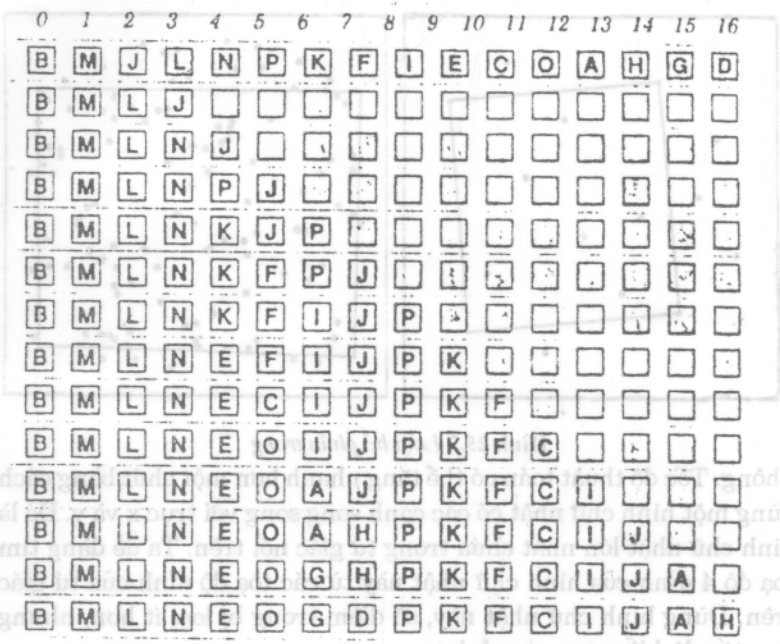
```
function grahamscan : integer;
   var  i, j, min, M : integer;
        l : line; t : point;
   begin
   min:=1;
   for i:=2 to N do
      if p[i].y < p[min].y then min:=i;
   for i:=1 to N do
      if (p[i].y=p[min].y) and (p[i].x>p[min].x) then min:=i;
   t:=p[1]; p[1]:=p[min]; p[min]:=t;
   shellsort;
   p[0]:=p[N];
   M:=3;
   for i:=4 to N do
      begin
      while ccw(p[M], p[M-1], p[i]) = 0 do M:=M-1;
      M:=M+1;
      t:=p[M]; p[M]:=p[i]; p[i]:=t;
      end;
   grahamscan := M;
   end;
```

Vòng lặp lưu giữ bao trong mảng *p[1...M]*. Với mỗi giá trị i đang xét, *M* được giảm nếu có các điểm bị loại khỏi bao, sau đó đưa *p[i]* vào bao bằng cách đổi chỗ với *p[M+1]*. Hình 25.6 cho thấy nội dung của mảng *p* sau mỗi lần xét một điểm mới.

Thuật toán này lý thú ở chỗ nó là một dạng đơn giản của backtracking - kỹ thuật thiết kế thuật toán kiểu "thử làm một điều gì đó, nếu không thành công thì lại thử một điều khác".

LOẠI BỎ PHẦN TRONG

Hầu như mọi phương pháp tìm bao lồi có thể được cải tiến nhiều

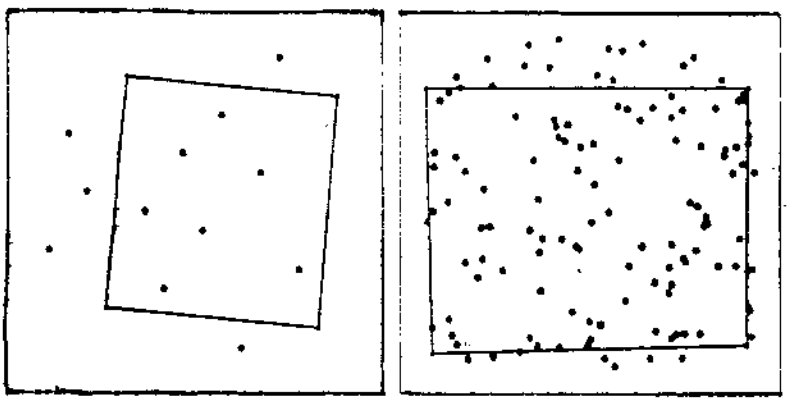


Hình 25.6 *Di chuyển dữ liệu trong phép quét Graham*

bằng cách dùng một kỹ thuật đơn giản để nhanh chóng loại bỏ ngay từ đầu phần lớn tập điểm. Ý tưởng tổng quát rất đơn giản: lấy 4 điểm đã biết trên bao, rồi bỏ đi mọi điểm bên trong tứ giác do 4 điểm trên tạo thành. Điều này làm giảm khá nhiều số điểm phải xét, sau đó các điểm còn lại được xử lý bằng quét Graham hay kỹ thuật bọc-gói.

Bốn điểm đã biết trên biên nên được chọn theo các thông tin về tập điểm đã cho. Nói chung, việc chọn các điểm tối ưu phải phù hợp với sự phân bố những phần tử của tập điểm đã cho. Trong hai tập điểm ví dụ trên hình 25.7 cho thấy kỹ thuật này loại bỏ đa số các điểm không thuộc biên.

Khi cài đặt việc loại bỏ phần trong, có một vòng lặp để kiểm tra mỗi điểm của tập điểm đã cho có nằm trong tứ giác kiểm tra hay



Hình 25.7 *Loại bỏ phần trong*

không. Tốc độ thuật toán có thể tăng nhanh hơn một chút bằng cách dùng một hình chữ nhật có các cạnh song song với trục x và y. Đó là hình chữ nhật lớn nhất chứa trong tứ giác nói trên. Ta dễ dàng tìm toạ độ 4 đỉnh của hình chữ nhật này từ các toạ độ đỉnh của tứ giác trên. Dùng hình chữ nhật này, số điểm trong bị loại ít hơn, nhưng làm tốc độ kiểm tra nhanh hơn.

KẾT QUẢ THỰC HIỆN

Như đã lưu ý trong chương trước, các thuật toán hình học thì khó phân tích hơn các thuật toán trong những lĩnh vực khác mà ta đã nghiên cứu; vì khó mô tả đặc điểm của đầu vào (và đầu ra) hơn. Các thuật toán đã xét phụ thuộc vào đặc điểm phân bố của các điểm và như vậy không thể so sánh các thuật toán với nhau; bởi vì sự so sánh chúng yêu cầu có một sự hiểu biết về các tác động qua lại rất phức tạp giữa các đặc tính được hiểu biết ít ỏi của các tập điểm. Nhưng cũng có một vài điều về việc thực hiện các thuật toán trên, giúp ta chọn lựa thuật toán thích hợp trong các ứng dụng cụ thể.

Tính chất 25.1 *Sau khi sắp xếp, quét Graham là một tiến trình có*

thời gian tuyến tính.

Cần phải suy nghĩ để chứng minh tính chất này là đúng, vì có một "vòng lặp trong vòng lặp" trong chương trình. Tuy nhiên, dễ thấy rằng không điểm nào bị "loại bỏ" hơn một lần, như vậy phần mã trong vòng lặp kép được lặp lại ít hơn N lần. Thời gian tổng cộng để tìm bao lồi bằng phương pháp này là $O(NlogN)$, nhưng vòng lặp trong tự bản thân nó là một phép sắp xếp, có thể được làm hiệu quả hơn bằng những kỹ thuật trong các chương 8-12.

Tính chất 25.2 *Nếu bao có M đỉnh, thì thuật toán bọc-gói yêu cầu khoảng MN bước lặp.*

Đầu tiên, ta phải tính $N-1$ góc để tìm góc nhỏ nhất, lần lặp kế tiếp cần $N-2$, rồi $N-3$,... như vậy tổng cộng số phép tính góc là $(N-1)+(N-2)+...+(N-M+1) = MN - M(M-1)/2$

Tính chất 25.3 *Phép loại bỏ phần trong là tuyến tính về thời gian trung bình*

BÀI TẬP

1. Giả sử bạn đã biết trước bao lồi của một tập điểm là hình tam giác. Tìm một thuật toán để xác định nhanh chóng tam giác này. Tương tự, tìm bao lồi nếu biết trước nó là một tứ giác.

2. Tìm một phương pháp hiệu quả để xác định xem một điểm có nằm trong một đa giác lồi hay không.

3. Hoàn thiện thuật toán tìm bao lồi giống như thuật toán sắp xếp chèn, dùng phương pháp của bạn trong bài tập trước.

4. Trong quét Graham, có thật sự cần thiết bắt đầu từ một điểm được biết chắc là ở trên bao? Giải thích tại sao có, tại sao không.

5. Trong phương pháp bọc-gói, có thật sự cần thiết bắt đầu từ một điểm được biết chắc là ở trên bao? Giải thích tại sao có, tại sao không.

6. Tìm một tập điểm sao cho phương pháp quét Graham tìm bao lồi mất hiệu quả.

7. Phương pháp quét Graham có tìm được bao lồi của một tập điểm tạo ra các đỉnh của một đa giác đơn bất kỳ không? Giải thích tại sao và cho một phần ví dụ chứng tỏ tại sao không.

8. Tìm 4 điểm để dùng cho phương pháp loại bỏ phần trong nếu tập điểm được cho là phân bố ngẫu nhiên trên hình tròn.

9. So sánh theo kinh nghiệm phương pháp quét Graham và phương pháp bọc gói cho một tập nhiều điểm với các toạ độ x, y trong khoảng 0 đến 100.

10. Cài đặt phương pháp khử phần trong và xác định theo kinh nghiệm N nên lớn bao nhiêu để loại được khoảng 50 điểm khi dùng phương pháp này trên tập điểm với x và y xấp xỉ nhau trong khoảng 0 và 100.

26

TÌM THEO KHOẢNG

Cho một tập điểm trong mặt phẳng, một câu hỏi tự nhiên là tìm các điểm rơi trong một vùng đã cho nào đó. "Liệt kê tất cả các thành phố trong phạm vi 50 dặm kể từ Princeton" là một câu hỏi thuộc loại này với tập điểm là các thành phố trên bản đồ Mỹ. Khi dạng hình học của vùng tìm được giới hạn là hình chữ nhật, vấn đề này dễ dàng được mở rộng cho các bài toán không hình học. Ví dụ, "Liệt kê những người từ 21 đến 25 tuổi với thu nhập từ 60000$ đến 100000$" tìm những "điểm" trong tập tin dữ liệu chứa tên-tuổi-thu nhập rơi trong mặt phẳng tuổi-thu nhập.

Việc mở rộng ngay trước mắt là cho nhiều chiều hơn. Nếu ta muốn liệt kê tất cả những ngôi sao trong phạm vi 50 năm ánh sáng từ mặt trời, ta có bài toán 3 chiều. Và nếu ta muốn tìm những người giàu có trẻ tuổi như trên mà lại cao và là phái nữ, ta có bài toán 4 chiều. Trong thực tế, số chiều của những bài toán như vậy có thể rất cao.

Tổng quát, giả sử ta có một tập các bản ghi với những thuộc tính nhất định có giá trị sắp xếp được (đôi khi được gọi là cơ sở dữ liệu, dù những định nghĩa hoàn chỉnh và chi tiết hơn cho thuật ngữ quan trọng này đã được trình bày). Tìm kiếm vùng là tìm tất cả những bản ghi thoả mãn các hạn chế phạm vi cho trước trên một tập các thuộc tính. Đây là một bài toán khó và quan trọng trong các ứng dụng thực tế. Trong chương này, ta sẽ tập trung trên bài toán hình

học 2 chiều; trong đó các bản ghi là các điểm và các thuộc tính là các trục toạ độ của điểm.

Ta sẽ xét một phương pháp được tổng quát hoá trực tiếp từ các phương pháp đã biết về việc tìm kiếm theo những khoá đơn (một chiều). Ta coi các câu hỏi là được đặt trên cùng một tập điểm, như vậy bài toán có thể chia làm hai phần : ta cần một thuật toán *tiền xử lý* để đưa các điểm đã cho vào một cấu trúc hỗ trợ có hiệu quả cho việc tìm kiếm vùng và một thuật toán tìm kiếm vùng, dùng cấu trúc trên để trả về các điểm rơi trong vùng (nhiều chiều) đã cho. Sự phân biệt này là để so sánh các phương pháp khác nhau vì tổng chi phí không chỉ phụ thuộc vào sự phân bố của các điểm mà còn vào số lượng và bản chất của các câu hỏi.

Bài toán tìm theo khoảng nghĩa là tìm tất cả các điểm rơi trong khoảng cho trước. Điều này có thể làm được bằng cách sắp các điểm và dùng phép tìm nhị phân để dò các điểm cuối của khoảng đã cho, rồi trả lại tất cả các điểm rơi trong khoảng trên. Một biện pháp khác là xây dựng cây tìm kiếm nhị phân và thực hiện một phép duyệt đệ quy trên cây, trả về các điểm trong khoảng tìm và bỏ qua các phần của cây nằm ngoài khoảng tìm. Trong chương trình ở trang sau có chứa một phép duyệt cây đệ quy đơn giản (xem Chương 4). Nếu điểm cuối bên trái của khoảng tìm rơi vào bên trái của nút gốc, ta tìm (đệ quy) theo cây con bên trái, và tương tự cho bên phải; với mỗi nút ta gặp, kiểm tra xem giá trị của nút có rơi trong khoảng tìm hay không.

Chương trình có thể được cải tiến hiệu quả hơn chút ít bằng cách dùng một biến toàn cục *int* để lưu khoảng tìm hơn là truyền biến cho các lời gọi đệ quy mà không thay đổi giá trị của nó. Hình 26.1 cho thấy các điểm tìm được khi chương trình vận hành trên cây ví dụ. Chú ý rằng các điểm trả về không cần liên thông với nhau trên cây.

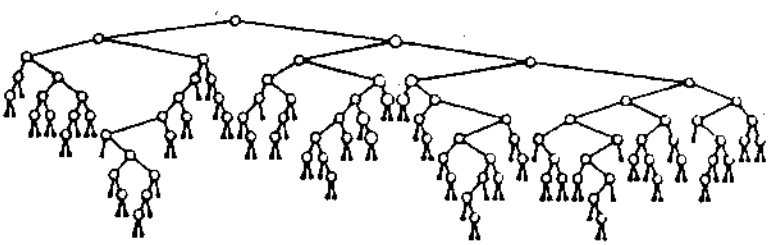
Tính chất 26.1 *Tìm kiếm theo khoảng có thể được thực hiện với O(NlogN) bước xử lý trước và O(R+logR) để tìm kiếm vùng, ở đây*

```
type interval = record x1, x2 :integer end;
procedure treerange (t:link, int:interval);
   var tx1, tx2 : boolean;
   begin
   if t<>z then
      begin
      tx1 := t↑.key >= int.x1;
      tx2 := t↑.key <= int.x2;
      if tx1 then treerange(t ↑.l, int);
      if tx1 and tx2
      then t.key is within the range
      if tx2 then treerange(t ↑.r, int);
      end
   end;
```

R là số các điểm thực sự rơi trong khoảng tìm.

Những điều này trực tiếp từ các đặc tính cơ bản của cấu trúc tìm kiếm (xem Chương 14 và 15). Nếu muốn, ta cũng có thể dùng cây cân bằng.

Mục đích của chúng ta trong chương này là sẽ đạt tới cùng thời gian thực hiện như trên khi tìm kiếm trên vùng nhiều chiều. Tham số R rất quan trọng : tạo thuận lợi để sinh các câu hỏi vùng, người dùng có thể dễ dàng phát biểu những câu hỏi yêu cầu tất cả hoặc hầu hết các điểm. Kiểu câu hỏi này được dùng trong nhiều ứng dụng,



Hình 26.1 *Tìm theo khoảng (một chiều) bằng một cây tìm kiếm nhị phân*

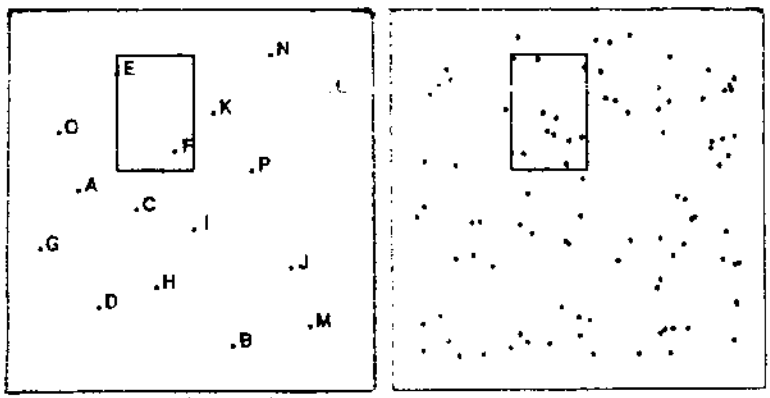
ngoài ra sẽ không cần đến các thuật giải phức tạp nếu mọi câu hỏi thuộc vào kiểu trên. Ta sẽ xét các thuật toán được thiết kế có hiệu quả cho các câu hỏi không trả về một số lớn các điểm.

CÁC PHƯƠNG PHÁP CƠ BẢN

Trong trường hợp 2 chiều, "vùng" là một miền phẳng. Để đơn giản, ta sẽ xét bài toán tìm tất cả các điểm có toạ độ x, y rơi trong các khoảng cho trước; nghĩa là ta sẽ tìm các điểm rơi trong một hình chữ nhật cho trước. Như vậy, ta sẽ giả thiết rằng kiểu rectangle là một bản ghi của 4 số nguyên cho các điểm cuối theo chiều dọc va chiều ngang. Thao tác cơ bản của chúng ta là kiểm tra xem một điểm có rơi vào hình chữ nhật đã cho hay không, như vậy ta sẽ dùng hàm *insiderect(p:point; rect:rectangle)* để kiểm tra điều này, trả về true nếu điểm p rơi trong *rect*. Mục tiêu của chúng ta là tìm tất cả các điểm rơi trong hình chữ nhật đã cho, dùng càng ít lời gọi hàm *insiderect* càng tốt.

Phương pháp đơn giản nhất để giải bài toán này là tìm kiếm tuần tự: quét tất cả các điểm, kiểm tra xem mỗi điểm có rơi trong khoảng đã cho hay không (bằng cách gọi *insiderect* cho từng điểm). Trong thực tế, phương pháp này được dùng trong nhiều cơ sở dữ liệu vì nó dễ dàng được cải tiến bằng cách "làm thành đợt" các câu hỏi vùng, kiểm tra nhiều câu hỏi khác nhau trong cùng một lần quét qua các điểm. Trong cơ sở dữ liệu rất lớn, nơi mà các dữ liệu được lưu trong các thiết bị ngoài và thời gian đọc dữ liệu là vượt xa chi phí cho các câu hỏi, các câu hỏi được gom lại trong bộ nhớ trong và tìm các điểm thoả các câu hỏi trong một lần duyệt tập tin dữ liệu ngoài. Tuy nhiên, nếu cách làm thành đợt không thuận tiện hay cơ sở dữ liệu nhỏ hơn, thì có những phương pháp tốt hơn nhiều.

Một cải tiến đơn giản đầu tiên của phương pháp tìm kiếm tuần tự là áp dụng trực tiếp phương pháp cho vùng một chiều trên các chiều



Hình 26.2 *Tìm theo khoảng hai chiều*

được tìm kiếm. Hình 26.2 cho một ví dụ tìm các điểm trong một hình chữ nhật.

Một cách làm là tìm các điểm có hoành độ x rơi trong khoảng tìm x của hình chữ nhật, rồi kiểm tra tung độ y của các điểm này để xét các điểm có thuộc hình chữ nhật đã cho hay không. Như vậy, các điểm không thuộc hình chữ nhật, do hoành độ x của chúng ở ngoài khoảng tìm, sẽ không bao giờ được xét đến. Kỹ thuật này gọi là phép chiếu; rõ ràng ta cũng có thể dùng cách trên cho tung độ y. Trong ví dụ, ta kiểm tra các điểm E C H F và I khi chiếu theo x và ta cũng có thể kiểm tra O E F K P N và L khi chiếu theo y.

Nếu các điểm phân bố đều trong một miền hình chữ nhật, dễ tính được số trung bình các điểm được kiểm tra. Phần các điểm ta muốn tìm trong hình chữ nhật đã cho là tỉ số của vùng hình chữ nhật với toàn miền; phần các điểm ta kiểm tra theo chiều x là tỉ số giữa bề rộng của hình chữ nhật với chiều rộng của toàn miền; và tương tự theo chiều y. Trong ví dụ, ta dùng hình chữ nhật 4x6 trong miền 16x16 nghĩa là ta trông có 3/32 điểm trong hình chữ nhật, 1/4 chúng trong phép chiếu x và 3/8 chúng theo phép chiếu y. Rõ ràng, dưới

những tình huống như vậy, tốt nhất là chiếu lên trục ứng với chiều ngắn hơn của hình chữ nhật. Nói cách khác, ta dễ dàng xây dựng những tình huống trong đó kỹ thuật chiếu là tệ hại; ví dụ, nếu tập điểm có dạng L và việc tìm kiếm với vùng tìm chỉ bao được các điểm ở gần góc của "L", sau đó phép chiếu trên cả hai trục loại bỏ chỉ được một nửa các điểm.

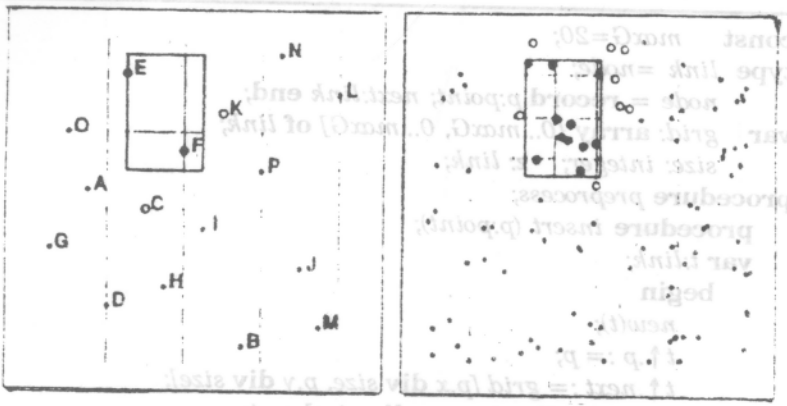
Nhìn lướt qua lần đầu, có vẻ là kỹ thuật chiếu được cải tiến để cho các điểm rơi trong khoảng tìm x và các điểm rơi trong khoảng tìm y là "cắt nhau". Hãy cố gắng làm điều này mà không xét tới trường hợp xấu nhất là tất cả các điểm trong khoảng tìm x hoặc tất cả các điểm rơi trong khoảng tìm y, sẽ giúp hiểu rõ hơn các phương pháp phức tạp mà ta nghiên cứu sau đây.

PHƯƠNG PHÁP LƯỚI

Một kỹ thuật đơn giản nhưng hiệu quả là duy trì quan hệ gần nhau giữa các điểm trong mặt phẳng để cấu trúc thành một lưới, chia vùng được tìm thành nhiều ô (ví dụ, kỹ thuật này được dùng trong khảo cổ học). Kế tiếp, khi các điểm nằm trong hình chữ nhật tìm kiếm được xét, ta chỉ xét các ô có giao với hình chữ nhật. Trong ví dụ, chỉ các điểm E, C, F và K là được xét (như trong hình 26.3)

Trong việc cài đặt phương pháp này, điều quyết định chủ yếu là xác định kích thước của lưới : nếu quá to, mỗi ô lưới sẽ chứa quá nhiều điểm, và nếu quá nhỏ, phải tìm kiếm trên quá nhiều ô lưới (phần lớn các ô là rỗng). Một cách nhằm vào sự cân đối giữa hai cực đoan trên là chọn kích thước ô sao cho số các ô là một phân số trên tổng số các điểm. Như vậy số các điểm trong một ô được hy vọng là xấp xỉ một hằng số nhỏ nào đó. Trong ví dụ, ta dùng lưới 4x4 cho tập 16 điểm nghĩa là mỗi ô có thể chứa một điểm.

Sau đây là một cài đặt đơn giản của chương trình xây dựng lưới chứa các điểm trong mảng *p[0...N]* các điểm theo type được định



Hình 26.3 *Phương pháp Lưới để tìm theo khoảng*

nghĩa trong phần đầu Chương 24. Biến *size* dùng để xác định cỡ của ô lưới. Để đơn giản, ta giả thiết các toạ độ của điểm trong khoảng 0 và một giá trị max nào đó. Vì biến size được lấy làm cỡ của ô lưới nên ta có *(max/size)*(max/size)* ô lưới. Để tìm ô lưới chứa một điểm nào đó, ta chia các toạ độ của điểm cho *size*, như trong cài đặt ở trang sau:

Chương trình này dùng cách biểu diễn danh sách liên kết chuẩn, với nút rỗng *z* ở cuối. Biến max được cho làm biến toàn cục, có giá trị ban đầu là giá trị toạ độ lớn nhất trong số các điểm được nhập vào.

Như đã mô tả ở trên, giá trị biến size phụ thuộc vào số các điểm, lượng bộ nhớ sử dụng và vùng các giá trị toạ độ. Một cách ước chừng, để có *M* điểm trong mỗi ô, ta nên chọn *size* là số nguyên lớn nhất gần với *max/sqrt(N/M)*. Điều này dẫn đến có khoảng *N/M* ô. Sự ước lượng này không chính xác cho các giá trị nhỏ của các biến, nhưng nó dùng được trong hầu hết trường hợp, và những ước lượng tương tự có thể dễ dàng công thức hoá trong các ứng dụng đặc biệt. Giá trị ước lượng trên có thể không cần chính xác và trong cài đặt này, ta dùng

```
const    maxG=20;
type  link =node;
      node = record p:point; next:link end;
var   grid: array [0...maxG, 0...maxG] of link;
      size: integer;    z: link;
procedure preprocess;
   procedure insert (p:point);
   var t:link;
      begin
         new(t);
         t↑.p := p;
         t↑.next := grid [p.x div size, p.y div size];
         grid [p.x div size, p.y div size] := t;
      end;
   begin
   new (z);
   size := 1;
   while size*size < max*max/N do size := size+size;
   for i:=0 to maxG do
      for j:=0 to maxG do grid[i,j]:=z;
   for i:=0 to N do insert (p[i]);
   end;
```

size là luỹ thừa của 2 bằng cách nhân và chia với *size*, sẽ hiệu quả hơn rất nhiều trong hầu hết các môi trường lập trình.

Một cách chọn thông thường là lấy $M=1$ và ta dùng giá trị M này trong cài đặt. Nếu khoảng trống quá nhiều, có thể lấy giá trị M lớn hơn, nhưng giá trị M nhỏ hơn thì ít được dùng ngoại trừ trong những tình huống đặc biệt.

Lúc này, ta đã nắm được hầu hết công việc để tìm kiếm vùng theo chỉ mục trên mảng *grid*:

Thời gian thực hiện của chương trình này tỉ lệ với số ô được xét. Vì ta đã thận trọng sắp xếp sao cho mỗi ô chứa (về trung bình) một

```
procedure gridrange (rect : rectangle);
   var   t: link;    i,j : integer;
   begin
   for i:=(rect.x1 div size) to (rect.x2 div size) do
      for j:=(rect.y1 div size) to (rect.y2 div size) do
         begin
         t:=grid[i,j];
         while t<>z do
            begin
            if insiderect (t↑.p.rect)
               then point t↑.p is within the range
            t := t↑.next;
         end
      end
   end;
```

hằng số các điểm và số các ô được xét cũng tỉ lệ (về trung bình) với số các điểm được xét.

Tính chất 26.2 *Tính trung bình, phương pháp lưới có thời gian tuyến tính đối với số các điểm trong vùng tìm, và tuyến tính đối với tổng số điểm trong trường hợp xấu nhất.*

Nếu số điểm trong hình chữ nhật tìm kiếm là R, thì số ô được xét sẽ tỉ lệ với R. Số ô được xét, mà lại không rơi trọn trong hình chữ nhật tìm kiếm, chắc chắn sẽ nhỏ hơn R; như vậy tổng số thời gian thực hiện (về trung bình) là tuyến tính với R. Với R lớn, số các điểm không rơi trong vùng tìm, mà vẫn được xét, thì khá nhỏ: tất cả những điểm như vậy rơi trong các ô có phần giao với hình chữ nhật tìm kiếm và số những ô như thế tỉ lệ với căn bậc hai của R khi R lớn. Chú ý là lập luận này có phần sai nếu các ô quá nhỏ (có quá nhiều ô rỗng trong hình chữ nhật tìm kiếm) hoặc quá lớn (chứa quá nhiều điểm trong các ô nằm trên chu vi của hình chữ nhật tìm kiếm) hoặc nếu hình chữ nhật tìm kiếm mảnh hơn các ô (nó có thể giao với

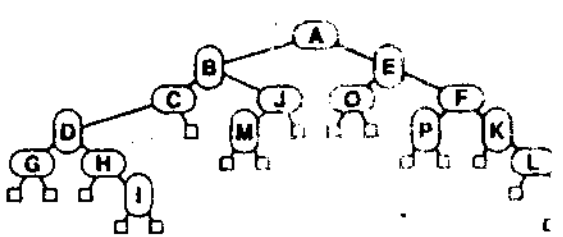
nhiều ô nhưng chỉ có ít điểm rơi trong đó).

Phương pháp lưới làm việc tốt nếu các điểm rải đều trong vùng tìm, nhưng sẽ tồi tệ nếu các điểm cụm lại thành đám (ví dụ, tất cả các điểm có thể rơi trong một ô, nghĩa là các ô khác sẽ không chứa gì cả). Phương pháp được xét kế tiếp làm cho trường hợp xấu nhất này khác hẳn đi bằng cách chia nhỏ không gian tìm theo nhiều cách khác nhau, phù hợp với tập điểm.

CÂY HAI CHIỀU

Cây hai chiều là một cấu trúc dữ liệu động, rất giống với cây nhị phân ngoại trừ điều là nó chia không gian hình học theo một cách thuận tiện cho việc tìm kiếm vùng. Ý tưởng ở đây là xây dựng cây tìm kiếm nhị phân với các điểm được chứa trong các nút, dùng các toạ độ x và y của các điểm như là các khoá theo một trình tự thay đổi nghiêm ngặt.

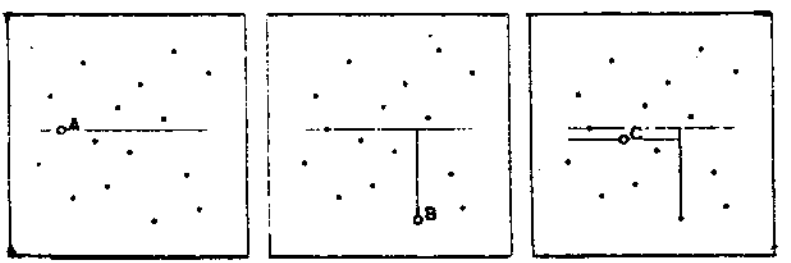
Cũng cùng một thuật giải được dùng để chèn các điểm vào cây 2D như trong cây tìm kiếm nhị phân thông thường; nhưng ở nút gốc, ta dùng tung độ y (nếu điểm được chèn có tung độ y nhỏ hơn điểm ở nút gốc, thì sang trái; nếu không sang phải), sau đó ở mức tiếp theo ta dùng hoành độ x, rồi ở mức tiếp theo nữa thì dùng tung độ y, ... cứ luân phiên như vậy cho đến khi gặp nút ngoài cùng. Hình



Hình26.4 *Một cây hai chiều (2D)*

26.4 cho thấy cây 2D tương ứng với tập điểm trong ví dụ.

Sự đáng chú ý của kỹ thuật này là việc chia mặt phẳng một cách đơn giản: tất cả những điểm dưới nút gốc thuộc cây con bên trái, tất cả những điểm ở trên thuộc cây con bên phải; sau đó tất cả những điểm ở trên nút gốc và ở bên trái điểm thuộc cây con bên phải thì thuộc cây con bên trái của cây con bên phải của nút gốc, ...

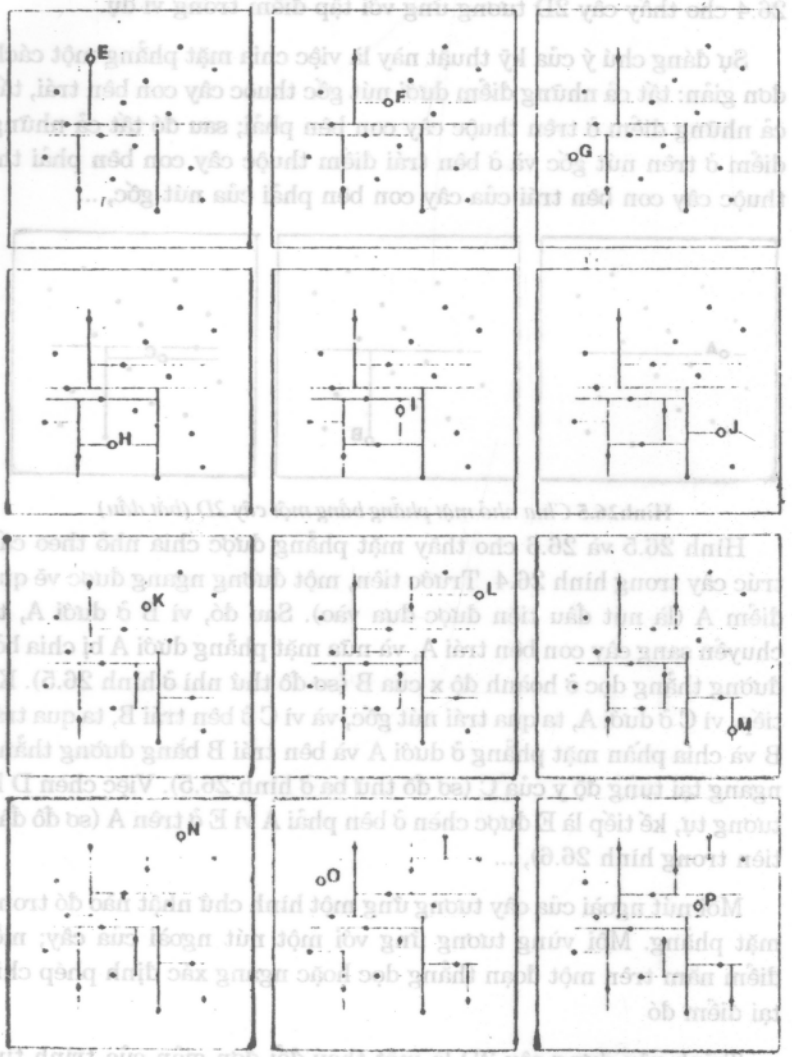


Hình26.5 *Chia nhỏ mặt phẳng bằng một cây 2D (bắt đầu)*

Hình 26.5 và 26.6 cho thấy mặt phẳng được chia nhỏ theo cấu trúc cây trong hình 26.4. Trước tiên, một đường ngang được vẽ qua điểm A (là nút đầu tiên được đưa vào). Sau đó, vì B ở dưới A, ta chuyển sang cây con bên trái A, và nữa mặt phẳng dưới A bị chia bởi đường thẳng dọc ở hoành độ x của B (sơ đồ thứ nhì ở hình 26.5). Kế tiếp, vì C ở dưới A, ta qua trái nút gốc, và vì C ở bên trái B, ta qua trái B và chia phần mặt phẳng ở dưới A và bên trái B bằng đường thẳng ngang tại tung độ y của C (sơ đồ thứ ba ở hình 26.5). Việc chèn D là tương tự, kế tiếp là E được chèn ở bên phải A vì E ở trên A (sơ đồ đầu tiên trong hình 26.6), ...

Mỗi nút ngoài của cây tương ứng một hình chữ nhật nào đó trong mặt phẳng. Mỗi vùng tương ứng với một nút ngoài của cây; mỗi điểm nằm trên một đoạn thẳng dọc hoặc ngang xác định phép chia tại điểm đó

Trình xây dựng cây 2D là một thay đổi đơn giản của trình tìm kiếm trên cây nhị phân chuẩn để thay đổi việc tìm theo *x* và *y* tại mỗi



Hình26.6 *Chia nhỏ mặt phẳng bằng một cây 2D (các bước tiếp theo)*

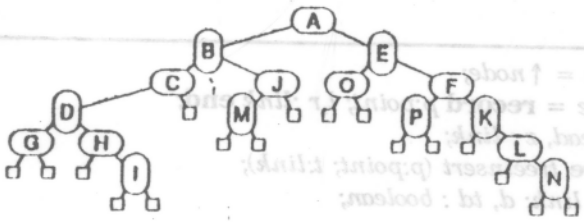
mức

```
type  link = ↑node;
      node = record p:point; l,r :link end;
var   t, head, z : link;
procedure treeinsert (p:point; t:link);
   var f : link; d, td : boolean;
   begin
   d := true;
   repeat
      if d   then td := p.x  t↑.p.x;
             else td := p.y  t↑.p.y;
      f := t;
      if td  then t := t↑.l
             else t := t↑.r;
      d := not d;
   until t=z;
   new (t); t↑.p := p; t↑.l := z; t↑.r := z;
   if td then f↑.l := t else f↑.r := t;
   end;
```

Như thường lệ, ta dùng một nút đầu head với điểm giả *(0, 0)*. Nút head "nhỏ hơn" mọi nút khác để cây lùi qua liên kết phải của nút head, và nút giả *z* được dùng để biểu diễn mọi nút ngoài. Lời gọi *treeinsert (p, head)* chèn một nút mới chứa điểm *p* vào cây. Biến logic *d* dùng để thực hiện việc kiểm tra luân phiên các toạ độ *x* và *y* khi đi xuống trên cây. Ngoài ra, thủ tục này là giống như thủ tục chuẩn ở Chương 14.

Tính chất 26.3 *Việc xây dựng cây 2D từ N điểm ngẫu nhiên đòi hỏi trung bình 2NlnN phép so sánh.*

Thực vậy, với các điểm phân bố ngẫu nhiên, cây 2D có cùng cách biểu diễn đặc trưng như cây nhị phân. Cả hai toạ độ giữ nhiệm vụ

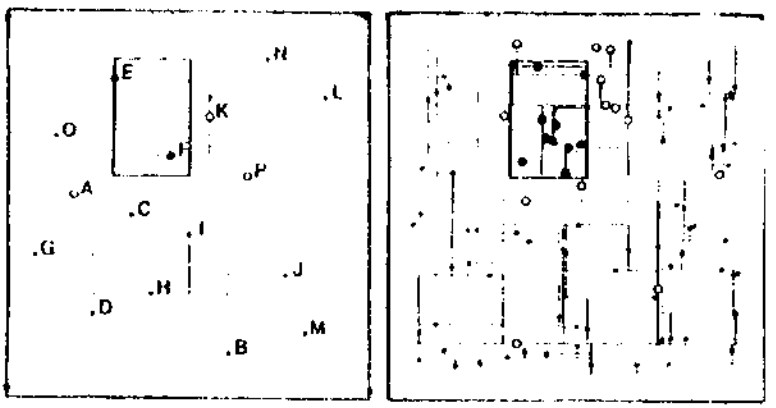


Hình 26.7 *Tìm theo khoảng nhờ một cây 2D*

như là các khoá ngẫu nhiên. Để tìm kiếm vùng bằng cây 2D, trước tiên ta xây dựng cây 2D từ các điểm trong giai đoạn tiền xử lý:

```
procedure preprocess;
   function initialize:link;
   var t : link;
      begin
      p[0].x := 0; p[0].y := 0; p[0].info := 0;
      new(t); t↑.p := p[0]; t↑.r := z;
      initialize := t;
      end;
   begin
      new(z); head := initialize;
      for i:=1 to N do treeinsert(p[i],head);
   end;
```

Tiếp theo, để tìm kiếm vùng, ta kiểm tra điểm ở mỗi nút với vùng tìm với chiều được dùng để chia mặt phẳng của nút đó. Trong ví dụ, ta bắt đầu bằng cách qua phải ở nút gốc và qua phải ở nút E, vì hình chữ nhật tìm kiếm hoàn toàn ở trên A và ở bên phải nút E. Kế đó, ở nút F, ta phải lần xuống theo cả hai cây con vì F rơi trong vùng (theo chiều x) xác định bởi hình chữ nhật (chú ý cẩn thận rằng điều này không đồng nghĩa với F rơi trong hình chữ nhật). Kế tiếp, các cây con bên trái của P và K được kiểm tra, tương ứng với việc kiểm tra

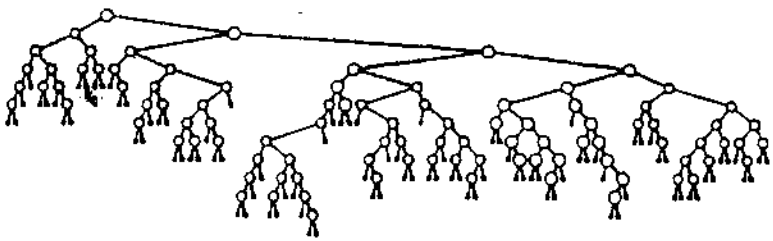


Hình 26.8 *Tìm theo khoảng nhờ một cây 2D (chia nhỏ mặt phẳng)*

các vùng phủ lên hình chữ nhật tìm kiếm. (xem hình 26.7 và 26.8).

Việc cài đặt tiến trình này là dễ dàng với sự mở rộng đơn giản từ thủ tục range 1D đã được xét ở đầu chương này:

```
procedure range (t:link; rect:rectangle; d:boolean);
   var t1, t2, tx1, tx2, ty1,ty2 : boolean;
   begin
   if tz then
      begin    tx1 := rect.x1  t↑.p.x; tx2 := t↑.p.x  rect.x2;
               ty1 := rect.y1  t↑.p.y; ty2 := t↑.p.y  rect.y2;
               if d    then begin t1 := tx1; t2:=tx2; end
                       else begin t1 := ty1; t2:=ty2; end;
               if t1 then range(t↑.l, rect, not d);
               if insiderect (t↑.p, rect)
                  then point t↑.p is within the range;
               if t2 then range (t↑.r, rect, not d);
      end;
   end;
```



Hình 26.9 *Tìm theo khoảng nhờ một cây 2D lớn*

Thủ tục này lần xuống hai cây con chỉ khi đường phân cách cắt ngang hình chữ nhật, điều này thường không xảy ra với những hình chữ nhật tương đối nhỏ. Hình 26.8 cho thấy các lần chia nhỏ mặt phẳng và các điểm được xét trong các ví dụ.

Tính chất 26.4 *Tìm kiếm vùng bằng cây 2D dùng khoảng R+logN bước để tìm R điểm trong các khoảng tìm trong miền chứa N điểm*

Phương pháp này vẫn còn đang được phân tích và tính chất nêu trên là một ước đoán hoàn toàn kinh nghiệm. Dĩ nhiên, sự thực hiện (và sự phân tích) luôn phụ thuộc vào kiểu của vùng được dùng. Nhưng phương pháp này rất đáng kể so với phương pháp lưới, và phần nào ít phụ thuộc vào "tính ngẫu nhiên" của tập điểm. Hình 26.9 cho thấy cây 2D của trường hợp có nhiều điểm.

TÌM KIẾM TRÊN VÙNG NHIỀU CHIỀU

Phương pháp lưới và cây 2D tổng quát hoá trực tiếp được cho nhiều chiều hơn: sự mở rộng đơn giản của các thuật toán trên tức khắc cho các phương pháp tìm kiếm làm việc trên vùng có số chiều lớn hơn 2. Tuy nhiên, bản chất của không gian nhiều chiều đòi hỏi phải thận trọng và tính năng của các thuật toán này có lẽ khó mà nói trước được cho một ứng dụng cụ thể nào đó.

Để cài đặt phương pháp lưới cho việc tìm kiếm trên vùng k chiều, ta tạo một lưới là mảng k chiều và dùng một chỉ mục cho mỗi chiều.

Vấn đề chính là tìm một giá trị hợp lý cho biến *size*. Vấn đề này hoàn toàn rõ ràng khi k lớn: kiểu lưới gì được dùng khi tìm kiếm trên vùng 10 chiều? Ngay như nếu ta chỉ dùng 3 lần chia cho mỗi chiều, ta cũng cần đến 3 lũy thừa 10 ô lưới, hầu hết các ô này là rỗng với những giá trị N bình thường.

Việc tổng quát từ cây 2D thành cây kD cũng đơn giản: xoay vòng lần lượt qua các chiều (như ta đã làm trong trường hợp 2 chiều bằng cách thay đổi luân phiên giữa x và y) khi lần xuống cây. Như trên, trong trường hợp ngẫu nhiên, cây kết quả có cùng các đặc trưng như cây tìm kiếm nhị phân. Cũng như trên, có sự tương ứng tự nhiên giữa các cây và các xử lý hình học. Trong trường hợp 3 chiều, sự phân cành ở mỗi nút tương ứng với việc dùng một mặt phẳng cắt miền tìm kiếm 3 chiều; một cách tổng quát, ta cắt miền k chiều bằng một siêu phẳng $(k-1)$ chiều.

Nếu k rất lớn, có sự chú ý nhiều về sự không cân bằng của cây kD, vì các tập điểm cụ thể có thể không đủ lớn để biểu lộ được sự ngẫu nhiên trên một số lớn các chiều. Đặc biệt, tất cả các điểm trong một cây con có thể sẽ có cùng một giá trị trên nhiều chiều, dẫn đến một số các cành một hướng trong cây. Một cách làm giảm vấn đề này, là thay vì xoay vòng qua các chiều, ta luôn chia các điểm theo chiều tốt nhất. Kỹ thuật này cũng áp dụng được cho cây 2D. Nó đòi hỏi có thông tin mở rộng (để chọn chiều) được lưu trong mỗi nút, nhưng nó làm giảm sự không cân bằng của cây, đặc biệt là trong các cây có số chiều cao.

Tóm lại, mặc dù dễ có được sự tổng quát các chương trình này trong bài toán tìm kiếm trên vùng nhiều chiều, nhưng biện pháp như vậy khó áp dụng hữu hiệu cho một ứng dụng lớn. Những cơ sở dữ liệu lớn, với nhiều thuộc tính trong một mẩu tin, có thể là những đối tượng rất phức tạp; và một sự hiểu biết rõ ràng về các đặc trưng của cơ sở dữ liệu là điều cần thiết để phát triển một phương pháp tìm kiếm vùng có hiệu quả cho một ứng dụng cụ thể. Đây là bài toán rất quan trọng vẫn còn đang được nghiên cứu.

BÀI TẬP

1. Viết bản không đệ quy cho chương trình tìm kiếm trên vùng 1 chiều

2. Viết chương trình in tất cả các điểm trong cây nhị phân mà các điểm này không rơi trong khoảng tìm

3. Cho số lớn nhất và nhỏ nhất các ô được xét trong phương pháp lưới như là các hàm theo số chiều của lưới và hình chữ nhật tìm kiếm.

4. Khảo sát ý đồ tránh sự tìm kiếm ở các ô rỗng bằng cách dùng danh sách liên kết: mỗi ô lưới có thể liên kết với một ô không rỗng kế tiếp cùng hàng và một ô không rỗng kế cận cùng cột. Dùng sơ đồ như vậy ảnh hưởng như thế nào đến kích thước của ô lưới?

5. Vẽ cây và các lần chia mặt phẳng nếu ta xây dựng cây 2D cho tập điểm bắt đầu với đường chia dọc. (Nghĩa là, gọi *range* với tham biến thứ ba là *false* thay vì *true*)

6. Hãy cho một tập điểm dẫn đến cây 2D xấu nhất, trong đó không nút nào có hai con; hãy chỉ ra các lần chia mặt phẳng

7. Mô tả cách thay đổi như thế nào trong mỗi phương pháp để tìm các điểm rơi trong một vòng tròn cho trước.

8. Với tất cả hình chữ nhật tìm kiếm trong cùng một vùng, dạng nào làm cho mỗi phương pháp có sự thực hiện kém nhất?

9. Phương pháp tìm kiếm vùng nào được ưa chuộng hơn khi các điểm co cụm thành các đám lớn cách xa nhau

10. Vẽ cây 3D do việc chèn các điểm (3,1,5), (4,8,3), (8,3,9), (6,2,7), (1,6,3), (1,3,5), (6,4,2) vào cây rỗng ban đầu

27

GIAO CỦA CÁC ĐỐI TƯỢNG

Một bài toán thường gặp trong các ứng dụng liên quan đến các dữ liệu hình học là: "Cho một tập N đối tượng, có hai đối tượng nào giao nhau không ?". "Đối tượng" có thể là đường thẳng, hình chữ nhật, hình tròn, đa giác hay bất kỳ kiểu đối tượng hình học nào. Ví dụ, trong hệ thống thiết kế và xử lý mạch điện tích hợp hoặc in các tấm mạch điện, điều quan trọng là xét xem có hai dây nào giao nhau gây ra sự đoản mạch không. Trong nghiên cứu sản xuất, công thức toán học của nhiều vấn đề quan trọng cũng dẫn đến bài toán giao hình học.

Giải pháp hiển nhiên cho bài toán tìm giao là kiểm tra mỗi cặp đối tượng có giao nhau hay không. Vì có khoảng $N^2/2$ cặp đối tượng, thời gian thực hiện của thuật toán này là tỉ lệ với N^2. Với một vài áp dụng, điều này không đáng kể vì có những hệ số giới hạn số đối tượng được xử lý. Tuy nhiên, nhiều ứng dụng thông thường khác, phải giải quyết hàng trăm, hàng ngàn và thậm chí hàng triệu đối tượng. Thuật toán thô thiển cỡ N^2 thì rõ ràng không thích hợp cho các ứng dụng như vậy. Trong chương này, ta sẽ nghiên cứu một phương pháp tổng quát để xác định có 2 đối tượng giao nhau hay không trong một tập có N đối tượng; phương pháp này dựa trên các thuật toán của M.Shamos và D.Hoey.

Trước tiên, ta sẽ khảo sát thuật toán tìm tất cả các cặp giao nhau trong tập các đoạn thẳng dọc và ngang. Điều này làm vấn đề có vẻ

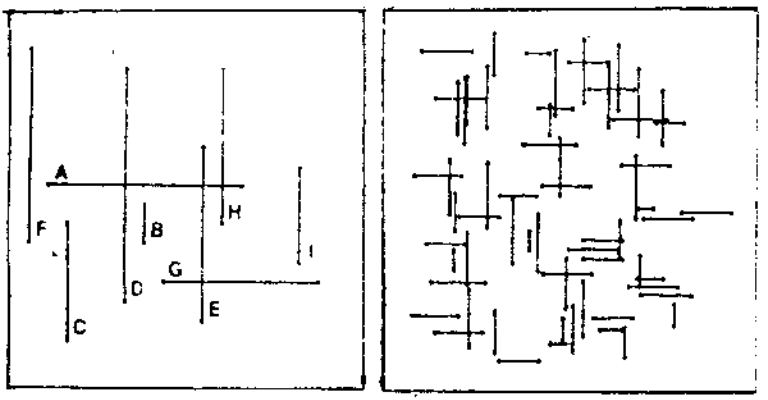
dễ hơn (các đoạn thẳng dọc và ngang là các đối tượng hình học đơn giản) và khó hơn (tìm tất cả các cặp giao nhau khó hơn là xác định xem có cặp nào giao nhau không). Ta sẽ cài đặt thuật toán này bằng cách ứng dụng cây tìm kiếm nhị phân và trình tìm kiếm vùng đã xét ở chương trước trong cùng một thủ tục đệ quy kép.

Sau đó, ta sẽ xét bài toán tìm một cặp giao nhau trong tập N đoạn thẳng có phương tự do. Chiến lược tổng quát được áp dụng ở đây là tương tự như chiến lược trong trường hợp các đoạn thẳng dọc-ngang. Thực ra, việc tìm giao của các đối tượng hình học khác được thực hiện theo cùng một ý tưởng cơ bản. Tuy nhiên, với các đoạn thẳng và các đối tượng khác, việc mở rộng để trả về tất cả các cặp giao nhau thì hơi phức tạp hơn trường hợp các đoạn thẳng dọc-ngang.

CÁC ĐƯỜNG THẲNG DỌC VÀ NGANG

Để bắt đầu, ta giả thiết là tất cả các đoạn thẳng là dọc hoặc ngang: đoạn thẳng được xác định bằng hai đầu mút hoặc có cùng hoành độ hoặc có cùng tung độ (xem ví dụ trong hình 27.1). (Điều này thường được gọi là *hình học Manhattan* vì thành phố Manhattan gồm nhiều đường ngang và dọc). Giả thiết hạn chế về các đoạn thẳng không làm bài toán trở thành vô nghĩa. Thực vậy, sự hạn chế này là nhằm vào các ứng dụng cụ thể: ví dụ, việc thiết kế các mạch điện tích hợp rất lớn. Trong hình 27.1 bên phải, các đoạn thẳng tương đối ngắn là điều thường xảy ra trong nhiều ứng dụng (dù rằng có thể có một số đoạn rất dài).

Sơ đồ tổng quát của thuật toán tìm một giao điểm trong tập các đoạn thẳng là dùng một đường thẳng ngang quét từ dưới lên trên. Chiếu lên đường quét này, những đoạn thẳng dọc biến thành các điểm và những đoạn thẳng ngang biến thành các khoảng. Khi đường quét ngang đi từ dưới lên, các điểm (biểu diễn các đoạn thẳng dọc)

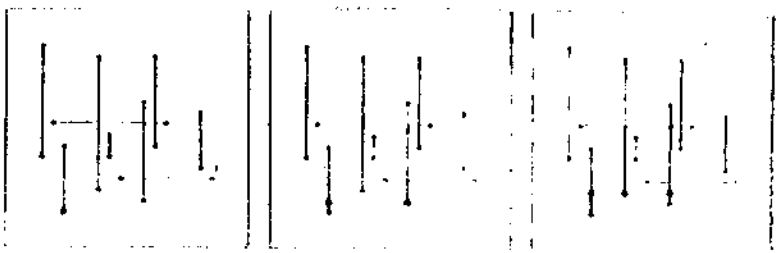


Hình 27.1 *Những bài toán giao của hai đường thẳng*

xuất hiện rồi biến mất và các đoạn thẳng ngang thì xuất hiện cách quãng. Giao điểm xuất hiện khi một khoảng trên đường quét (biểu diễn một đoạn thẳng ngang) chứa một điểm (biểu diễn một đoạn thẳng dọc). Bằng cách này, bài toán 2 chiều tìm giao điểm của một cặp đường thẳng được thu lại thành bài toán tìm kiếm vùng 1 chiều.

Dĩ nhiên, không thực sự cần thiết phải quét đường thẳng ngang qua toàn bộ tập các đoạn thẳng. Vì chỉ cần thao tác với những điểm cuối của các đoạn thẳng nên ta bắt đầu bằng cách sắp xếp các đoạn thẳng theo tung độ y, rồi xử lý chúng theo thứ tự này. Nếu gặp đầu thấp của một đoạn thẳng dọc, ta thêm hoành độ x của đoạn thẳng này vào cây tìm kiếm nhị phân (ở đây gọi là cây x); nếu gặp đầu cao của đoạn thẳng dọc, ta xoá đoạn thẳng đó khỏi cây và nếu gặp đoạn thẳng ngang, ta thực hiện việc tìm kiếm vùng theo 2 hoành độ. Như ta thấy, cần một số chú ý khi xử lý các đầu mút đoạn thẳng có toạ độ bằng nhau.

Hình 27.2 cho thấy các bước đầu tiên của việc quét tìm các giao điểm trong ví dụ bên trái hình 27.1. Việc quét bắt đầu từ điểm có tung độ y thấp nhất, là đầu thấp của đoạn thẳng C. Sau đó tới E, rồi



Hình 27.2 *Các bước đầu tiên của việc quét tìm các giao điểm*

D. Hình 27. 3 cho thấy phần còn lại của tiến trình: đoạn thẳng ngang G được xử lý kế tiếp, được kiểm tra có giao với C, D và E hay không (là các đoạn dọc cắt đường quét ngang)

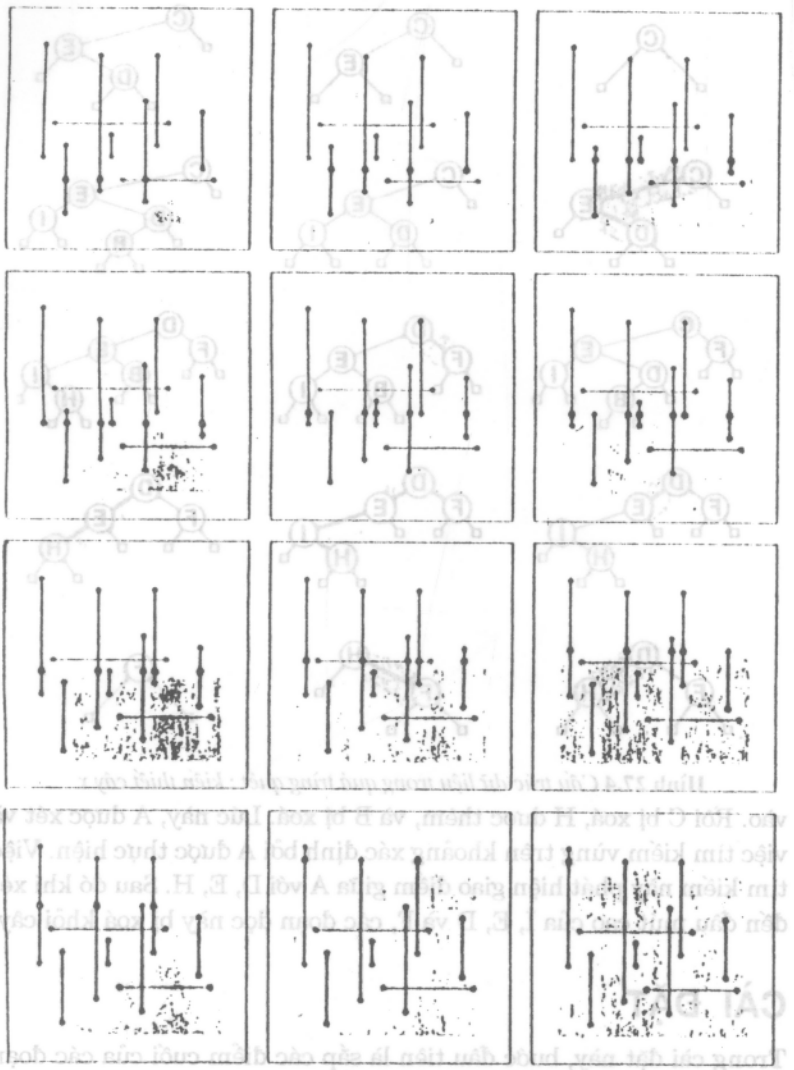
Để cài đặt việc quét, ta chỉ cần sắp các điểm cuối của các đoạn thẳng theo tung độ *y*. Trong ví dụ, ta được danh sách

C E D G I B F C H B A I E D H F

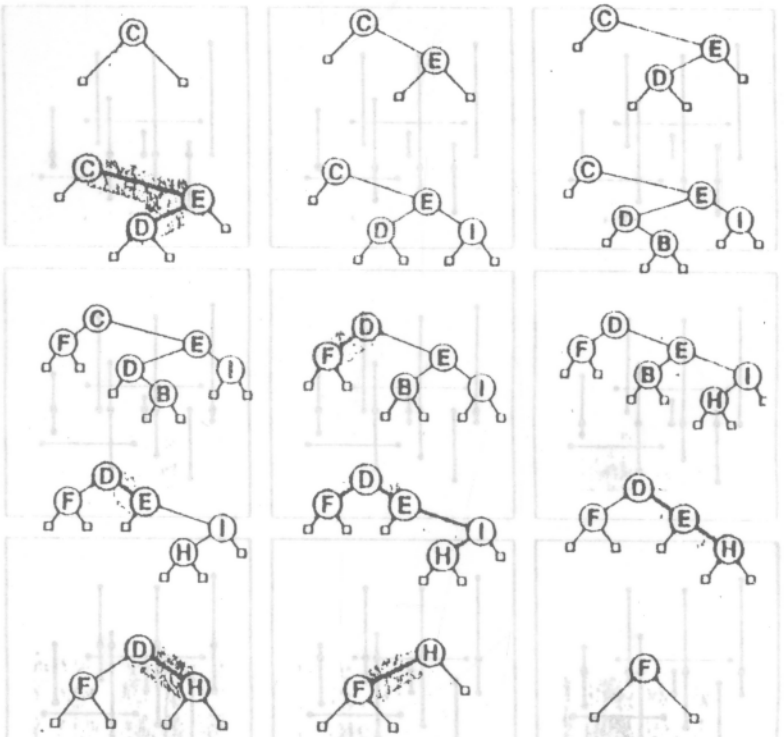
Trong danh sách này, mỗi đoạn thẳng dọc xuất hiện 2 lần và mỗi đường thẳng ngang xuất hiện 1 lần. Danh sách đã sắp này được coi như một dãy các lệnh insert (khi xét đầu thấp của các đoạn thẳng dọc) và delete (khi xét đầu cao của các đoạn thẳng dọc). Tất cả những "lệnh" này là các lời gọi đơn giản đến các thủ tục cây nhị phân chuẩn trong Chương 14 và 26, dùng hoành độ *x* làm khoá.

Hình 27.4 cho thấy việc xử lý cấu trúc cây x trong suốt quá trình quét. Mỗi nút trong cây tương ứng với một đoạn thẳng dọc và trong quá trình cấu trúc cây, khoá được dùng là hoành độ x. Vì E ở bên phải C, nó là cây con bên phải của C. Dòng đầu tiên trong hình 27.4 tương ứng với hình 27.2 và phần còn lại tương ứng với hình 27.3.

Khi gặp một đoạn thẳng ngang, nó thường được dùng để thực hiện việc tìm kiếm vùng trên cây : các giao điểm sẽ tương ứng với tất cả đoạn thẳng dọc ở trong vùng tạo bởi đoạn thẳng ngang. Trong ví dụ này, ta tìm được giao điểm giữa E và G; rồi I, B và F được thêm



Hình 27.3 *Các bước đầy đủ của việc quét tìm các giao điểm*



Hình 27.4 *Cấu trúc dữ liệu trong quá trình quét : kiến thiết cây x*

vào. Rồi C bị xoá, H được thêm, và B bị xoá. Lúc này, A được xét và việc tìm kiếm vùng trên khoảng xác định bởi A được thực hiện. Việc tìm kiếm này phát hiện giao điểm giữa A với D, E, H. Sau đó khi xét đến đầu mút cao của I, E, D và F, các đoạn dọc này bị xoá khỏi cây.

CÀI ĐẶT

Trong cài đặt này, bước đầu tiên là sắp các điểm cuối của các đoạn thẳng theo tung độ. Nhưng vì các cây nhị phân, sẽ được dùng để duy

trì trạng thái của các đoạn thẳng dọc mà vẫn giữ nguyên các đoạn thẳng ngang, các cây này cũng được dùng để khởi tạo cho việc sắp xếp theo *y*! Đặc biệt, ta sẽ dùng 2 cây nhị phân "gián tiếp", với nút đầu *hy* và *hx*. Cây *y* sẽ chứa tất cả điểm cuối các đoạn thẳng và các điểm này được xử lý từng cái một theo thứ tự trên. Cây *x* chứa các đoạn thẳng đang cắt đường quét ngang. Ta khởi tạo *hx* và *hy* có khoá là 0 và các con trỏ chỉ đến một nút rỗng (như trong Chương 14). Sau đó, ta xây dựng cây *hy* bằng cách thêm vào tung độ *y* của các đoạn thẳng dọc và ngang:

```
procedure buildytree;
  var x1, y1, x2, y2 : integer;
  begin
    hy := bstinitialize; N:=0;
    repeat
      N := N+1;
      read (x1, y1, x2, y2); if eoln then readln;
      lines[N].p1.x := x1; lines[N].p1.y := y1;
      lines[N].p2.x := x2; lines[N].p2.y := y2;
      bstinsert (N, y1, hy);
      if y2<>y1 then bstinsert (N, y2, hy);
    until eof;
  end;
```

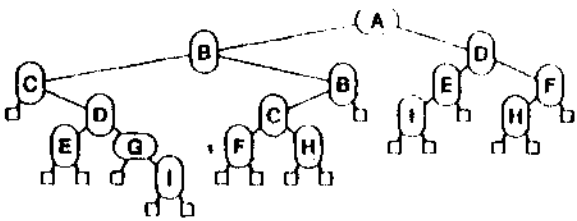
Chương trình này đọc nhóm 4 số (tương ứng với một đoạn thẳng) và mang các số này vào mảng *lines* và vào cây tìm kiếm nhị phân theo *y*. Ta dùng thủ tục chuẩn *bstinsert* trong Chương 14, với khoá là tung độ *y*, và lập chỉ mục cho mảng lines theo trường info. Hình 27.5 minh hoạ việc xây dựng cây.

Bây giờ, việc sắp xếp có hiệu quả theo *y* được thực hiện bằng cách dùng thủ tục duyệt cây theo thứ tự giữa (xem chương 4 và 14). Ta thăm tất cả các nút theo thứ tự *y* tăng bằng cách thăm tất cả nút của

cây con trái, thăm gốc và thăm tất cả nút của cây con phải của cây *hy*. Cùng lúc, ta duy trì một cây khác (gốc là hx) như đã nói trên, để làm phép duyệt theo đường thẳng quét ngang.

```
procedure scan (next : link);
   var t, x1, x2, y1, y2 : integer; int : interval;
   begin
   if next<>z then
      begin
      scan(next↑.l);
      x1:=lines[next↑.info].p1.x; y1:=lines[next↑.info].p1.y;
      x2:=lines[next↑.info].p2.x; y2:=lines[next↑.info].p2.y;
      if x2<x1 then begin t:=x2; x2:=x1; x1:=t; end;
      if y2<y1 then begin t:=y2; y2:=y1; y1:=t; end;
      if next↑.key=y1 then bstinsert(next↑.info.x1.hx);
      if next↑.key=y2 then
         begin
            bstdelete(next↑.info.x1.hx);
            int.x1:=x1; int.x2:=x2;
            bstrange(hx↑.r, int);
         end;
      scan (next↑.r);
      end;
   end;
```

Từ mô tả trên, dễ dàng viết thủ tục thăm một nút. Đầu tiên, các toạ độ của đầu mút đoạn thẳng tương ứng được lấy lại từ mảng *lines* (đã lập chỉ mục theo trường *info* của nút). Rồi đem so sánh trường key của nút với các toạ độ này để xác định xem nút này tương ứng với đầu cao hay đầu thấp của đoạn thẳng : nếu là đầu thấp, nút này được đưa vào cây hx và nếu là đầu cao thì xoá khỏi cây hy; sau đó ta tiến hành tìm kiếm vùng. Cách cài đặt hơi khác với mô tả này ở chỗ các đoạn thẳng ngang được chèn thực sự vào cây *hx*, rồi được xoá



Hình 27.5 *Sắp xếp để quét bằng cây y*

ngay lập tức và việc tìm kiếm vùng được thực hiện cho khoảng "dài 1 điểm" của các đoạn thẳng dọc. Điều này làm chương trình vận hành đúng đắn trong cả trường hợp các đoạn dọc phủ lên nhau (được xem là giao nhau).

Cách tiếp cận trộn các thủ tục đệ quy thao tác trên *x* và *y* là rất quan trọng trong các thuật toán hình học. Một ví dụ khác là thuật toán cây 2D trong chương trước, và ta sẽ xét thêm một ví dụ trong chương sau.

Tính chất 27.1 *Có thể tìm tất cả giao điểm của N đoạn thẳng dọc và ngang trong thời gian tỉ lệ với NlogN + I (với I là số giao điểm).*

Tính trung bình, thời gian thao tác trên cây tỉ lệ với *logN* (nếu cây cân bằng được dùng, trường hợp xấu nhất tốn thời gian cỡ *logN*), nhưng thời gian tiêu tốn cho *bstrange* cũng phụ thuộc vào tổng số giao điểm. Nói chung, số giao điểm có thể rất lớn. Ví dụ, nếu ta có *N*/2 đoạn thẳng ngang và *N*/2 đoạn thẳng dọc được sắp theo kiểu các đường chéo song song, thì tổng số giao điểm là N^2.

Cùng với tìm kiếm vùng, nếu đã biết trước số giao điểm là rất lớn, thì một số tiếp cận đơn giản cũng nên được dùng. Điển hình, các ứng dụng thuộc kiểu "mò kim đáy biển" có rất nhiều đoạn thẳng được kiểm tra nhưng chỉ có vài giao điểm.

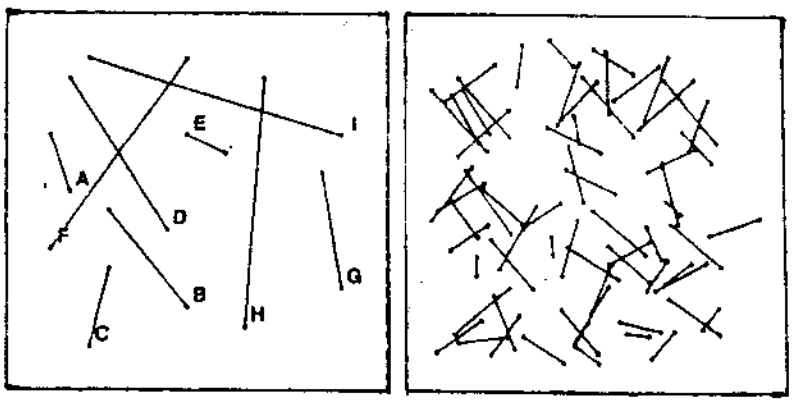
GIAO CỦA CÁC ĐƯỜNG THẲNG TỔNG QUÁT

Khi cho phép các đoạn thẳng có độ dốc tuỳ ý, tình thế trở nên phức tạp hơn, như trong hình 27.6. Đầu tiên, các đoạn thẳng có phương tuỳ ý dẫn đến sự cần thiết phải kiểm tra tường minh xem một cặp đoạn thẳng là có giao nhau hay không - ta không thể tìm giao điểm chỉ bằng một phép kiểm tra khoảng đơn giản. Thứ nhì, quan hệ thứ tự giữa các đoạn thẳng trên cây nhị phân là phức tạp hơn trước đây, vì nó phụ thuộc vào vùng y đang xét. Thứ ba, các giao điểm bất kỳ làm nảy sinh việc thêm các giá trị y mới mà có thể là sẽ khác với tập các giá trị y ta lấy từ các đầu mút đoạn thẳng.

Thành ra những vấn đề này có thể xử lý trong một thuật toán với cùng cấu trúc cơ bản đã nêu. Để việc thảo luận được đơn giản, ta sẽ xét thuật toán xác định có hay không một cặp giao nhau trong tập N đoạn thẳng và thảo luận việc mở rộng thuật toán này như thế nào để tìm tất cả giao điểm.

Như trên, trước tiên ta sắp theo y để chia không gian thành các dải không chứa các đầu mút đoạn thẳng. Rồi ta cũng duyệt qua danh sách các điểm, thêm các đoạn thẳng vào cây tìm kiếm nhị phân khi gặp đầu mút thấp và xoá nó khi gặp đầu mút cao. Cũng như trên, cây nhị phân cho biết thứ tự các đoạn thẳng xuất hiện trong dải ngang giữa hai giá trị y liên tiếp. Ví dụ, trong dải giữa đầu thấp của D và đầu cao của B trong hình 27.6, các đoạn thẳng sẽ xuất hiện theo thứ tự F B D H G. Ta giả sử rằng không có giao điểm nào trong dải ngang đang xét: mục tiêu của chúng ta là giữ nguyên cấu trúc cây và dùng nó để tìm giao điểm.

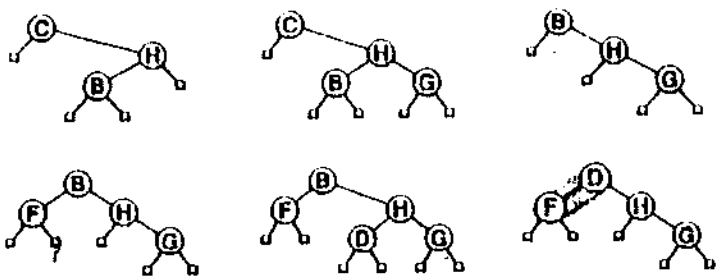
Để xây dựng cây, ta không thể chỉ đơn giản dùng hoành độ của các đầu mút đoạn thẳng làm khoá (trong ví dụ trên, thực hiện điều này sẽ làm sai thứ tự của B và D). Thay vào đó, ta dùng một quan hệ thứ tự phức tạp hơn: một đoạn thẳng x được định nghĩa là ở bên phải



Hình 27.6 *Hai bài toán giao của các đường thẳng tổng quát*

đoạn thẳng *y* nếu cả hai đầu mút của *x* đều ở cùng một bên của *y* như là một điểm xa về bên phải; ta cũng định nghĩa tương tự cho bên trái. Như vậy, trong sơ đồ trên, B ở bên phải A và B ở bên phải C (vì C ở bên trái B). Nếu *x* không ở bên trái mà cũng không ở bên phải *y*, thì chúng phải giao nhau. Thao tác "so sánh đoạn thẳng" tổng quát được cài đặt bằng cách dùng thủ tục *ccw* ở chương 24. Các thủ tục về cây tìm kiếm nhị phân chuẩn cũng được dùng (kể cả cây cân bằng nếu cần thiết). Hình 27. 7 cho thấy sự thay đổi của cây khi gặp đoạn thẳng C, rồi gặp D. Thực tế, mỗi thao tác được hoàn thành trong suốt thủ tục xử lý cây là một phép kiểm tra giao điểm các đoạn thẳng: nếu thủ tục cây tìm kiếm nhị phân không thể quyết định sang trái hay phải thì các đoạn thẳng này là phải giao nhau.

Nhưng dây chưa phải là tất cả, bởi vì thao tác so sánh tổng quát chưa được *chuyển đổi*. Trong ví dụ trên, F ở bên trái B (vì B là ở bên phải F) và B là ở bên trái D, nhưng F *không ở bên trái* D. Cần chú ý điều này, vì thủ tục xoá trên cây nhị phân giả thiết rằng phép so sánh là đã thay đổi: khi B bị xoá trên cây cuối cùng trong dãy trên,



Hình 27.7 *Cấu trúc dữ liệu (cây X) trong bài toán tổng quát*

cây trên hình 27.7 được tạo thành mà không có bất kỳ sự so sánh tường minh nào của F và D. Để thuật toán tìm giao điểm này làm việc đúng đắn, ta phải kiểm tra tường minh các phép so sánh là hợp lệ mỗi khi ta thay đổi cấu trúc cây. Đặc biệt, mỗi lúc đặt liên kết bên trái của nút x chỉ đến nút y, ta kiểm tra một cách rõ ràng theo định nghĩa trên xem đoạn thẳng tương ứng với x là có ở bên trái đoạn thẳng tương ứng với y hay không; và tương tự cho bên phải. Dĩ nhiên, sự so sánh này được dùng để tìm giao điểm.

Tóm lại, để tìm một giao điểm của N đoạn thẳng, ta dùng lại chương trình trước, ngoại trừ lời gọi *range* được bỏ qua và mở rộng thủ tục cây nhị phân bằng cách dùng sự so sánh tổng quát như đã nói. Nếu không có giao điểm nào, ta bắt đầu bằng cây rỗng và kết thúc bằng một cây rỗng không có việc tìm bất kỳ những đoạn thẳng nào không thể so sánh được. Nếu có giao điểm, thì 2 đường thẳng phải được so sánh với nhau trong tiến trình quét và sẽ tìm được giao điểm.

Tuy nhiên, mỗi khi tìm được giao điểm, ta không thể chỉ chú ý và hy vọng tìm được một giao điểm khác, bởi vì hai đoạn thẳng giao nhau sẽ đổi chỗ theo thứ tự ngay lập tức, cách xử lý vấn đề này là dùng một hàng chờ có độ ưu tiên thay cho cây nhị phân trong việc sắp theo y: đầu tiên, đặt các đoạn thẳng vào hàng đợi có ưu tiên tuỳ

theo toạ độ y của các điểm cuối của chúng, rồi chuyển dần đường quét lên bằng cách lấy liên tiếp toạ độ y nhỏ nhất từ hàng chờ và thực hiện việc thêm, xoá cây nhị phân như đã nêu. Khi tìm thấy giao điểm, các thành phần mới được thêm từng cái một vào hàng chờ, dùng giao điểm này làm điểm thấp hơn cho mỗi cái.

Một cách khác để tìm tất cả giao điểm, chỉ thích hợp nếu không có quá nhiều giao điểm, là chỉ đơn giản bỏ đi các đoạn thẳng giao nhau khi giao điểm được tìm thấy. Rồi sau khi việc quét hoàn tất, ta biết rằng tất cả cặp giao nhau phải dính líu đến một trong các đoạn thẳng đó và ta có thể dùng một phương pháp đơn giản để liệt kê tất cả giao điểm.

Tính chất 27.2 *Tất cả giao điểm của N đoạn thẳng có thể tìm trong thời gian tỉ lệ với NlogN+I, ở đây I là số các giao điểm.*

Điều này được suy trực tiếp từ các thảo luận ở trên.

Đặc điểm thú vị của thủ tục trên là nó có thể được làm thích hợp bằng cách chỉ thay đổi thủ tục so sánh tổng quát để kiểm tra sự tồn tại của một cặp giao nhau trong tập các dạng hình học tổng quát hơn.

Ví dụ, ta cài đặt một thủ tục so sánh 2 hình chữ nhật mà các cạnh của chúng là dọc hoặc ngang. Hai hình chữ nhật được so sánh theo quy tắc bình thường là hình chữ nhật *x* là ở bên trái hình chữ nhật *y* nếu cạnh phải của *x* ở bên trái của cạnh trái của *y*. Như vậy ta có thể dùng phương pháp trên để kiểm tra sự giao nhau của tập các hình chữ nhật này.

Với đường tròn, ta có thể dùng hoành độ *x* của tâm để sắp thứ tự và kiểm tra tường minh sự giao nhau (ví dụ, so sánh khoảng cách giữa hai tâm và tổng bán kính của chúng). Lần nữa, nếu thủ tục so sánh này được dùng trong phương pháp trên, ta có một thuật toán tìm giao điểm trong tập các đường tròn. Bài toán tìm tất cả giao điểm trong những trường hợp như vậy thì phức tạp hơn rất nhiều

mặc dù phương pháp đơn giản đề cập trong chương trước vẫn dùng được nếu chỉ có vài giao điểm. Một tiếp cận khác đủ dùng cho nhiều ứng dụng khác là xem xét những đối tượng phức tạp như tập các đoạn thẳng và dùng thủ tục tìm giao điểm các đoạn thẳng.

BÀI TẬP

1. Tìm cách xác định hai tam giác có giao nhau không? Hình vuông? Đa giác n cạnh với n>4?

2. Trong thuật toán tìm giao các đoạn thẳng dọc-ngang, ở trường hợp xấu nhất có bao nhiêu cặp được kiểm tra nhưng không có giao điểm? Cho một sơ đồ minh hoạ câu trả lời của bạn.

3. Điều gì xảy ra khi thủ tục tìm giao các đoạn thẳng dọc và ngang được áp dụng cho tập các đoạn thẳng có phương tuỳ ý?

4. Viết chương trình tìm số cặp giao nhau trong tập N đoạn thẳng dọc và ngang ngẫu nhiên, mỗi đoạn thẳng được tạo với các toạ độ nguyên ngẫu nhiên từ 0 đến 1000 và một bit ngẫu nhiên để phân biệt đoạn thẳng là dọc hay ngang.

5. Hãy cho một phương pháp để kiểm tra xem một đa giác là đơn hay không (đa giác không tự cắt)?

6. Hãy cho một phương pháp để kiểm tra xem một đa giác là có chứa một đa giác khác?

7. Giải bài toán tìm giao của các đoạn thẳng tổng quát với giả thiết sự phân cách tối thiểu giữa hai đoạn thẳng là chiều dài lớn nhất của chúng.

8. Cho cấu trúc cây nhị phân khi dùng thuật toán tìm giao các đoạn thẳng cho tập các đoạn thẳng trong hình 27.6 được quay 90 độ.

9. Các thủ tục so sánh cho các vòng tròn và các hình chữ nhật Manhattan như mô tả trong chương này có chuyển đổi với nhau được không ?

10. Viết chương trình tìm các cặp giao nhau trong tập N đoạn thẳng ngẫu nhiên, mỗi đoạn được tạo với các toạ độ nguyên ngẫu nhiên từ 0 đến 1000.

28

BÀI TOÁN ĐIỂM GẦN NHẤT

Các bài toán hình học thường xét đến các khoảng cách giữa các điểm. Ví dụ, một bài toán quen thuộc trong nhiều ứng dụng là bài toán *tìm láng giềng gần nhất*: tìm điểm gần nhất của một điểm cho trước trong một tập cho trước. Điều này có lẽ bao gồm việc kiểm tra khoảng cách giữa điểm được cho với mỗi điểm trong tập điểm đã cho, nhưng có những cách giải quyết khác tốt hơn. Trong đoạn này, ta sẽ xem xét một vài bài toán khác về khoảng cách, một thuật toán mẫu và một cấu trúc hình học cơ sở là *biểu đồ Voronoi* được dùng có hiệu quả trong các biến thể của những bài toán như vậy. Cách tiếp cận của chúng ta là sẽ thảo luận một phương pháp tổng quát để giải các bài toán điểm gần nhất thông qua việc xem xét kỹ lưỡng một cài đặt mẫu giải quyết một bài toán đơn giản.

Một số bài toán trong chương này tương tự với các bài toán tìm kiếm vùng của Chương 26, và các phương pháp lưới, phương pháp cây 2D cũng được triển khai thích hợp cho bài toán láng giềng gần nhất và các bài toán khác. Tuy nhiên, thiếu sót lớn nhất của những phương pháp này là chúng dựa vào tính ngẫu nhiên của tập điểm: chúng sẽ làm việc rất tồi trong trường hợp xấu nhất. Mục đích chúng ta trong chương này là khảo sát một tiếp cận tổng quát khác mà vẫn bảo đảm sự thực hiện tốt trong nhiều bài toán, không phụ thuộc số liệu vào. Một vài phương pháp quá phức tạp để cài đặt đầy đủ, và chúng đòi hỏi tổng phí đủ để các phương pháp đơn giản hơn

lại có thể làm việc tốt hơn trên những tập không lớn lắm hoặc khi tập điểm này được phân tán đủ tốt. Tuy nhiên, sự nghiên cứu các phương pháp như vậy với việc thực hiện trong những trường hợp xấu nhất sẽ cho thấy một vài đặc tính cơ bản cần được hiểu rõ của các tập điểm, ngay như nếu các phương pháp đơn giản là tốt hơn trong các tình huống đặc biệt.

Ta sẽ xét một tiếp cận tổng quát, dùng các thủ tục đệ quy kép để liên kết các xử lý theo hai trục toạ độ. Hai phương pháp trước đã xét (cây kD và giao điểm) dựa trên cây tìm kiếm nhị phân; phương pháp ở đây là "chia để trị" dựa trên *mergesort*.

BÀI TOÁN TÌM CẶP ĐIỂM GẦN NHẤT

Bài toán cặp điểm gần nhất tìm hai điểm gần nhau nhất trong một tập điểm cho trước. Bài toán này có quan hệ với bài toán tìm láng giềng gần nhất; mặc dù không được ứng dụng rộng rãi bằng.

Có vẻ như ta phải tính khoảng cách giữa mọi cặp điểm để tìm khoảng cách nhỏ nhất : cho N điểm sẽ có thời gian thực hiện tỉ lệ với N^2. Tuy nhiên, ta có thể dùng một thuật toán sắp xếp để chỉ phải tính chừng $NlogN$ khoảng cách trong trường hợp xấu nhất (về mặt trung bình sẽ ít hơn nhiều) và thời gian thực hiện trong trường hợp xấu nhất là $NlogN$ (về mặt trung bình sẽ tốt hơn nhiều). Trong đoạn này, ta sẽ xem xét chi tiết một thuật toán như vậy.

Ta sẽ dùng một thuật toán dựa trên chiến lược "chia để trị". Ý tưởng ở đây là sắp xếp các điểm theo một trục toạ độ, như trục x chẳng hạn, rồi dùng thứ tự này để chia tập điểm thành hai phần. Trong toàn bộ tập điểm đã cho, cặp điểm gần nhất hoặc là cặp gần nhất trong cùng một bên nào đó, hoặc là một cặp điểm cắt ngang đường thẳng phân giới giữa hai tập điểm thành phần. Dĩ nhiên, trường hợp đáng chú ý là khi cặp điểm gần nhất cắt ngang đường phân giới. Cặp điểm gần nhất trong mỗi nữa bên rõ ràng là tìm được

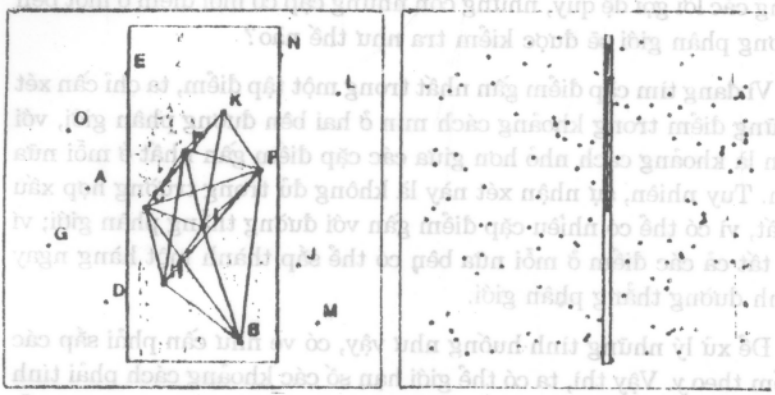
bằng các lời gọi đệ quy, nhưng còn những cặp có mỗi điểm ở một bên đường phân giới sẽ được kiểm tra như thế nào?

Vì đang tìm cặp điểm gần nhất trong một tập điểm, ta chỉ cần xét những điểm trong khoảng cách min ở hai bên đường phân giới, với min là khoảng cách nhỏ hơn giữa các cặp điểm gần nhất ở mỗi nửa bên. Tuy nhiên, sự nhận xét này là không đủ trong trường hợp xấu nhất, vì có thể có nhiều cặp điểm gần với đường thẳng phân giới; ví dụ tất cả các điểm ở mỗi nửa bên có thể sắp thành một hàng ngay cạnh đường thẳng phân giới.

Để xử lý những tình huống như vậy, có vẻ như cần phải sắp các điểm theo y. Vậy thì, ta có thể giới hạn số các khoảng cách phải tính như sau: xử lý các điểm theo chiều tăng của y, kiểm tra xem mỗi điểm có nằm trong dải đứng chứa các điểm trong phạm vi min kể từ đường phân giới. Với mỗi điểm trong dải trên, tính khoảng cách giữa điểm này với các điểm cũng trong dải và có tung độ y nhỏ hơn tung độ của điểm đang xét nhưng không nhỏ quá min. Do khoảng cách giữa các điểm ở mỗi nửa bên tối thiểu là min nên số điểm phải kiểm tra sẽ ít hơn.

Trong tập điểm ở bên trái hình 28.1, đường phân cách thẳng đứng tưởng tượng ở ngay bên phải điểm F; có 8 điểm ở bên trái và 8 điểm bên phải đường phân cách. Cặp điểm gần nhất ở nửa trái là AC (hoặc AO), cặp điểm gần nhất ở nửa phải là JM. Nếu các điểm được sắp theo y, thì cặp điểm gần nhất bị chia bởi đường phân giới được tìm bằng cách xét các cặp HI, CI, FK (cặp gần nhất trong toàn bộ tập điểm), và EK. Với tập nhiều điểm hơn, dải chứa các cặp điểm gần nhất cắt ngang đường phân giới sẽ hẹp hơn (xem phần phải của hình 28.1)

Mặc dù thuật toán này đơn giản, có một vài chú ý để cài đặt có hiệu quả. Ví dụ, có thể sẽ trả giá đắt để sắp các điểm theo y bằng một thủ tục đệ quy. Ta đã xét nhiều thuật toán có thời gian thi hành được mô tả bằng biểu thức quy nạp $C_N = 2C_{N/2} + N$, dẫn đến C_N là tỉ lệ



Hình 28.1 *Tiếp cận "chia-để-trị" để tìm cặp điểm gần nhất*

với $NlogN$; nếu ta sắp theo y thì biểu thức quy nạp sẽ là $CN = 2*CN/2 + NlogN$, dẫn đến C_N tỉ lệ với $Nlog^2N$ (xem Chương 6) . Để loại bỏ điều này, ta tránh sắp theo y.

Lời giải vấn đề này thì đơn giản nhưng tinh vi. Phương pháp *mergesort* trong Chương 12 dựa trên việc chia những phần tử được sắp, cũng giống như việc chia những điểm đã nêu trên. Ta có hai vấn đề và có chung một lời giải cho chúng, như vậy ta có thể giải chúng đồng thời ! Đặc biệt, ta sẽ viết một thủ tục đệ quy để vừa sắp theo y và vừa tìm cặp điểm gần nhất. Thủ tục này sẽ chia đôi tập điểm, rồi gọi lại chính nó để sắp hai nữa bên theo y và tìm cặp điểm gần nhất trong mỗi nữa, sau đó trộn hai nửa bên để hoàn tất việc sắp theo y và áp dụng lại thủ tục trên để hoàn tất việc tính cặp điểm gần nhất. Trong cách này, ta tránh được chi phí cho sự thêm việc sắp theo y bằng cách trộn dữ liệu được yêu cầu trong việc sắp xếp với dữ liệu được yêu cầu khi tính cặp điểm gần nhất.

Khi sắp theo y, việc chia đôi có thể được làm bằng bất kỳ cách nào, nhưng với phép tính cặp điểm gần nhất, việc chia đôi yêu cầu có một nửa bên có hoành độ x nhỏ hơn nửa bên còn lại. Điều này được

cài đặt dễ dàng bằng cách sắp theo x trước khi chia đôi tập điểm. Thực tế, ta có thể dùng cũng thủ tục này để sắp theo x ! Khi đã hiểu dự định tổng thể, việc cài đặt không còn khó khăn nữa.

Như đã lưu ý ở trên, việc cài đặt sẽ dùng các thủ tục *sort* và *merge* đệ quy ở Chương 12. Bước đầu tiên là thay đổi các cấu trúc danh sách để lưu các điểm thay cho các khoá, và thay đổi thủ tục *merge* để kiểm tra biến toàn cục *pass* dùng cho việc chọn cách so sánh. Nếu *pass=1*, ta sẽ so sánh hoành độ x của hai điểm; nếu *pass=2* ta so sánh tung độ y giữa hai điểm. Trình *merge* được cài đặt như sau:

```
function merge (a,b : link) : link;
  var c: link; comp: boolean;
  begin
  c:= z;
  repeat
    if pass=1
    then comp := a↑.p.x < b↑.p.x
    else comp := a↑.p.y < b↑.p.y;
    if comp
    then begin c↑.next:=a; c:=a; a:=a↑.next; end
    else begin c↑.next:=b; c:=b; b:=b↑.next; end
  until c=z;
  merge:=z↑.next;
  z↑.next:=z;
  end;
```

Nút rỗng z ở cuối danh sách làm điểm cầm canh với toạ độ x, y giả. Ta dùng một thủ tục đơn giản khác để kiểm tra khoảng cách giữa hai điểm đã cho có nhỏ hơn biến toàn cục *min* hay không. Nếu nhỏ hơn, biến *min* sẽ lưu lại khoảng cách này và các điểm nút được lưu vào hai biến toàn cục *cp1* và *cp2*.

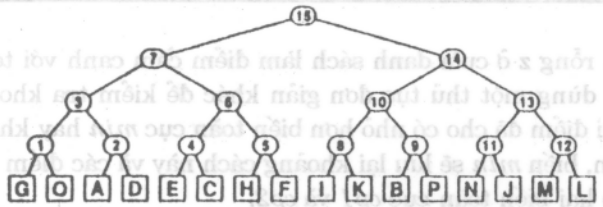
```
procedure check(p1, p2 : point);
  var dist : real;
  begin if (p1.y <> z↑.p.y) and (p2.y <> z↑.p.y) then
        begin
        dist:=sqrt((p1.x-p2.x)*(p1.x-p2.x)+(p1.y-p2.y)*(p1.y-p2.y));
        if (dist < min) then
        begin min:=dist; cp1:=p1; cp2:=p2; end;
        end;
  end;
```

Như vậy, min luôn chứa khoảng cách giữa *cp1* và *cp2*, là hai điểm gần nhất được thấy tới lúc này.

Bước thứ hai là thay đổi thủ tục *sort* đệ quy ở Chương 12 để tìm điểm gần nhất khi *pass=2* (Xem chương trình hàm *sort* ở trang sau).

Nếu *pass=1*, đây chính là thủ tục *mergesort* đệ quy ở Chương 12: nó trả lại một danh sách kết nối các điểm được sắp theo hoành độ *x* (vì *merge* được thay đổi như đã nêu trên để so sánh theo *x* trong pass1). Điểm hay của cài đặt này là khi *pass=2*. chương trình không chỉ sắp xếp theo *y* (vì *merge* được thay đổi như đã nêu trên để so sánh theo *y* trong *pass2*) mà còn hoàn thành việc tính điểm gần nhất sẽ được mô tả chi tiết sau.

Trước tiên, ta sắp xếp theo x, rồi sắp xếp theo y và tìm cặp điểm gần nhất bằng lời gọi sort như đoạn trình thứ hai ở trang sau.



Hình 28.2 *Cây đệ quy trong việc tính cặp điểm gần nhất*

```
function sort (c:link; N:integer) : link;
  var a,b:link; i:integer; middle:real; p1,p2,p3,p4:point;
  begin
    if c↑.next = z
    then sort:=c
    else  begin    a := c;
          for i:=2 to N div 2 do c:=c↑.next;
          b := c↑.next; c↑.next := z;
          if pass=2 then middle:=b↑.p.x;
          c := merge(sort(a,N div 2), sort (b,N-(N div 2)));
          sort := c;
          if pass=2 then
            begin a:=c;
            p1:=z↑.p; p2:=z↑.p; p3:=z↑.p; p4:=z↑.p;
            repeat
            if abs(a↑.p.x - middle) min then
              begin    check(a↑.p,p1);    check(a↑.p,p2);
                       check(a↑.p,p3); check(a↑.p,p4);
                       p1:=p2; p2:=p3; p3:=p4; p4:=a↑.p;
              end;
            a := a↑.next;
            until a = z
            end;
          end;
  end;
```

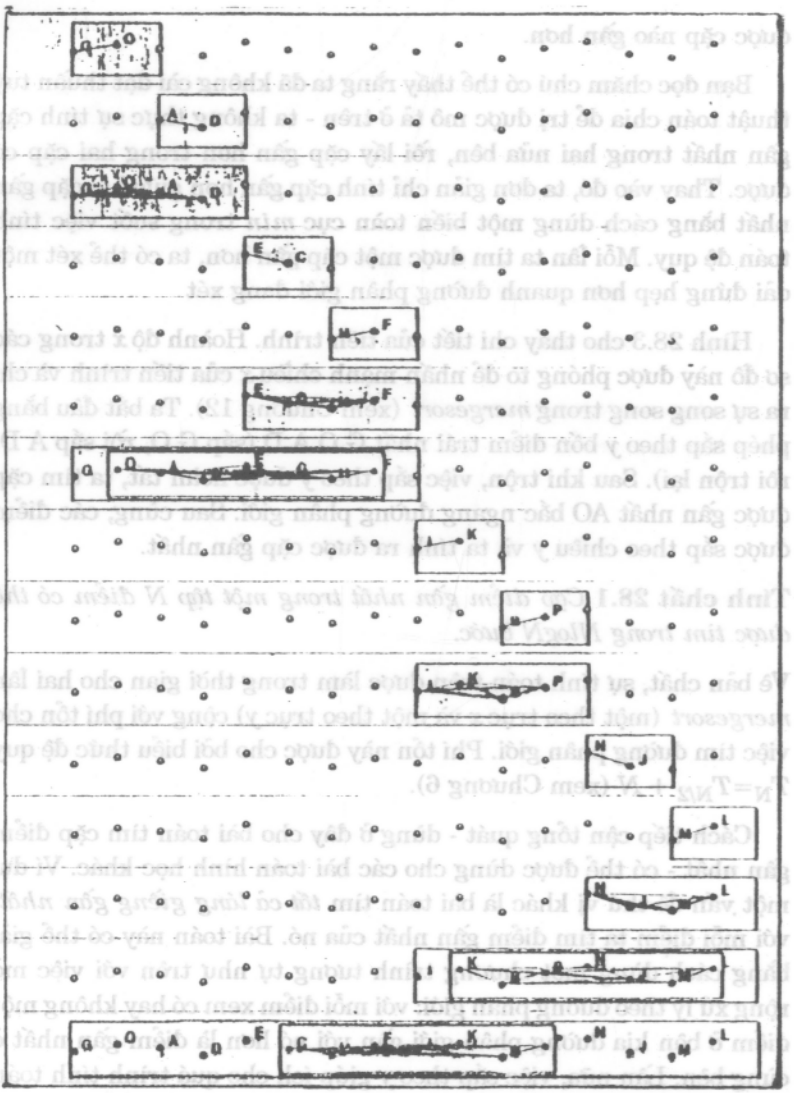
```
new(z); z↑.next:=z;
z↑.p.x:=maxint;
z↑.p.y:=maxint;
new(h); h↑.next:=readlist;
min:=maxint;
pass:=1; h↑.next:=sort(h↑.next, N);
pass:=2; h↑.next:=sort(h↑.next, N);
```

Sau đó, hai biến toàn cục *cp1* và *cp2* (trong thủ tục *check* "tìm cái nhỏ nhất") chứa cặp điểm gần nhất.

Vấn đề then chốt trong cài đặt này là thao tác *sort* khi *pass*=2. Trước khi vào các lời gọi đệ quy, các điểm đã được sắp theo *x*; trật tự này được dùng để chia đôi các điểm và tìm hoành độ *x* của đường phân giới. Sau các lời gọi đệ quy, các điểm được sắp theo *y* và khoảng cách giữa các cặp điểm trong mỗi nửa bên là lớn hơn *min*. Trật tự theo *y* được dùng để quét các điểm ở gần đường phân giới; giá trị *min* được dùng để giới hạn số điểm kiểm tra. Mỗi điểm ở trong phạm vi *min* kể từ đường phân giới được kiểm tra với từng điểm trong 4 điểm trước đó và 4 điểm này cũng ở trong phạm vi *min* kể từ đường phân giới (độc giả có thể tự kiểm tra lại điều này). Sự kiểm tra này bảo đảm tìm ra cặp điểm gần nhau hơn *min* và mỗi điểm nằm ở một bên của đường phân giới. Ta biết rằng các điểm rơi vào cùng một bên của đường phân giới cách nhau một khoảng tối thiểu là *min*, như vậy số các điểm rơi vào vòng tròn có bán kính *min* sẽ được giới hạn lại.

Hình 28.2 cho thấy cây biểu diễn các lời gọi đệ quy, mô tả sự hoạt động của thuật toán trên một tập điểm. Trên cây này, nút trong biểu diễn đường phân giới chia tập điểm vào cây con trái và phải. Các nút được đánh số theo thứ tự các trục đứng được dùng đến trong thuật toán. Sự đánh số này đúng với cách duyệt trên cây theo thứ tự trước - bởi vì đường phân giới được dùng đến sau các lời gọi đệ quy trong chương trình và là một cách đơn giản để xem xét trật tự của các lần trộn được thực hiện trong suốt quá trình đệ quy của *mergesort* (xem Chương 12)

Như vậy, đầu tiên là đường thẳng giữa G và O được xét và cặp GO được lưu lại như là một cặp gần nhất được tìm thấy. Sau đó, đường thẳng giữa A và D được xét, nhưng A và D quá xa - không làm thay đổi *min*. Rồi đường thẳng giữa O và A được xét và các cặp GD, GA và OA lần lượt là các cặp gần nhất. Trong ví dụ này, cho tới khi cặp cuối cùng FK được kiểm tra trong lần chia cuối cùng, ta không tìm



Hình 28.3 *Tính cặp điểm gần nhất (hoành độ x được phóng to)*

được cặp nào gần hơn.

Bạn đọc chăm chú có thể thấy rằng ta đã không cài đặt thuần tuý thuật toán chia để trị được mô tả ở trên - ta không thực sự tính cặp gần nhất trong hai nửa bên, rồi lấy cặp gần hơn trong hai cặp có được. Thay vào đó, ta đơn giản chỉ tính cặp gần hơn giữa hai cặp gần nhất bằng cách dùng một biến toàn cục *min* trong suốt việc tính toán đệ quy. Mỗi lần ta tìm được một cặp gần hơn, ta có thể xét một dải đứng hẹp hơn quanh đường phân giới đang xét

Hình 28.3 cho thấy chi tiết của tiến trình. Hoành độ x trong các sơ đồ này được phóng to để nhấn mạnh chiều x của tiến trình và chỉ ra sự song song trong *mergesort* (xem Chương 12). Ta bắt đầu bằng phép sắp theo y bốn điểm trái nhất G O A D (sắp G O, rồi sắp A D, rồi trộn lại). Sau khi trộn, việc sắp theo y được hoàn tất, ta tìm cặp được gần nhất AO bắc ngang đường phân giới. Sau cùng, các điểm được sắp theo chiều y và ta tính ra được cặp gần nhất.

Tính chất 28.1 *Cặp điểm gần nhất trong một tập N điểm có thể được tìm trong NlogN bước.*

Về bản chất, sự tính toán trên được làm trong thời gian cho hai lần *mergesort* (một theo trục x và một theo trục y) cộng với phí tổn cho việc tìm đường phân giới. Phí tổn này được cho bởi biểu thức đệ quy $T_N=T_{N/2} + N$ (xem Chương 6).

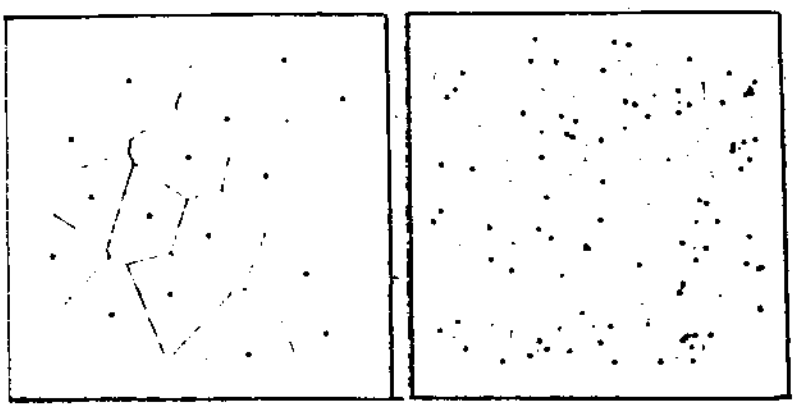
Cách tiếp cận tổng quát - dùng ở đây cho bài toán tìm cặp điểm gần nhất - có thể được dùng cho các bài toán hình học khác. Ví dụ, một vấn đề thú vị khác là bài toán tìm *tất cả láng giềng gần nhất:* với mỗi điểm ta tìm điểm gần nhất của nó. Bài toán này có thể giải bằng cách dùng một chương trình tương tự như trên với việc mở rộng xử lý theo đường phân giới: với mỗi điểm xem có hay không một điểm ở bên kia đường phân giới gần với nó hơn là điểm gần nhất ở cùng bên. Lần nữa, việc sắp theo y giúp ích cho quá trình tính toán này.

BIỂU ĐỒ VORONOI

Trong một tập cho trước, tập các điểm gần với một điểm cho trước hơn các điểm khác, là một cấu trúc hình học đặc biệt - gọi là *đa giác Voronoi* của điểm đó. Hợp tất cả đa giác Voronoi trong một tập điểm gọi là *biểu đồ Voronoi* của tập điểm đó. Đây là nguyên tắc cơ bản trong tính toán điểm gần nhất; ta sẽ thấy hầu hết các bài toán gặp phải, bao gồm các khoảng cách giữa các điểm, có lời giải thú vị và tự nhiên dựa trên biểu đồ Voronoi. Các sơ đồ - trong tập điểm ví dụ trên- được chỉ trong hình 28.4.

Đa giác Voronoi cho một điểm được dựng bằng các đường trung trực của các đoạn liên kết điểm này với các điểm gần nhất. Định nghĩa thực sự của nó là một cách đi vòng khác: Đa giác Voronoi được định nghĩa là chu vi của tập điểm phẳng gần với điểm được cho hơn các điểm khác trong tập điểm này, và mỗi cạnh của đa giác Voronoi ngăn cách điểm đã cho với một điểm "gần nhất" của nó.

Đối ngẫu của biểu đồ Voronoi được chỉ trong hình 28.5, tạo ra một sự tương ứng rõ ràng: trong đối ngẫu, một đường thẳng được vẽ giữa mỗi điểm và tất cả điểm "gần nhất" của điểm này. Hình này

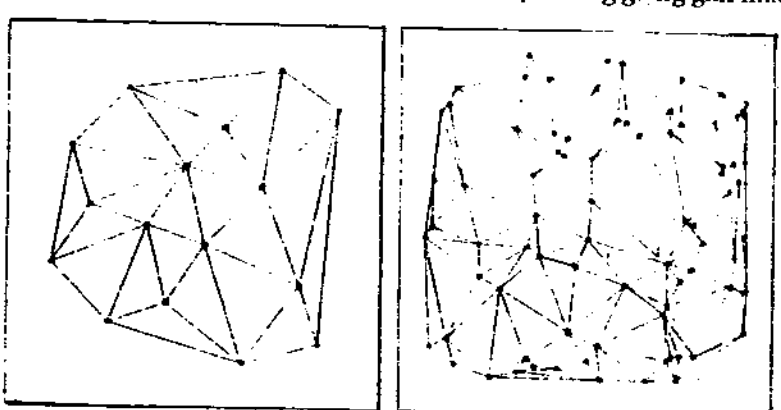


Hình 28.4 *Biểu đồ Voronoi*

cũng được gọi là hình ba cạnh Delaunay. Nói cách khác, x và y là liên thông trong đối ngẫu Voronoi nếu đa giác Voronoi của chúng có một cạnh chung.

Biểu đồ Voronoi và hình ba cạnh Delaunay có nhiều đặc tính dẫn đến các thuật toán hiệu quả cho bài toán điểm gần nhất. Đặc tính làm những thuật toán này hiệu quả là số đường thẳng trong cả biểu đồ và đối ngẫu là tỉ lệ với một hằng số nhỏ N. Ví dụ, đường thẳng nối cặp điểm gần nhất phải là đối ngẫu, như vậy bài toán của phần trước có thể được giải bằng cách tìm đối ngẫu và sau đó chỉ còn tìm đoạn thẳng ngắn nhất trong số các đoạn thẳng đối ngẫu. Tương tự, đường thẳng nối mỗi điểm với láng giềng gần nhất của nó phải là đối ngẫu, như vậy bài toán tìm tất cả láng giềng gần nhất thu gọn lại thành việc tìm đối ngẫu. Bao đóng của một tập điểm là một phần của đối ngẫu, như vậy việc tìm đối ngẫu Voronoi cũng là một thuật toán khác tìm bao lồi. Chúng tôi sẽ chỉ một ví dụ khác trong Chương 31 mà nó được giải quyết hiệu quả bằng cách trước tiên tìm đối ngẫu Voronoi.

Sự định nghĩa của biểu đồ Voronoi hàm chứa việc nó được dùng để giải bài toán láng giềng gần nhất : để xác định láng giềng gần nhất



Hình 28.5 *Hình ba cạnh Delaunay*

của một điểm trong một tập điểm, ta chỉ cần tìm một đa giác Voronoi mà điểm đó rơi vào. Có thể tổ chức các đa giác Voronoi trong một cấu trúc như cây 2D để cho sự tìm kiếm có hiệu quả.

Biểu đồ Voronoi có thể được tính bằng cách dùng một thuật toán với cùng một cấu trúc như trong thuật toán tìm cặp điểm gần nhất ở trên. Trước tiên, các điểm được sắp theo hoành độ x của chúng. Sau đó, trật tự này được dùng để chia đôi tập điểm, dẫn đến hai lời gọi đệ quy để tìm biểu đồ Voronoi cho mỗi nữa tập điểm. Cùng lúc, các điểm được sắp theo tung độ y; cuối cùng, hai sơ đồ Voronoi của hai nữa bên được trộn lại với nhau. Cũng như trên, sự trộn này (được thực hiện với *pass=2*) có thể khai thác sự kiện các điểm được sắp theo x trước các lời gọi đệ quy và các điểm được sắp theo y và các biểu đồ Voronoi của hai nữa được xây dựng sau các lời gọi đệ quy. Tuy nhiên, ngay với những sự giúp đỡ này, việc trộn cũng đã là một công việc hoàn toàn rắc rối và việc thực hiện một cài đặt đầy đủ là ra ngoài phạm vi cuốn sách này.

Biểu đồ Voronoi tất nhiên là một cấu trúc bản chất của bài toán điểm gần nhất, và việc hiểu các đặc trưng của một bài toán theo nghĩa biểu đồ Voronoi hay đối ngẫu của nó là một bài toán đáng giá. Tuy nhiên, trong nhiều vấn đề cụ thể, một cài đặt trực tiếp dựa trên sơ đồ tổng quát đã cho trong chương này có thể là đã thích hợp. Sơ đồ này là đủ mạnh để tìm biểu đồ Voronoi, và cũng đủ cho các thuật giải dựa trên biểu đồ Voronoi, và nó có thể cho mã hiệu quả hơn, đơn giản hơn, như ta đã thấy trong bài toán cặp điểm gần nhất.

BÀI TẬP

1. Viết chương trình giải bài toán láng giềng gần nhất, trước tiên dùng phương pháp lưới, sau đó dùng cây 2D.

2. Mô tả điều gì xảy ra khi thủ tục tìm cặp điểm gần nhất được dùng cho một tập điểm rơi trên cùng một đường thẳng dọc và cách đều nhau.

3. Mô tả điều gì xảy ra khi thủ tục tìm cặp điểm gần nhất được dùng cho một tập điểm rơi trên cùng một đường thẳng ngang và cách đều nhau.

4. Cho tập có $2N$ điểm, một nửa có hoành độ x dương, một nửa có hoành độ x âm, tìm cặp điểm gần nhất với mỗi điểm nằm ở một bên.

5. Tìm các cặp điểm liên tiếp được gán trong *cp1* và *cp2* khi chương trình trên được thực hiện với tập điểm trong ví dụ và đã bỏ đi điểm A.

6. Kiểm tra tính hiệu quả của việc cho biến min là biến toàn cục bằng cách so sánh sự thực hiện trên một tập ngẫu nhiên nhiều điểm của cài đặt đã cho với cài đặt đệ quy thuần tuý.

7. Cho thuật toán tìm cặp gần nhất trong một tập đường thẳng.

8. Vẽ sơ đồ Voronoi và đối ngẫu của nó cho các điểm A B C D E F trong ví dụ.

9. Cho một phương pháp thô thiển (có thể cần thời gian thực hiện tỉ lệ với N^2) để tìm sơ đồ Voronoi.

10. Viết chương trình, cũng dùng cấu trúc đệ quy như trong cài đặt tìm cặp điểm gần nhất, để tìm bao lồi của một tập điểm.

29

CÁC THUẬT TOÁN CƠ BẢN VỀ ĐỒ THỊ

Rất nhiều bài toán có thể diễn đạt dưới dạng những đối tượng và các đường nối chúng với nhau. Ví dụ, trên một bản đồ đường hàng không của miền đông Nước Mỹ, chúng ta cần trả lời các câu hỏi chẳng hạn như: "Muốn đi từ Providence đến Princeton thì đường bay nào nhanh nhất?" Hoặc nếu chúng ta quan tâm đến chi phí hơn là thời gian thì chúng ta lại cần chọn đường bay nào rẻ nhất khi đi từ Providence đến Princeton. Để trả lời những câu hỏi như thế chúng ta chỉ cần các thông tin về các đường nối (đường hàng không) giữa các đối tượng (các tỉnh) với nhau.

Các mạch điện cũng là một ví dụ khác mà trong đó các đường nối đóng vai trò chính. Các thiết bị điện như bóng bán dẫn, điện trở, điện dung được mắc với nhau một cách phức tạp. Các mạch như thế có thể được biểu diễn và xử lý trên máy tính để trả lời từ các câu hỏi đơn giản như: "Tất cả các thiết bị có được nối lại với nhau hay không?" cho đến những câu hỏi phức tạp như: "Nếu mạch này được chế tạo, nó sẽ hoạt động như thế nào ?" Trong tình huống này, trả lời cho câu hỏi đầu tiên chỉ phụ thuộc vào tính chất của các đường nối (dây điện), trong khi trả lời cho câu hỏi thứ hai lại yêu cầu thông tin chi tiết cả về các đường nối lẫn các đối tượng mà chúng nối lại.

Một ví dụ thứ ba là "Bảng điều phối công việc", trong trường hợp nầy các đối tượng là các công việc cần phải thực hiện (ví dụ trong một qui trình sản xuất), và các đường nối cho biết việc nào nên làm trước các việc khác. Đối với ví dụ nầy ta cần trả lời các câu hỏi giống như: "Mỗi công việc nên được thực hiện vào lúc nào?"

Đồ Thị là một công cụ toán học xây dựng mô hình cho các vấn đề nói trên. Chương nầy sẽ điểm qua vài tính chất cơ bản của đồ thị và các chương sắp tới sẽ nghiên cứu các thuật toán khác nhau giải quyết các câu hỏi có dạng như trên.

Thật ra, chúng ta cũng đã gặp các đồ thị trong những chương trước. Các cấu trúc dữ liệu liên kết chính là sự biểu diễn của các đồ thị, và chúng ta sẽ thấy rằng một vài thuật toán về đồ thị lại tương tự với các thuật toán đã gặp đối với cây và các cấu trúc khác. Ví dụ, các máy trạng thái hữu hạn trong Chương 19 và Chương 20 đã được thể hiện dưới dạng các cấu trúc đồ thị.

Lý thuyết đồ thị là một nhánh quan trọng của toán học tổ hợp đã được nghiên cứu sâu sắc trong hàng trăm năm. Nhiều tính chất quan trọng và hữu ích của các đồ thị đã được chứng minh, nhưng cũng còn rất nhiều vấn đề khó chưa giải quyết xong. Ở đây, với mục đích tìm hiểu về các thuật toán cơ bản của đồ thị, chúng ta chỉ phác họa sơ qua một số vấn đề trong lãnh vực nầy.

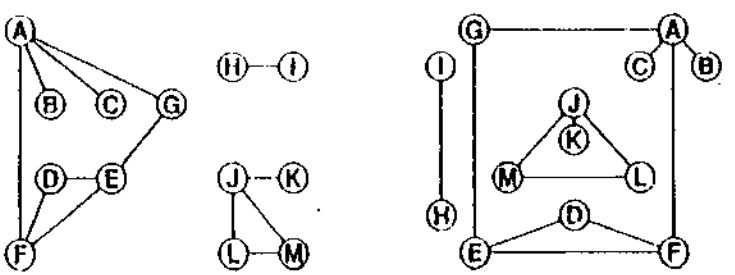
Cũng giống như các vấn đề khác mà chúng ta đã nghiên cứu, các đồ thị chỉ được xem xét dưới góc độ thuật toán trong những năm gần đây. Bên cạnh một vài thuật toán cơ bản rất cũ là nhiều thuật toán lý thú đã được khám phá trong vòng mười năm đổ lại. Ngay cả những thuật toán đồ thị tầm thường cũng đưa đến các chương trình máy tính thú vị, và các thuật toán không tầm thường (mặc dù có thể khó hiểu) mà chúng ta sẽ trình bày sẽ là các thuật toán đẹp và thú vị nhất.

THUẬT NGỮ

Có rất nhiều thuật ngữ liên quan đến các đồ thị. Chúng ta sẽ lần lượt định nghĩa hầu hết các thuật ngữ này, mặc dù một số thuật ngữ có thể chưa được sử dụng ngay.

Đồ thị là một tập hợp gồm các **đỉnh** và **cạnh**. Các đỉnh là các đối tượng đơn giản được đặt tên và gắn thêm một số tính chất khác; một cạnh là một đường nối giữa hai đỉnh. Người ta có thể vẽ một đồ thị bằng cách đánh dấu các đỉnh bởi các điểm trong mặt phẳng và vẽ các cạnh của đồ thị bằng các đoạn thẳng nối các điểm với nhau, nhưng cần phải nhớ rằng một đồ thị luôn được định nghĩa độc lập với biểu diễn của nó. Ví dụ, hai bản vẽ trong Hình 29.1 biểu diễn cùng một đồ thị. Chúng ta định nghĩa đồ thị này bằng cách nói rằng nó gồm một tập hợp đỉnh A B C D E F G H I J K L M và một tập hợp cạnh AG AB AC LM JM JL JK ED FD HI FE AF GE.

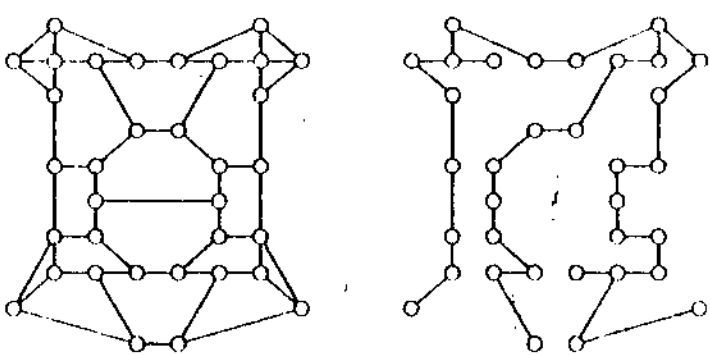
Trong một vài ứng dụng, chẳng hạn như đường hàng không trong ví dụ nói trên, có lẽ không nên sắp xếp lại các đỉnh như trong Hình 29.1. Nhưng với một số ứng dụng khác, chẳng hạn như các mạch điện, thì tốt nhất là chỉ tập trung vào các cạnh, các đỉnh và không phụ thuộc vào bất kỳ một vị trí hình học cụ thể nào. Một ví dụ



Hình 29.1 *Hai thể hiện của cùng một đồ thị.*

khác là các máy hữu hạn trong Chương 19 và 20 cũng không cần bàn đến vị trí hình học của các nút. Mối quan hệ giữa các thuật toán đồ thị và các bài toán hình học sẽ được thảo luận trong chi tiết hơn trong Chương 31. Bây giờ chúng ta sẽ tập trung vào các thuật toán đồ thị "thuần túy" làm việc trên một tập hợp đơn giản gồm các cạnh và nút.

Đường đi từ đỉnh x tới đỉnh y trong một đồ thị là một danh sách các đỉnh mà hai đỉnh liên tiếp nhau trong danh sách đó được nối bởi một cạnh của đồ thị. Ví dụ, BAFEG là một đường từ B tới G trong Hình 29.1. Một đồ thị được gọi là **liên thông** nếu hai đỉnh bất kỳ của nó luôn có đường đi từ đỉnh nầy tới đỉnh kia. Về mặt trực giác, nếu tưởng tượng các đỉnh là các đồ vật và các cạnh là dây nối chúng với nhau, thì một đồ thị liên thông sẽ giữ nguyên một chùm (không có vật nào bị rớt ra) nếu ta nhấc bất kỳ một vật lên. Một đồ thị không liên thông bao gồm các **thành phần liên thông**; ví dụ đồ thị trong Hình 29.1 có ba thành phần liên thông. Một **đường đi đơn** là một



Hình 29.2 *Một đồ thị lớn và cây tối đại của nó*

đường đi mà không có đỉnh nào được lặp lại. (Ví dụ BAFEGAC không là đường đi đơn.) Một **chu trình** là một đường đi đơn ngoại trừ đỉnh đầu tiên và đỉnh cuối cùng phải trùng nhau (nghĩa là một đường đi khởi đầu từ một đỉnh và trở về chính nó), ví dụ đường đi AFEGA là một chu trình.

Đồ thị không có bất kỳ chu trình nào được gọi là một cây (xem Chương 4). Một nhóm cây (trong đó không có hai cây nào được nối với nhau) được gọi là một rừng. **Cây bao trùm** của một đồ thị là một là một đồ thị con chứa tất cả các đỉnh nhưng chỉ chứa vừa đủ một số cạnh để tạo nên một cây. Ví dụ, các cạnh AB AD AF DE EG tạo nên một cây bao trùm cho bộ phận lớn của đồ thị trong Hình 29.1, Hình 29.2 minh họa một đồ thị lớn hơn và một trong các cây tối đại của nó.

Chú ý rằng nếu chúng ta thêm một cạnh bất kỳ vào một cây thì sẽ có một chu trình (bởi vì đã có sẵn một đường đi giữa hai đỉnh là các đầu mút của cạnh thêm vào). Như đã thấy trong Chương 4, một cây có V đỉnh sẽ có chính xác $V-1$ cạnh. Nếu một đồ thị có V đỉnh mà ít hơn $V-1$ cạnh thì nó sẽ không liên thông, nếu đồ thị có nhiều hơn $V-1$ cạnh thì nó phải có một chu trình.

Chúng ta sẽ ký hiệu số đỉnh trong một đồ thị đã cho là V, số cạnh là E. Chú ý rằng E có thể lấy giá trị bất kỳ từ 0 đến $V(V-1)/2$. Một đồ thị có tất cả các cạnh có thể có (tức là V đỉnh và $V(V-1)/2$ cạnh) được gọi là **đồ thị đầy đủ**. Đồ thị có ít cạnh (nhỏ hơn $V\log V$) được gọi là **đồ thị thưa**. Đồ thị mà hầu hết các đỉnh đều được nối với nhau được gọi là **đồ thị dày**.

Sự phụ thuộc vào hai tham số cạnh và đỉnh làm cho việc nghiên cứu so sánh các thuật toán về đồ thị phức tạp hơn nhiều so với nhiều thuật toán mà chúng ta đã nghiên cứu, bởi vì nhiều khả năng có thể xảy ra. Ví dụ một thuật toán A có thể cần V^2 bước, trong khi một thuật toán B (giải quyết cùng một vấn đề như A) lại cần $(E+V)\log E$ bước. Thuật toán B có thể tốt đối với các đồ thị thưa, nhưng thuật

toán A có thể được chuộng hơn đối với các đồ thị dày.

Các đồ thị đã được định nghĩa ở trên được gọi là các **đồ thị vô hướng**, đây là dạng đồ thị đơn giản nhất. Chúng ta cũng khảo sát các đồ thị phức tạp hơn, có nhiều thông tin gắn với các đỉnh và cạnh. Trong **đồ thị có trọng số**, các số được gắn với mỗi cạnh được gọi là **khoảng cách** hay **phí tổn**. Trong đồ thị có hướng, các cạnh hướng theo "một chiều": một cạnh có thể đi từ x tới y nhưng không đi ngược lại từ y tới x. Các đồ thị **vừa có trọng số vừa có hướng** đôi khi được gọi là các mạng lưới. Chúng ta sẽ thấy rằng các thông tin thêm vào như trọng lượng và hướng làm cho các thao tác phức tạp hơn các thao tác trong đồ thị vô hướng đơn giản.

BIỂU DIỄN CỦA ĐỒ THỊ

Để xử lý các đồ thị bằng chương trình máy tính, trước tiên chúng ta phải tìm cách để biểu diễn chúng trong máy tính. Chúng ta sẽ xem hai phương pháp biểu diễn thông dụng nhất, việc lựa chọn giữa hai phương pháp này tùy thuộc vào đồ thị dày hay thưa, mặt khác tính tự nhiên của các thao tác sẽ đóng một vai trò quan trọng.

Bước đầu tiên trong việc biểu diễn một đồ thị là đánh số các đỉnh của nó bằng các số nguyên từ 1 tới V. Lý lẽ chính để làm điều này là chúng ta cần truy xuất nhanh thông tin liên kết với mỗi đỉnh bằng cách dùng chỉ số mảng. Bất kỳ một phương pháp tìm kiếm nào (đã trình bày trong các chương trước) cũng có thể áp dụng vào vấn đề này; ví dụ chúng ta có thể chuyển các tên đỉnh thành các số nguyên giữa 1 và V bằng cách duy trì một bảng băm hay một cây nhị phân sao cho có thể tìm kiếm một số nguyên dương tương ứng với một tên đỉnh bất kỳ. Bởi vì chúng ta đã nghiên cứu các kỹ thuật này, chúng ta sẽ giả sử rằng có sẵn một **hàm chỉ mục** để đổi từ các tên đỉnh thành các số nguyên giữa 1 và V, và một **hàm tên** để đổi ngược lại từ các số nguyên thành các tên đỉnh.

Hình 29.3 *Biểu diễn đồ thị bằng ma trận kề*

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	0	0	1	1	0	0	0	0	0	0
B	1	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	1	0	0	0	0	0	0
F	1	0	0	1	1	1	0	0	0	0	0	0	0
G	1	0	0	0	1	0	1	0	0	0	0	0	0
H	0	0	0	0	0	0	0	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	1	0	0	0
J	0	0	0	0	0	0	0	0	1	1	1	0	0
K	0	0	0	0	0	0	0	0	0	1	1	1	0
L	0	0	0	0	0	0	0	0	0	0	1	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

Để các thuật toán đơn giản hơn, chúng ta ký hiệu tên mỗi đỉnh bằng một chữ cái, chữ thứ i trong bảng chữ cái sẽ được tương ứng với số nguyên i. Do đó việc cài đặt hai hàm tên và chỉ mục sẽ trở nên tầm thường đối với các thí dụ của chúng ta, có thể dễ dàng mở rộng các thuật toán để xử lý các đồ thị có tên đỉnh thật (trong thực tế) bằng cách dùng các kỹ thuật trong các Chương từ 14 đến 17.

Phương pháp biểu diễn trực quan nhất được gọi là biểu diễn ma trận kề. Chúng ta dùng một mảng hai chiều kích thước VxV, mỗi phần tử của nhận giá trị boolean, trong đó *a[x, y]* nhận giá trị *true* nếu có một cạnh từ đỉnh x nối tới đỉnh y và nhận giá trị false nếu ngược lại. Ma trận kề của đồ thị trong Hình 29.1 đượxc trình bày trong Hình 29.3 (1 có nghĩa là *true* và 0 có nghĩa là *false*).

Chú ý rằng mỗi cạnh được biểu diễn thực sự bằng hai bit, một cạnh nối x và y được biểu diễn bằng giá trị true trong cả hai phần tử của mảng *a[x, y]* và *a[y, x]*. Mặc dù có thể tiết kiệm bộ nhớ bằng cách chỉ lưu trữ một nữa của ma trận đối xứng nầy, nhưng để cho các thuật toán đơn giản người ta thường qui ước biểu diễn bằng ma trận đầy. Tương tự, chúng ta thường qui ước có một "cạnh" nối từ mỗi

đỉnh tới chính nó, do đó *a[x, x]* luôn nhận giá trị *1* với mọi *x* từ *1* tới *V*. (Trong một vài trường hợp thì các phần tử đường chéo nầy được giả sử nhận giá trị *0*, chúng ta cũng tự do làm như vậy khi thấy thích hợp.)

Một đồ thị được định nghĩa bằng một tập hợp đỉnh và một tập hợp cạnh nối các đỉnh với nhau. Để nhập một đồ thị vào máy tính, chúng ta phải quy định một cách thức để nhập các thành phần của nó, chẳng hạn chúng ta có thể dùng ma trận kề để nhập dữ liệu, nhưng giải pháp nầy không thích hợp đối với đồ thị thưa. Chúng ta cũng có thể dùng một cách thức nhập trực tiếp hơn như sau: trước tiên đọc các tên đỉnh, kế đến lần lượt từng cặp đỉnh tạo nên cạnh đồ thị. Như đã chú ý ở trên, một phương pháp có thể thực hiện dễ dàng là đọc tên các đỉnh vào một bảng băm hay cây nhị phân và gán cho tên mỗi đỉnh một số nguyên để dùng trong việc truy xuất chỉ số đỉnh giống như ma trận kề. Đỉnh được đọc lần thứ *i* có thể được tương ứng với số nguyên *i*. Để các chương trình đơn giản, chúng ta đọc *V* và *E* trước tiên, kế đến là các đỉnh và các cạnh. Một cách làm khác là dữ liệu nhập được sắp xếp nhờ vào các dấu cách giữa đỉnh và cạnh, chương trình có thể xác định *V* và *E* bằng cách dựa vào dữ liệu nhập. Thứ tự xuất hiện các cạnh đóng vai trò không quan trọng, nó không ảnh hưởng đến dạng của đồ thị và ma trận kề, chương trình ở trang sau cho phép nhập đồ thị vào máy tính.

Kiểu của V1 và V2 không được nói tới trong chương trình nầy, và đoạn chương trình cho hàm chỉ số cũng được bỏ qua. Tùy thuộc vào biểu diễn dữ liệu nhập, chúng ta sẽ xác định cụ thể, chẳng hạn có thể chọn kiểu *char* cho *v1*, *v2* và hàm chỉ số chỉ đơn giản là hàm *ord* của ngôn ngữ Pascal.

Biểu diễn ma trận kề chỉ thích hợp với đồ thị dày: ta cần V^2 bit để lưu trữ ma trận và V^2 bước để khởi tạo nó. Nếu số cạnh (tức là số lượng các bit 1) trong ma trận xấp xỉ V^2 thì biểu diễn ma trận kề phù hợp bởi vì chúng ta cần khoảng V^2 bước để đọc các cạnh của đồ thị.

```
program adjmatrix(input, output);
const maxV=50;
var j, x, y, V, E : integer; a : array [1..maxV,1..maxV] of boolean;
begin   readln(V,E);
        for x:=1 to V do for y:=1 to V do a[x,y]:=false;
        for x:=1 to V do a[x,x]:=true;
        for j:=1 to E do
        begin   readln(v1,v2);
                x:=index(v1); y:=index(v2);
                a[x,y]:=true;  a[y,x]:=true;
        end;
end;
```

Tuy nhiên nếu đồ thị thưa thì việc khởi tạo mà trận sẽ tốn thời gian nhiều so với toàn bộ thời gian chạy của thuật toán.

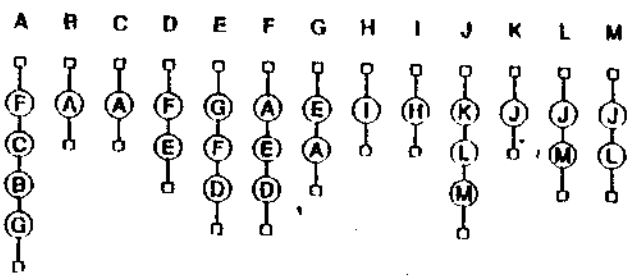
Bây giờ chúng ta hãy xem một biểu diễn thích hợp hơn cho các đồ thị không dày. Trong biểu diễn **cấu trúc kề**, tất cả các đỉnh được nối tới một đỉnh trong đồ thị được đặt vào **danh sách kề** của đỉnh đó. Cấu trúc này có thể cài đặt dễ dàng bằng cách dùng một xâu liên kết, chương trình dưới đây sẽ xây dựng cấu trúc kề cho đồ thị mẫu của chúng ta. Các xâu liên kết được tạo như thường lệ, với nút kết thúc là *z* (trỏ tới chính nó). Các nút bắt đầu xâu được lưu trong mảng *adj* và đặt chỉ mục dựa vào các đỉnh. Để thêm vào một cạnh nối đỉnh *x* tới đỉnh *y* trong biểu diễn này, chúng ta thêm *x* vào **xâu kề** của *y* và thêm *y* vào xâu kề của x như trong chương trình ở trang sau

Nếu nhập các cạnh vào theo thứ tự AG AB AC LM JM JL JK ED FD HI FE AF GE thì chương trình sẽ tạo nên cấu trúc xâu kề như mô tả trong Hình 29.4. Một lần nữa chúng ta hãy chú ý rằng mỗi cạnh được biểu diễn hai lần: một cạnh nối *x* và *y* được biểu diễn bằng một nút (chứa *x*) nằm trong danh sách kề của *y* và một nút (chứa *y*) nằm trong danh sách kề của *x*. Việc lưu trữ hai lần như trên rất quan trọng, bởi vì nếu ngược lại thì các câu hỏi chẳng hạn như "các nút

```
program adjlist(input, output);
const maxV=1000;
type link=↑node;
     node=record v:integer; next:link end;
var  j, x, y, V, E : integer; t, z : link;
     adj : array [1..maxV] of link;
begin   readln(V,E);    new(z); z↑.next:=z;
        for j:=1 to V do adj[j]:=z;
        for j:=1 to E do
        begin   readln(v1,v2);
                x:=index(v1); y:=index(v2);
                new(t); t↑.v:=x; t↑.next:=adj[y]; adj[y]:=t;
                new(t); t↑.v:=y; t↑.next:=adj[x]; adj[x]:=t;
        end;
end;
```

nào được nối trực tiếp tới nút x ?" sẽ không được trả lời nhanh chóng.

Trong biểu diễn này, thứ tự xuất hiện cạnh trong dữ liệu nhập vô cùng quan trọng. Cùng với phương pháp chèn vào xâu được dùng, thứ tự này qui định một thứ tự của các đỉnh trong các xâu kề. Do đó, cùng một đồ thị có thể biểu diễn nhiều cách khác nhau bằng cấu trúc



Hình 29.4 *Một biểu diễn cấu trúc kề*

xâu kề. Thật ra, rất khó có thể tiên đoán xâu kề như thế nào nếu chỉ dựa vào các cạnh, bởi vì mỗi cạnh phải cần hai lần chèn vào xâu kề.

Mặt khác thứ tự xuất hiện các cạnh trong xâu kề ảnh hưởng đến thứ tự các cạnh được xử lý trong các thuật toán. Trong khi mỗi thuật toán nên cho ra một câu trả lời đúng bất chấp thứ tự các cạnh trong xâu kề, nó lại phải tìm câu trả lời nhờ vào các dãy hoàn toàn khác nhau của các tính toán cho các thứ tự khác nhau, và nếu có nhiều hơn một "câu trả lời đúng", thì các thứ tự nhập vào khác nhau có thể dẫn tới các kết xuất khác nhau.

Một vài thao tác đơn giản không được cung cấp sẵn nếu chúng ta dùng cách biểu diễn này. Ví dụ người ta muốn xóa một đỉnh x và tất cả các cạnh nối với nó. Nếu chỉ xóa các nút khỏi xâu kề của x thì sẽ không đúng, bởi vì mỗi nút trong xâu kề của x lại qui định một đỉnh khác mà xâu kề của đỉnh này phải được duyệt toàn bộ để xóa một nút tương ứng với x. Vấn đề này có thể được giải quyết bằng cách liên kết tất cả các nút tương ứng với một cạnh cụ thể và tạo một xâu kép. Khi đó nếu một cạnh được xóa thì tất cả các nút tương ứng với cạnh đó có thể được xóa nhanh chóng. Dĩ nhiên rằng các liên kết thêm vào này sẽ cồng kềnh và không nên dùng chúng trừ khi các thao tác giống như thao tác xóa nói trên thực sự là cần thiết.

Các khảo sát nói trên đưa đến câu hỏi là tại sao chúng ta không dùng một biểu diễn "trực tiếp" cho các đồ thị: một cấu trúc dữ liệu mà mô hình hóa đồ thị một cách chính xác, trong đó các đỉnh được biểu diễn như các mẩu tin được cấp phát và các danh sách cạnh chứa các liên kết tới các đỉnh thay vì các tên đỉnh. Làm thế nào để thêm cạnh vào một đồ thị được biểu diễn bằng phương pháp này?

Đối với các đồ thị có hướng và có trọng số ta biểu diễn chúng bằng các cấu trúc tương tự như trên. Với đồ thị có hướng, mọi việc đều y hệt, ngoại trừ mỗi cạnh được biểu diễn chỉ một lần: một cạnh từ x tới y được biểu diễn bằng giá trị true trong phần tử $a[x,y]$ của ma trận kề hay một lần xuất hiện của y trong xâu kề của x nếu dùng cấu trúc

kề. Do đó một đồ thị vô hướng có thể được xem như một đồ thị có hướng mà với mỗi cặp đỉnh x, y nếu có cạnh (vô hướng) nối x và y thì sẽ có hai cạnh có hướng, trong đó một cạnh nối x tới y và một cạnh nối y tới x. Với đồ thị có trọng, mọi việc cũng y hệt, ngoại trừ chúng ta sẽ đặt các phần tử trong ma trận kề với các trọng lượng thay vì giá trị bool (quy ước trọng lượng không tồn tại bởi giá trị *false*); nếu dùng cấu trúc kề thì ta thêm vào một trường (để lưu trọng lượng cạnh) vào mỗi mẩu tin của xâu kề.

Thông thường các thông tin khác được kết hợp với mỗi đỉnh của đồ thị sẽ rất cần thiết khi mô hình hóa các đối tượng phức tạp hơn. Thông tin thêm vào mỗi đỉnh có thể được lưu trữ được đặt chỉ số bởi chỉ số đỉnh hay tạo một mảng adj gồm các mẩu tin trong biểu diễn cấu trúc kề. Thông tin trợ giúp cho mỗi cạnh có thể được đặt vào các nút của xâu kề (hay trong một mảng a gồm các mẩu tin trong biểu diễn ma trận kề); hoặc trong mảng phụ được đặt chỉ số bằng chỉ số cạnh (trường hợp này cần phải đánh số cạnh).

TÌM KIẾM ƯU TIÊN ĐỘ SÂU

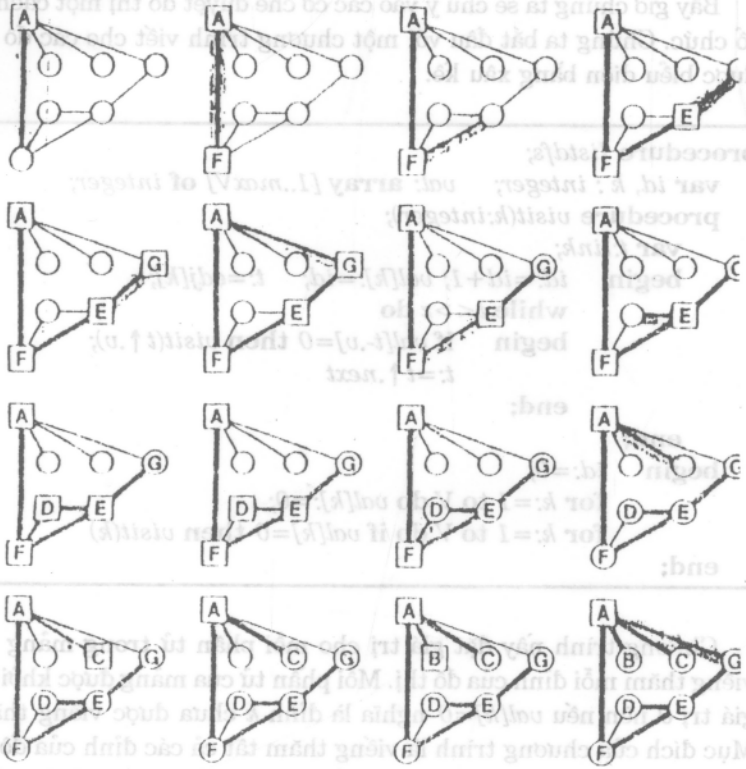
(DEPTH-FIRST)

Trong phần đầu của chương này, chúng ta thấy nhiều câu hỏi nảy sinh trong việc xử lý một đồ thị. Đồ thị có liên thông hay không? Nếu không hãy cho biết các thành phần liên thông của đồ thị? Đồ thị có một chu trình nào không ? Những câu hỏi này và nhiều vấn đề khác có thể được giải quyết dễ dàng bằng một kỹ thuật được gọi là **tìm kiếm ưu tiên độ sâu**, đây là một phương pháp tự nhiên để "viếng thăm" mỗi nút và kiểm tra mỗi cạnh của đồ thị một cách có hệ thống. Trong chương này, chúng ta sẽ thấy các biến dạng đơn giản tổng quát hóa của phương pháp này có thể được dùng để giải các bài toán đồ thị đa dạng.

Bây giờ chúng ta sẽ chú ý vào các cơ chế duyệt đồ thị một cách có tổ chức. Chúng ta bắt đầu với một chương trình viết cho các đồ thị được biểu diễn bằng xâu kề:

```
procedure listdfs;
   var id, k : integer;    val: array [1..maxV] of integer;
   procedure visit(k:integer);
      var t:link;
      begin    id:=id+1; val[k]:=id;    t:=adj[k];
               while t<>z do
               begin    if val[t.v]=0 then visit(t↑.v);
                        t:=t↑.next
               end;
      end;
   begin    id:=0;
            for k:=1 to V do val[k]:=0;
            for k:=1 to V do if val[k]=0 then visit(k)
   end;
```

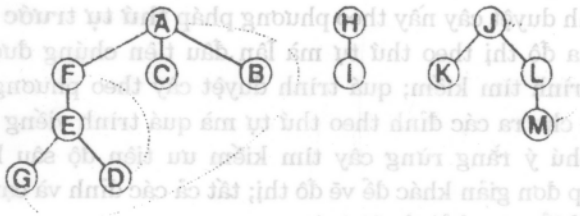
Chương trình này đặt giá trị cho mỗi phần tử trong mảng khi viếng thăm mỗi đỉnh của đồ thị. Mỗi phần tử của mảng được khởi tạo giá trị *0* nên nếu *val[k]=0* nghĩa là đỉnh *k* chưa được viếng thăm. Mục đích của chương trình là viếng thăm tất cả các đỉnh của đồ thị một cách có hệ thống, quá trình viếng sẽ đặt một giá trị id cho phần tử của mảng val ứng với đỉnh được viếng lần thứ id, trong đó *id=1, 2, .., V*. Chương trình sử dụng một thủ tục đệ qui visit duyệt qua tất cả các đỉnh trong cùng một thành phần liên thông với đỉnh được qui đỉnh bởi đối số của thủ tục này. Để viếng thăm một đỉnh, chúng ta kiểm tra tất cả các cạnh của nó xem chúng có dẫn tới các đỉnh chưa được viếng hay không (nhờ vào giá trị *0* trong các phần tử của mảng val), nếu đúng như vậy thì ta sẽ viếng thăm chúng. Trước tiên thủ tục visit được gọi cho đỉnh thứ nhất và kết quả là các phần tử trong mảng *val* (ứng với các đỉnh được nối tới đỉnh thứ nhất) sẽ được đặt



Hình 29.5 *Tìm kiếm ưu tiên độ sâu bằng đệ quy*

giá trị *0*. Kế đến là duyệt qua mảng *val* để tìm một phần tử *0* (tương ứng với một đỉnh chưa được viếng) và gọi thủ tục *visit* cho đỉnh đó, quá trình sẽ tiếp tục cho tới khi viếng xong tất cả các đỉnh của đồ thị.

Hình 29.5 minh họa mỗi bước trong quá trình tìm kiếm ưu tiên độ sâu trên thành phần liên thông lớn của đồ thị trong ví dụ mẫu của chúng ta và cho biết mỗi cạnh trong thành phần liên thông được chạm đến như thế nào khi gọi *visit(1)* (sau khi xâu kề trong Hình 29.4 được xây dựng). Thông thường mỗi cạnh được gặp hai lần bởi



Hình 29.6 *Rừng tìm kiếm ưu tiên độ sâu*

vì mỗi cạnh được biểu diễn trong cả hai xâu kề của hai đỉnh mà nó nối. Mỗi sơ đồ trong Hình 29.5 tương ứng với mỗi cạnh được duyệt, cạnh "hiện hành" trong mỗi sơ đồ được vẽ có bóng và nút mà có xâu kề của nó chứa cạnh đang xét được đánh dấu bởi một ô vuông bao quanh. Hơn nữa, khi một nút được viếng lần đầu tiên (tương ứng với một lần gọi thủ tục *visit*), cạnh nối tới nút đó sẽ được vẽ đậm lên. Các nút chưa được tới lần nào sẽ có bóng nhưng không được đánh dấu bởi ô vuông, và các nút đã được viếng xong lần đầu tiên thì có bóng và được đánh dấu bởi ô vuông.

Cạnh đầu tiên được đi qua là AF, nó chính là nút đầu tiên trong xâu kề đầu tiên. Kế đến thủ tục *visit* được gọi cho nút F và cạnh FA được viếng bởi vì A là nút đầu tiên trong xâu kề của F. Nhưng lúc này nút có một giá trị (tương ứng trong mảng *val*) khác 0 nên ta lấy cạnh FE là phần tử kế tiếp trong xâu kề của F. Kế đến là cạnh EG, GE được viếng bởi vì G và E là các phần tử trong mỗi xâu khác. Cạnh GA được viếng, quá trình viếng đỉnh G hoàn tất và thuật toán tiếp tục viếng đỉnh E duyệt qua các cạnh EF, ED. Kế tiếp là quá trình viếng đỉnh D bao gồm việc duyệt qua DE và DF. Cuối cùng chúng ta trở về A và duyệt qua AC, CA, AB, BA và AG.

Một phương pháp khác để theo dõi các bước trong tìm kiếm ưu tiên độ sâu là vẽ lại đồ thị đang xét bằng cách gọi đệ qui thủ tục visit như minh họa trong Hình 29.6. Mỗi thành phần liên thông cho ra một cây được gọi là **cây tìm kiếm ưu tiên độ sâu** của thành phần đó. Quá trình duyệt cây này theo phương pháp **thứ tự trước** cho ra các đỉnh của đồ thị theo thứ tự mà lần đầu tiên chúng được gặp trong quá trình tìm kiếm; quá trình duyệt cây theo phương pháp **thứ tự sau** cho ra các đỉnh theo thứ tự mà quá trình viếng chúng hoàn tất. Chú ý rằng rừng cây tìm kiếm ưu tiên độ sâu là một phương pháp đơn giản khác để vẽ đồ thị; tất cả các đỉnh và cạnh của đồ thị được kiểm tra bởi thuật toán.

Các đoạn liền nét trong Hình 29.6 chỉ rằng đỉnh được tìm thấy thấp hơn (khi thuật toán hoạt động) nằm trong danh sách cạnh của đỉnh cao hơn và chưa được viếng thăm vào lúc đó, điều này đưa đến cần một lần gọi đệ quy. Các đoạn đứt nét tương ứng với các đỉnh đã được viếng, vì vậy điều kiện của lệnh *if* trong thủ tục *visit* không thỏa, và cạnh không được "theo sau" với một lệnh gọi đệ quy.

Một tính chất quyết định cho các cây tìm kiếm ưu tiên độ sâu đối với đồ thị có hướng là các liên kết "đứt nét" luôn đi từ một nút đến tổ tiên của nó trong cây. Trong suốt quá trình thuật toán thực hiện, các đỉnh được phân chia thành ba lớp: một lớp gồm những đỉnh mà thủ tục *visit* đã hoàn tất, một lớp gồm những đỉnh mà thủ tục *visit* chỉ hoàn tất một nửa, và một lớp gồm những đỉnh chưa được thấy lần nào. Trong thủ tục *visit*, chúng ta sẽ không gặp một cạnh nào hướng tới bất kỳ một đỉnh trong lớp thứ nhất, và nếu chúng ta gặp một cạnh hướng tới một đỉnh trong lớp thứ ba (cạnh liền nét trong cây tìm kiếm ưu tiên độ sâu) thì một lệnh gọi đệ quy sẽ được thực hiện. Còn lại là những đỉnh trong lớp thứ hai, chúng chính là các đỉnh nằm trên đường đi từ đỉnh hiện hành tới gốc của cây, các cạnh tương ứng là các "liên kết" đứt nét trong cây tìm kiếm ưu tiên độ sâu.

Tính chất 29.1 *Thời gian tìm kiếm ưu tiên độ sâu trên một đồ thị (biểu diễn bằng xâu kề) xấp xỉ với $V+E$.*

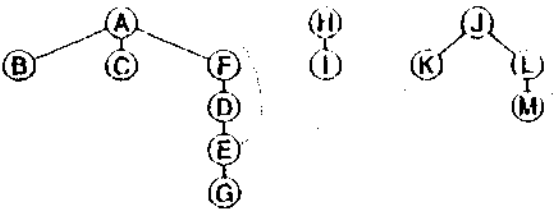
Chúng ta đặt giá trị cho mỗi phần tử trong V phần tử của mảng val (do đó có thành phần V trong $V+E$) và kiểm tra mỗi cạnh hai lần (do đó có thành phần E). Có thể gặp một cây quá thưa với $E < V$, nhưng nếu các đỉnh cô lập không được thừa nhận (ví dụ chúng được khử bỏ trong quá trình tiền xử lý), chúng ta xem như thời gian cần thiết cho tìm kiếm ưu tiên độ sâu là tuyến tính theo số cạnh.

Bằng cách dùng thủ tục *visit* sau đây, phương pháp vừa trình bày có thể áp dụng cho các đồ thị được biểu diễn bởi ma trận kề.

```
procedure visit(k:integer);
   var t:integer;
   begin   id:=id+1; val[k]:=id;
           for t:=1 to V do
              if a[k,t] then if val[t]=0 then visit(t);
   end;
```

Việc duyệt trên một xâu kề được chuyển thành việc quét một dòng ma trận kề, tìm kiếm các giá trị true (tương ứng với các cạnh). Giống như trước, bất kỳ cạnh nào tương ứng với một đỉnh chưa thấy sẽ được "tiếp theo" bởi một lệnh gọi đệ qui. Bây giờ, các cạnh (nối tới mỗi đỉnh) được kiểm tra theo một thứ tự khác nhau, vì vậy chúng ta có được một một rừng tìm kiếm ưu tiên độ sâu khác như được minh họa trong Hình 29.7. Điều này nhấn mạnh rằng rừng tìm kiếm ưu tiên độ sâu là một biểu diễn đơn giản khác của đồ thị, một biểu diễn mà cấu trúc cụ thể của nó phụ thuộc vào thuật toán tìm kiếm và biểu diễn trong được sử dụng.

Tính chất 29.2 *Thời gian tìm kiếm ưu tiên độ sâu trên đồ thị biểu diễn bằng ma trận kề xấp xỉ với V^2.*



Hình 29.7 *Rừng tìm kiếm ưu tiên độ sâu (biểu diễn ma trận kề)*

Chứng minh tính chất này cực kỳ đơn giản, bởi vì mỗi bit trong ma trận kề đều được kiểm tra.

Phương pháp tìm kiếm ưu tiên độ sâu có thể giải trực tiếp một số bài toán xử lý đồ thị cơ bản. Ví dụ, số thành phần liên thông của đồ thị là số lần gọi thủ tục visit ở dòng cuối của chương trình. Nếu kiểm tra xem một đồ thị có chu trình hay không ta chỉ cần sửa chương trình trên một ít, một đồ thị có chu trình nếu và chỉ nếu một phần tử khác *0* trong mảng val được tìm thấy trong quá trình duyệt. Nghĩa là nếu chúng ta chọn một cạnh mà hướng tới một đỉnh đã viếng thì đồ thị có một chu trình. Tất cả các liên kết "đứt nét" trong các cây tìm kiếm ưu tiên độ sâu đều thuộc về các chu trình.

TÌM KIẾM ƯU TIÊN ĐỘ SÂU KHÔNG ĐỆ QUY

Tìm kiếm ưu tiên độ sâu trên một đồ thị là sự tổng quát hóa của thao tác duyệt cây. Khi xét trong một cây thì phương pháp này chính là duyệt cây; trong các đồ thị thì phương pháp này tương ứng với việc duyệt **cây bao trùm** của đồ thị. Như chúng ta đã thấy, việc duyệt cây phụ thuộc vào đồ thị được biểu diễn như thế nào.

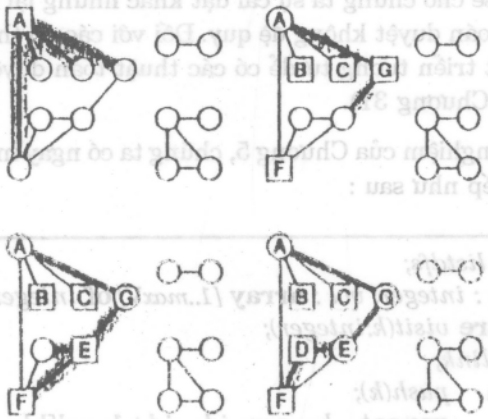
Sự đệ quy trong tìm kiếm ưu tiên độ sâu có thể được khử bằng cách dùng một ngăn xếp theo cách thức tương tự như chúng ta đã

khử đệ quy trong thuật toán duyệt cây ở Chương 5. Đối với cây, việc khử đệ quy sẽ cho chúng ta sự cài đặt khác nhưng lại đơn giản hơn, đó là thuật toán duyệt không đệ quy. Đối với các đồ thị, chúng ta sẽ thấy sự phát triển tương tự để có các thuật toán duyệt đồ thị tổng quát (trong Chương 31).

Với kinh nghiệm của Chương 5, chúng ta có ngay một cài đặt dựa trên ngăn xếp như sau :

```
procedure listdfs;
   var id, k : integer; val : array [1..maxV] of integer;
   procedure visit(k:integer);
      var t:link;
      begin   push(k);
              repeat  k:=pop; id:=id+1; val[k]:=id; t:=adj[k];
                 while t<>z do
                 begin   if val[t↑.v]=0 then
                         begin push(t↑.v); val[t↑.v]:=-1 end;
                         t:=t↑.next
                 end;
              until stackempty;
      end;
   begin   id:=0; stackinit;
           for k:=1 to V do val[k]:=0;
           for k:=1 to V do if val[k]=0 then visit(k)
   end;
```

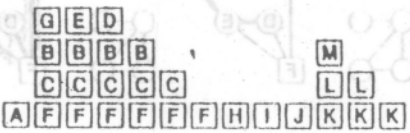
Các đỉnh đã được gặp nhưng chưa được viếng sẽ được lưu trong một ngăn xếp. Để viếng một đỉnh, chúng ta duyệt qua các cạnh của nó và đặt vào ngăn xếp bất kỳ đỉnh nào chưa được viếng và chưa có trong ngăn xếp. Trong cài đặt đệ quy, việc đánh dấu các đỉnh "được viếng một phần" được che giấu bằng một biến cục bộ t trong thủ tục đệ quy. Chúng ta có thể cài đặt điều nầy một cách trực tiếp nhờ vào



Hình 29.8 *Tìm kiếm ưu tiên độ sâu (dựa vào ngăn xếp)*

duy trì các con trỏ (tương ứng với *t*) vào các xâu kề. Thay vì làm như vậy, chúng ta chỉ cần mở rộng ý nghĩa của các phần tử trong mảng val để đánh dấu các đỉnh đã được đặt vào ngăn xếp: các đỉnh ứng với các phần tử của mảng val có giá trị *0* là các đỉnh chưa được gặp, những đỉnh ứng với các phần tử âm là các đỉnh đã ở trên ngăn xếp, những đỉnh ứng với các phần tử dương là những đỉnh đã được viếng (là những đỉnh mà tất cả các cạnh trong các xâu kề của chúng đã được đặt vào ngăn xếp).

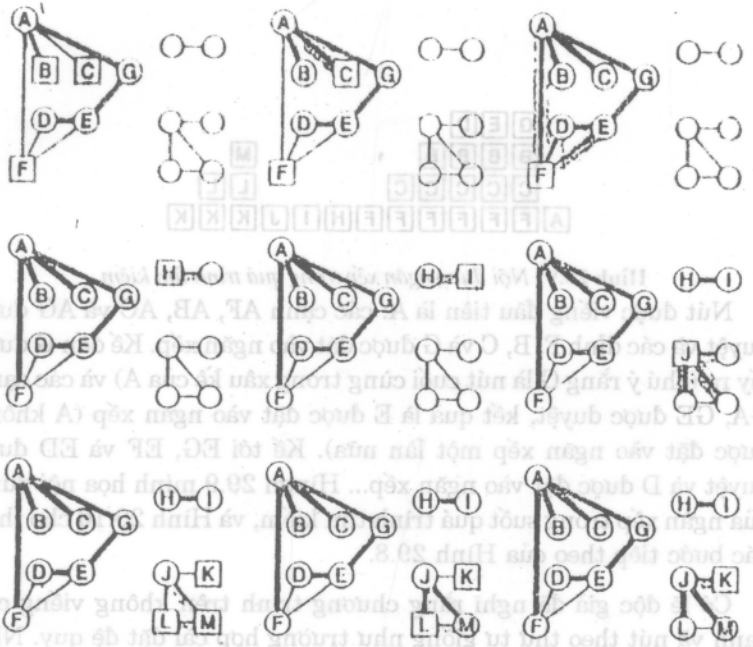
Hình 29.8 cho thấy lần lượt các thao tác của thủ tục tìm kiếm ưu tiên độ sâu dựa ngăn xếp khi duyệt qua bốn nút đầu tiên trong đồ thị mẫu của chúng ta. Mỗi sơ đồ trong hình này tương ứng với việc viếng một nút: nút đã được viếng được đánh dấu bởi một ô vuông, và tất cả các cạnh trong xâu kề được làm bóng. Giống như lúc trước, các nút chưa được gặp sẽ không được đánh dấu và không được đánh bóng, các nút được viếng xong sẽ được đánh dấu và không được đánh bóng, và mỗi nút (ví dụ nút *x*) được nối bởi một cạnh tô đậm nối tới nút làm cho nút *x* được đặt vào ngăn xếp.



Hình 29.9 *Nội dung ngăn xếp trong quá trình tìm kiếm*

Nút được viếng đầu tiên là A: các cạnh AF, AB, AC và AG được duyệt và các đỉnh F, B, C và G được đặt vào ngăn xếp. Kế đến G được lấy ra (chú ý rằng G là nút cuối cùng trong xâu kề của A) và các cạnh GA, GE được duyệt, kết quả là E được đặt vào ngăn xếp (A không được đặt vào ngăn xếp một lần nữa). Kế tới EG, EF và ED được duyệt và D được đặt vào ngăn xếp... Hình 29.9 minh họa nội dung của ngăn xếp trong suốt quá trình tìm kiếm, và Hình 29.10 cho thấy các bước tiếp theo của Hình 29.8.

Có lẽ độc giả đã nghĩ rằng chương trình trên không viếng các cạnh và nút theo thứ tự giống như trường hợp cài đặt đệ quy. Như trong Chương 5, điều này có thể thấy được nhờ chú ý vào thứ tự các cạnh được đưa vào ngăn xếp, trong chương trình trên chúng ta viếng cạnh trong xâu kề của mỗi nút ngược lại với thứ tự xuất hiện của chúng trong xâu. Nếu chúng ta duyệt xâu theo thứ tự đảo thì thứ tự viếng thăm các nút giống như trong cài đặt đệ qui. (Giống như Chương 5, lúc đó cây con phải được đặt vào ngăn xếp trước cây con trái trong cài đặt không đệ qui.) Tuy nhiên đối với các đồ thị không nhất thiết phải tuân thủ nghiêm ngặt vào thứ tự duyệt cạnh trong cài đặt đệ qui, bởi vì như đã thấy, nhiều nhân tố khác cũng ảnh hưởng đến thứ tự duyệt cạnh. Trong Chương 31 chúng ta sẽ thấy có rất nhiều lựa chọn cho phép chúng ta lợi dụng khả năng linh động này vào các thuật toán.



Hình 29.10 *Tìm kiếm ưu tiên độ sâu (dựa vào ngăn xếp)*

TÌM KIẾM ƯU TIÊN BỀ RỘNG

(BREADTH-FIRST)

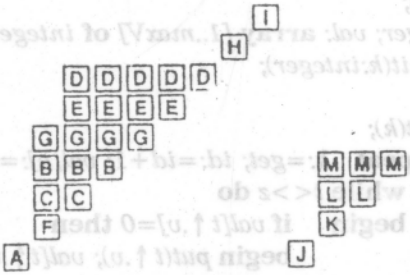
Giống như duyệt cây (xem Chương 4), chúng ta có thể dùng mộtt hàng đợi thay vì ngăn xếp làm cấu trúc dữ liệu để lưu các đỉnh. Điều nầy đưa tới một thuật toán duyệt đồ thị cổ điển thứ hai được gọi là **tìm kiếm ưu tiên bề rộng**. Để cài đặt thuật toán tìm kiếm này, chúng ta cần thay đổi các phép toán trên ngăn xếp bằng các phép toán trên hàng đợi như sau:

```
procedure listbfs;
  var id, k :integer; val: array [1..maxV] of integer;
  procedure visit(k:integer);
    var t:link;
    begin   put(k);
            repeat  k:=get; id:=id+1; val[k]:=id; t:=adj[k];
              while t<>z do
              begin   if val[t↑.v]=0 then
                      begin put(t↑.v); val[t↑.v]:=-1 end;
                      t:=t↑.next
              end;
            until queueempty;
    end;
  begin   id:=0; queueinitialize;
          for k:=1 to V do val[k]:=0;
          for k:=1 to V do if val[k]=0 then visit(k)
  end;
```

Việc thay đổi cấu trúc dữ liệu bằng cách vừa trình bày sẽ ảnh hưởng đến thứ tự viếng các nút. Trong đồ thị mẫu của chúng ta, các cạnh được viếng theo thứ tự AF AC AB AG FA FE FD CA BA GE GA DE EG EF ED HI IH JK JL JM KJ LJ LM MJ ML. Hình 29.11 cho thấy nội dung của hàng đợi trong suốt quá trình duyệt.

Tương tự như tìm kiếm ưu tiên độ sâu, chúng ta có thể định nghĩa một rừng nhờ vào các cạnh dẫn tới mỗi nút như trong Hình 29.12. Tìm kiếm ưu tiên độ rộng tương ứng với việc duyệt cây trong rừng nầy theo **thứ tự tầng**.

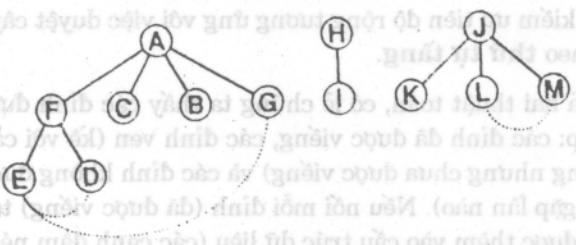
Trong cả hai thuật toán, có lẽ chúng ta thấy các đỉnh được chia thành ba lớp: các đỉnh đã được viếng, các đỉnh ven (kề với các đỉnh đã được viếng nhưng chưa được viếng) và các đỉnh không được thấy (chưa được gặp lần nào). Nếu nối mỗi đỉnh (đã được viếng) tới cạnh làm cho nó được thêm vào cấu trúc dữ liệu (các cạnh đậm nét trong



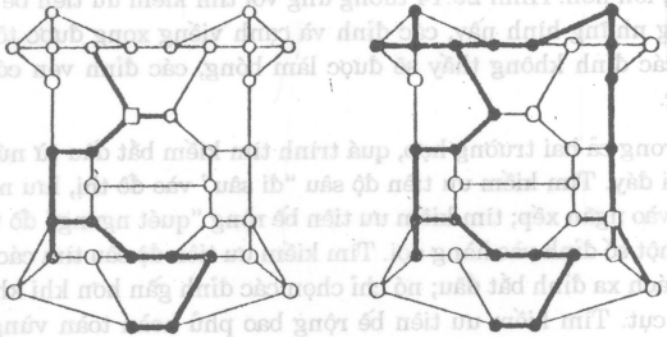
Hình 29.11 *Nội dung hàng đợi trong tìm kiếm ưu tiên bề rộng*

Hình 29.8 và 29.10) thì các cạnh này tạo thành một cây.

Để tìm một thành phần liên thông của đồ thị (cài đặt một thủ tục visit), chúng ta bắt đầu với một đỉnh ven, tất cả các đỉnh không được thấy, và thực hiện thao tác sau đây cho tới khi tất cả các đỉnh được viếng xong: "di chuyển một đỉnh (gọi nó là x) từ tập các đỉnh ven vào tập các đỉnh đã được viếng, đặt tất cả các đỉnh không thấy kề với x vào tập các đỉnh ven". Các phương pháp duyệt đồ thị khác nhau ở chỗ chúng quyết định đỉnh nào nên được di chuyển vào tập các đỉnh ven vào tập các đỉnh đã được viếng. Với tìm kiếm ưu tiên độ sâu chúng ta muốn chọn một đỉnh từ tập các đỉnh ven được gặp lần sau cùng, điều này tương ứng với việc dùng một ngăn xếp để lưu các đỉnh ven. Với tìm kiếm ưu tiên bề rộng chúng ta muốn chọn một đỉnh từ tập các đỉnh ven được chạm tới trước nhất, điều này tương



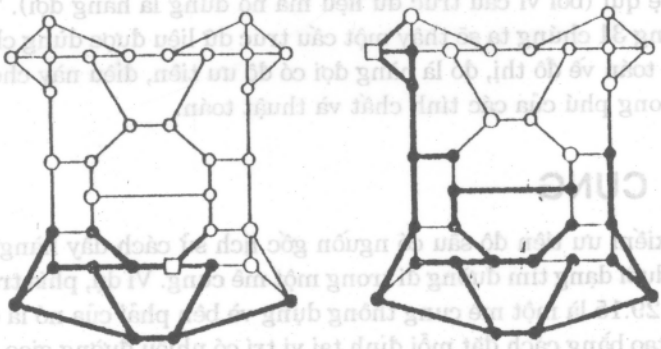
Hình 29.12 *Rừng tìm kiếm ưu tiên bề rộng*



Hình 29.13 *Tìm kiếm ưu tiên độ sâu trong một đồ thị lớn hơn*

ứng với việc dùng một hàng đợi để lưu các đỉnh ven. Trong Chương 31, chúng ta sẽ thấy ảnh hưởng của một **hàng đợi có độ ưu tiên** cho tập hợp các đỉnh ven.

Sự tương phản giữa tìm kiếm ưu tiên độ sâu và tìm kiếm ưu tiên bề rộng sẽ hoàn toàn hiển nhiên khi chúng ta khảo sát một đồ thị lớn



Hình 29.14 *Tìm kiếm ưu tiên bề rộng trong một đồ thị lớn hơn*

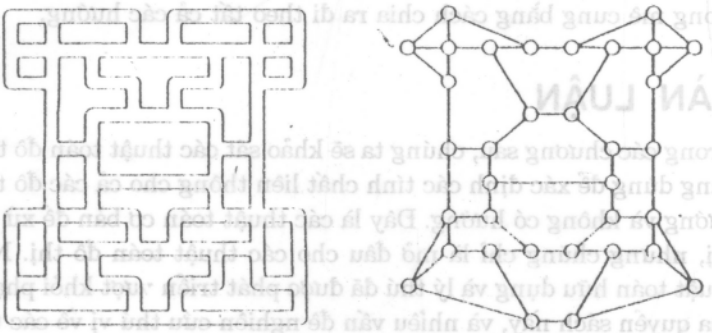
hơn. Hình 29.13 minh họa thao tác tìm kiếm ưu tiên độ sâu cho một đồ thị lớn hơn. Hình 29.14 tương ứng với tìm kiếm ưu tiên bề rộng. Trong những hình nầy, các đỉnh và cạnh viếng xong được tô đậm lên, các đỉnh không thấy sẽ được làm bóng, các đỉnh ven có màu trắng.

Trong cả hai trường hợp, quá trình tìm kiếm bắt đầu từ nút trái ở dưới đáy. Tìm kiếm ưu tiên độ sâu "đi sâu" vào đồ thị, lưu một số đỉnh vào ngăn xếp; tìm kiếm ưu tiên bề rộng "quét ngang" đồ thị và nhớ một số đỉnh vào hàng đợi. Tìm kiếm ưu tiên độ sâu tìm các đỉnh mới cách xa đỉnh bắt đầu; nó chỉ chọn các đỉnh gần hơn khi khi gặp điểm cụt. Tìm kiếm ưu tiên bề rộng bao phủ hoàn toàn vùng gần điểm bắt đầu, nó chỉ đi ra xa mỗi khi đỉnh gần đã được nhìn thấy. Một lần nữa, thứ tự viếng các nút phụ thuộc nhiều vào thứ tự xuất hiện các cạnh lúc nhập vào và ảnh hưởng của thứ tự nầy vào thứ tự xuất hiện các đỉnh trong xâu kề.

Những điểm khác nhau vừa nói đưa đến khác nhau cơ bản trong cài đặt những phương pháp nầy. Tìm kiếm ưu tiên độ sâu có thể cài đặt đệ qui rất đơn giản (bởi vì cấu trúc dữ liệu mà nó dùng là ngăn xếp) và tìm kiếm ưu tiên bề rộng có thể cài đặt đơn giản mà không cần đệ qui (bởi vì cấu trúc dữ liệu mà nó dùng là hàng đợi). Trong Chương 31 chúng ta sẽ thấy một cấu trúc dữ liệu được dùng cho các thuật toán về đồ thị, đó là hàng đợi có độ ưu tiên, điều này cho thấy sự phong phú của các tính chất và thuật toán.

MÊ CUNG

Tìm kiếm ưu tiên độ sâu có nguồn gốc lịch sử cách đây hàng trăm năm dưới dạng tìm đường đi trong một mê cung. Ví dụ, phía trái của Hình 29.15 là một mê cung thông dụng và bên phải của nó là đồ thị được tạo bằng cách đặt mỗi đỉnh tại vị trí có nhiều đường giao nhau, kế đến nối các đỉnh theo các đường đi trong mê cung. Mê cung này



Hình 29.15 *Một mê cung và một đồ thị biểu diễn của nó*

phức tạp hơn so với các mê cung trước đây trong các vườn giải trí ở Anh quốc được xây dựng bằng các con đường xuyên qua các bức tường rào cao. Trong những mê cung này, tất cả các bức tường được nối tới bức tường ngoài cùng sao cho những quí ông và quí bà có thể đi dạo trong đó và những người khôn ngoan có thể tìm đường đi ra ngoài bằng cách giữ tay phải của họ vào tường. Khi xuất hiện thêm các bức tường độc lập ở phía trong, cần phải có chiến lược tinh vi hơn để đi trong mê cung, đây là lý do đưa đến phương pháp tìm kiếm ưu tiên độ sâu.

Muốn dùng phương pháp tìm kiếm ưu tiên độ sâu để đi từ một nơi nầy đến nơi khác trong mê cung, chúng ta dùng thủ tục *visit*, khởi đầu từ đỉnh của đồ thị tương ứng với điểm bắt đầu của chúng ta. Mỗi khi thủ tục *visit* "theo sau" một cạnh bởi một lần gọi đệ quy, chúng ta đi dọc theo con đường tương ứng trong mê cung. Điểm cốt lõi là khi thủ tục *visit* hoàn tất một đỉnh, chúng ta phải đi trở lại dọc theo con đường vừa đi qua. Cách thức này đưa chúng ta trở lại đỉnh cũ với một bước cao hơn trong cây tìm kiếm ưu tiên độ sâu, sẵn sàng dò theo cạnh kế của nó. Tìm kiếm ưu tiên độ sâu thích hợp khi một

người đang tìm kiếm một vật nào đó trong mê cung, tìm kiếm ưu tiên bề rộng sẽ thích hợp hơn khi có nhiều người tìm một vật nào đó trong mê cung bằng cách chia ra đi theo tất cả các hướng.

BÀN LUẬN

Trong các chương sau, chúng ta sẽ khảo sát các thuật toán đồ thị đa dạng dùng để xác định các tính chất liên thông cho cả các đồ thị có hướng và không có hướng. Đây là các thuật toán cơ bản để xử lý đồ thị, nhưng chúng chỉ là mở đầu cho các thuật toán đồ thị. Nhiều thuật toán hữu dụng và lý thú đã được phát triển vượt khỏi phạm vi của quyển sách này, và nhiều vấn đề nghiên cứu thú vị về các thuật toán vẫn chưa được giải quyết.

Một số thuật toán rất hiệu quả đã được phát triển nhưng quá phức tạp đến nỗi không thể trình bày ở đây. Ví dụ, xét xem một đồ thị có thể vẽ được trên mặt phẳng mà không có các đoạn thẳng giao nhau hay không. Bài toán này được gọi là **bài toán đồ thị phẳng**, bài toán không có thuật giải hiệu quả cho tới khi R. E. Tarjan đã phát triển một thuật toán thời gian tuyến tính dựa trên tìm kiếm ưu tiên độ sâu.

Một số bài toán về đồ thị đã nảy sinh một cách tự nhiên, rất khó và không có thuật toán tốt để giải chúng. Ví dụ, không có thuật toán hiệu quả để tìm ra giá trị nhỏ nhất khi duyệt qua mỗi đỉnh trong đồ thị có trọng. Bài toán này được gọi là **bài toán nhà buôn đi du lịch**, nó thuộc lớp các bài toán khó mà chúng ta sẽ không thảo luận chi tiết. Hầu hết các chuyên gia đều tin rằng không tồn tại thuật toán hiệu quả cho bài toán này.

Các bài toán đồ thị khác có lẽ có các thuật toán hiệu quả để giải chúng mặc dù chưa được tìm thấy. Một ví dụ là **bài toán đẳng cấu đồ thị**: xác định xem hai đồ thị bất kỳ có đồng nhất nhau hay không nếu ta đổi tên các đỉnh. Đối với nhiều dạng đồ thị đặc biệt, đã có các

thuật toán hiệu quả để bài toán này, nhưng bài toán tổng quát cũng còn mở.

Có rất nhiều bài toán và thuật toán liên quan đến các đồ thị và các thuật toán về đồ thị đóng vai trò rất lớn trong ứng dụng vào thực tế.

BÀI TẬP

1. Biểu diễn nào cho đồ thị vô hướng là thích hợp nhất để xác định nhanh xem một đỉnh của đồ thị có cô lập (không được nối tới bất kỳ đỉnh nào) hay không?

2. Giả sử thuật toán tìm kiếm ưu tiên độ sâu được áp dụng vào cây tìm kiếm nhị phân và cạnh bên phải được lấy trước khi rời mỗi nút. Các nút được viếng thăm theo thứ tự như thế nào?

3. Cần bao nhiêu bit để lưu trữ trong biểu diễn ma trận kề của đồ thị có hướng V đỉnh và E cạnh? Cần bao nhiêu đối với biểu diễn xâu kề?

4. Hãy cho ví dụ một đồ thị không thể vẽ được ra giấy mà không có hai cạnh cắt nhau.

5. Viết một chương trình để xóa đi một cạnh khỏi một đồ thị được biểu diễn bằng xâu kề.

6. Viết một phiên bản của thủ tục *adjlist* để duy trì các xâu kề theo thứ tự của chỉ số đỉnh. Thảo luận về ưu điểm của cách tiếp cận nầy.

7. Hãy vẽ ra rừng tìm kiếm ưu tiên độ sâu có được (đối với ví dụ trong chương nầy) khi thủ tục *dfs* quét qua các đỉnh theo thứ tự đảo ngược (từ V tới 1) chỉ cả hai cách biểu diễn.

8. Thủ tục visit được gọi chính xác bao nhiêu lần trong tìm kiếm ưu tiên độ sâu trên một đồ thị vô hướng (đếm theo số đỉnh V, số cạnh E, số thành phần liên thông C)?

9. Hãy cho biết xâu kề có được nếu các cạnh của đồ thị mẫu (trong chương nầy) được đọc theo thứ tự đảo ngược so với thứ tự đã dùng để tạo nên cấu trúc trong Hình 29.4.

10. Hãy cho biết rừng tìm kiếm ưu tiên độ sâu cho đồ thị mẫu khi thủ tục đệ quy *listdfs* được dùng trên xâu kề của bài tập trước.

30

ĐỒ THỊ LIÊN THÔNG

Trong chương trước, chúng ta thấy có thể sử dụng thủ tục tìm kiếm theo phương pháp ưu tiên độ sâu (*DFS - Depth first search*) để tìm các thành phần liên thông của đồ thị; trong chương này, ta sẽ xét các thuật toán và các vấn đề liên quan đến tính liên thông của đồ thị.

Sau khi xét một vài ứng dụng trực tiếp của DFS để lấy các thông tin liên thông, ta sẽ xét sự tổng quát hoá của sự liên thông là song liên thông. Ở đây, ta chú ý đến trường hợp có nhiều hơn một đường dẫn giữa hai nút. Một đồ thị được gọi là song liên thông nếu và chỉ nếu với mỗi cặp đỉnh có ít nhất hai đường dẫn khác biệt nối các đỉnh này với nhau. Như vậy, ngay khi một đỉnh và tất cả các cạnh nối với đỉnh này được bỏ đi thì đồ thị vẫn là liên thông. Nếu trong ứng dụng, bài toán liên thông là quan trọng thì việc đồ thị vẫn giữ được tính liên thông cũng là một bài toán trọng yếu. Cách giải bài toán này thì phức tạp hơn thuật toán duyệt trong Chương trước, nhưng vẫn dựa trên DFS.

Có một trường hợp đặc thù hay gặp ở tình huống động của bài toán liên thông - trong đó đồ thị được thêm từng cạnh một, là những câu hỏi như hai điểm có thuộc cùng một thành phần liên thông không. Đây là bài toán đáng để nghiên cứu, và ta sẽ xem xét tỉ mỉ hai thuật toán kinh điển cho bài toán này. Đây không chỉ là các phương pháp đơn giản được áp dụng rộng rãi, mà chúng còn minh hoạ sự khó khăn có thể gặp khi phân tích một thuật toán đơn giản. Vấn đề

k	1	2	3	4	5	6	7	8	9	10	11	12	13
name[k]	A	B	C	D	E	F	G	H	I	J	K	L	M
val[k]	1	7	6	5	3	2	4	8	9	10	11	12	13
inval[k]	-1	6	5	7	4	3	2	-8	9	-10	11	12	13

Hình 30.1 *Những cấu trúc dữ liệu cho các thành phần liên thông*

này đôi khi được gọi là bài toán "tìm hội", thuật ngữ này có từ ứng dụng những thuật toán làm các thao tác đơn giản trên một tập hợp.

CÁC THÀNH PHẦN LIÊN THÔNG

Bất kỳ phương pháp duyệt đồ thị nào cũng có thể dùng để tìm các thành phần liên thông, vì tất cả phương pháp này đều dựa trên cùng một chiến lược tổng quát là thăm tất cả các nút trong một thành phần liên thông trước khi chuyển sang nút kế tiếp. Một cách dễ dàng để in ra các thành phần liên thông là thay đổi trình đệ quy *DFS* để in ra các đỉnh đang được thăm (bằng cách thêm *write(name(k))* ngay trước khi rời khỏi thủ tục). Sau đó, ngay trước lời gọi (không đệ quy) đến *visit*, ta in thông báo cho biết thành phần liên thông mới sẽ bắt đầu (bằng cách thêm vào hai lệnh *writeln*). Với *DFS* dùng cách biểu diễn danh sách kế cận (ở hình 29.1), kỹ thuật này sẽ tạo ra output sau:

G D E F C B A
I H
K M L J

Những biến thể khác, như ma trận kế cận, *DFS* dựa trên ngăn xếp, và *BFS (breadth first search)*, có thể dùng để tính các thành phần liên thông; nhưng thứ tự các đỉnh được in ra sẽ khác.

Ta dễ dàng mở rộng để làm những thao tác phức tạp hơn trên các thành phần liên thông. Ví dụ, bằng cách thêm *inval[id] = k* ở sau *val[k] = id*, ta sẽ được cái "đảo ngược" của mảng *val*, mà phần tử

thứ *id* của mảng *inval* là chỉ số của đỉnh được duyệt lần thứ *id*. Các đỉnh trong cùng một thành phần liên thông là liên tục trên mảng *inval*, mỗi thành phần liên thông có chỉ số là giá trị của id trong mỗi lần gọi *visit*. Những giá trị này có thể được lưu trữ hoặc để đánh dấu các ranh giới trong mảng *inval* (ví dụ, phần tử đầu tiên trong mỗi thành phần liên thông được quy ước là số âm).

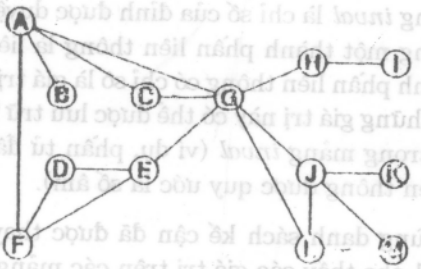
Với *DFS* dùng danh sách kế cận đã được thay đổi như đã nêu trên, hình 30.1 cho thấy các giá trị trên các mảng trong ví dụ. Đặc biệt, những kỹ thuật như vậy được dùng để chia đồ thị thành các thành phần liên thông trong các chương trình rắc rối khác, để tránh khỏi những chi tiết vụn vặt gặp phải khi xét các thành phần không liên thông.

SONG LIÊN THÔNG

Đôi khi có ích để thiết kế nhiều hơn một đường đi giữa các điểm trên một đồ thị, nhằm dự phòng khả năng không liên lạc được với các đỉnh. Ví dụ, ta có thể bay từ Providence đến Princeton ngay như nếu New York có tuyết bằng cách đi qua phía Philadelphia. Các đường dây chính của một mạch điện thường là song liên, như vậy phần còn lại của mạch điện vẫn có thể làm việc nếu một thành phần của mạch bị hỏng. Một ứng dụng khác (không thực tế lắm nhưng là một minh hoạ tự nhiên cho khái niệm này) là trong tình trạng chiến tranh, ta làm sao để địch phải dội bom ít nhất hai nơi mới cắt được đường tiếp vận.

Nút bản lề trong một đồ thị liên thông là một nút mà nếu xoá nó, sẽ tách đồ thị này thành 2 thành phần. Đồ thị không có nút bản lề được gọi là song liên thông. Trong đồ thị song liên, mỗi cặp điểm có ít nhất 2 đường dẫn nối chúng với nhau. Đồ thị không song liên, sẽ được chia thành các thành phần song liên.

Hình 30.2 cho thấy một đồ thị liên thông nhưng không phải là



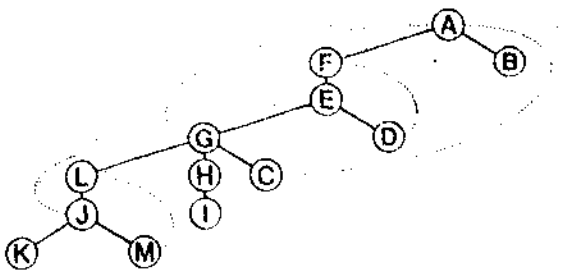
Hình 30.2 *Một đồ thị không song liên thông*

song liên (đồ thị này có được từ đồ thị trong Chương trước bằng cách thêm các cạnh GC, GH, GJ và GL. Trong ví dụ, ta sẽ thêm 4 cạnh này theo thứ tự trên vào cuối input (để các danh sách kế cận tương tự như trong hình 29.4) và trong danh sách sẽ có 8 thành phần mới biểu diễn 4 cạnh vừa được thêm. Các điểm bản lề của đồ thị này là A (vì A nối tới B là phần cuối của đồ thị) và G (vì nếu xoá G, đồ thị sẽ bị phân thành 3 mảnh). Có 6 thành phần song liên: A C G D E F, G J L M, và các nút riêng lẻ B, H, I và K.

Việc xác định các nút bản lề trở nên một mở rộng đơn giản của *DFS*. Ta sẽ khảo sát cây *DFS* cho đồ thị ví dụ này (xem hình 30.3). Xoá nút E không làm đồ thị mất liên thông vì G và D đều có các đường nối tới nút F (ở trên E) và làm thành những đường dẫn khác từ chúng tới F (là cha của E trên cây). Mặt khác, xoá G sẽ làm đồ thị mất liên thông vì không có đường dẫn nào khác đi từ L hoặc H đến E (cha của G).

Một đỉnh x không phải là nút bản lề nếu mỗi nút con y có một nút thấp hơn được nối với nút cao hơn trên cây (theo các đường chấm), từ đó cho một liên kết khác từ x đến y. Sự kiểm chứng này không đúng cho nút gốc của cây *DFS*, vì không có nút nào cao hơn nút gốc (xem hình 30.3).

Nút gốc là một nút bản lề nếu nó có 2 con hoặc nhiều hơn, bởi vì



Hình 30.3 *DFS để xét song liên thông*

đường dẫn nối các con của nút gốc phải đi xuyên qua nút gốc. Ta dễ dàng đưa những sự kiểm tra này vào *DFS* bằng cách thay đổi thủ tục *node-visit* có chức năng trả lại nút cao nhất trên cây (có giá trị val thấp nhất) trong quá trình tìm kiếm.

```
function visit (k:integer) : integer;
   var t : link; m, min : integer;
   begin
   id:=id+1; val[k]:=id; min:=id; t:=adj[k];
   while t<>z do
      begin    if val[t↑.v]=0 then
               begin m:=visit(t↑.v);
                     if m<min then min:=m;
                     if m>=val[k] then write(name(k));
               end
               else
               if val[t↑.v] then min:=val[t↑.v];
               t := t↑.next;
      end;
   visit:=min;
   end;
```

Một cách đệ quy, thủ tục này từ các nút con của đỉnh *k* đi dọc theo các đường chấm để xác định nút cao nhất có thể với đến trên cây, và dùng thông tin trên để xác định *k* có là nút bản lề hay không. Thông thường, sự tính toán này chỉ đơn giản là phép kiểm tra nút con (có giá trị *val* nhỏ nhất) có là cao hơn nút *k* trên cây hay không.

Tuy nhiên, ta cần có một phép kiểm tra thêm để xác định *k* có là gốc của cây *DFS* (một cách tương đương là xét xem đây có phải là lần gọi *visit* đầu tiên cho thành phần liên thông chứa *k* hay không), bởi vì ta đang sử dụng cùng một chương trình đệ quy cho cả hai trường hợp. Phép kiểm tra này được thực hiện bên ngoài *visit* và vì vậy trong đoạn mã trên không có phép kiểm tra này.

Tính chất 30.1 *các thành phần song liên của một đồ thị được tìm trong thời gian tuyến tính.*

Mặc dù trình này chỉ đơn giản in ra các nút bản lề, nó dễ dàng được mở rộng, như ta đã làm cho các thành phần liên thông, để làm các xử lý thêm trên các nút bản lề và các thành phần song liên. Bởi vì đây là một thủ tục *DFS*, thời gian thực hiện là tỉ lệ với $V+E$ (nếu dựa trên ma trận kế cận, một trình tương tự sẽ thực hiện với khoảng $O(N^2)$ bước).

Trong các loại ứng dụng nêu trên, tính song liên được dùng để tăng độ tin cậy, có ích để phân rã các đồ thị lớn thành các phần quản lý được. Rõ ràng là trong nhiều ứng dụng, các thành phần liên thông được xử lý từng cái một; nhưng thỉnh thoảng cũng cần xử lý từng thành phần song liên.

CÁC THUẬT TOÁN TÌM PHẦN HỢP

Trong một số ứng dụng, ta muốn biết đỉnh *x* có liên thông với đỉnh *y* hay không mà không nhất thiết phải xét đường dẫn cụ thể nối chúng với nhau. Vấn đề này được nghiên cứu kỹ lưỡng trong những năm gần đây; việc phát triển các thuật toán hiệu quả đang là một sự

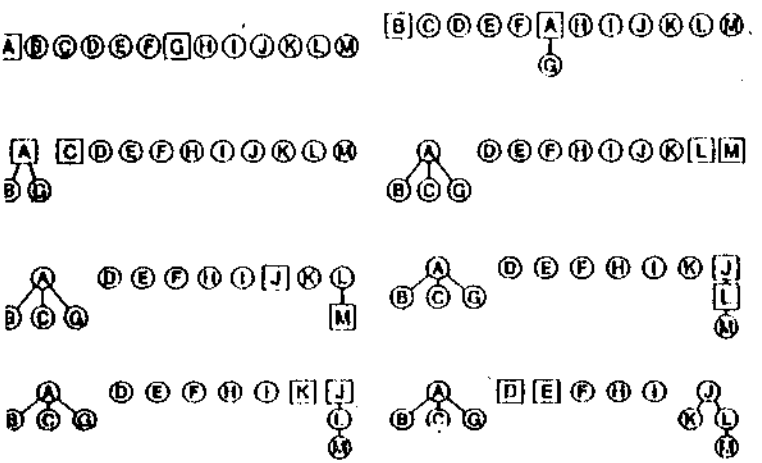
chú ý bởi vì chúng cũng được dùng để xử lý các tập hợp.

Các đồ thị tương ứng một cách tự nhiên với các tập đối tượng: các đỉnh tương ứng với các đối tượng và các cạnh có ý nghĩa "trong cùng một tập". Như vậy, đồ thị ví dụ trong Chương trước tương ứng với các tập A B C D E F G, H I và J K L M. Mỗi thành phần liên thông tương ứng với một tập hợp khác nhau. Với các tập hợp trên, ta chú ý đến câu hỏi "x có trong cùng một tập hợp với y?". Rõ ràng, điều này tương ứng với câu hỏi "đỉnh x có liên thông với đỉnh y?"

Cho một tập các cạnh, ta xây dựng danh sách kế cận của đồ thị tương ứng và dùng *DFS* để gán cho mỗi đỉnh một giá trị là chỉ số của thành phần liên thông chứa đỉnh đó, và những câu hỏi dạng "x có liên thông với y?" có thể được trả lời chỉ với hai lần truy xuất mảng (để lấy chỉ số thành phần liên thông của x và y) và một phép so sánh. Hướng mở rộng những phương pháp được xét ở đây là tính động của chúng : các thuật toán này cho phép thêm các cạnh mới trộn lẫn với các câu hỏi và trả lời chính xác các câu hỏi này bằng cách dùng các thông tin nhận được. Tương ứng với bài toán tập hợp, việc thêm các cạnh mới gọi là thao tác *union* và các câu hỏi gọi là thao tác *find*.

Mục đích chúng ta là viết một hàm để kiểm tra hai đỉnh x và y có trong cùng một tập hợp (theo cách nói đồ thị, có trong cùng một thành phần liên thông) và nếu không, thì đặt chúng trong cùng một tập hợp (thêm một cạnh giữa chúng trong đồ thị). Thay vì xây dựng một danh sách kế cận hay một biểu diễn đồ thị khác, ta sẽ làm tăng tính hiệu quả bằng cách dùng một cấu trúc nội đặc biệt, hỗ trợ các thao tác *union* và *find*. Cấu trúc nội này sẽ là một rừng các cây, mỗi cây là một thành phần liên thông. Ta cần có khả năng kiểm tra hai đỉnh có thuộc cùng một cây hay không và khả năng trộn hai cây làm một. Điều này dẫn đến khả năng cài đặt hiệu quả cho cả hai thao tác này.

Để minh hoạ thuật toán làm việc như thế nào, ta sẽ xét cấu trúc rừng khi các cạnh của đồ thị (trong hình 30.1) được xử lý theo thứ

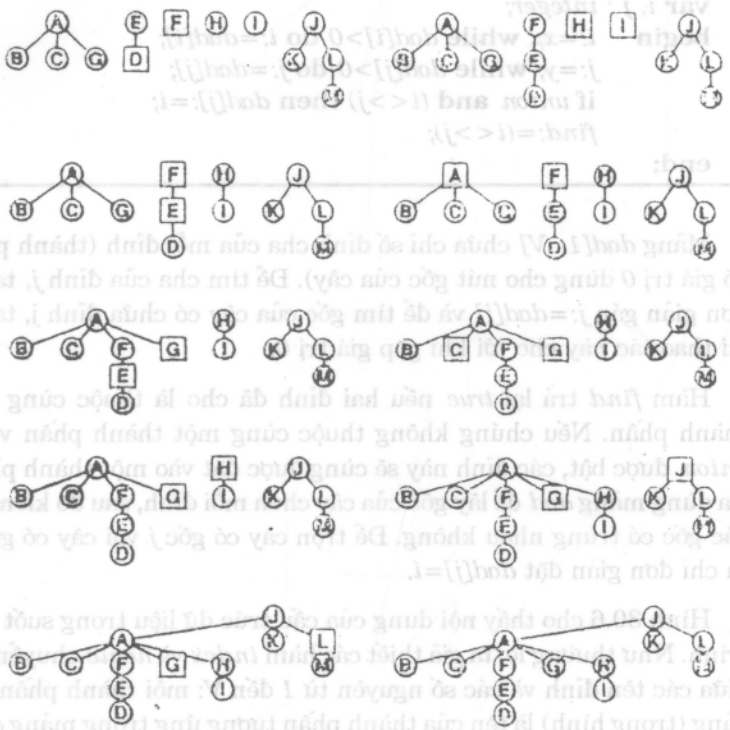


Hình 30.4 *Các bước bắt đầu của việc tìm phần hội*

tự AG AB AC LM JM JL JK ED FD HI FE AF GE GC GH JG LG. 7 bước đầu tiên được chỉ trong hình 30.4

Ban đầu, tất cả các nút ở những cây tách rời nhau. Kế đó AG tạo ra cây 2 nút với A là nút gốc. (Sự chọn lựa này là tuỳ ý, ta cũng có thể chọn G làm gốc). Cùng cách trên, các cạnh AB và AC thêm B và C vào cây này. Kế đó, các cạnh LM, JM, JL và JK tạo nên cây chứa J, K, L và M; cây này có cấu trúc hơi khác (chú ý là JL không tạo thêm gì, vì LM và JM đặt L và J trong cùng một thành phần)

Hình 30.5 trang sau cho thấy toàn bộ tiến trình. Các cạnh ED, FD và HI tạo thêm 2 cây nữa, thành một rừng 4 cây. Rừng này cho biết các cạnh được xử lý ở thời điểm mô tả đồ thị có 4 thành phần liên thông, hay một cách tương đương, lúc này các thao tác union dẫn đến 4 tập {A B C G}, {J K L M}, {D E F} và {H I}. Bây giờ, cạnh FE không góp thêm gì vào cấu trúc, vì F và E đã cùng thành phần; nhưng cạnh AE kết hợp hai cây đầu tiên; GE và GC không thêm gì; nhưng GH và JG tạo ra sự hợp nhất tất cả vào một cây.



Hình 30.5 *Toàn bộ các bước tìm phần hợp*

Cần phải nhấn mạnh rằng - không giống như các cây *DFS* - giữa các cây *union-find* và đồ thị ẩn phía dưới với các cạnh đã cho, chỉ có quan hệ là chúng chia các đỉnh thành các tập hợp theo cùng một phương thức. Ví dụ, không có sự tương ứng giữa những đường nối các nút trong cây và những đường nối các nút trong đồ thị.

Rất dễ dàng cài đặt các thao tác *union* và *find* bằng cách dùng "liên kết cha" cho các cây (xem Chương 4).

```
function find (x, y :integer; union : boolean) : boolean;
  var i, j : integer;
  begin   i:=x; while dad[i]>0 do i:=dad[i];
          j:=y; while dad[j]>0 do j:=dad[j];
          if union and (i<>j) then dad[j]:=i;
          find:=(i<>j);
  end;
```

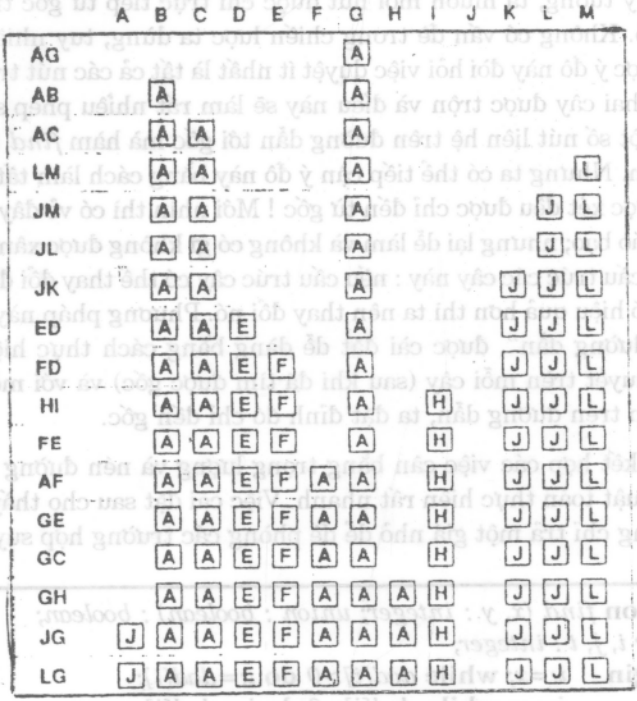
Mảng *dad[1...V]* chứa chỉ số đỉnh cha của mỗi đỉnh (thành phần có giá trị *0* dùng cho nút gốc của cây). Để tìm cha của đỉnh *j*, ta chỉ đơn giản gán *j:=dad[j]* và để tìm gốc của cây có chứa đỉnh j, ta lặp lại thao tác này cho tới khi gặp giá trị *0*.

Hàm *find* trả lại *true* nếu hai đỉnh đã cho là thuộc cùng một thành phần. Nếu chúng không thuộc cùng một thành phần và cờ *union* được bật, các đỉnh này sẽ cùng được đặt vào một thành phần. Ta dùng mảng *dad* để lấy gốc của cây chứa mỗi đỉnh, sau đó kiểm tra các gốc có trùng nhau không. Để trộn cây có gốc *j* với cây có gốc *i*, ta chỉ đơn giản đặt *dad[j]=i*.

Hình 30.6 cho thấy nội dung của cấu trúc dữ liệu trong suốt tiến trình. Như thường lệ, ta giả thiết các hàm *index* và *name* chuyển đổi giữa các tên đỉnh và các số nguyên từ *1* đến *V*: mỗi thành phần của bảng (trong hình) là tên của thành phần tương ứng trong mảng *dad*. Ví dụ, lời gọi hàm *find(index (x), index(y), false)* sẽ kiểm tra đỉnh tên *x* có cùng thuộc một thành phần như đỉnh tên *y* hay không (không thêm cạnh nối chúng).

Thuật toán mô tả trên có sự thực hiện tồi tệ trong trường hợp xấu nhất là khi cây thoái hoá. Ví dụ, lấy các cạnh theo thứ tự AB BC CD DE EF FG GH HI IJ... YZ tạo ra một chuỗi dài với Z chỉ đến Y, Y chỉ đến X... Cần thời gian tỉ lệ với V^2 để xây dựng cấu trúc này và cần thời gian tỉ lệ với *V* cho một kiểm tra tương đương.

Nhiều phương pháp được đề nghị để giải bài toán này. Một



Hình 30.6 *Cấu trúc dữ liệu của tiến trình tìm phần hợp*

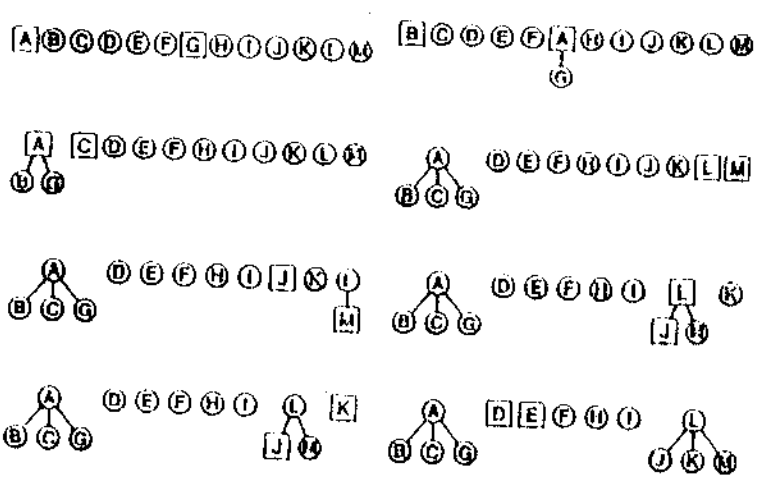
phương pháp tự nhiên là thử làm một điều "hợp lý" khi trộn hai cây hơn là áp đặt *dad[j]=i*. Khi cây có gốc *i* được trộn với cây có gốc *j*, một trong các nút này sẽ vẫn là nút gốc và nút còn lại (và tất cả con của nó) phải dời xuống một mức trên cây. Để giảm thiểu khoảng cách từ gốc đến đa số nút, một cách trực quan, ta lấy nút gốc là nút có nhiều con hơn. Ý tưởng này, được gọi là cân bằng trọng lượng, thì dễ dàng cài đặt bằng cách lưu kích thước của mỗi cây (là số con của gốc) trong mảng *dad* cho mỗi nút gốc và mã hoá kích thước này thành một số không dương (để tìm lại được nút gốc khi lần ngược trên cây trong hàm *find*).

Về ý tưởng, ta muốn mọi nút được chỉ trực tiếp từ gốc trên cây của nó. Không có vấn đề trong chiến lược ta dùng; tuy nhiên, việc đạt được ý đồ này đòi hỏi việc duyệt ít nhất là tất cả các nút trên một trong hai cây được trộn và điều này sẽ làm rất nhiều phép so sánh cho một số nút liên hệ trên đường dẫn tới gốc mà hàm *find* thường xét đến. Nhưng ta có thể tiếp cận ý đồ này bằng cách làm tất cả các nút được xét đều được chỉ đến từ gốc ! Mới nhìn thì có vẻ đây là một bước táo bạo; nhưng lại dễ làm và không có gì không được xâm phạm trong cấu trúc các cây này : nếu cấu trúc cây có thể thay đổi để thuật toán có hiệu quả hơn thì ta nên thay đổi nó. Phương pháp này, gọi là "nén đường dẫn", được cài đặt dễ dàng bằng cách thực hiện một phép duyệt trên mỗi cây (sau khi đã tìm được gốc) và với mỗi đỉnh bắt gặp trên đường dẫn, ta đặt đỉnh đó chỉ đến gốc.

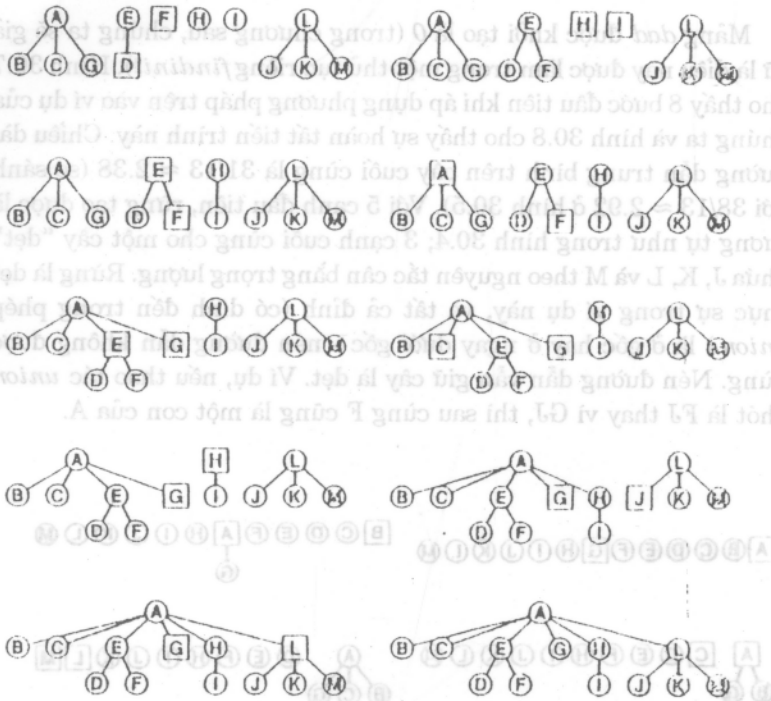
Sự kết hợp các việc cân bằng trọng lượng và nén đường dẫn sẽ làm thuật toán thực hiện rất nhanh. Việc cài đặt sau cho thấy trình mở rộng chỉ trả một giá nhỏ để đề phòng các trường hợp suy biến.

```
function find (x, y : integer; union : boolean) : boolean;
   var i, j, t : integer;
   begin    i:=x; while dad[i]>0 do i:=dad[i];
            j:=y; while dad[j]>0 do j:=dad[j];
            while dad[x]>0 do
            begin t:=x; x:=dad[x]; dad[t]:=i; end;
            while dad[y]>0 do
            begin t:=y; y:=dad[y]; dad[t]:=j; end;
            if union and (i<>j) then
               if dad[j]<dad[i]
               then begin dad[j]:=dad[j]+dad[i]-1; dad[i]:=j;
                    end
               else begin dad[i]:=dad[i]+dad[j]-1; dad[j]:=i;
                    end;
            find := (i<>j);
end;
```

Mảng *dad* được khởi tạo là *0* (trong chương sau, chúng ta sẽ giả sử là điều này được làm trong một thủ tục riêng *findinit*). Hình 30.7 cho thấy 8 bước đầu tiên khi áp dụng phương pháp trên vào ví dụ của chúng ta và hình 30.8 cho thấy sự hoàn tất tiến trình này. Chiều dài đường dẫn trung bình trên cây cuối cùng là 31/13 ≈ 2.38 (so sánh với 38/13 ≈ 2.92 ở hình 30.5). Với 5 cạnh đầu tiên, rừng tạo được là tương tự như trong hình 30.4; 3 cạnh cuối cùng cho một cây "dẹt" chứa J, K, L và M theo nguyên tắc cân bằng trọng lượng. Rừng là dẹt thực sự trong ví dụ này, và tất cả đỉnh (có đính đến trong phép *union*) là ở gốc hay ở ngay dưới gốc - nén đường dẫn không được dùng. Nén đường dẫn vẫn giữ cây là dẹt. Ví dụ, nếu thao tác *union* chót là FJ thay vì GJ, thì sau cùng F cũng là một con của A.



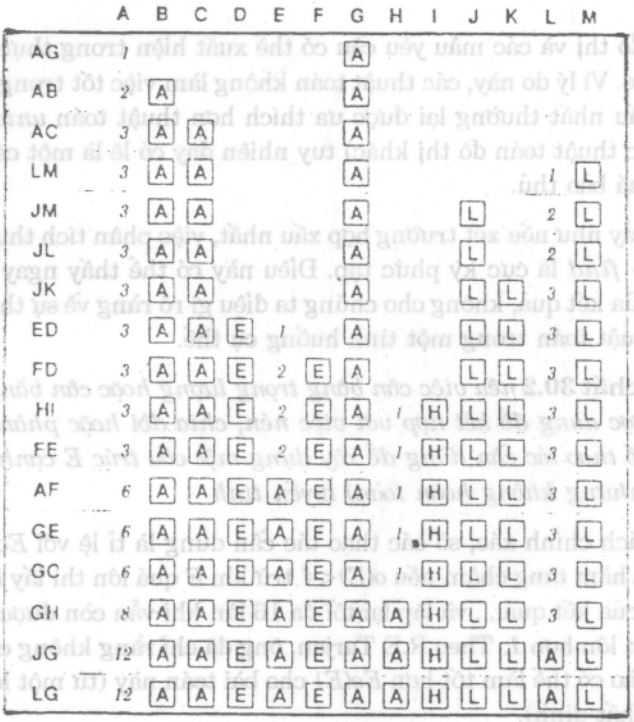
Hình 30.7 *Những bước đầu tiên của tiến trình tìm phần hợp (trường hợp có trọng số và nén đường dẫn)*



Hình 30.8 *Hoàn tất tiến trình tìm phần hợp (trường hợp có trọng số và nén đường dẫn)*

Hình 30.9 cho nội dung của mảng *dad* khi rừng được xây dựng. Để bảng này được rõ ràng, mỗi thành phần dương *i* được thay bằng chữ cái thứ *i* trong bảng chữ cái (tên của cha mẹ), và mỗi thành phần âm được bổ sung một số dương (trọng lượng của cây).

Nhiều kỹ thuật khác được phát triển nhằm tránh các trường hợp suy biến. Ví dụ, nén đường dẫn có nhược điểm là yêu cầu một phép duyệt ngược khác trên cây. Một kỹ thuật khác, gọi là *halving*, làm mỗi nút chỉ đến tổ tiên của nút đó trên đường lân ngược trên cây. Một kỹ thuật nữa là *splitting*, tựa như *halving*, nhưng chỉ áp dụng cho tất cả các nút khác trên đường dẫn. Những kỹ thuật này có thể



Hình 30.9 *Cấu trúc dữ liệu của tiến trình tìm phần hợp (trường hợp có trọng số và nén đường dẫn)*

được dùng kết hợp với cân bằng trọng lượng hay cân bằng chiều cao (cũng tương tự như cân bằng trọng lượng nhưng dùng chiều cao thay vì dùng kích thước của cây để quyết định phương pháp trộn cây).

Làm thế nào để chọn được một trong các phương pháp này ? và các cây tạo ra sẽ "dẹt" như thế nào ? Phân tích bài toán này thì hoàn toàn khó bởi vì sự thực hiện không chỉ phụ thuộc vào các tham số V, E mà còn vào các thao tác *find* và cái tệ nhất là theo trật tự xuất hiện của thao tác *union* và *find*. Không như trong việc sắp xếp, nơi mà các tập tin thường là hoàn toàn ngẫu nhiên, khó mà thấy các mô

hình đồ thị và các mẫu yêu cầu có thể xuất hiện trong thực tế như thế nào. Vì lý do này, các thuật toán không làm việc tốt trong trường hợp xấu nhất thường lại được ưa thích hơn thuật toán *union-find* (và các thuật toán đồ thị khác) tuy nhiên đây có lẽ là một cách tiếp cận quá bảo thủ.

Ngay như nếu xét trường hợp xấu nhất, việc phân tích thuật toán *union- find* là cực kỳ phức tạp. Điều này có thể thấy ngay từ bản chất của kết quả, không cho chúng ta điều gì rõ ràng về sự thực hiện của thuật toán trong một tình huống cụ thể.

Tính chất 30.2 *nếu việc cân bằng trọng lượng hoặc cân bằng chiều cao được dùng để kết hợp với việc nén, chia đôi hoặc phân rã, thì tổng số thao tác cần dùng để xây dựng một cấu trúc E cạnh là gần như (nhưng không hoàn toàn) tuyến tính*

Một cách chính xác, số các thao tác cần dùng là tỉ lệ với $Eo(E)$ với $o(E)$ là hàm tăng chậm nếu $o(E)<4$ trừ khi E quá lớn thì lấy lgE, rồi lấy lg của kết quả,... và lặp lại tối đa 16 lần khi vẫn còn được một số nguyên lớn hơn *1*. Theo R.E Tarjan, ông đã chỉ rằng không có thuật toán nào có thể làm tốt hơn $Eo(E)$ cho bài toán này (từ một lớp tổng quát nhất định).

Một áp dụng cụ thể quan trọng cho thuật toán *union-find* là xác định một đồ thị có V đỉnh và E cạnh là có liên thông hay không với thời gian tỉ lệ với V và trong thời gian hầu như tuyến tính. Đây là ưu điểm so với *DFS* trong một số tình huống: ở đó ta không cần phải lưu các cạnh. Với đồ thị có hàng ngàn đỉnh và hàng triệu cạnh, ta có thể xác định được sự liên thông của nó chỉ trong một lần duyệt các cạnh.

BÀI TẬP

1. Từ đồ thị ví dụ, xoá cạnh GJ và thêm cạnh IK. Hãy tìm các điểm bản lề và các thành phần liên thông của đồ thị mới này.

2. Vẽ cây *DFS* cho đồ thị trong bài tập 1.

3. Trong đồ thị song liên có *V* đỉnh, số cạnh nhỏ nhất là bao nhiêu ?

4. Viết chương trình in ra các thành phần song liên của một đồ thị

5. Vẽ rừng *union-find* được xây dựng trong ví dụ, với hàm *find* được thay đổi là *a[i]=j* thay cho *a[j]=i*.

6. Giải bài tập trên có sử dụng việc nén đường dẫn.

7. Vẽ rừng *union-find* có từ các cạnh AB BC CD DE EF... YZ, với giả thiết việc cân bằng trọng lượng không có nén đường dẫn, sau đó lại là nén đường dẫn mà không có cân bằng trọng lượng.

8. Giải bài tập trên, sử dụng cả việc nén đường dẫn và cân bằng trọng lượng.

9. Cài đặt các biến thể *union-find* đã mô tả trong Chương này, và xác định theo kinh nghiệm việc thực hiện các phép so sánh của chúng cho 1000 thao tác *union* với các tham biến là những số nguyên ngẫu nhiên *1...V*

10. Viết chương trình sinh một đồ thị liên thông ngẫu nhiên có *V* đỉnh bằng cách sinh ngẫu nhiên các cặp số nguyên từ *1...V*. Ước lượng số cạnh cần thiết để tạo ra một đồ thị liên thông theo *V*.

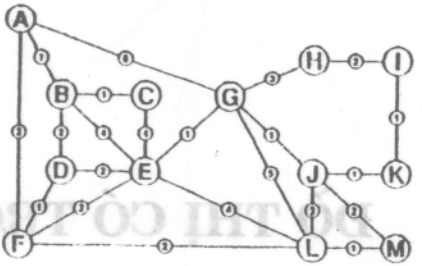
31

ĐỒ THỊ CÓ TRỌNG SỐ

Chúng ta thường muốn mô hình hóa các vấn đề thực tế bằng cách sử dụng đồ thị trong đó mỗi cạnh được gán một trọng số hay chi phí thích hợp. Trong bản đồ hàng không các cạnh đồ thị tượng trưng cho các tuyến đường bay, thì trọng số của các cạnh biểu diễn cho khoảng cách hoặc giá vé. Trong một mạch điện, thì các cạnh tượng trưng cho những sợi dây điện, trọng số thường được sử dụng là độ dài hay chi phí. Với một lưu đồ thời khóa biểu, trọng số có thể tượng trưng cho thời gian hay chi phí thực hiện hoặc chờ đợi một công việc được hoàn thành.

Trong các tình huống như trên, vấn đề cực tiểu hóa chi phí tất yếu nảy sinh. Chúng ta sẽ dành chương này để khảo sát chi tiết các thuật toán nhằm giải quyết hai bài toán: "tìm con đường đi qua tất cả các đỉnh đồ thị và có chi phí thấp nhất" và "tìm đường đi nối hai đỉnh cho trước với chi phí nhỏ nhất". Bài toán đầu tiên rõ ràng rất hữu dụng cho các đồ thị mô phỏng những vấn đề tương tự mạch điện, được gọi là bài toán tìm *cây bao trùm tối tiểu;* bài toán thứ nhì lại cần thiết với các đồ thị biểu diễn những vấn đề tương tự bản đồ hàng không, được gọi là bài toán *đường đi ngắn nhất*. Các bài toán này là tiêu biểu cho nhiều vấn đề khác nảy sinh trên các đồ thị có trọng số.

Các thuật toán của chúng ta liên quan đến việc tìm kiếm xuyên qua đồ thị, và đôi khi theo trực giác chúng ta thường có khuynh



Hình 31.1 *Một đồ thị vô hướng có trọng số*

hướng xem trọng số của một cạnh như là khoảng cách giữa hai đỉnh tạo nên cạnh đó. Nhưng cần nhớ là trọng số không bắt buộc phải thể hiện khoảng cách, nó có thể tượng trưng cho thời gian, chi phí hay mang một ý nghĩa hoàn toàn khác. Khi trọng số thật sự biểu diễn cho khoảng cách, nhiều thuật toán khác có thể thích hợp. Chúng ta sẽ thảo luận điều này chi tiết hơn ở cuối chương.

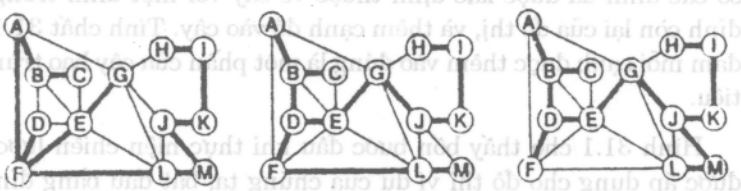
Hình 31.1 cho thấy một ví dụ về đồ thị vô hướng có trọng số. Khi thể hiện một đồ thị có trọng số, trong ma trận liên hệ sẽ lưu trữ các trọng số của các cạnh chứ không chỉ là các giá trị boolean cho biết cạnh đó tồn tại hay không; trong cấu trúc liên hệ biểu diễn cho mỗi cạnh, ta phải thêm vào một trường biểu diễn trọng số. Giả sử các trọng số đều có giá trị dương. Một vài thuật toán có thể được viết thêm những phần phụ vào để giải quyết trường hợp trọng số âm, nhưng chúng làm cho vấn đề trở nên phức tạp hơn rất nhiều. Trong các trường hợp khác, trọng số âm làm thay đổi bản chất vấn đề và đòi hỏi những thuật toán tinh vi hơn các thuật toán chúng ta xem xét ở đây. Một ví dụ cho những khó khăn có thể gặp phải là đồ thị có chu trình với tổng trọng số các cạnh trong chu trình là âm, khi đó một đường đi ngắn nhất không xác định có thể phát sinh do đi vòng chu trình. Nhiều thuật toán cổ điển đã được phát triển để giải bài toán cây bao trùm tối tiểu và đường đi ngắn nhất. Các thuật toán này là một trong những thuật toán nổi tiếng nhất trong cuốn sách

này. Như chúng ta đã biết, các thuật toán cổ điển bao giờ cũng cung cấp một cách tiếp cận tổng quát, nhưng các cấu trúc dữ liệu hiện đại cho phép cài đặt gọn gàng và hiệu quả. Trong chương này chúng ta sẽ xem xét cách sử dụng các hàng ưu tiên trong sự tổng quát của phương pháp du đồ ở chương 29 để giải cả hai bài toán trên một cách hiệu quả cho các đồ thị thưa; chúng ta cũng xem xét mối liên hệ giữa phương pháp này với các phương pháp cổ điển cho các đồ thị phức tạp; và cũng xem xét một phương pháp giải bài toán cây bao trùm tối tiểu sử dụng cách tiếp cận hoàn toàn khác.

CÂY BAO TRÙM TỐI TIỂU

Cây bao trùm tối tiểu của một đồ thị có trọng số là một tập hợp các cạnh nối tất cả các đỉnh sao cho tổng trọng số của các cạnh trong tập hợp này là nhỏ nhất so với tổng trọng số của các tập hợp cạnh khác thỏa điều kiện nối tất cả các đỉnh. Cây bao trùm tối tiểu không bắt buộc phải là duy nhất: hình 31.2 cho thấy ba cây bao trùm tối tiểu của đồ thị ví dụ ban đầu. Có thể dễ dàng chứng tỏ rằng tập hợp các cạnh ta vừa định nghĩa ở trên phải tạo thành một cây bao trùm: nếu có bất kỳ một chu trình nào, thì có thể xóa bớt vài cạnh trong chu trình mà vẫn có tập hợp các cạnh nối tất cả đỉnh đồ thị với tổng trọng số nhỏ hơn.

Trong chương 29 chúng ta đã thấy nhiều thủ tục xác định một cây bao trùm của đồ thị. Liệu chúng ta có thể sắp xếp mọi việc đối



Hình 31.2 *Cây bao trùm tối thiểu*

với một đồ thị có trọng số để cây bao trùm xác định được chính là cây bao trùm tối thiểu? Có nhiều cách để làm được điều đó, tất cả đều dựa trên tính chất tổng quát sau đây của các cây bao trùm tối thiểu.

Tính chất 31.1 *Nếu chia tất cả các đỉnh của đồ thị theo bất kỳ cách nào thành hai tập hợp đỉnh, thì cây bao trùm tối thiểu sẽ chứa cạnh ngắn nhất trong số các cạnh nối một đỉnh trong tập hợp này với một đỉnh trong tập hợp kia.*

Ví dụ, ta chia các đỉnh trong đồ thị ở trên thành hai tập hợp A B C D và E F G H I J K L M lúc đó hàm ý DF phải ở trong mọi cây bao trùm tối thiểu. Có thể dễ dàng chứng minh tính chất này bằng phản chứng. Gọi cạnh ngắn nhất nối các đỉnh giữa hai tập là *s*, giả sử *s* không ở trong một cây bao trùm tối thiểu. Bây giờ ta xét đến đồ thị được tạo bằng cách thêm *s* vào một cây bao trùm tối thiểu của đồ thị gốc đã xác định được trước đó. Đồ thị mới này có một chu trình; trong chu trình đó một vài cạnh khác *s* phải nối các đỉnh trong hai tập đỉnh. Nếu xóa bớt cạnh này trong đồ thị mới thì ta sẽ vẫn có cây bao trùm nhưng trọng số nhỏ hơn và chứa *s* (vì *s* có trọng số nhỏ nhất trong số các cạnh nối hai tập đỉnh), vậy điều đó trái với giả thiết ban đầu, suy ra *s* phải nằm trong cây bao trùm tối thiểu.

Vì vậy chúng ta có thể xây dựng cây bao trùm tối thiểu bằng cách khởi đầu từ một đỉnh bất kỳ và luôn luôn tìm đỉnh kế tiếp "gần nhất" đối với các đỉnh đã xác định trước đó. Nói cách khác, chúng ta tìm cạnh có trọng số nhỏ nhất trong số các cạnh nối giữa một đỉnh trong số các đỉnh đã được xác định thuộc về cây với một đỉnh trong tập đỉnh còn lại của đồ thị, và thêm cạnh đó vào cây. Tính chất 31.1 bảo đảm mỗi cạnh được thêm vào đúng là một phần của cây bao trùm tối thiểu.

Hình 31.1 cho thấy bốn bước đầu khi thực hiện chiến lược này được áp dụng cho đồ thị ví dụ của chúng ta, bắt đầu bằng đỉnh A. Đỉnh gần với A nhất (được nối với A bằng cạnh có trọng số nhỏ nhất) là B, vì vậy AB thuộc về cây bao trùm tối thiểu. Trong tất cả các cạnh

liên thông với AB, cạnh BC có trọng số nhỏ nhất, vì thế nó được thêm vào cây và đỉnh C được xét tiếp theo. Sau đó, đỉnh gần nhất đối với A, B, C là D vì thế BD được thêm vào cây. Hình 31.5 thể hiện kết quả cuối cùng.

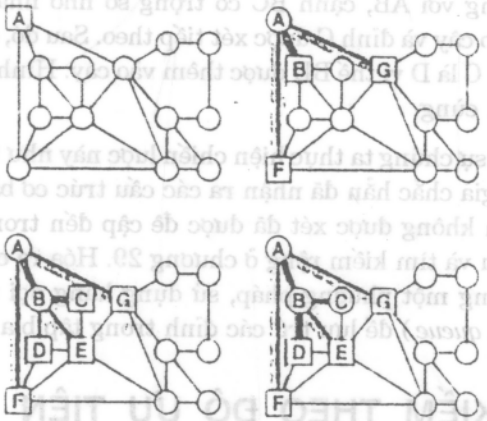
Thật sự chúng ta thực hiện chiến lược này như thế nào ? Đến bây giờ đọc giả chắc hẳn đã nhận ra các cấu trúc cơ bản của cây, bìa và các đỉnh không được xét đã được đề cập đến trong chiến lược tìm kiếm sâu và tìm kiếm rộng ở chương 29. Hóa ra chúng ta có thể sử dụng cùng một phương pháp, sử dụng hàng đợi ưu tiên (thay cho *stack* và *queue*) để lưu trữ các đỉnh trong tập bìa.

TÌM KIẾM THEO ĐỘ ƯU TIÊN

(PRIORITY-FIRST SEARCH)

Từ chương 29 chúng ta đã biết việc tìm kiếm trên đồ thị có thể mô tả như là việc chia các đỉnh thành ba tập hợp: các *đỉnh cây* (tree), là các đỉnh ứng với các cạnh đã được xem xét; các *đỉnh bìa* (fringe), là các đỉnh đang chờ được xử lý và đang được lưu trữ trong một cấu trúc dữ liệu nào đó; và các *đỉnh không nhìn thấy* (unseen), là các đỉnh chưa được xem xét đến. Phương pháp tìm kiếm căn bản trên đồ thị mà chúng ta sử dụng dựa trên các bước "chuyển một đỉnh x từ tập bìa sang tập cây, sau đó đặt vào bìa một đỉnh bất kỳ kề với x và chưa được xét". Chúng ta sử dụng thuật ngữ priority-first search để nói đến chiến lược tổng quát sử dụng hàng đợi có ưu tiên (priority queue) để quyết định đỉnh nào sẽ lấy từ fringe ra để xử lý. Điều này cho phép một sự linh động đáng kể. Như chúng ta đã thấy, nhiều thuật toán cổ điển (bao gồm cả tìm kiếm sâu và tìm kiếm rộng) chỉ khác biệt trong cách chọn sự ưu tiên.

Để xác định cây bao trùm tối tiểu, độ ưu tiên của mỗi đỉnh trong tập bìa nên là độ dài của cạnh ngắn nhất nối đỉnh đó vào cây đang tạo lập. Hình 31.4 thể hiện nội dung của hàng đợi ưu tiên trong suốt quá



Hình 31.3 *Các bước khởi đầu xây dựng cây bao trùm tối tiểu*

trình tạo dựng cây bao trùm tối tiểu được minh họa ở hình 31.3 và 31.5. Để rõ ràng, các phần tử trong hàng đợi được thể hiện trong trình tự đã sắp. Sự cài đặt hàng đợi ưu tiên theo kiểu danh sách được sắp này có thể thích hợp với các đồ thị nhỏ, nhưng nên sử dụng heap cho các đồ thị lớn để bảo đảm tất cả các thao tác được hoàn thành trong O(logN) bước (xem chương 11).

Trước tiên chúng ta xem xét các đồ thị thưa biểu biển bằng một danh sách kề. Như đã nhắc tới ở trên, ta thêm trường trọng số *w* vào bản ghi *edge* (và sửa chữa chương trình ứng với đoạn nhập trọng số). Sau đó, sử dụng một hàng đợi ưu tiên cho tập bìa, chúng ta có đoạn chương trình ở trang sau :

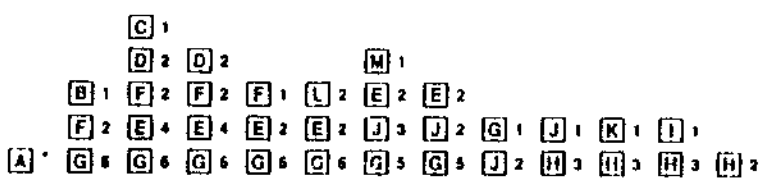
Để xác định cây bao trùm tôi tiểu, thay thế cả hai lần xuất hiện của *priority* bằng $t\uparrow.w$ Các thủ tục *pqinitialize* và *pqremove* là những thủ tục xử lý hàng đợi ưu tiên như đã mô tả trong chương 11. Hàm *pqupdate* được cài đặt thêm vào tập các thủ tục xử lý hàng đợi ưu tiên nhằm đảm bảo một đỉnh đã cho xuất hiện trong hàng đợi với tối thiểu một độ ưu tiên cho trước: nếu đỉnh đó không trong hàng

```
procedure listpfs;
   var id, k : integer; val : array [1..maxV] of integer;
   procedure visit(k:integer);
      var t : link;
      begin
         if pqupdate(k,unseen) then dad[k]:=0;
         repeat
            id:=id+1;
            k:=pqremove; val[k]:=-val[k];
            if val[k]=unseen then val[k]:=0;
            t:=adj[k];
            while t<>z do
            begin
               if val[t↑.v]<0 then
                  if pqupdate(t↑.v, priority) then
                     begin val[t↑.v]:=-priority; dad[t↑.v]:=k
                     end;
               t:=t↑.next
            end;
         until pqempty
      end;
   begin
      id:=0; pqinitialize;
      for k:=1 to V do val[k]:=-unseen;
      for k:=1 to V do
         if val[k]=-unseen then visit(k)
   end;
```

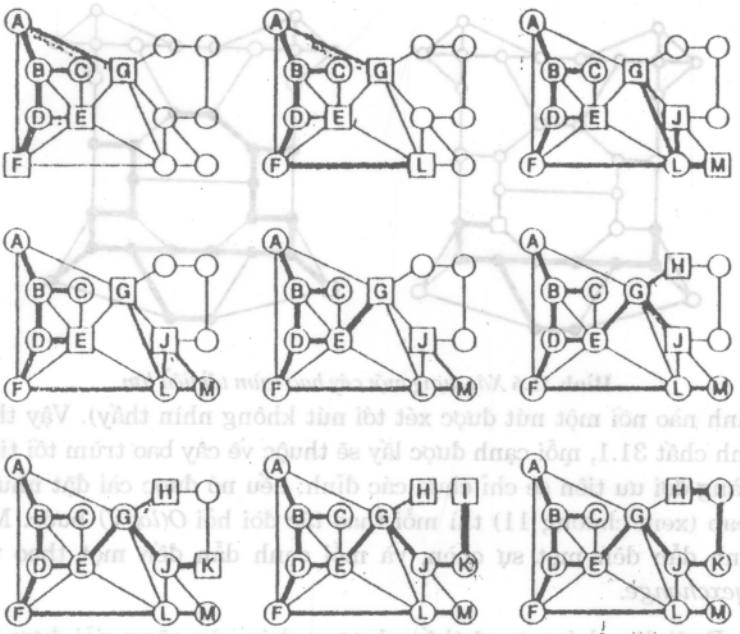
đợi, thủ tục *pqinsert* được thi hành; nếu đỉnh nằm trong hàng đợi nhưng có độ ưu tiên lớn hơn thì thi hành thủ tục *pqchange* để thay đổi độ ưu tiên. Bất kỳ một sự thay đổi nào xảy ra cũng làm cho *pqupdate* trả về giá trị true. Điều này cho phép chương trình trên có thể giữ các mảng *val* và *dad* hiện tại (current). Mảng *val* tự nó có thể thật sự giữ hàng đợi ưu tiên theo kiểu gián tiếp; trong chương trình trên chúng ta đã tách rời các thao tác trên hàng đợi ra cho rõ ràng.

Ngoài việc thay đổi cấu trúc dữ liệu thành hàng đợi ưu tiên, chương trình trên giống các chương trình sử dụng cho tìm kiếm rộng và sâu, với hai ngoại lệ. Trước hết, cần đến một thao tác bổ sung khi cạnh được xét đã nằm trong tập bìa rồi: với tìm kiếm rộng và sâu, các cạnh như thế được bỏ qua, nhưng trong chương trình trên ta cần phải kiểm tra xem cạnh mới có làm giảm độ ưu tiên hay không. Điều này bảo đảm chúng ta luôn xét đến đỉnh kế tiếp trong tập bìa thỏa điều kiện "gần" với cây nhất như mong muốn. Thứ đến, chương trình này giữ vết của cây một cách tường minh bằng cách duy trì mảng dad mảng lưu trữ cha của mỗi nút trong cây tìm kiếm ưu tiên (tên của nút khiến cho nút đó được đưa từ tập bìa vào tập cây). Cũng thế, với mỗi nút k trên cây, *val[k]* là trọng số của cạnh nối giữa k và *dad[k]*. Các đỉnh trong tập bìa được đánh dấu bằng các trọng số âm như ở phần trước; các đỉnh không nhìn thấy được đánh dấu bằng giá trị *-unseen* thay vì 0. Lý do của sự thay đổi này sẽ trở nên rõ ràng dưới đây.

Hình 31.5 cho thấy kết quả hoàn tất của việc xây dựng cây bao trùm tối tiểu. Như thường lệ, các đỉnh được khoanh trong hình vuông là các đỉnh trên bìa, các đỉnh được khoanh tròn và có nhãn là các đỉnh trên cây, và các đỉnh được khoanh tròn, không nhãn là các đỉnh không nhìn thấy. Các cạnh của cây là các đường đen dày, cạnh có bóng là cạnh ngắn nhất nối mỗi đỉnh trên bìa vào cây. Độc giả nên



Hình 31.4 *Nội dung của hàng đợi ưu tiên trong tiến trình xây dựng cây bao trùm tối tiểu*

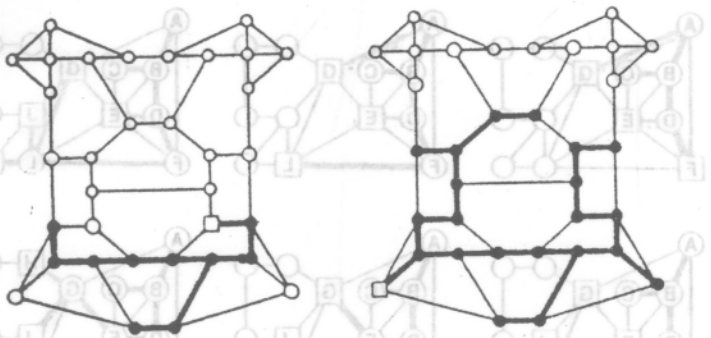


Hình 31.5 *Kết quả hoàn tất của việc xây dựng cây bao trùm tối tiểu*

theo dõi quá trình tạo cây từ hình 31.4. Đặc biệt, chú ý cách thức nút G được thêm vào cây sau nhiều bước ở tập bìa. Đầu tiên, khoảng cách từ G đến cây là 6 (cạnh GA). Sau khi L được thêm vào cây, GL làm cho khoảng cách từ G đến cây chỉ còn 5, sau đó E lại được thêm vào cây nên khoảng cách chỉ còn 1, và G được thêm vào cây trước J. Hình 31.6 thể hiện sự tạo thành của cây bao trùm tối tiểu cho một đồ thị lớn và "hỗn độn", với chiều dài cạnh được xem như là trọng số.

Tính chất 31.2 *Tìm kiếm theo thứ tự ưu tiên trên đồ thị thưa xác định được cây bao trùm tối tiểu trong O((E+V)logV) bước.*

Áp dụng tính chất 31.1: hai tập đang được quan tâm đến là tập các nút được xét và tập các nút không được xét đến. Tại mỗi bước, chúng ta lấy một cạnh ngắn nhất từ một nút được xét nút bìa (Không có



Hình 31.6 *Xây dựng một cây bao trùm tối tiểu lớn*

cạnh nào nối một nút được xét tới nút không nhìn thấy). Vậy theo tính chất 31.1, mỗi cạnh được lấy sẽ thuộc về cây bao trùm tối tiểu. Hàng đợi ưu tiên sẽ chỉ chứa các đỉnh: nếu nó được cài đặt như là heap (xem chương 11) thì mỗi thao tác đòi hỏi $O(logV)$ bước. Mỗi đỉnh dẫn đến một sự chèn, và mỗi cạnh dẫn đến một thao tác *pqexchange*.

Dưới đây chúng ta sẽ thấy phương pháp này cũng giải được bài toán đường đi ngắn nhất với độ ưu tiên được chọn lựa thích hợp. Chúng ta cũng xem xét một cách cài đặt khác của hàng đợi có điều kiện có thể cho một thuật toán V^2, thích hợp cho các đồ thị dày. Thuật toán này tương đương với các thuật toán cũ từ năm 1957: với bài toán cây bao trùm tối tiểu, ta có thuật toán R.Prim; với bài toán đường đi ngắn nhất ta có thuật toán E.Dijkstra cho đồ thị dày, và thuật toán vừa nhắc đến ở trên gọi là "giải pháp tìm kiếm ưu tiên".

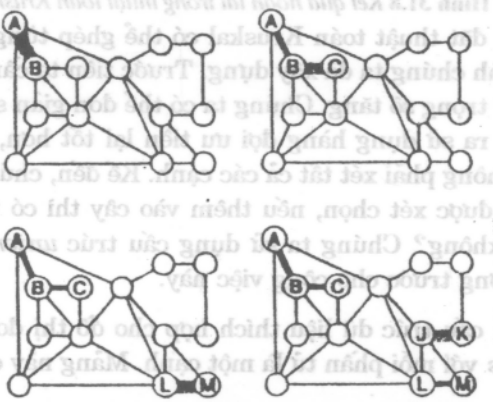
Tìm kiếm ưu tiên là một sự tổng quát hóa của các phương pháp tìm kiếm rộng và tìm kiếm sâu vì các phương pháp này có thể được dẫn xuất từ tìm kiếm ưu tiên với độ ưu tiên thích hợp. Nên nhớ rằng *id* tăng từ 1 đến V trong suốt quá trình xử lý của thuật toán và vì thế có thể dùng để gán một độ ưu tiên duy nhất cho các đỉnh. Nếu thay *priority* trong *listpfs* bằng *V-id*, ta sẽ có tìm kiếm sâu, vì đỉnh vừa

mới xét có độ ưu tiên cao nhất. Nếu sử dụng *id* thay cho *priority* ta lại có tìm kiếm theo chiều rộng vì các đỉnh vừa mới xét xong có độ ưu tiên cao nhất. Việc gán độ ưu tiên này làm cho hàng đợi thao tác giống stack và queue.

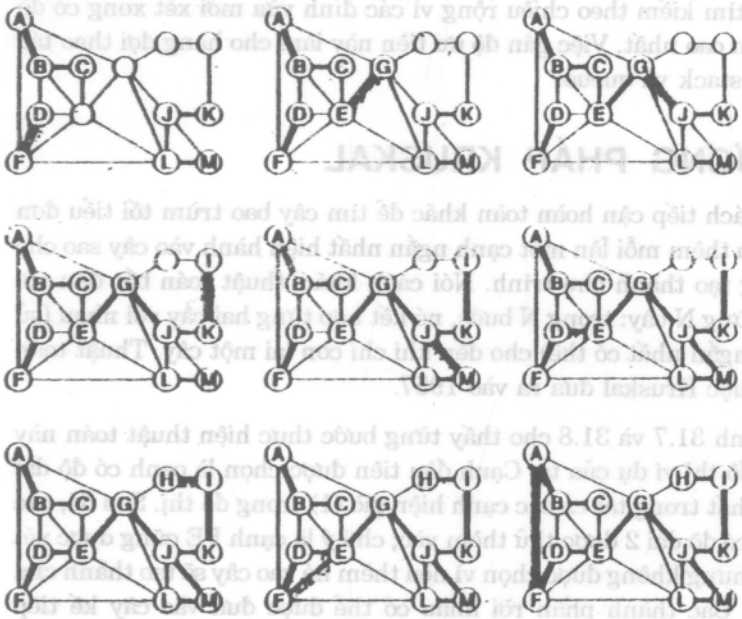
PHƯƠNG PHÁP KRUSKAL

Một cách tiếp cận hoàn toàn khác để tìm cây bao trùm tối tiểu đơn giản là thêm mỗi lần một cạnh ngắn nhất hiện hành vào cây sao cho không tạo thành chu trình. Nói cách khác, thuật toán bắt đầu với một rừng N cây: trong N bước, nó kết hợp từng hai cây với nhau (sử cạnh ngắn nhất có thể) cho đến khi chỉ còn lại một cây. Thuật toán này được Kruskal đưa ra vào 1957.

Hình 31.7 và 31.8 cho thấy từng bước thực hiện thuật toán này trên đồ thị ví dụ của ta. Cạnh đầu tiên được chọn là cạnh có độ dài nhỏ nhất trong tất cả các cạnh hiện giờ (1) trong đồ thị. Sau đó, các cạnh có độ dài 2 được thử thêm vào; chú ý là cạnh FE cũng được xét đến nhưng không được chọn vì nếu thêm nó vào cây sẽ tạo thành chu trình. Các thành phần rời nhau có thể được đưa vào cây kế tiếp nhau, khác với thuật toán "tìm kiếm ưu tiên" xây dựng cây tăng trưởng theo từng nhánh.



Hình 31.7 *Những bước bắt đầu trong thuật toán Kruskal*



Hình 31.8 *Kết quả hoàn tất trong thuật toán Kruskal*

Việc cài đặt thuật toán Kruskal có thể ghép từng mảnh từ các chương trình chúng ta đã xây dựng. Trước tiên ta cần xét các cạnh theo thứ tự trọng số tăng. Chúng ta có thể đơn giản sắp xếp chúng, nhưng hóa ra sử dụng hàng đợi ưu tiên lại tốt hơn, chủ yếu là vì chúng ta không phải xét tất cả các cạnh. Kế đến, chúng ta cần xem cạnh đang được xét chọn, nếu thêm vào cây thì có tạo thành chu trình hay không? Chúng ta sử dụng cấu trúc *unionfind* đã thảo luận ở chương trước cho công việc này.

Bây giờ, cấu trúc dữ liệu thích hợp cho đồ thị đơn giản là một mảng *edges* với mỗi phần tử là một cạnh. Mảng này có thể dễ dàng

xây dựng từ danh sách kề hay ma trận kề của đồ thị với tìm kiếm sâu, hoặc một thủ tục đơn giản hơn. Tuy nhiên trong chương trình dưới đây, Chúng tôi chỉ trực tiếp khởi động mảng này bằng các giá trị nhập. Các thủ tục gián tiếp xử lý hàng đợi ưu tiên *pqconstruct* và pqremove ở chương 11 vẫn được dùng để bảo trì hàng đợi, dùng các trường trọng số (w) trong mảng edges làm các độ ưu tiên, và thủ tục *findinit* và *find* ở chương 30 được sử dụng để kiểm tra chu trình có tồn tại hay không. Chương trình dưới đây đơn giản chỉ gọi thủ tục *edgefound* cho mỗi cạnh trong cây bao trùm; với một chút xử lý nữa thì có thể xác định luôn cả mảng *dad* hay một cấu trúc biểu diễn khác.

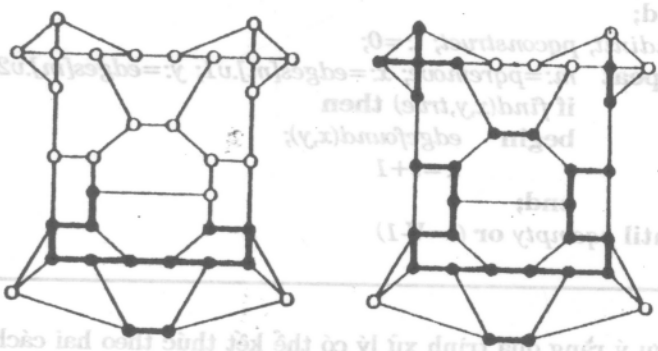
```
program kruskal(input, output);
const maxV=50; maxE=2500;
type edge= record v1, v2, w : integer end;
var i, j, m, , y, V, E : integer; edges: array [0..maxE] of edge;
begin
   readln(V,E);
   for j:=1 to E do
   begin    readln(c,d,edges[j].w);
            edges[j].v1:=index(c); edges[j].v2:=index(d);
   end;
   findinit; pqconstruct; i:=0;
   repeat   m:=pqremove; x:=edges[m].v1; y:=edges[m].v2;
            if find(x,y,true) then
            begin    edgefound(x,y);
                     i:=i+1
            end;
   until pqempty or (i=V-1)
end.
```

Lưu ý rằng quá trình xử lý có thể kết thúc theo hai cách. Nếu chúng ta tìm thấy *V-1* cạnh, thì chúng ta đã có cây và có thể dừng.

Nếu hàng đợi ưu tiên đã rỗng, thì chúng ta đã duyệt tất cả các cạnh mà không xây dựng được cây bao trùm, điều này có thể xảy ra khi đồ thị không liên thông. Thời gian thực hiện phụ thuộc vào thời gian xử lý các cạnh trong hàng đợi.

Tính chất 31.3 *Thuật toán Kruskal xác định cây bao trùm tối tiểu của đồ thị trong O(ElogE) bước.*

Sự đúng đắn của thuật toán cũng dựa trên tính chất 31.1. Hai tập hợp đỉnh đang được quan tâm là tập các đỉnh ứng với các cạnh đã được chọn để xây dựng cây, và tập các đỉnh chưa đụng chạm tới. Mỗi cạnh được thêm vào là cạnh ngắn nhất giữa các đỉnh ở hai tập. Trường hợp xấu nhất là đồ thị không liên thông, lúc đó tất cả các cạnh đều phải được xem xét. Ngay cả một đồ thị liên thông, trường hợp xấu nhất cũng có thể xảy ra vì đồ thị có thể bao gồm hai phần mà trong mỗi phần các đỉnh được nối với nhau bằng các cạnh ngắn, và chỉ có một cạnh dài nối liền hai phần. Khi đó cạnh dài nhất trong đồ thị sẽ ở trong cây bao trùm tối tiểu, nhưng sẽ là cạnh cuối cùng được lấy ra khỏi hàng đợi ưu tiên. Đối với các đồ thị điển hình, chúng ta có thể chờ đợi cây bao trùm tối tiểu được hoàn tất trước khi phải lấy cạnh dài nhất của đồ thị (cây chỉ có *V-1* cạnh), tuy nhiên luôn cần



Hình 31.9 *Tìm một cây bao trùm tối tiểu bằng thuật toán Kruskal*

thời gian tỉ lệ với E để xây dựng hàng đợi ưu tiên ban đầu (xem tính chất 11.2).

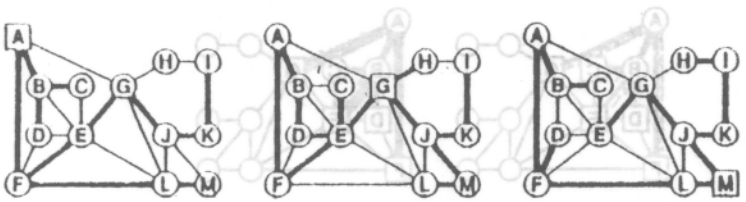
Hình 31.9 minh họa việc tạo dựng một cây bao trùm tối tiểu lớn hơn với thuật toán Kruskal. Sơ đồ này cho thấy một cách rõ ràng các phương pháp lấy tất cả các cạnh ngắn nhất trước tiên: nó sẽ thêm các cạnh dài hơn vào sau cùng.

Bạn có thể không sử dụng hàng đợi ưu tiên, mà chỉ sắp xếp các cạnh theo trọng số, sau đó xử lý chúng theo thứ tự. Cũng vậy, việc kiểm tra chu trình có thể thực hiện trong thời gian tỉ lệ với $ElogE$ với một chiến lược đơn giản hơn *unionfind*, để được một cây bao trùm tối tiểu luôn luôn phải trải qua $E\ logE$ bước. Đó là phương pháp nguyên thủy do Kruskal đề nghị, nhưng phần trên chúng tôi đã đề cập đến một phiên bản hiện đại hơn với sự sử dụng hàng đợi ưu tiên và cấu trúc *unionfind* như là thuật toán Kruskal.

ĐƯỜNG ĐI NGẮN NHẤT

Mục đích của bài toán đường đi ngắn nhất trong đồ thị có hướng là tìm đường đi nối hai đỉnh x, y cho trước sao cho tổng trọng số của tất cả các cạnh trên đường đi là nhỏ nhất.

Nếu tất cả các trọng số đều là 1 thì bài toán vẫn còn thú vị: hãy tìm đường đi có ít cạnh nhất nối hai đỉnh x, y cho trước. Chúng ta sẽ nghiên cứu thuật toán tìm kiếm ưu tiên độ rộng để giải bài toán này.

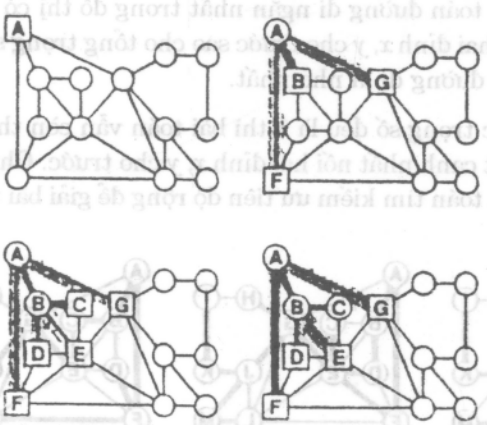


Hình 31.10 *Cây bao trùm đường đi ngắn nhất*

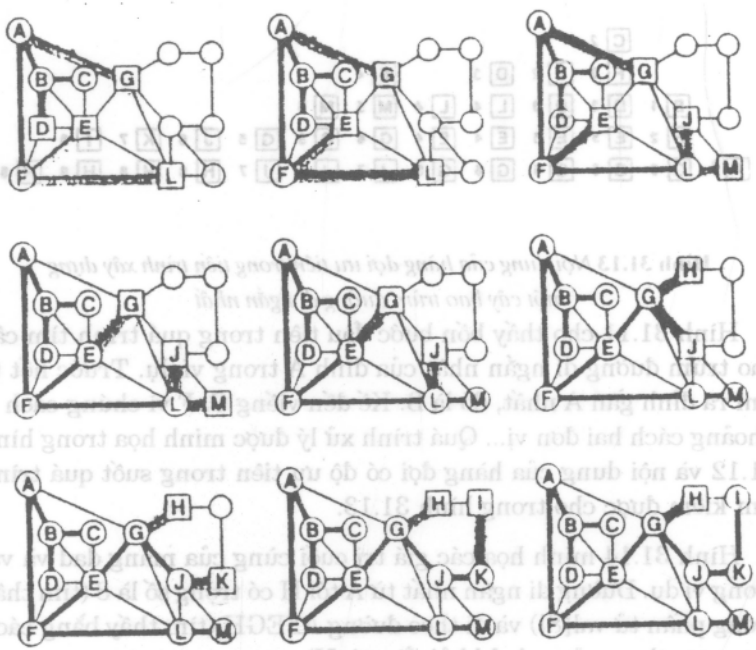
Có thể chứng minh dễ dàng bằng qui nạp rằng quá trình tìm kiếm ưu tiên độ rộng (khởi đầu từ một đỉnh x) trước tiên sẽ viếng tất cả các đỉnh được nối tới x bởi một cạnh trong đồ thị, kế đến là viếng tất cả các đỉnh có thể được nối tới x bởi cạnh của đồ thị... Do đó, khi y được gặp lần đầu tiên thì đường đi ngắn nhất từ x tới y được tìm thấy bởi vì không có đường đi nào ngắn hơn (xem lại hình 29.14).

Nói chung, đường đi từ x tới y có thể qua tất cả các đỉnh, vì vậy thông thường chúng xét bài toán tìm đường đi ngắn nhất nối một đỉnh x đã cho với tất cả các đỉnh khác trong đồ thị. Bây giờ thì bài toán lại có thể giải quyết đơn giản bằng thuật toán duyệt đồ thị theo độ ưu tiên đã được trình bày bên trên.

Nếu chúng ta vẽ đường đi ngắn nhất từ đỉnh x tới mỗi đỉnh khác trong đồ thị thì rõ ràng chúng ta sẽ không gặp bất kỳ chu trình nào và chúng ta có được một **cây bao trùm đường đi ngắn nhất**, ví dụ hình 31.10 cho thấy các cây bao trùm có được khi đỉnh x lần lượt là A, G và M.

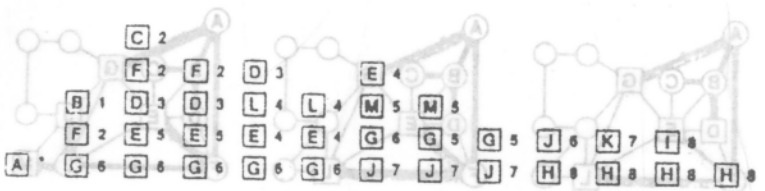


Hình 31.11 *Các bước bắt đầu xây dựng một cây bao trùm đường đi ngắn nhất*



Hình 31.12 *Kết quả hoàn tất xây dựng một cây bao trùm đường đi ngắn nhất*

Việc giải bài toán này nhờ vào phương pháp tìm kiếm theo độ ưu tiên cũng giống như giải bài toán cây bao trùm tối tiểu: chúng ta xây dựng cây cho mỗi đỉnh x bằng cách thêm vào các đỉnh "bìa" gần x nhất trong mỗi bước (trước khi chúng ta đã thêm vào một đỉnh gần cây nhất). Để tìm ra đỉnh ven nào gần x nhất, chúng ta dùng một biến mảng *val:* với mỗi đỉnh k, *val[k]* là khoảng cách từ đỉnh đó tới x dọc theo đường đi ngắn nhất. Khi k được thêm vào cây, chúng ta cập nhật xâu kề của k. Với mỗi nút t trong xâu, khoảng cách ngắn nhất tới x xuyên qua từ $t.v$ là $val[k]+t.w$. Do đó thuật toán được cài đặt dễ dàng bằng cách dùng đại lượng này làm độ ưu tiên trong thuật toán tìm kiếm theo độ ưu tiên.



Hình 31.13 *Nội dung của hàng đợi ưu tiên trong tiến trình xây dựng một cây bao trùm đường đi ngắn nhất*

Hình 31.11 cho thấy bốn bước đầu tiên trong quá trình tìm cây bao trùm đường đi ngắn nhất của đỉnh A trong ví dụ. Trước hết ta tìm ra đỉnh gần A nhất, đó là B. Kế đến viếng C, F vì chúng cách A khoảng cách hai đơn vị... Quá trình xử lý được minh họa trong hình 31.12 và nội dung của hàng đợi có độ ưu tiên trong suốt quá trình tìm kiếm được cho trong hình 31.13.

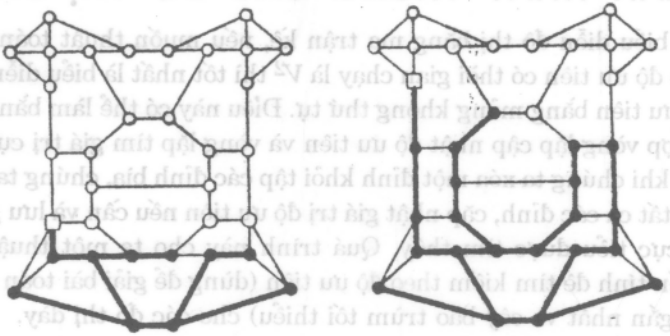
Hình 31.14 minh họa các giá trị cuối cùng của mảng dad và val trong ví dụ. Đường đi ngắn nhất từ A tới H có trọng số là 8 (tìm thấy trong phần tử val[8]) và đi theo đường AFEGH (tìm thấy bằng cách đi ngược theo mảng dad khởi đầu từ H).

Tính chất 31.4 *Tìm kiếm dựa vào độ ưu tiên trong một đồ thị thưa tìm ra cây đường đi ngắn nhất sao O((E + V)logV) bước.*

Sự chính xác của thuật toán có thể được chứng minh bằng phương pháp tương tự như Tính chất 31.1. Với hàng đợi có độ ưu tiên được

k	1	2	3	4	5	6	7	8	9	10	11	12	13
*name*dad[k]	A	B	B	F	A	E	G	K	G	I	F	L	
val[k]	0	1	2	3	4	2	5	8	8	6	7	4	5

Hình 31.14 *Biểu hiện cây bao trùm đường đi ngắn nhất*



Hình 31.15 *Xây dựng một cây bao trùm đường đi ngắn nhất lớn*

cài đặt dựa vào cấu trúc heap như chương 12, thì thuật toán tìm kiếm theo độ ưu tiên luôn luôn bảo đảm thời gian được ước lượng như trên không phụ thuộc vào việc dùng độ ưu tiên nào.

Dưới đây chúng ta sẽ thấy một cách cài đặt khác của hàng đợi có độ ưu tiên để có được một thuật toán cấp V^2 thích hợp cho các đồ thị dày. Đối với bài toán tìm đường đi ngắn nhất, điều này đưa đến một phương pháp có lẽ do E.Dijkstra tìm ra vào khoảng năm 1959.

Hình 31.15 minh họa một cây đường đi ngắn nhất lớn hơn. Giống như lúc trước, chiều dài các cạnh được dùng như trọng số trong đồ thị này, và chúng ta cần tìm được đi ngắn nhất từ đỉnh đáy bên trái tới mỗi đỉnh trong đồ thị. Sau này chúng ta sẽ thảo luận về một cải tiến có thể thích hợp cho các đồ thị như thế. Nhưng ngay cả trong đồ thị này, có lẽ thích hợp hơn khi dùng các giá trị khác làm trọng số: ví dụ, nếu đồ thị này biểu diễn một mê cung (xem chương 29), trọng số của cạnh có thể được biểu diễn bởi chính các khoảng cách trong mê cung.

CÂY BAO TRÙM TỐI TIỂU VÀ ĐƯỜNG ĐI NGẮN NHẤT TRONG CÁC ĐỒ THỊ DÀY

Khi biểu diễn đồ thị bằng ma trận kề, nếu muốn thuật toán duyệt theo độ ưu tiên có thời gian chạy là V^2 thì tốt nhất là biểu diễn hàng đợi ưu tiên bằng mảng không thứ tự. Điều này có thể làm bằng cách tổ hợp vòng lặp cập nhật độ ưu tiên và vòng lặp tìm giá trị cực tiểu: mỗi khi chúng ta xóa một đỉnh khỏi tập các đỉnh bìa, chúng ta duyệt qua tất cả các đỉnh, cập nhật giá trị độ ưu tiên nếu cần và lưu giữ giá trị cực tiểu được tìm thấy. Quá trình này cho ta một thuật toán tuyến tính để tìm kiếm theo độ ưu tiên (dùng để giải bài toán đường đi ngắn nhất và cây bao trùm tối thiểu) cho các đồ thị dày.

Đặc biệt, chúng ta cài đặt cấu trúc hàng đợi ưu tiên trong mảng *val* (cũng có thể cài đặt như trong thủ tục *listpfs* như đã thảo luận trước đây) và cài đặt trực tiếp các thao tác trên cấu trúc này thay vì dùng cấu trúc heap. Dấu của mỗi phần tử trong mảng *val* cho biết đỉnh tương ứng ở trên cây hay trong hàng đợi. Lúc đầu thì tất cả các đỉnh đều ở trong hàng đợi và có độ ưu tiên mặc nhiên *unseen*, để thay đổi độ ưu tiên cho một đỉnh chúng ta chỉ cần gán độ ưu tiên mới vào phần tử trong mảng *val* tương ứng với đỉnh đó. Sau khi hiệu chỉnh chương trình *listpfs* chúng ta có chương trình *matrixpfs* ở trang sau:

Chú ý rằng giá trị *unseen* phải nhỏ hơn *maxint* một ít (bởi vì cần có một giá trị lớn hơn để dùng làm "lính canh" cho việc tìm giá trị nhỏ nhất) và *-unseen* phải có nghĩa trong kiểu nguyên đang sử dụng.

Nếu chúng ta lưu các trọng số trong ma trận kề và dùng *a[k,t]* làm độ ưu tiên trong chương trình này thì sẽ có thuật toán Prim (tìm cây bao trùm tối tiểu); Nếu chúng ta sử dụng *val[k]+a[k,t]* như độ ưu tiên thì sẽ có thuật toán Dijkstra (bài toán đường đi ngắn nhất); nếu chúng ta thêm vài lệnh để dùng *id* làm số đỉnh đã được duyệt và

```
procedure matrixpfs;
  var k, min, t : integer;
  begin   for k:=1 to V do
             begin val[k]:=-unseen; dad[k]:=0 end;
          val[0]:=-(unseen+1); min:=1;
          repeat
             k:=min; val[k]:=-val[k]; min:=0;
             if val[k]=unseen then val[k]:=0;
             for t:=1 to V do
                if val[t]<0 then
                begin
                if (a[k,t]<>0) and (val[t]<-priority) then
                   begin val[t]:=-priority; dad[t]:=k; end;
                if val[t]>val[min] then min:=t
                end;
          until min=0;
  end;
```

V-id làm độ ưu tiên thì sẽ được thuật toán tìm kiếm ưu tiên độ sâu; nếu dùng id thì có thuật toán tìm kiếm ưu tiên độ rộng. Chương trình này khác với chương trình trình tìm kiếm theo độ ưu tiên đã được viết cho đồ thị thưa về mặt biểu diễn đồ thị (dùng ma trận kề thay vì xâu kề) và cài đặt hàng đợi ưu tiên (dùng mảng không thứ tự thay vì dùng heap).

Tính chất 31.5 *Bài toán đường đi ngắn nhất và cây bao trùm tối tiểu có thể được giải trong khoảng thời gian tuyến tính đối với đồ thị thưa.*

Thời gian hoạt động của thuật toán xấp xỉ với V^2, mỗi lần một đỉnh được viếng thì thuật toán duyệt qua V phần tử trong một dòng của ma trận kề để kiểm tra tất cả các cạnh kề, cập nhật và tìm giá trị nhỏ nhất kế tiếp trong hàng đợi ưu tiên. Do đó thời gian chạy sẽ tuyến tính khi E xấp xỉ với V^2.

Chúng ta vừa xem qua ba chương trình tìm cây bao trùm tối tiểu với các tính năng hoàn toàn khác nhau: phương pháp tìm kiếm theo độ ưu tiên, thuật toán Kruskal và thuật toán Prim. Đối với một số dạng đồ thị, thuật toán Prim gần như nhanh nhất, nhưng đối với các dạng khác thì có thể hai thuật toán trước lại hiệu quả hơn. Trong trường hợp xấu nhất, thuật toán tìm kiếm theo độ ưu tiên đòi hỏi thời gian là $(E+V)logV$, trong khi thuật toán Prim là V^2 và thuật toán Kruskal là $ElogE$. Thuật toán tìm kiếm theo độ ưu tiên và thuật toán Kruskal có thời gian xấp xỉ với E đối với các đồ thị thông thường trong thực tế, với các đồ thị thưa thì thuật toán tìm kiếm theo độ ưu tiên sẽ hiệu quả nhất, đối với các đồ thị dày thì thuật toán Prim có thời gian xấp xỉ E.

CÁC BÀI TOÁN HÌNH HỌC

Cho trước N điểm trong mặt phẳng, hãy tìm tập các đoạn thẳng nối tất cả các điểm với nhau sao cho tổng độ dài của chúng ngắn nhất. Bài toán hình học này được gọi là **bài toán cây bao trùm Euclid tối tiểu**. Bài toán có thể được giải bằng các thuật toán nói trên, nhưng rõ ràng các thông tin hình học thêm vào sẽ giúp chúng ta tìm ra thuật toán hiệu quả hơn nhiều.

Một phương pháp để giải bài toán Euclid nói trên là xây dựng một đồ thị đầy đủ có N đỉnh và $N(N-1)/2$ cạnh, mỗi cạnh nối một cặp đỉnh và có trọng số là khoảng cách giữa hai đỉnh đó. Cây bao trùm tối tiểu có thể tìm được với thời gian xấp xỉ N^2 nhờ chương trình *matrixpfs*.

Trong thực tế chúng ta có thể làm tốt hơn, cấu trúc hình cho phép chúng ta loại bỏ bớt hầu hết các cạnh trong đồ thị đầy đủ trước khi tìm cây bao trùm tối tiểu. Có thể chứng minh rằng cây bao trùm tối tiểu là tập con của đồ thị chỉ gồm các cạnh trong đối ngẫu của biểu đồ Voronoi (xem chương 28). Chúng ta biết rằng đồ thị này có số cạnh xấp xỉ N và có thể áp dụng thuật toán tìm kiếm theo độ ưu

tiên hay thuật toán Kruskal cho đồ thị thưa này. Do đó trước hết chúng ta tìm đối ngẫu Voronoi (thời gian xấp xỉ $NlogN$) và kế đến sử dụng thuật toán tìm kiếm theo độ ưu tiên hay thuật toán Kruskal để tìm ra cây bao trùm tối tiểu, thời gian phải trả tổng cộng sẽ xấp xỉ $NlogN$. Tuy nhiên, việc viết phương trình để tìm đối ngẫu Voronoi sẽ không dễ dàng (ngay cả đối với một lập trình viên có kinh nghiệm), vì thế cách tiếp cận này không có ích trong thực tế.

Một tiếp cận khác là lợi dụng tính phân bố các điểm để giới hạn số cạnh trong đồ thị giống như các ô lưới được dùng trong Chương 26 cho bài toán tìm theo khoảng. Nếu chúng ta chia mặt phẳng thành các ô vuông sao cho mỗi ô có thể chứa khoảng $lgN/2$ điểm, kế đến chỉ chọn các cạnh nối mỗi điểm tới các điểm trong các ô vuông lân cận và dùng thuật toán Kruskal hay thuật toán tìm kiếm theo độ ưu tiên.

Một ý tưởng thú vị là phân tích mối quan hệ giữa các đồ thị và các thuật toán hình học đã trình bày trong những phần trước. Chắc chắn nhiều bài toán có thể đưa về bài toán hình học hay bài toán đồ thị. Nếu vị trí của các đối tượng là yếu tố quan trọng thì các thuật toán hình học trong các chương 24-28 sẽ thích hợp, nhưng nếu mối quan hệ giữa chúng là yếu tố quan trọng thì các thuật toán đồ thị trong phần này sẽ tốt hơn. Bài toán cây bao trùm Euclid tối thiểu dường như nằm trung gian giữa hai cách tiếp cận này (dữ liệu nhập là dữ liệu hình học trong khi dữ liệu xuất lại là các mối nối các điểm).

Một bài toán tương tự là tìm đường đi ngắn nhất từ x tới y trong đồ thị gồm các điểm trong mặt phẳng và các cạnh là các đoạn thẳng nối chúng. Đồ thị mê cung chúng ta đã dùng là ví dụ cho loại đồ thị như thế. Lời giải của bài toán này rất đơn giản: áp dụng thuật toán tìm kiếm theo độ ưu tiên, đặt độ ưu tiên cho mỗi đỉnh bìa được gặp là khoảng cách trong cây từ x tới đỉnh bìa đó cộng với khoảng cách từ đỉnh bìa tới y. Chúng ta dừng khi y được thêm vào cây. Phương pháp này sẽ tìm đường đi ngắn nhất từ đỉnh x tới đỉnh y rất nhanh

bởi vì nó luôn đi "hướng về" y, trong khi các thuật toán đồ thị chuẩn phải "tìm kiếm" y. Việc di chuyển từ một góc đến một góc khác trong một mê cung lớn có lẽ cần phải kiểm tra một số đỉnh xấp xỉ căn bậc hai của V, trong khi các thuật toán đồ thị chuẩn phải kiểm tra gần như tất cả các đỉnh của đồ thị.

BÀI TẬP

1. Tìm một cây bao trùm tối tiểu khác của đồ thị trong ví dụ đầu chương này.
2. Hãy đưa ra thuật toán tìm rừng bao trùm tối tiểu của một đồ thị liên thông (nghĩa là mỗi đỉnh phải được nối bởi một cạnh nào đó, nhưng đồ thị có được không cần liên thông).
3. Có đồ thị nào V đỉnh và E cạnh mà cần thời gian xấp xỉ $(E+V)\log V$ khi tìm cây bao trùm tối tiểu bằng phương pháp tìm kiếm theo độ ưu tiên hay không?
4. Giả sử chúng ta cài đặt hàng đợi ưu tiên bằng một xâu có thứ tự giống như cài đặt thuật toán duyệt đồ thị tổng quát. Cho biết thời gian chạy trong trường hợp xấu nhất ? Khi nào thì thuật toán này không thích hợp ?
5. Tìm một phản ví dụ để cho thấy chiến lược "tham lam" sau đây sẽ không đúng khi giải bài toán cây bao trùm tối tiểu hay đường đi ngắn nhất: "trong mỗi bước, viếng đỉnh chưa được viếng gần đỉnh vừa viếng nhất".
6. Hãy vẽ các cây đường đi ngắn nhất cho mỗi đỉnh của đồ thị ví dụ.
7. Làm thế nào để tìm cây bao trùm tối tiểu của đồ thị "khổng lồ" (quá lớn đến nỗi không thể chứa nó trong bộ nhớ chính).
8. Viết chương trình để phát sinh các đồ thị liên thông ngẫu nhiên V đỉnh, kế đó tìm đường đi ngắn nhất và cây bao trùm tối tiểu. Hãy dùng trọng số ngẫu nhiên từ *1* đến V. So sánh trọng số các cây có được tùy theo các giá trị của V.
9. Viết chương trình để phát sinh đồ thị đầy đủ có trọng số ngẫu nhiên gồm V đỉnh bằng cách đặt các giá trị trong ma trận kề bởi các số ngẫu nhiên từ *1* đến V. Chạy chương trình để so sánh thuật toán tìm cây bao trùm tối tiểu của Kruskal và Prim khi $V=10, 25, 100$.
10. Cho một phản ví dụ để thấy thuật toán tìm cây bao trùm tối tiểu sau đây sẽ không đúng: "Sắp xếp các điểm theo hoành độ, tìm cây bao trùm tối tiểu cho một nửa đầu tiên và một nửa thứ hai, kế đến tìm cạnh ngắn nhất nối chúng với nhau".

32

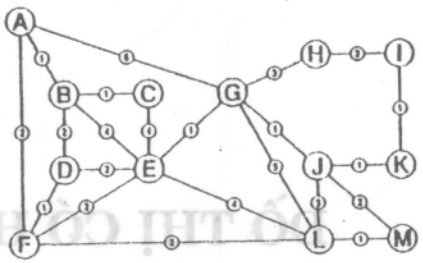
ĐỒ THỊ CÓ HƯỚNG

Đồ thị có hướng là đồ thị mà các cạnh nối các nút theo một chiều, điều này đưa đến nhiều khó khăn hơn trong việc xác định các tính chất của đồ thị. Việc xử lý các đồ thị dạng này tương tự như đi dạo trong một thành phố có nhiều đường một chiều hay đi du lịch bằng đường hàng không trong một quốc gia, trong tình huống này chúng ta cần phải đi từ một vị trí này đến vị trí khác trên đồ thị.

Thường thì hướng của cạnh sẽ phản ánh một ý nghĩa thực tế trong các ứng dụng đang được mô phỏng. Ví dụ, đồ thị có hướng có thể dùng để mô phỏng việc sản xuất theo dây chuyền: các nút của đồ thị là các công việc cần phải làm và một cạng nối từ nút x đến nút y có nghĩa là công việc x phải được làm xong trước công việc y. Phải tiến hành các công việc như thế nào để các quy định không bị vi phạm?

Trong chương này chúng ta sẽ quan sát việc tìm kiếm ưu tiên độ sâu trên các đồ thị có hướng, cũng như các thuật toán tìm bao đóng liên thông, các thuật toán sắp xếp tôpô, các thuật toán tìm các thành phần liên thông mạnh (quan tâm đến hướng của cạnh).

Như đã lưu ý trong Chương 29, việc biểu diễn các đồ thị có hướng chỉ là mở rộng (thật ra là thu hẹp) đơn giản của biểu diễn các đồ thị vô hướng. Trong biểu diễn xâu kề, mỗi cạnh xuất hiện chỉ một lần: cạnh nối x tới y được biểu diễn bởi một nút chứa y trong xâu liên kết



Hình 32.1 *Một đồ thị có hướng*

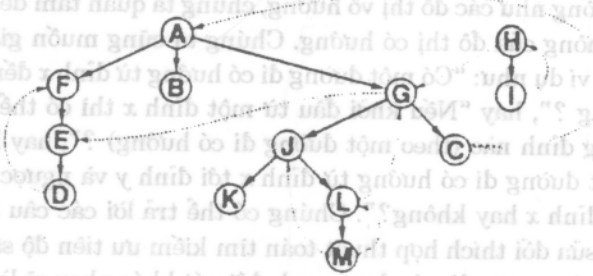
tương ứng với x. Trong biểu diễn ma trận kề, chúng ta phải duy trì một ma trận đầy đủ với kích thước VxV và đặt bit 1 trong dòng x cột y của ma trận nếu có một cạnh nối x tới y.

Hình 32.1 cho thấy một đồ thị có hướng tương tự với một đồ thị vô hướng mà chúng ta đã thấy trước đây. Đồ thị này gồm các cạnh AG AB CA LM JM JL JK ED DF HI FE AF GE GC HG GJ LG IH ML. Bây giờ thì thứ tự của đỉnh trong một cạnh sẽ rất quan trọng: ký hiệu AG mô tả một cạnh nối A tới G, và không phải nối G tới A. Nhưng cũng có thể có hai cạnh giữa hai nút, mỗi cạnh theo một hướng khác nhau (đồ thị trong Hình 32.1 có cả hai cạnh HI và IH, cũng như LM và ML).

Chú ý rằng đối với các biểu diễn này thì không có sự khác nhau giữa hai cạnh có hướng đối nhau trong đồ thị có hướng và mỗi cạnh trong đồ thị vô hướng. Do đó, một số thuật toán trong chương này có thể được xem như tổng quát hóa của các thuật toán trong các chương trước.

TÌM KIẾM ƯU TIÊN ĐỘ SÂU

Thuật toán tìm kiếm ưu tiên độ sâu của Chương 29 cũng hoạt động chính xác đối với các đồ thị có hướng. Thật ra các thao tác lại đơn



Hình 32.2 *Rừng tìm kiếm ưu tiên độ sâu cho một đồ thị có hướng*

giản hơn một ít so với các đồ thị vô hướng bởi vì chúng ta không cần quan tâm đến các cạnh kép giữa các nút trừ khi chúng được biểu thị tường minh trong đồ thị. Tuy nhiên các cây tìm kiếm sẽ có cấu trúc phức tạp hơn. Ví dụ, Hình 32.2 minh họa cấu trúc tìm kiếm ưu tiên độ sâu dùng để mô tả các thao tác của thuật toán đệ qui trong Chương 29 cho đồ thị mẫu của chúng ta. Giống như trước, đây là một phiên bản vẽ lại của đồ thị: các cạnh liền nét thì tương ứng với cạnh đã được dùng thực sự để viếng các đỉnh do các lệnh gọi đệ qui và các cạnh đứt nét tương ứng với các cạnh trỏ tới các đỉnh đã được viếng vào lúc gặp cạnh. Các nút được viếng theo thứ tự A F E D B G J K L M C H I.

Chú ý rằng hướng của cạnh làm cho rừng tìm kiếm ưu tiên độ sâu hoàn toàn khác với rừng tìm kiếm ưu tiên độ sâu cho các đồ thị vô hướng. Ví dụ, ngay cả đồ thị đồ thị gốc liên thông, cấu trúc tìm kiếm ưu tiên độ sâu được định nghĩa bởi các cạnh liền nét sẽ không liên thông: cấu trúc nầy là một rừng chứ không phải là một cây.

Đối với các đồ thị vô hướng, chúng ta chỉ có một loại cạnh đứt nét, loại cạnh nầy nối một đỉnh tới tổ tiên của nó. Đối với đồ thị có hướng, có ba loại cạnh đứt nét: loại hướng lên, nối một đỉnh tới tổ tiên của nó; loại hướng xuống, nối một đỉnh tới con cháu của nó; và loại vắt ngang, nối một đỉnh tới đỉnh khác không phải con cháu và cũng không phải là tổ tiên của nó.

Cũng giống như các đồ thị vô hướng, chúng ta quan tâm đến tính chất liên thông của đồ thị có hướng. Chúng ta cũng muốn giải đáp các câu hỏi ví dụ như: "Có một đường đi có hướng từ đỉnh x đến đỉnh y hay không ?", hay "Nếu khởi đầu từ một đỉnh x thì có thể đi tới được những đỉnh nào (theo một đường đi có hướng) ?", hay là "Có tồn tại một đường đi có hướng từ đỉnh x tới đỉnh y và ngược lại từ đỉnh y tới đỉnh x hay không?". Chúng có thể trả lời các câu hỏi đó bằng cách sửa đổi thích hợp thuật toán tìm kiếm ưu tiên độ sâu cho đồ thị vô hướng, mặc dù các dạng cạnh đứt nét khác nhau sẽ làm cho việc hiệu chỉnh có đôi điều phức tạp.

BAO ĐÓNG

Trong các đồ thị vô hướng, các đỉnh nằm trong cùng một thành phần liên thông có thể liên lạc được với nhau bằng cách duyệt theo cạnh. Đối với các đồ thị có hướng cũng tương tự, chúng ta thường quan tâm đến tập hợp các đỉnh có thể liên lạc với nhau bằng cách duyệt theo các cạnh có hướng trong đồ thị. Bài toán cho đồ thị có hướng sẽ phức tạp hơn tính liên thông đơn giản của đồ thị vô hướng.

Có thể dễ dàng chứng minh rằng thủ tục đệ quy *visit* về tìm kiếm ưu tiên độ sâu trong Chương 29 sẽ qua tất cả các nút có thể đi tới được từ nút khởi đầu. Do đó, nếu chúng ta sửa đổi thủ tục này để in ra tất cả những nút mà nó đang viếng (chẳng hạn bằng cách chèn thêm lệnh *write(name(k))*). Cũng cần phải chú ý rằng không nhất thiết là mỗi cây trong rừng tìm kiếm ưu tiên độ sâu chứa tất cả các nút mà có thể tới được nếu khởi đầu từ gốc của cây đó: trong ví dụ của chúng ta thì tất cả các nút đều có thể tới được nếu khởi đầu từ H hay I. Để có thể tới được tất cả những nút (có thể tới được) khi khởi đầu từ một nút, chúng ta chỉ cần gọi thủ tục *visit* V lần, một lần gọi cho mỗi nút:

```
for k:=1 to V do
   begin   id:=0;
           for j:=1 to V do val[j]:=0;
           visit(k);
           writeln
   end;
```

Chương trình này sẽ cho ra kết xuất sau đây khi áp dụng vào đồ thị có hướng trong Hình 32.2. *Tập hợp* các nút trên mỗi dòng phản ánh tính chất cấu trúc của chính bản thân đồ thị: mỗi dòng sẽ chứa những nút mà có thể đến được bằng một đường đi có hướng nếu khởi đầu từ nút đầu tiên trên dòng.

```
A F E D B G J K L M C
B
C A F E D B G J K L M
D F E
E D F
F E D
G J K L M C A F E D B
H G J K L M C A F E D B I
I H G J K L M C A F E D B
J K L G C A F E D B M
K
L G J K M C A F E D B
M L G J K C A F E D B
```

Đối với các đồ thị vô hướng, quá trình tính toán này sẽ cho ra một bảng trong đó mỗi dòng là tập hợp tất cả các nút trong cùng một thành phần liên thông. Dưới đây chúng ta sẽ nghiên cứu sự tổng quát hóa của tính chất liên thông trong đồ thị vô hướng.

Như thường lệ, đương nhiên là chúng ta muốn làm những công việc có ích hơn là chỉ liệt kê các đỉnh thành một bảng. Ví dụ chúng

ta muốn thêm vào một cạnh nối trực tiếp x tới y nếu tồn tại một đường đi có hướng từ x tới y. Đồ thị có được bằng cách thêm tất cả các cạnh vào một đồ thị có hướng theo qui tắc vừa nói sẽ được gọi là **bao đóng** của đồ thị đã cho. Nói chung thì sẽ cần thêm vào rất nhiều cạnh nên bao đóng sẽ là một đồ thị dày và do đó chúng ta nên dùng biểu diễn ma trận kề. Đây là một khái niệm tương tự với các thành phần liên thông trong đồ thị vô hướng; chúng ta chỉ cần tìm bao đóng một lần, sau nầy chúng ta có thể trả lời rất nhanh các câu hỏi tương tự như: "Có tồn tại một đường đi từ x tới y hay không?"

Tính chất 32.1 *Quá trình tìm kiếm ưu tiên độ sâu có thể được dùng để tính bao đóng của một đồ thị có hướng trong khoảng $O(V(E+V))$ bước đối với một đồ thị thưa, và trong khoảng $O(V^3)$ bước đối với đồ thị dày.*

Tính chất nầy được suy ra trực tiếp từ các tính chất cơ sở trong Chương 29: chúng ta thực hiện thủ tục tìm kiếm ưu tiên độ sâu lần lượt cho mỗi đỉnh trong V đỉnh của đồ thị. Kết quả tương tự cũng đúng đối với trường hợp tìm kiếm ưu tiên chiều rộng: như đã chú ý ở trên, thứ tự các đỉnh được viếng không thích hợp cho vấn đề cụ thể nầy.

Nếu dùng biểu diễn ma trận kề thì bài toán tìm bao đóng có thể giải quyết bằng một chương trình không đệ quy đơn giản nhưng rất đáng chú ý như sau:

```
for y:=1 to V do
  for x:=1 to V do
    if a[x,y] then
      for j:=1 to V do
        if a[y,j] then a[x,j]:=true;
```

S. Warshall đã khám phá ra phương pháp nầy vào năm 1962 bằng cách dựa vào quan sát đơn giản như sau: "nếu có một đường đi từ

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	0	0	0	1	1	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	0	1	0	0	0	0	0	0	0	0	0	0
D	0	0	0	1	0	1	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0	0	0	0
F	0	0	0	0	1	1	0	0	0	0	0	0	0
G	0	0	1	0	1	0	1	0	0	1	0	0	0
H	0	0	0	0	0	0	1	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	0	0	0	0	0	0	1	0	0	0	0	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	0	0	0	1	1	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	0	0	1	1	0	0	0	0	0	0
D	0	0	0	1	0	1	0	0	0	0	0	0	0
E	0	0	0	1	1	0	0	0	0	0	0	0	0
F	0	0	0	0	1	1	0	0	0	0	0	0	0
G	0	0	1	0	1	0	1	0	0	1	0	0	0
H	0	0	0	0	0	0	1	1	1	0	0	0	0
I	0	0	0	0	0	0	0	1	1	0	0	0	0
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	0	0	0	0	0	0	1	0	0	0	0	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

Hình 32.3 *Các trạng thái khởi động của thuật toán Warshall*

nút x đến nút y và một đường đi từ nút y tới nút j thì sẽ có một đường đi từ nút x tới nút j". Thủ thuật là chú ý quan sát kỹ hơn để cho quá trình xử lý chỉ duyệt qua ma trận một lần, sự sáng suốt ở chỗ: "Nếu có một đường đi từ nút x tới nút y chỉ đi qua các nút với chỉ số nhỏ hơn y và có một đường đi từ nút y tới nút j thì sẽ có một đường đi từ nút x tới nút j mà chỉ đi qua các nút có chỉ số nhỏ hơn $y+1$." Chương trình vừa nói trên là một cài đặt trực tiếp của phương pháp này.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	1	1	1	1	0	0	1	1	1	1
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	0	0	1	1	1	1
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	0	0	0	0	0	0	0
F	0	0	0	1	1	1	0	0	0	0	0	0	0
G	1	1	1	1	1	1	1	0	0	1	1	1	1
H	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1
J	0	0	0	0	0	0	0	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	1	1	1	1	1	1	1	0	0	1	1	1	1
M	0	0	0	0	0	0	0	0	0	0	0	1	1

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	1	1	1	1	1	1	1	0	0	1	1	1	1
B	0	1	0	0	0	0	0	0	0	0	0	0	0
C	1	1	1	1	1	1	1	0	0	1	1	1	1
D	0	0	0	1	1	1	0	0	0	0	0	0	0
E	0	0	0	1	1	1	0	0	0	0	0	0	0
F	0	0	0	1	1	1	0	0	0	0	0	0	0
G	1	1	1	1	1	1	1	0	0	1	1	1	1
H	1	1	1	1	1	1	1	1	1	1	1	1	1
I	1	1	1	1	1	1	1	1	1	1	1	1	1
J	1	1	1	1	1	1	1	0	0	1	1	1	1
K	0	0	0	0	0	0	0	0	0	0	1	0	0
L	1	1	1	1	1	1	1	0	0	1	1	1	1
M	1	1	1	1	1	1	1	0	0	1	1	1	1

Hình 32.4 *Các trạng thái cuối cùng của thuật toán Warshall*

Phương pháp Warshall chuyển ma trận kề của một đồ thị thành ma trận kề của bao đóng của đồ thị. Một phương pháp để theo dõi thuật toán là có thể xem thuật toán xử lý mỗi lần một dòng của ma trận. Xử lý cho cột *y* là thay thế mỗi dòng có giá trị 1 trong cột *y* bởi chính dòng đó *or* luận lý với dòng *y*. Hình 32.3 cho thấy ma trận khởi động của ví dụ mẫu và trạng thái của ma trận sau khi hai cột đầu tiên và một nửa cột thứ ba được xử lý: đến lúc này chỉ có dòng *C* bị ảnh hưởng. Hình 32.4 minh họa trạng thái ma trận trước khi một vài cột cuối cùng được xử lý và trạng thái cuối cùng (tức là ma trận của bao đóng).

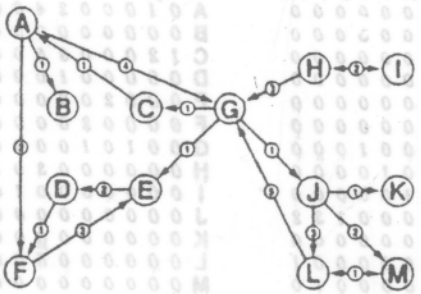
Tính chất 32.2 *Thuật toán Warshall tìm được bao đóng sau* $O(V^3)$ *bước.*

Nếu ma trận khởi đầu hoàn toàn là 1 thì điều nầy hiển nhiên vì chúng ta có ba vòng lặp lồng nhau. Nhưng ngay cả các đồ thị thưa cũng có thể được đưa nhanh về trường hợp nầy - ví dụ nếu nút đầu tiên được nối tới mỗi nút khác thì ma trận sẽ được lấp đầy bởi các bit 1 trước khi *y* nhận giá trị 2.

Với các đồ thị lớn, quá trình nầy có thể được tổ chức sao cho các phép toán trên bit có thể được làm mỗi lần một từ của máy tính, do đó sẽ đưa đến tiết kiệm đáng kể trong nhiều môi trường.

TÌM TẤT CẢ CÁC ĐƯỜNG ĐI NGẮN NHẤT

Với mọi cặp đỉnh *x*, *y*, bao đóng của đồ thị không trọng lượng (có hướng hay vô hướng) trả lời câu hỏi "Có tồn tại đường đi từ *x* tới *y* hay không ?". Với các đồ thị có trọng (có hướng hay vô hướng), có thể người ta cần xây dựng một bảng cho phép tìm ra đường đi ngắn nhất từ *x* đến *y* cho mỗi cặp đỉnh của đồ thị. Đây chính là **bài toán đường đi ngắn nhất cho tất cả các cặp đỉnh**. Ví dụ, trong đồ thị có trọng của hình 32.5, chúng ta muốn biết rằng đường đi ngắn nhất từ M tới K có chiều dài 6, đường đi ngắn nhất từ J tới F có chiều dài



Hình 32.5 *Đồ thị có hướng và có trọng*

10 ... mà chỉ cần tham khảo bảng đã được xây dựng.

Thuật toán tìm đường đi ngắn nhất trong chương trước tìm ra đường đi từ một đỉnh tới mỗi đỉnh khác của đồ thị, vì vậy chúng ta chỉ cần chạy thuật toán đó V lần. Điều nầy đưa tới một thuật toán chạy trong $O(EV+V^2logV)$ bước. Nhưng chúng ta cũng có thể dùng phương pháp của R. W. Floyd giống như phương pháp của Warshall như sau:

```
for y:=1 to V do
  for x:=1 to V do
    if a[x,y]>0 then
      for j:=1 to V do
        if a[y,j]>0 then
          if (a[x,j]=0) or (a[x,y]+a[y,j]<a[x,j])
            then a[x,j]:=a[x,y]+a[y,j];
```

Cấu trúc của thuật toán thì gần như y hệt với phương pháp Warshall, nhưng thay vì dùng phép logic *"or"* để theo dõi các đường đi, chúng ta thêm một ít tính toán cho mỗi cạnh để xác định xem nó có là bộ phận của đường đi ngắn nhất mới hay không: "đường đi

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	3	0	0	0
C	1	0	0	0	0	0	0	0	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	1	0	1	0	0	0	0	1	0	0	0
H	0	0	0	0	0	0	3	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	3	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	5	0	0	0	0	0	1
M	0	0	0	0	0	0	0	0	0	0	1	0	0

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	0	1	0	0	0	2	4	0	0	0	0	0	0
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	0	0	0	0	3	6	0	0	0	0	0
D	0	0	0	0	0	1	0	0	0	0	0	0	0
E	0	0	0	2	0	0	0	0	0	0	0	0	0
F	0	0	0	0	2	0	0	0	0	0	0	0	0
G	0	0	1	0	1	0	0	0	0	1	0	0	0
H	0	0	0	0	0	0	3	0	1	0	0	0	0
I	0	0	0	0	0	0	0	1	0	0	0	0	0
J	0	0	0	0	0	0	0	0	0	0	1	3	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	5	0	0	0	0	0	1
M	0	0	0	0	0	0	0	0	0	0	1	0	0

Hình 32.6 *Trạng thái khởi động của thuật toán Floyd*

ngắn nhất từ nút x tới nút j chỉ đi qua các nút có chỉ số nhỏ hơn $y+1$ thì hoặc là đường đi ngắn nhất từ nút x tới nút j chỉ đi qua các nút có chỉ số nhỏ hơn y hoặc nếu ngắn hơn thì nó là đường đi ngắn nhất từ x tới y cộng với đường đi ngắn nhất từ y tới j." Như thường lệ, phần tử *0* trong ma trận tương ứng không có cạnh nối hai đỉnh, chương trình có thể cải tiến đơn giản hơn (bỏ đi các phép so sánh với *0*) bằng cách dùng một biến "lính canh" để ký hiệu cạnh có trọng số vô hạn.

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	6	1	5	6	4	2	4	0	0	5	6	8	7
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	6	7	5	3	5	0	0	6	7	9	8
D	0	0	0	5	3	1	0	0	0	0	0	0	0
E	0	0	0	2	5	3	0	0	0	0	0	0	0
F	0	0	0	4	2	5	0	0	0	0	0	0	0
G	2	3	1	3	1	4	6	0	0	1	2	4	3
H	5	6	4	6	4	7	3	2	1	4	5	7	6
I	6	7	5	7	5	8	4	1	2	5	6	8	7
J	0	0	0	0	0	0	0	0	0	1	3	2	
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	7	8	6	8	6	9	5	0	0	6	7	9	1
M	0	0	0	0	0	0	0	0	0	0	1	0	

	A	B	C	D	E	F	G	H	I	J	K	L	M
A	6	1	5	6	4	2	4	0	0	5	6	8	7
B	0	0	0	0	0	0	0	0	0	0	0	0	0
C	1	2	6	7	5	3	5	0	0	6	7	9	8
D	0	0	0	5	3	1	0	0	0	0	0	0	0
E	0	0	0	2	5	3	0	0	0	0	0	0	0
F	0	0	0	4	2	5	0	0	0	0	0	0	0
G	2	3	1	3	1	4	6	0	0	1	2	4	3
H	5	6	4	6	4	7	3	2	1	4	5	7	6
I	6	7	5	7	5	8	4	1	2	5	6	8	7
J	10	11	9	11	9	12	8	9	0	0	5	7	2
K	0	0	0	0	0	0	0	0	0	0	0	0	0
L	7	8	6	8	6	9	5	0	0	6	7	2	1
M	8	9	7	9	7	10	6	0	0	7	8	1	2

Hình 32.7 *Trạng thái cuối cùng của thuật toán Floyd*

Tính chất 32.3 *Thuật toán Floyd cần $O(V^3)$ bước để giải bài toán đường đi ngắn nhất của cho mỗi cặp đỉnh.*

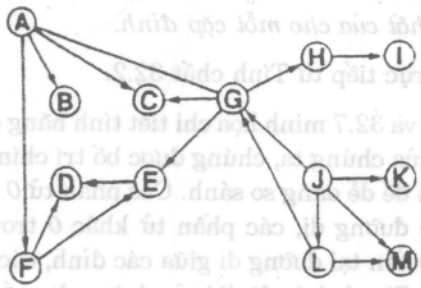
Điều này suy ra trực tiếp từ Tính chất 32.2.

Các Hình 32.6 và 32.7 minh họa chi tiết tính năng của thuật toán Floyd trên ví dụ của chúng ta, chúng được bố trí chính xác như các Hình 32.3 và 32.4 để dễ dàng so sánh. Các phần tử *0* tương ứng với ý nghĩa không có đường đi, các phần tử khác *0* trong thuật toán Warshall cho thấy tồn tại đường đi giữa các đỉnh, các phần tử khác *0* trong thuật toán Floyd chứa độ dài của đường đi ngắn nhất. Đường đi ngắn nhất cũng có thể được tìm ra bằng cách dùng phiên bản ma trận của mảng *dad* trong các chương trước: cho phần tử trong dòng *x* cột *j* nhận tên của đỉnh trước trong đường đi ngắn nhất từ *x* tới *j* (trong đoạn chương trình trên chính là đỉnh *y* ở vòng lặp trong).

SẮP XẾP TÔPÔ

Các đồ thị có chu trình xuất hiện nhiều trong các ứng dụng cần đến đồ thị có hướng. Tuy nhiên, ví dụ như nếu đồ thị trong Hình 32.1 mô hình một dây chuyền sản xuất thì ta thấy rằng công việc A phải được làm trước công việc G, mà G thì phải được làm trước C, còn C lại phải làm trước A. Rõ ràng đây là một tình huống không nhất quán, không phù hợp với thực tế. Một số ứng dụng sẽ cần các đồ thị không có chu trình có hướng, các đồ thị như thế được gọi tắt là *DAG* (*directed acyclic graph*). Một dag cũng có khả năng có chu trình nếu chúng ta không kể đến hướng của các cạnh. Hình 32.8 cho thấy một *DAG* tương tự với đồ thị có hướng trong Hình 32.1, trong đó một số cạnh bị xóa bớt hay đổi hướng. Danh sách các cạnh của đồ thị này giống như đồ thị liên thông trong Chương 30, nhưng ở đây thứ tự đỉnh trong mỗi cạnh có khác một ít.

Các *DAG* hoàn toàn khác với các đồ thị có hướng tổng quát: chúng là cây từng phần, do đó chúng ta có thể lợi dụng cấu trúc đặc



Hình 32.8 *Đồ thị không có chu trình có hướng (DAG)*

biệt này khi xử lý chúng. Khi nhìn từ một đỉnh tùy ý, một *DAG* trông giống như một cây; nói cách khác, rừng tìm kiếm ưu tiên độ sâu của một *DAG* không có các cạnh hướng lên. Hình 32.9 cho ta thấy rừng tìm kiếm ưu tiên độ sâu mô tả các thao tác của thủ tục dfs khi áp dụng vào *DAG* trong Hình 32.8.

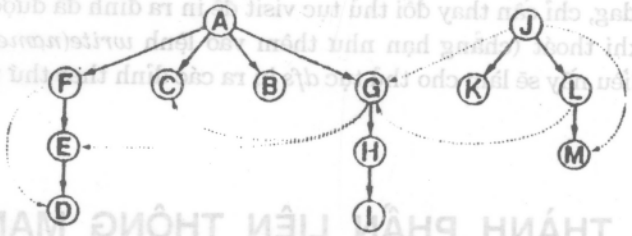
Thao tác cơ bản trên *DAG* là xử lý các đỉnh của đồ thị sao cho không có đỉnh nào của đồ thị được xử lý trước đỉnh chỉ tới nó. Ví dụ, các nút trong đồ thị trên có thể được xử lý theo thứ tự:

J K L M A G H I F E D B C

Nếu các cạnh được vẽ theo thứ tự đỉnh như thế này thì các cạnh luôn đi từ trái tới phải. Như đã chú ý ở trên, điều này hiển nhiên có ứng dụng trong thực tế, ví dụ trong các đồ thị biểu diễn xử lý trong sản xuất. Thao tác xếp thứ tự các đỉnh như trên được gọi là sắp xếp tôpô.

Nói chung, thứ tự các đỉnh có được từ quá trình sắp xếp tôpô sẽ không duy nhất. Chẳng hạn thứ tự sau đây cũng là thứ tự tôpô hợp lệ đối với ví dụ của chúng ta (đương nhiên có thể có nhiều thứ tự khác)

A J G F K L E M B H C I D.



Hình 32.9 *Rừng tìm kiếm ưu tiên độ sâu trong một DAG*

Trong các ứng dụng vào dây chuyền sản xuất, hoàn cảnh này xuất hiện khi một công việc không phụ thuộc trực tiếp hay gián tiếp vào công việc khác và do đó có nhiều cách thức để tiến hành công việc.

Đôi khi sẽ rất có ích nếu chúng ta diễn dịch các cạnh của đồ thị theo cách khác: chẳng hạn cạnh nối x tới y nghĩa là đỉnh x "phụ thuộc" vào đỉnh y. Ví dụ, các đỉnh của đồ thị biểu diễn các khái niệm được định nghĩa trong một tài liệu hướng dẫn của một ngôn ngữ lập trình (hay một quyển sách về thuật toán !), trong đó một cạnh nối từ x tới y nếu định nghĩa của x sử dụng y. Trong trường hợp này sẽ có ích lợi nếu tìm được một thứ tự với tính chất mỗi khái niệm được định nghĩa trước khi nó được dùng trong một định nghĩa khác. Điều này tương ứng với việc quy định vị trí các đỉnh trên một dòng sao cho các cạnh luôn đi từ phải tới trái. Một thứ tự tôpô đảo cho đồ thị trong ví dụ là:

D E F C B I H G A K M L J

Sự khác nhau (giữa hai thứ tự) sẽ không quan trọng, bởi vì việc sắp xếp thứ tự tôpô đảo tương đương với việc sắp xếp thứ tự tôpô trong đồ thị bằng cách đổi chiều tất cả các cạnh.

Chúng ta cũng đã thấy một thuật toán sắp thứ tự tôpô đảo: thủ tục tìm kiếm đệ quy ưu tiên độ sâu trong Chương 29. Khi đồ thị nhập là một dag, chỉ cần thay đổi thủ tục visit để in ra đỉnh đã được viếng trước khi thoát (chẳng hạn như thêm vào lệnh *write(name[k])*) ở cuối), điều này sẽ làm cho thủ tục *dfs* in ra các đỉnh theo thứ tự tôpô đảo.

CÁC THÀNH PHẦN LIÊN THÔNG MẠNH

Nếu đồ thị chứa một chu trình có hướng (nghĩa là chúng ta có thể khởi đầu từ một đỉnh và đi trở về chính nó bằng cách theo đi theo đúng hướng của các cạnh) thì nó không phải là một *DAG* và không thể xếp các đỉnh theo thứ tự tôpô. Trường hợp này chúng ta sẽ quan tâm đến tính chất liên thông mạnh, nghĩa là có thể đi từ mỗi trong đồ thị đến một đỉnh bất kỳ khác theo một đường đi có hướng. Đối với đồ thị không liên thông mạnh thì các nút trong cùng một thành phần liên thông mạnh sẽ đi đến được nút khác. Các thành phần liên thông mạnh của đồ thị có hướng trong Hình 32.1 gồm hai nút đơn lẻ B, K, một cặp liên thông H I, một bộ ba liên thông D E F, một bộ sáu liên thông A C G J L M. Ví dụ, đỉnh A và F không nằm trong cùng một thành phần liên thông bởi vì có một đường đi từ A đến F nhưng không có đường nào đi từ F đến A.

Có thể tìm được các thành phần liên thông mạnh bằng cách thay đổi thuật toán tìm kiếm ưu tiên độ sâu, phương pháp mà chúng ta sẽ trình bày đã được R. E. Tarjan khám phá năm 1972. Bởi vì thuật toán dựa trên phương pháp tìm kiếm ưu tiên độ sâu, thời gian hoạt động của nó xấp xỉ $V+E$, nhưng nó thực sự là một phương pháp rất thông minh. Chỉ cần sửa đổi một ít trong thủ tục *visit* cơ sở của chúng ta thì có ngay thuật toán này, tuy nhiên trước khi Tarjan tìm ra thuật toán thì không có một thuật toán thời gian tuyến tính nào cho bài toán này mặc dù có rất nhiều người nghiên cứu nó.

Tương tự với chương trình mà chúng ta đã nghiên cứu trong Chương 30 khi tìm các thành phần đông liên thông, cũng có một phiên bản hiệu chỉnh của phương pháp tìm kiếm ưu tiên độ sâu để tìm các thành phần liên thông của đồ thị. Hàm đệ quy *visit* được cho dưới đây dùng biến *min* để tìm đỉnh cao nhất mà có thể tới được từ một cháu chắt bất kỳ của đỉnh *k*, nhưng giá trị của min được sử dụng khác một ít khi in ra các thành phần liên thông mạnh:

```
function visit(k:integer):integer;
  var t:link; m, min: integer;
  begin id:=id+1; val[k]:=id; stack[p]:=k; p:=p+1; t:=adj[k];
        while t<>z do
        begin   if val[t↑.v]=0
                then m:=visit(t↑.v)
                else m:=val[t↑.v];
                t:=t↑.next
        end;
        if min=val[k] then
        begin   repeat  p:=p-1; write(name(stack[p]));
                        val[stack[p]]:=V+1;
                until stack[p]=k;
                writeln
        end;
        visit:=min;
  end;
```

Chương trình này đặt các đỉnh vào ngăn xếp khi vào thủ tục *visit*, kế đến sẽ lấy chúng khỏi ngăn xếp và in ra khi viếng xong phần tử cuối cùng của mỗi thành phần liên thông mạnh. Cuối cùng nếu *min=val[k]* thì tất cả các đỉnh đã gặp (ngoại trừ những đỉnh đã được in) thuộc vào cùng một thành phần liên thông mạnh với *k*. Chương trình trên có thể được sửa đổi dễ dàng để làm nhiều điều thú vị hơn là chỉ đơn giản in ra các thành phần liên thông.

Tính chất 32.4 *Có thể tìm được các thành phần liên thông mạnh của đồ thị trong khoảng thời gian tuyến tính.*

Chúng ta bỏ qua chứng minh của tính chất này.

BÀI TẬP

1. Hãy cho biết ma trận kề của bao đóng của dag trong Hình 32.8.

2. Nếu áp dụng thuật toán tìm bao đóng vào một đồ thị vô hướng biểu diễn bằng ma trận kề thì kết quả như thế nào ?

3. Hãy viết một chương trình để tìm số cạnh của bao đóng của một đồ thị có hướng bằng cách dùng biểu diễn xâu kề.

4. Hãy so sánh thuật toán Warshall với thuật toán tìm bao đóng bằng cách dùng kỹ thuật tìm kiếm ưu tiên độ sâu nhưng sử dụng dạng ma trận kề cho thủ tục *visit* và khử bỏ đệ quy.

5. Hãy cho biết thứ tự tôpô có được đối với *DAG* trong Hình 32.8 khi dùng phương pháp đã nói với biểu diễn ma trận kề nhưng thủ tục *dfs* quét qua các đỉnh theo thứ tự đảo ngược (từ V xuống 1) trong quá trình tìm kiếm các đỉnh chưa được viếng.

6. Thuật toán tìm đường đi ngắn nhất trong Chương 31 có đúng đối với đồ thị có hướng hay không ? Giải thích tại sao, và cho một ví dụ nếu nó sai.

7. Viết một chương trình để kiểm tra xem một đồ thị đã cho có là một *DAG* hay không.

8. Có bao nhiêu thành phần liên thông mạnh trong một *DAG* ? Có bao nhiêu thành phần liên thông mạnh trong một đồ thị có một chu trình có hướng kích thước V ?

9. Sử dụng chương trình trong Chương 29 và Chương 30 để tìm ra các đồ thị có hướng ngẫu nhiên với V đỉnh. Có bao nhiêu thành phần liên thông mạnh trong các đồ thị như thế ?

10. Viết một chương trình có chức năng tương tự như thủ tục *find* trong Chương 30, nhưng chú ý các thành phần liên thông mạnh của đồ thị có hướng (bài tập này không dễ, chắc chắn bạn sẽ không thể có được một chương trình hiệu quả như thủ tục *find*.)

33

DÒNG CHẢY TRONG MẠNG LƯỚI

Các đồ thị có hướng và có trọng số là các mô hình hữu dụng cho các ứng dụng cần đến sự di chuyển trong một mạng lưới. Một ví dụ là mạng lưới gồm các ống dẫn dầu với nhiều kích cỡ khác nhau được nối lại rất đa dạng với các công tắc điều khiển hướng đi của dầu trong ống tại các điểm nối. Giả sử mạng lưới có một nơi bắt nguồn (chẳng hạn bể chứa dầu) và một nơi kết thúc (đủ lớn) cho tất cả các ống dẫn nối tới chúng. Làm thế nào để lượng dầu chuyển được từ nơi bắt nguồn tới nơi nhận là nhiều nhất ? Đây là bài toán **dòng chảy trong mạng lưới**, một bài toán không tầm thường.

Một ví dụ khác cho bài toán này là luồng lưu thông trên xa lộ và hàng hoá cần được di chuyển qua các nhà máy... Nhiều phiên bản của bài toán này đã được nghiên cứu tương ứng với nhiều tình huống thực tế khác nhau. Rõ ràng là chúng ta cần phải lưu tâm nhiều để tìm một thuật toán hiệu quả cho bài toán này. Đây là bài toán nằm trung gian giữa khoa học máy tính và lĩnh vực nghiên cứu ứng dụng. Các chuyên viên nghiên cứu ứng dụng thường lưu tâm tới mô hình toán học của các hệ thống phức tạp nhằm mục đích đưa ra những quyết định có hiệu quả trong thực tế. Dòng chảy trong mạng lưới là một ví dụ về bài toán nghiên cứu ứng dụng, chúng ta sẽ tìm hiểu sơ lược về một số bài toán nghiên cứu ứng dụng khác trong các Chương

42-43.

Trong Chương 43 chúng ta sẽ nghiên cứu về **quy hoạch tuyến tính**, đây là một tiếp cận tổng quát để giải các phương trình toán học phức tạp có được từ các mô hình nghiên cứu ứng dụng. Với các bài toán cụ thể, chẳng hạn bài toán dòng chảy trong mạng lưới đã có các lời giải cổ điển giống như các thuật toán đồ thị mà chúng ta đã khảo cứu và có thể cài đặt rất dễ dàng. Tuy nhiên không giống với các bài toán khác, đây là bài toán vẫn còn đang được nghiên cứu, lời giải "tốt nhất" vẫn chưa được tìm thấy và nhiều thuật toán mới vẫn còn đang được khám phá.

BÀI TOÁN DÒNG CHẢY TRONG MẠNG LƯỚI

Hãy quan sát hình vẽ lý tưởng của một mạng lưới nhỏ gồm các ống dẫn dầu trong Hình 33.1. Các ống có khả năng chứa xấp xỉ với kích thước của chúng và dầu chỉ có thể chảy theo hướng từ đỉnh tới đáy. Hơn nữa, các công tắc ở mỗi mối nối sẽ điều khiển lượng dầu chảy theo mỗi hướng. Hệ thống sẽ đạt được trạng thái cân bằng khi lượng dầu chảy vào đỉnh bằng với lượng dầu chảy ra từ đáy (chúng ta muốn đại lượng này lớn nhất) và lượng dầu đi vào mỗi mối nối bằng với lượng dầu đi ra khỏi mối nối đó. Chúng ta đo luồng dầu và sức chứa của các ống bằng một đơn vị nguyên (chẳng hạn gallon/s).

Hình 33.1 cho thấy việc sử dụng các công tắc tại những mối nối có thể ảnh hưởng tới luồng dầu cực đại. Trước tiên ta giả sử rằng công tắc điều khiển đường ống AB được mở, sau đó là BD và DF như trong sơ đồ bên trái của hình. Kế đến giả sử ống AC được mở và chỉnh công tắc C để đóng CD và mở CE (có thể người trực tại công tắc D đã thông báo cho người trực tại công tắc C rằng anh ta không thể lấy thêm dầu nữa bởi vì đã lấy từ B). Kết quả được minh họa

trong biểu đồ giữa của Hình: các ống BD và CE đã đầy. Bây giờ dòng chảy có thể được gia tăng bằng cách đưa đủ dầu qua đường ACDF để lấp đầy ống DF, nhưng chúng ta lại có một cách giải quyết tốt hơn như trong sơ đồ thứ ba. Bằng cách thay đổi công tắc tại B để đổi hướng một lượng dầu thích hợp theo ống BE, chúng ta mở ống DF thích hợp để công tắc C đưa dầu đầy vào ống CD đã mở sẵn. Tổng số dầu vào và ra trong mạng lưới đã được gia tăng bằng cách điều chỉnh các công tắc một cách thích hợp.

Việc cần làm của chúng ta là phát triển một thuật toán để điều chỉnh các công tắc. Hơn nữa, chúng ta muốn bảo đảm rằng không có một cách điều chỉnh khác tốt hơn cách được đưa ra.

Tình huống này có thể được mô phỏng dễ dàng bằng một đồ thị có hướng, từ đó có thể dùng các chương trình mà chúng ta đã nghiên cứu. Chúng ta định nghĩa mạng lưới là một đồ thị có hướng có trọng và có hai đỉnh đặc biệt: một đỉnh không có cạnh nào hướng tới nó (đỉnh nguồn) và một đỉnh không có cạnh nào hướng ra ngoài nó (đỉnh đích). Trọng lượng của các cạnh là các số nguyên không âm gọi là **sức chứa của cạnh**. Kế đến chúng ta định nghĩa **dòng chảy**

Hình 33.1 *Dòng chảy cực đại trong một mạng lưới đơn giản*

là một tập hợp khác tương ứng với các cạnh sao cho dòng chảy trên mỗi cạnh nhỏ hơn hay bằng sức chứa, và dòng chảy vào mỗi đỉnh bằng với dòng chảy đi ra khỏi đỉnh đó. **Giá trị** của dòng chảy được định nghĩa là dòng chảy đi vào đỉnh nguồn (hay ra khỏi đỉnh đích). Bài toán dòng chảy trong mạng lưới là hãy tìm ra giá trị dòng chảy lớn nhất đối với mỗi mạng lưới đã cho.

Hiển nhiên có thể biểu diễn mạng lưới bằng ma trận kề hay các xâu kề mà chúng ta đã dùng trong các chương trước. Thay vì một chúng ta sẽ dùng hai trọng số cho mỗi cạnh gồm sức chứa và giá trị luồng chảy qua cạnh. Hai trọng số này có thể biểu diễn bằng hai trường trong mỗi nút của xâu kề, hay dùng hai ma trận trong biểu diễn ma trận kề, hay hai trường trong mỗi bản ghi của ma trận kề. Mặc dù mạng lưới là một đồ thị có hướng, các thuật toán mà chúng ta khảo sát cũng cần phải duyệt cạnh theo một hướng "ngược", vì vậy chúng ta dùng biểu diễn của đồ thị vô hướng: nếu có một cạnh nối x tới y với kích thước (sức chứa) s và dòng chảy f, chúng ta lưu một cạnh từ y tới x với kích thước -s và dòng chảy -f. Trong biểu diễn xâu kề, cần thiết phải duy trì các liên kết nối hai nút trong danh sách nút biểu diễn mỗi cạnh, sao cho nếu chúng ta thay đổi giá trị trong dòng chảy một cạnh thì có thể cập nhật được giá trị nầy trong cạnh khác.

PHƯƠNG PHÁP FORD-FULKERSON

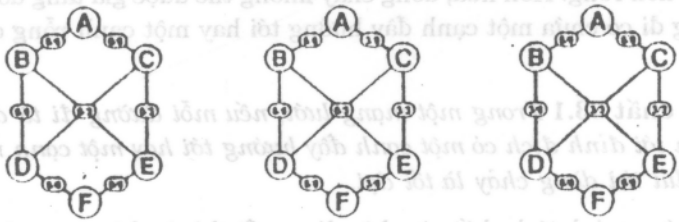
Tiếp cận cổ điển cho bài toán dòng chảy trong mạng lưới được xây dựng bởi L. R. Ford và D. R. Fulkerson vào năm 1962. Họ cung cấp một phương pháp để cải tiến bất kỳ một dòng chảy hợp lệ nào (dĩ nhiên là ngoại trừ dòng chảy tối đa). Bắt đầu với một dòng chảy 0, chúng ta sẽ lập đi lặp lại phương pháp này. Sau khi phương pháp được áp dụng nhiều lần, nó cho ra một dòng chảy mạnh hơn, nếu không thể cải tiến tiếp thì tìm thấy được dòng chảy tối đa. Thật ra dòng chảy trong Hình 33.1 có được nhờ áp dụng phương pháp này;

bây giờ chúng ta sẽ thí nghiệm phương pháp này cho đồ thì trong Hình 32.2.

Để đơn giản, chúng ta bỏ qua các mũi tên, bởi vì tất cả chúng đều hướng xuống dưới. Phương pháp chúng ta áp dụng đương nhiên không chỉ giới hạn vào các đồ thị có thể vẽ sao cho các cạnh chỉ theo cùng một hướng. Hiện tại chúng ta dùng các đồ thị như thế để dễ dàng hình dung dòng chảy trong mạng lưới dưới dạng dung dịch chảy trong các ống.

Hãy xem một đường đi có hướng bất kỳ xuyên qua mạng lưới từ nguồn tới đích. Rõ ràng dòng chảy có thể được gia tăng thêm ít nhất là một lượng bằng với sức chứa chưa dùng trên mỗi cạnh của đường đi. Trong sơ đồ bên trái của Hình 33.2, cách thức này được áp dụng theo đường đi ABDF; kế đến là sơ đồ giữa, nó được áp dụng dọc theo đường đi ACEF.

Như đã chú ý trong phần trên, kế đến chúng ta có thể áp dụng cách thức vừa nói cho đường ACDF và tạo ra tình huống mà tất cả các đường đi có hướng trong mạng lưới đều có ít nhất một cạnh được lấp đầy bởi sức chứa. Nhưng cũng có một phương pháp khác để gia



Hình 33.2 *Dòng chảy cực đại trong một mạng lưới*

tăng dòng chảy: chúng ta có thể xét các đường đi tùy ý xuyên qua mạng lưới mà có thể chứa các cạnh chỉ theo hướng "sai" (từ đích hướng đến nguồn dọc theo đường đi). Dòng chảy có thể được gia tăng dọc theo một con đường như thế bằng cách tăng dòng chảy trên mỗi cạnh từ nguồn hướng tới đích và giảm dòng chảy trên các cạnh từ đích hướng tới nguồn với cùng một lượng đã tăng. Trong ví dụ của chúng ta dòng chảy xuyên qua mạng lưới có thể gia tăng 3 đơn vị dọc theo đường ACDBEF như trong sơ đồ thứ ba của Hình 33.2. Điều nầy tương ứng với việc thêm 3 đơn vị vào các dòng chảy qua AC và CD, kế đến là làm lệch đi 3 đơn vị ở công tắc B từ BD tới BE và EF. Chúng ta không bị mất bớt dòng chảy qua DF bởi vì 3 đơn vị đến từ BD sẽ đến từ CD.

Để đơn giản từ ngữ, chúng ta sẽ gọi các cạnh mà dòng chảy xuôi từ nguồn hướng tới đích (dọc theo một đường đi) là các cạnh hướng tới và các cạnh mà dòng chảy ngược lại thì được gọi là các cạnh quay lui. Chú ý rằng dòng chảy có thể gia tăng với một lượng bị giới hạn bởi đại lượng cực tiểu của các phần không dùng trong các cạnh hướng tới và đại lượng cực tiểu của của các cạnh quay lui. Nói cách khác, trong dòng chảy mới sẽ có ít nhất một cạnh hướng tới dọc theo đường đi trở nên đầy hay ít nhất một cạnh quay lui dọc theo đường đi trở nên rỗng. Hơn nữa, dòng chảy không thể được gia tăng đối với đường đi có chứa một cạnh đầy hướng tới hay một cạnh rỗng quay lui.

Tính chất 33.1 *Trong một mạng lưới, nếu mỗi đường đi từ đỉnh nguồn tới đỉnh đích có một cạnh đầy hướng tới hay một cạnh rỗng quay lui thì dòng chảy là tối đại.*

Để chứng minh tính chất này, hãy đi qua đồ thị và nhận ra cạnh đầy hướng tới hay cạnh rỗng quay lui thứ nhất trên mỗi đường đi. Tập hợp các cạnh đó sẽ cắt đồ thị thành hai phần (trong ví dụ của chúng ta, các cạnh AB, CD và CE minh họa một nhát cắt như thế). Với mỗi nhát cắt chia mạng lưới thành hai phần, chúng ta có thể đo dòng

chảy ngang qua nó: tổng số dòng chảy trên các cạnh đi từ nguồn hướng tới đích. Trường hợp tổng quát các cạnh có thể đi theo cả hai hướng xuyên qua nhát cắt, phải bớt đi tổng số dòng chảy trên các cạnh theo hướng ngược. Nhát cắt trong ví dụ của chúng ta có giá trị 12 bằng với tổng số dòng chảy trong mạng lưới. Bất cứ khi nào giá trị nhát cắt bằng tổng số dòng chảy, không những chúng ta biết dòng chảy là tối đại mà còn biết nhát cắt là tối tiểu (nghĩa là mỗi nhát cắt khác có ít nhất một dòng chảy ngang qua với mật độ cao hơn). Đây chính là định lý dòng chảy tối đa-nhát cắt tối thiểu: dòng chảy không thể đạt giá trị lớn hơn và nhát cắt không thể đạt giá trị nhỏ hơn. Chúng ta bỏ qua chứng minh chi tiết của định lý này.

TÌM KIẾM TRÊN MẠNG LƯỚI

Có thể tóm tắt phương pháp Ford-Fulkerson như sau: "bắt đầu với dòng chảy 0 và gia tăng dòng chảy dọc theo mỗi đường đi từ đỉnh nguồn đến đỉnh đích mà không có cạnh đầy hướng tới hay cạnh rỗng quay lui, tiếp tục đến khi không còn đường đi như thế trong mạng lưới". Tuy nhiên, do phương pháp tìm đường đi không trình bày cụ thể và có thể dùng bất cứ đường đi nào, ví dụ dựa vào trực quan ta có cảm giác rằng đường đi càng dài thì mạng lưới càng được làm đầy và do đó chúng ta sẽ thích các đường đi dài hơn. Nhưng ví dụ trong Hình 33.3 cho thấy cần phải cẩn thận trong một số trường hợp.

Trong mạng lưới này, nếu đường đi được chọn đầu tiên là ABCD thì dòng chảy chỉ tăng lên 1. Kế đến chọn đường đi ACBD, dòng chảy cũng gia tăng 1 và ta trở lại tình huống khởi đầu nhưng với dòng chảy trên các cạnh bao ngoài tăng lên 1. Có thể do dựa vào đường đi dài ta cứ tiếp tục với chiến lược này và phải cần 2x1000 lần lặp mới tìm ra được dòng chảy cực đại. Nếu các số trên cạnh là 1 tỷ thì phải cần 2 tỷ lần lặp. Hiển nhiên đây là tình huống không nên để xảy ra, bởi vì các đường đi ABC và ADC cho ra ngay dòng chảy cực đại chỉ

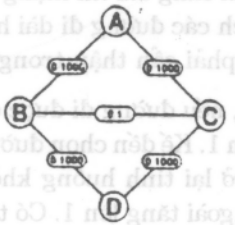
sau hai bước. Để thuật toán hoạt động tốt chúng ta phải tránh khả năng thời gian chạy phụ thuộc vào độ lớn của sức chứa. Thật ra, vấn đề này có thể được giải quyết bởi tính chất sau đây.

Tính chất 33.2 *Trong phương pháp Ford-Fulkerson, nếu dùng đường đi ngắn nhất từ đỉnh nguồn đến đỉnh đích thì cần ít hơn VE lần chọn các đường đi để tìm ra dòng chảy cực đại, trong đó V là số đỉnh và E là số cạnh của mạng lưới.*

Kết quả này đã được chứng minh bởi Edmonds và Karp vào năm 1972, các chi tiết trong chứng minh vượt khỏi giới hạn của quyển sách này.

Một phương án đơn giản là chỉ cần dùng một phiên bản sửa thích hợp của phương pháp **tìm kiếm ưu tiên độ rộng**. Chận trên trong định lý 33.2 là chận trên trong trường hợp xấu nhất, đối với mạng lưới thông thường thì có thể cần ít lần lặp hơn nhiều.

Dựa vào các phương pháp duyệt đồ thị có độ ưu tiên trong Chương 31, chúng ta có thể cài đặt một phương pháp khác được đề



Hình 33.3 *Mạng lưới có thể đòi hỏi nhiều lần lặp*

nghị bởi Edmonds và Karp: hãy tìm đường đi xuyên qua mạng lưới sao cho dòng chảy được gia tăng nhiều nhất. Có thể thực hiện điều này bằng cách dùng biến *priority* lưu độ ưu tiên (giá trị được chọn thích hợp) trong xâu kề hay ma trận kề của phương pháp tìm kiếm **dựa vào độ ưu tiên** trong Chương 31. Trường hợp biểu diễn ma trận kề, có thể tính độ ưu tiên nhờ vào đoạn chương trình sau, trường hợp biểu diễn xâu kề ta có thể làm hoàn toàn tương tự.

```
if size[k,t]>0
   then priority:=size[k,t]-flow[k,t]
   else priority:=-flow[k,t];
if priority>val[k] then priority:=val[k];
```

Kế đến bởi vì chúng ta muốn chọn nút với giá trị độ ưu tiên cao nhất chúng ta phải đổi thứ tự sơ đồ độ ưu tiên-hàng đợi trong các chương trình để trả về giá trị cực đại thay vì giá trị cực tiểu hay là sử dụng chúng bằng cách sửa *priority* thành *maxint-1-priority* (và xử lý ngược lại khi giá trị bị xóa). Chúng ta cũng sửa chữa thủ tục tìm kiếm dựa vào độ ưu tiên để đỉnh nguồn và đỉnh đích là các tham số, kế đến bắt đầu mỗi lần tìm kiếm từ đỉnh nguồn và dừng khi gặp đỉnh đích. Nếu không tìm ra được đường đi như thế, cây tìm kiếm ưu tiên riêng phần định nghĩa một nhát cắt tối thiểu cho mạng lưới; nếu ngược lại thì có thể cải tiến dòng chảy. Cuối cùng biến val của đỉnh nguồn phải nhận giá trị maxint trước khi quá trình tìm kiếm bắt đầu với ý nghĩa là một lượng tùy ý của dòng chảy có thể đạt được tại đỉnh nguồn (mặc dù nó bị giới hạn bởi tổng số sức chứa của tất cả các ống đi ra trực tiếp từ đỉnh nguồn).

Bằng cách dùng thủ tục *matrixpfs* trong phần trước, việc tìm dòng chảy cực đại rất đơn giản như chương trình sau đây:

```
repeat
   matrixpfs(1,V);
   y:=V; x:=dad[V];
   while x<>0 do
      begin flow[x,y]:=flow[x,y]+val[V];
            flow[y,x]:=-flow[x,y];
            y:=x; x:=dad[y]
      end;
until val[V]=1-maxint;
```

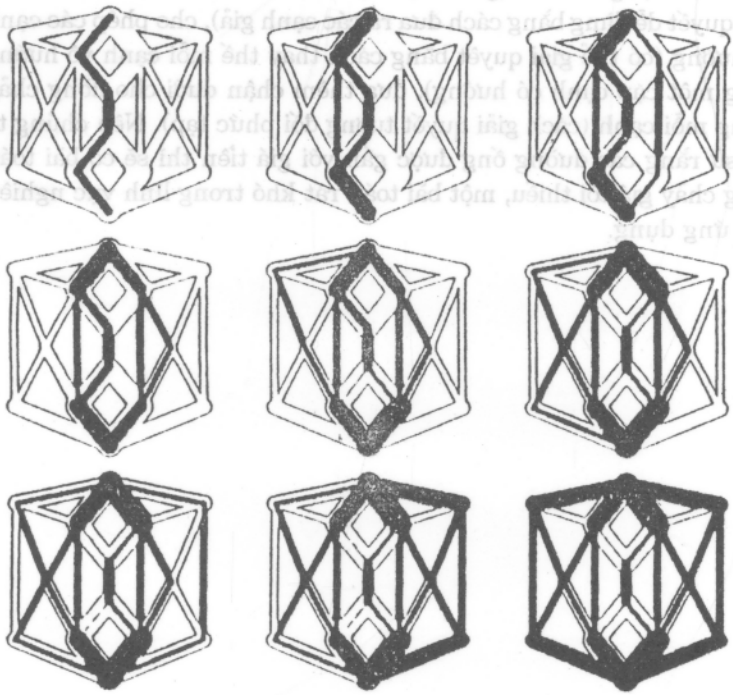
Trước tiên thuật toán gia tăng dọc theo đường ABDF, kế đến dọc theo ACDBEF, thuật toán đã không chọn đường đi thứ ba là ACDF vì đường đi nầy chỉ gia tăng dòng chảy 1 đơn vị thay vì 3 đơn vị có sẵn trong đường đi dài hơn. (Chú ý là chúng ta dựa vào Tính chất 33.2 để quyết định lựa chọn này.) Kế đến cần thêm một lần lặp để tìm ra dòng chảy cực đại.

Mặc dù thuật toán nầy dễ dàng cài đặt và nó hoạt động tốt cho các mạng lưới nảy sinh trong thực tế, sự phân tích của nó tương đối phức tạp. Trước hết thủ tục *matrixpfs* cần V^2 bước trong trường hợp xấu nhất, nếu thay thế bằng thủ tục *listpfs* thì cần $(E+V)logV$ bước cho mỗi lần lặp. Nhưng chúng ta phải cần tổng cộng bao nhiêu lần lặp?

Tính chất 33.3 *Nếu dùng phương pháp Ford-Fulkerson và gia tăng dòng chảy với một lượng lớn nhất thì số đường đi được dùng trước khi dòng chảy tối đa được tìm thấy trong mạng lưới sẽ nhỏ hơn* $1+\log_{M/M-1}f^*$ *trong đó* f^* *là giá của dòng chảy và M là số tối đa các cạnh trong mỗi nhát cắt của mạng lưới.*

Giống như trên, tính chất này được chứng minh bởi Edmonds và Karp, nó hoàn toàn vượt khỏi phạm vi quyển sách này.

Bài toán đang xét đã được nghiên cứu rộng rãi và các thuật toán phức tạp với trường hợp xấu nhất tương đối tốt của chúng đã được



Hình 33.4 *Tìm dòng chảy cực đại trong một mạng lưới lớn hơn*

phát triển. Tuy nhiên, thuật toán Edmonds-Karp như cài đặt trên đã rất tốt đối với các mạng lưới nảy sinh trong thực tế. Hình 33.4 cho thấy hoạt động của thuật toán đối với một mạng lưới lớn hơn.

Bài toán dòng chảy trong mạng lưới có thể được mở rộng theo nhiều hướng và nhiều biến dạng của nó đã được nghiên cứu cặn kẽ bởi vì chúng quan trọng trong các ứng dụng thực tế. Ví dụ, bài toán dòng chảy tổng quát gồm nhiều nguồn, nhiều đích và nhiều dạng vật chất truyền đi trong đồ thị. Trường hợp này bài toán khó hơn nhiều và cần phải có nhiều thuật toán nâng cao hơn. Các mở rộng khác của

bài toán mạng lưới bao gồm việc đặt sức chứa lên các đỉnh (có thể giải quyết dễ dàng bằng cách đưa ra các cạnh giả), cho phép các cạnh vô hướng (có thể giải quyết bằng cách thay thế mỗi cạnh vô hướng bằng một cặp cạnh có hướng), đưa thêm chận dưới cho dòng chảy trong mỗi cạnh (cách giải quyết tương đối phức tạp). Nếu chúng ta giả sử rằng các đường ống được gắn với giá tiền thì sẽ có bài toán dòng chảy giá tối thiểu, một bài toán rất khó trong lĩnh vực nghiên cứu ứng dụng.

BÀI TẬP

1. Hãy đưa ra một thuật toán để giải quyết bài toán dòng chảy trong mạng lưới trong trường hợp mạng lưới có dạng một cây nhờ vào xóa đi đỉnh đích.

2. Những đường đi nào được đi qua khi tìm dòng chảy tối đa trong mạng lưới có được bằng cách thêm các cạnh từ B đến C và từ E đến D với trọng lượng 3.

3. Hãy vẽ ra cây tìm kiếm có độ ưu tiên có được sau mỗi lần gọi đến thủ tục *matrixpfs* cho ví dụ mẫu trong chương này.

4. Cho biết nội dung của ma trận dòng chảy sau mỗi lần gọi thủ tục *matrixpfs* cho ví dụ mẫu.

5. Khẳng định sau đúng hay sai: không thuật toán nào có thể tìm được dòng chảy tối đại mà không kiểm tra mỗi cạnh trong mạng lưới.

6. Điều gì xảy ra đối với thuật toán Ford-Fulkerson khi mạng lưới có một chu trình có hướng.

7. Hãy cho biết một phiên bản của thuật toán chặn trên Edmonds-Karp đối với trường hợp tất cả sức chứa là $O(1)$.

8. Tìm một phản ví dụ cho thấy tại sao tìm kiếm ưu tiên độ sâu không thích hợp đối với bài toán dòng chảy trong mạng lưới.

9. Cài đặt lời giải yìm kiếm ưu tiên độ rộng cho bài toán dòng chảy bằng cách dùng thủ tục *sparsepfs*.

10. Viết một chương trình để tìm các dòng chảy tối đại trong các mạng lưới ngẫu nhiên có V đỉnh và khoảng $10V$ cạnh. Có bao nhiêu lần gọi đến thủ tục *sparsepfs* nếu $V=25, 50, 100$.

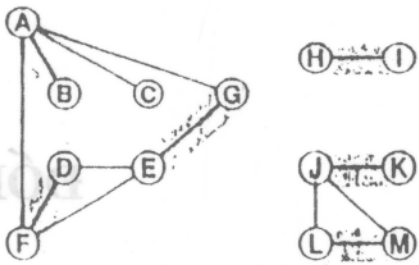
34

ĐỐI SÁNH

Một vấn đề thường gặp là "ghép cặp" các đối tượng thích hợp với nhau theo một số quan hệ nào đó. Ví dụ, cần bố trí các sinh viên y khoa sắp tốt nghiệp vào các bệnh viện. Mỗi sinh viên liệt kê ra các bệnh viện mà mình thích và mỗi bệnh viện đưa ra danh sách các sinh viên được ưu tiên. Một bài toán đặt ra là hãy bố trí các sinh viên vào các bệnh viện sao cho thích hợp. Bài toán này cần một thuật toán tinh xảo, bởi vì các sinh viên tốt nhất sẽ được nhiều bệnh viện ưa thích và các bệnh viện tốt nhất sẽ được nhiều sinh viên ưa chuộng.

Ví dụ trên là một trường hợp đặc biệt của một bài toán khó và cơ bản của lãnh vực đồ thị. Cho một đồ thị, một **bộ đối sánh** được định nghĩa gồm một số cạnh của đồ thị sao cho không có đỉnh nào xuất hiện nhiều hơn một lần. Nghĩa là các đỉnh kề với mỗi cạnh trong bộ đối sánh được ghép cặp với nhau, nhưng các đỉnh còn lại có thể không được ghép cặp. Ngay cả trường hợp một bộ đối sánh bao gồm nhiều cạnh nhất trong phạm vi cho phép thì các phương cách chọn cạnh khác nhau có thể đưa đến số lượng các đỉnh không được ghép cặp là khác nhau.

Một **bộ đối sánh cực đại** là một bộ đối sánh có nhiều cạnh nhất, điều này tương đương với việc tối thiểu hóa số các đỉnh không được đối sánh. Trường hợp tốt nhất mà chúng ta có thể hy vọng là có một tập hợp cạnh mà mỗi đỉnh của đồ thị xuất hiện chính xác một lần trong tập hợp đó (trong một đồ thị có $2V$ đỉnh một bộ đối sánh như



Hình 34.1 *Một bộ đối sánh cực đại (các cạnh bóng)*

vậy sẽ có V cạnh). Nhưng không phải lúc nào chúng ta cũng đạt được trường hợp tốt nhất này.

Hình 34.1 minh họa một bộ đối sánh cực đại (các cạnh được làm bóng) trong đồ thị mẫu của chúng ta. Với một đồ thị gồm 13 đỉnh chúng không thể có bộ đối sánh nào nhiều hơn sáu cạnh. Nhưng các thuật toán đơn giản tìm ra các bộ đối sánh sẽ gặp khó khăn ngay cả trong ví dụ này. Ví dụ nếu chúng ta chọn các cạnh thích hợp cho việc đối sánh khi chúng xuất hiện trong quá trình tìm kiếm ưu tiên độ sâu (xem Hình 29.7). Xem Hình 34.1, chúng ta chọn được các cạnh AF EG HI JK LM, nhưng chúng không phải là một bộ đối sánh cực đại. Như đã chú ý trước đây, không phải dễ dàng biết có bao nhiêu cạnh trong một bộ đối sánh cực đại của một đồ thị đã cho trước. Chẳng hạn xem một đồ thị con (của đồ thị đang xét) gồm 6 đỉnh từ A đến F, trong đồ thị con này không tồn tại một bộ đối sánh nào có ba cạnh. Trong khi đó chúng ta có thể tìm dễ dàng một bộ đối sánh cực đại của đồ thị lớn (ví dụ, việc tìm một bộ đối sánh cực đại cho đồ thị "mê cung" trong Chương 29 sẽ không phải là một bài toán khó). Do đó, trong trường hợp tổng quát việc tìm bộ đối sánh cực đại cho một đồ thị bất kỳ là một vấn đề khó khăn.

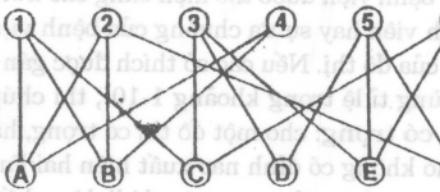
Với bài toán đối sánh bệnh viện-sinh viên như đã trình bày ở trên, các sinh viên và các bệnh viện được thể hiện bằng các nút của đồ thị; các sở thích của sinh viên hay sự ưa chuộng của bệnh viện được thể hiện bằng các cạnh của đồ thị. Nếu các sở thích được gán bởi các giá trị số (chẳng hạn dùng tỉ lệ trong khoảng 1-10), thì chúng ta sẽ có **bài toán đối sánh có trọng**: cho một đồ thị có trọng, hãy tìm một tập hợp cạnh sao cho không có đỉnh nào xuất hiện hai lần trong tập đó và tổng trọng lượng các cạnh trong tập đó là lớn nhất. Dưới đây chúng ta sẽ xem một dạng khác, trong dạng nầy chúng ta chỉ chú ý tới các sở thích nhưng không đòi hỏi gán giá trị số cho chúng.

Bài toán đối sánh đã thu hút sự chú ý lớn lao của nhiều nhà toán học bởi tính tự nhiên và khả năng ứng dụng rộng rãi của nó. Lời giải của nó trong trường hợp tổng quát là các công trình về toán học tổ hợp rất đẹp và rất phức tạp mà hoàn toàn vượt khỏi phạm vi của quyển sách này. Ở đây chúng tôi sẽ giúp độc giả tiếp cận bài toán qua việc khảo sát một vài trường hợp đặc biệt thú vị cùng với phát triển một số thuật toán hữu dụng.

ĐỒ THỊ HAI PHẦN

(BIPARTITE GRAPH)

Như đã lưu ý trong ví dụ trên, việc đối sánh các sinh viên với các bệnh viện chắc chắn là một trong nhiều ứng dụng đa dạng của bài toán đối sánh. Ví dụ chúng ta cần đối sánh các đàn ông và các phụ nữ trong một dịch vụ hôn nhân, các người xin việc vào các xí nghiệp, các cua giảng dạy vào các giờ học. Các đồ thị nảy sinh trong các trường hợp như thế được gọi là **đồ thị hai phần**, được định nghĩa gồm các cạnh nối hai tập hợp đỉnh. Nghĩa là tập hợp đỉnh được chia thành hai tập con và không có cạnh nào nối hai đỉnh trong cùng một tập con. Một ví dụ của đồ thị hai phần được trình bày trong trong Hình 34.2. Có lẻ độc giả thích thú với việc tìm một bộ đối sánh cực



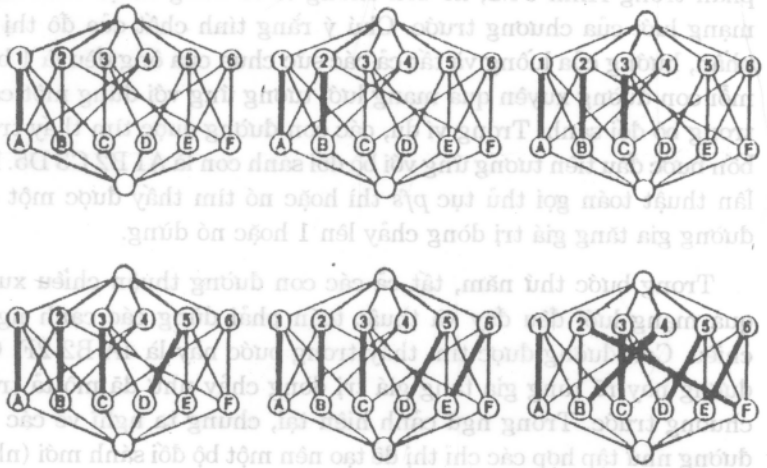
Hình 34.2 *Một đồ thị hai phần*

đại trong đồ thị này.

Trong biểu diễn ma trận kề cho các đồ thị hai phần, hiển nhiên người ta chỉ cần lưu các dòng cho một tập hợp đỉnh và các cột cho tập hợp đỉnh còn lại. Trong biểu diễn xâu kề thì lại không có thay đổi nào, ngoài ra phải đặt tên các đỉnh một cách thông minh để có thể xác định tên các đỉnh thuộc về tập hợp đỉnh nào.

Trong các ví dụ, chúng ta sẽ dùng các chữ cái để đánh dấu các đỉnh trong một tập hợp và các con số để đánh dấu các đỉnh trong tập hợp còn lại. Bài toán đối sánh cực đại cho các đồ thị hai phần có thể phát biểu đơn giản trong dạng biểu diễn này như sau: "hãy tìm một tập con lớn nhất gồm các cặp chữ cái-số sao cho không có hai cặp nào trong tập con đó có cùng một chữ cái hay số". Một bài tập thú vị là cố gắng tìm ra một lời giải trực tiếp cho bài toán đối sánh trong trường hợp các đồ thị hai phần. Nhìn sơ qua thì vấn đề này dường như có vẻ đơn giản, nhưng sự tế nhị lại rất cần thiết trong trườn hợp này. Chắc chắn sẽ có nhiều tổ hợp các cặp nếu chúng ta cố gắng vét cạn tất cả các khả năng có thể: một lời giải phải đủ thông minh để có thể chọn chỉ một vài phương án có thể ghép đôi các đỉnh.

Chúng ta sẽ khảo sát một lời giải gián tiếp: để giải một ví dụ cụ thể của bài toán đố sánh, chúng ta sẽ xây dựng một ví dụ của **bài toán dòng chảy trong mạng lưới**, dùng thuật toán trong chương trước, kế đến dùng lời giải cho bài toán mạng lưới để giải bài toán đối



Hình 34.3 *Sử dụng dòng chảy trong mạng lưới để tìm bộ đối sánh cực đại*

sánh. Nghĩa là chúng ta đưa bài toán đối sánh về bài toán dòng chảy trong mạng lưới. Đây là một phương pháp thiết kế thuật toán để tạo ra một thuật toán nhờ vào các thuật toán họ hàng của nó có sẵn từ một thư viện chương trình con được viết sẵn bởi người lập trình hệ thống. Đây là một điểm quan trọng cơ bản trong lý thuyết của các thuật toán tổ hợp nâng cao (xem Chương 40). Bây giờ cách tiếp cận nầy sẽ cho chúng ta một lời giải hiệu quả của bài toán đối sánh trong đồ thị hai phần.

Với một đồ thị hai phần cho sẵn, chúng ta xây dựng một mạng lưới bằng cách tạo một đỉnh nguồn với các cạnh nối tới tất cả các đỉnh trong một tập hợp của đồ thị hai phần, kế đến thêm vào một đỉnh đích được nối tới bởi tất cả các đỉnh trong tập còn lại của đồ thị hai phần. Tất cả các cạnh trong đồ thị có được sẽ được xem là có sức chứa (kích thước) 1.

Hình 34.3 cho thấy một mạng dòng chảy tương ứng với đồ thị hai phần trong Hình 34.2, kế đến chúng ta sẽ dùng thuật toán luồng mạng lưới của chương trước. Chú ý rằng tính chất của đồ thị hai phần, hướng của luồng và tất cả các sức chứa của ống đều là 1 buộc mỗi con đường xuyên qua mạng lưới tương ứng với đúng một cạnh trong bộ đối sánh. Trong ví dụ, các con đường được tìm thấy trong bốn bước đầu tiên tương ứng với bộ đối sánh con là A1 B2 C3 D5. Mỗi lần thuật toán gọi thủ tục *pfs* thì hoặc nó tìm thấy được một con đường gia tăng giá trị dòng chảy lên 1 hoặc nó dừng.

Trong bước thứ năm, tất cả các con đường thuận chiều xuyên qua mạng lưới đều đầy và thuật toán phải dùng các cạnh ngược chiều. Con đường được tìm thấy trong bước này là 4A B2 2F. Con đường này rõ ràng gia tăng giá trị dòng chảy như đã mô tả trong chương trước. Trong ngữ cảnh hiện tại, chúng ta nghĩ về các con đường như tập hợp các chỉ thị để tạo nên một bộ đối sánh mới (nhiều hơn một cạnh) từ bộ đối sánh hiện hành. Việc xây dựng này theo thứ tự như sau: "4A" nghĩa là thêm cạnh A4 vào bộ đối sánh, "B2" nghĩa là xoá bớt cạnh B2 và "2F" có nghĩa là thêm cạnh F2 vào bộ đối sánh. Do đó sau khi xử lý xong con đường này chúng ta có bộ đối sánh A1 B4 C3 D5 E6 F2. Thuật toán kết thúc khi gặp F6; bởi vì tất cả các đường ống khởi đầu từ đỉnh nguồn và đi vào đỉnh đích, vì vậy chúng ta có một bộ đối sánh cực đại.

Có thể chứng minh dễ dàng rằng bộ đối sánh tìm được nhờ vào thuật toán dòng chảy cực đại là một bộ đối sánh cực đại của đồ thị. Trước nhất là thuật toán luôn cho ta một bộ đối sánh hợp lệ: bởi vì mỗi đỉnh có một cạnh với sức chứa 1 đi vào đỉnh đích hay ra khỏi đỉnh nguồn, nhiều nhất một đơn vị dòng chảy đi qua mỗi đỉnh, nghĩa là mỗi đỉnh xuất hiện nhiều nhất một lần trong bộ đối sánh. Thứ hai là không thể có một bộ đối sánh nào có thể có nhiều cạnh hơn, bởi vì bất kỳ một bộ đối sánh nào như thế sẽ cho ta một dòng chảy tốt hơn dòng chảy có được từ thuật toán.

Như vậy để tìm bộ đối sánh cực đại cho đồ thị hai phần, chúng ta định dạng đồ thị sao cho nó thích hợp cho việc nhập dữ liệu vào thuật toán dòng chảy trong mạng lưới của chương trước. Dĩ nhiên, các đồ thị được biểu diễn dựa vào thuật toán dòng chảy nầy trong trường hợp nầy sẽ đơn giản hơn nhiều so với các đồ thị tổng quát mà thuật toán được thiết kế để giải quyết bài toán dòng chảy cực đại, chính vì thế mà trong trường hợp này thuật toán hoạt động hiệu quả hơn nhiều.

Tính chất 34.1 *Một bộ đối sánh cực đại trong đồ thị hai phần có thể được tìm thấy trong $O(V^3)$ bước nếu đồ thị dày và trong $O(V(E+V)\log V)$ bước nếu đồ thị thưa.*

Việc xây dựng trong phần trên bảo đảm rằng mỗi lần gọi tới thủ tục *pfs* sẽ thêm một cạnh vào bộ đối sánh, vì vậy chúng ta biết rằng có nhiều nhất là $V/2$ lần gọi tới thủ tục *pfs* trong suốt quá trình thực hiện của thuật toán. Do đó thời gian cần thiết sẽ xấp xỉ tới một hệ số V lớn hơn thời gian cần cho một lần tìm kiếm đã thảo luận trong Chương 31.

BÀI TOÁN HÔN NHÂN BỀN VỮNG

Giả sử có N chàng trai và N cô gái, nếu mỗi chàng trai cho biết chính xác cảm tưởng của anh ta về mỗi phụ nữ và ngược lại mỗi phụ nữ cũng cho biết chính xác về cảm tưởng của cô ta đối với mỗi chàng trai. Vấn đề cần giải quyết là tìm ra N cuộc hôn nhân đáp ứng ý muốn của mỗi người.

Làm thế nào để thể hiện sở thích ? Một phương pháp có thể dùng là sử dụng thang điểm 1-10, mỗi người sẽ cho điểm người khác phái một điểm trong thang điểm nầy. Khi đó bài toán hôn nhân sẽ trở thành bài toán đối sánh có trọng, một bài toán tương đối khó. Hơn nữa việc dùng các điểm tuyệt đối có thể dẫn tới sự không chính xác, bởi vì điểm số sẽ không nhất quán đối với tất cả mỗi người (điểm 10

Hình 34.4 *Các danh sách sở thích cho bài toán hôn nhân*

của cô gái này có thể tương đương với điểm 7 của một cô khác). Một phương pháp tự nhiên hơn là biểu diễn các sở thích bằng cách tạo ra cho mỗi người một danh sách những người khác phái được xếp theo thứ tự mà họ thích. Hình 34.4 minh họa các danh sách sở thích của 5 chàng trai và 5 cô gái. Như thường lệ, chúng ta giả sử đã dùng phương pháp băm hay một cách khác để chuyển tên thật thành các chữ số (đối với các cô gái) và thành các chữ cái (đối với các chàng trai).

Rõ ràng những sở thích này thường tranh chấp nhau, ví dụ cả A lẫn B đều thích 2 nhất và không ai thích 4 nhiều lắm (nhưng một trong hai có thể phải kết hôn với cô ta). Vấn đề là chọn cho mỗi chàng trai một cô gái để đáp ứng các sở thích của họ càng nhiều càng tốt. Một tập các cuộc hôn nhân gọi là **không bền vững** nếu tồn tại hai người thích lẫn nhau hơn so với vợ/chồng của họ. Ví dụ việc gán A1 B3 C2 D4 E5 là không bền vững bởi vì A thích 2 hơn 1 và 2 thích A hơn C, do đó để đáp ứng nguyện vọng của họ thì A nên bỏ 1 lấy 2 và 2 nên bỏ C lấy A.

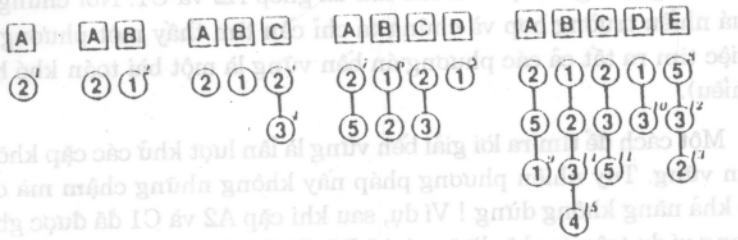
Việc tìm một lời giải bền vững là một bài toán khó, bởi vì có quá nhiều trường hợp có thể có. Ngay cả xác định xem một lời giải cho trước có bền vững hay không cũng không phải là vấn đề đơn giản, trước khi đọc đoạn văn sắp tới độc giả hãy thử tìm một cặp không

bền vững trong ví dụ trên sau khi đã ghép A2 và C1. Nói chung có quá nhiều trường hợp và chúng ta chỉ cần tìm thấy một phương án (việc tìm ra tất cả các phương án bền vững là một bài toán khó hơn nhiều).

Một cách để tìm ra lời giải bền vững là lần lượt khử các cặp không bền vững. Tuy nhiên phương pháp này không những chậm mà còn có khả năng không dừng ! Ví dụ, sau khi cặp A2 và C1 đã được ghép trong ví dụ trên (nghĩa là ta có A2 B3 C1 D4 E5), thì B và 2 tạo nên một cặp không bền vững (B và 2 thích nhau hơn so với những người đã được ghép) nên ta có phương án mới là A3 B3 C1 D4 E5. Kế đến B và 1 tạo nên một cặp không bền vững nên ta có A3 B1 C2 D4 E5. Cuối cùng A và 1 lại tạo nên một cặp không bền vững và điều này dẫn đến phương án trùng với phương án gốc. Như vậy cách làm này bị quẩn đối với ví dụ đang xét.

Bây giờ chúng ta sẽ nghiên cứu một thuật toán xây dựng các cặp bền vững dựa trên ý tưởng giống như đời sống thực của bài toán. Mỗi chàng trai X sẽ chọn cho mình một cô dâu, hiển nhiên là anh ta sẽ chọn cô gái đầu tiên trong danh sách của mình. Nếu cô này đã hứa hôn (đã được chọn trước đó) với chàng trai khác mà cô ta thích thì anh chàng X phải chọn người phụ nữ kế tiếp trong danh sách của anh ta, quá trình này tiếp tục tới khi X tìm được một cô Y chưa được hứa hôn hay là cô gái thích anh ta. Nếu Y chưa được hứa hôn thì cô ta sẽ được gả cho X và chúng ta sẽ tiếp tục cho người đàn ông kế tiếp. Nếu Y đã được hứa hôn thì sẽ hủy bỏ cuộc đính hôn và gả cô ta cho X (vì cô ta thích X), và chúng ta tiếp tục với anh chàng vừa bị từ hôn.

Phương pháp này có lẽ được mô hình từ một câu chuyện trong các tiểu thuyết của thế kỷ 19, chúng ta sẽ kiểm tra cẩn thận một số trường hợp để thấy nó cho ta phương án bền vững. Hình 34.5 minh họa tuần tự các bước của thuật toán. Trước tiên chàng A chọn nàng 2; kế đến B chọn 1; kế đến C chọn 2 nhưng không được và anh ta lại chọn 3; kế đến D chọn 1, vì 1 thích D hơn B nên hủy bỏ sự hứa hôn



Hình 34.5 *Giải bài toán hôn nhân bền vững*

của B và 1, và gả 1 cho D; chúng ta bắt đầu lại với chàng B, B chọn 2, vì 2 thích B hơn A nên hủy bỏ sự hứa hôn của A và 2, và gả 2 cho B; kế đến A chọn 5, chúng ta có một tình huống bền vững tạm thời. Độc giả thử kiểm tra tiếp cho chàng E. Chú ý rằng E đóng vai trò người hôn phu hai lần trong quá trình xử lý.

Bước đầu tiên trong việc cài đặt là thiết kế cấu trúc dữ liệu được dùng cho các danh sách sở thích. Đây là các xâu đơn mà chúng ta đã thấy qua các ví dụ của Chương 3 và nơi khác, nhưng việc lựa chọn một biểu diễn đặc thù cũng có khả năng ảnh hưởng đến tính năng của thuật toán.

Bởi vì tất cả các danh sách sở thích có cùng độ dài, cài đặt đơn giản nhất là dùng mảng hai chiều. Ví dụ *prefer[m,w]* là cô gái thứ *w* trong danh sách sở thích của chàng trai thứ m. Hơn nữa chúng ta cần theo dõi xem mỗi chàng trai đã xử lý đến đâu trong danh sách của anh ta. Điều này có thể được thực hiện nhờ một mảng một chiều *next*, mảng này khởi tạo đến *0* và *next[m]+1* là chỉ số của cô gái kế tiếp trong danh sách của chàng trai thứ *m*; chỉ danh của cô ta sẽ là *prefer[m,next[m]+1]*.

Với mỗi cô gái, chúng ta cần theo dõi vị hôn phu của cô ta (*fiancée[w]* sẽ là chàng trai đã hứa hôn với cô gái *w*) và chúng ta cần trả lời câu hỏi "người đàn ông thứ *s* có được thích hơn so với người

đàn ông *fiancée[w]* hay không?". Câu trả lời này có thể được giải đáp bằng cách tìm kiếm tuần tự cho tới khi *s* hoặc *fiancé[w]* được tìm thấy, nhưng cách nầy sẽ không hiệu quả nếu cả hai người đều ở gần cuối danh sách. Thay vì vậy, chúng ta sẽ dùng một danh sách "ngược": *rank[w,s]* là chỉ số của chàng trai thứ *s* trong danh sách sở thích của cô *w*. Trong ví dụ trên thì *rank[1,1]* là 2 bởi vì A (chàng trai thứ nhất) đứng thứ hai trong danh sách của sở thích của cô gái thứ nhất, *rank[5,4]* là 1 bởi vì D (chàng trai thứ tư) đứng đầu tiên trong danh sách sở thích của cô gái thứ năm...

Sự phù hợp của vị hôn phu *s* có thể được xác định rất nhanh chóng bằng cách kiểm tra xem *rank[w,s]* có nhỏ hơn *rank[w,fiancée[w]]* hay không. Các mảng nầy được xây dựng dễ dàng và trực tiếp từ các danh sách sở thích. Để khởi đầu, chúng ta dùng một "lính canh" là chàng trai *0* để khởi tạo cho vị hôn phu và đặt anh ta ở cuối tất cả các danh sách sở thích của các cô gái.

Với cấu trúc dữ liệu được khởi tạo bằng phương pháp này, việc cài đặt có thể thực hiện dễ dàng như sau:

```
for m:=1 to N do
   begin   s:=m;
           repeat
              next[s]:=next[s]+1; w:=prefer[s,next[s]];
              if rank[w,s]<rank[w,fiancée[w]] then
                 begin t:=fiancée[w]; fiancée[w]:=s; s:=t end;
           until s=0;
   end;
```

Mỗi vòng lặp sẽ bắt đầu bởi một chàng trai chưa hứa hôn và kết thúc với một cô gái chưa hứa hôn. Vòng lặp repeat phải kết thúc bởi vì mỗi danh sách của một chàng trai chứa mỗi cô gái và mỗi lần lặp đã tăng thêm danh sách của một chàng trai nào đó, và một cô gái

chưa hứa hôn phải được thấy trước khi danh sách của một chàng trai bất kỳ được xử lý. Thuật toán sẽ cho ta các cuộc đính hôn bền vững bởi vì mỗi cô gái nếu được một chàng trai nào đó thích hơn vợ chưa cưới của anh ta sẽ được gả cho người mà cô ta thích hơn anh ta.

Tính chất 34.2 *Bài toán hôn nhân bền vững có thể được giải với thời gian tuyến tính.*

Như đã chú ý ở trên, mỗi lần lặp trong vòng lặp gia tăng danh sách sở thích của một chàng trai nào đó. Trong trường hợp xấu nhất tất cả các phần tử của mỗi danh danh sách đều được kiểm tra (nhưng không có phần tử nào được kiểm tra hai lần). Thật ra thuật toán có thể đòi hỏi ít thời gian hơn nó cần để xây các danh sách, bởi vì một phương án bền vững có thể được tìm ra trước khi tất cả các danh sách được xử lý.

Có nhiều thiên vị hiển nhiên trong thuật toán này. Trước tiên, chàng trai duyệt qua các cô gái trong danh sách, trong khi các cô gái phải chờ đợi một chàng trai "đúng ý". Sự thiên vị này có thể được hiệu chỉnh bằng cách thay đổi thứ tự nhập của các danh sách sở thích. Việc làm này sẽ cho ta phương án bền vững 1E 2D 3A 4C 5B, trong đó mỗi cô gái đều chọn được người mình thích nhất ngoại trừ cô thứ 5 phải lấy chàng trai thứ hai trong sách của mình. Nói chung có thể có nhiều phương án bền vững, có thể nói đây là một phương án "tối ưu" cho các phụ nữ theo nghĩa không có một phương án bền vững nào khác làm cho các cô gái có thể chọn tốt hơn.

Một đặc điểm thiên vị khác của thuật toán là thứ tự mà các chàng trai được chọn làm hôn phu: giữa chàng trai đầu tiên và chàng trai cuối cùng thì ai nên được phép chọn trước? Câu trả lời là thứ tự này không quan trọng. Khi mỗi chàng trai lựa chọn và mỗi cô gái đồng ý nhờ dựa vào danh sách của họ thì các kết quả sẽ như nhau.

CÁC THUẬT TOÁN NÂNG CAO

Hai trường hợp đặc biệt mà chúng ta vừa khảo sát đã cho thấy sự rắc rối của bài toán đối sánh. Mặc dù các thuật toán cụ thể này có ích trong nhiều ứng dụng thực tế nhưng nhiều ứng dụng khác có thể đòi hỏi giải quyết tổng quát hơn.

Một số bài toán tổng quát đã được nghiên cứu như: bài toán đối sánh cực đại cho đồ thị tổng quát (không nhất thiết là đồ thị hai phần); bài toán đối sánh có trọng cho đồ thị hai phần, trong đó các cạnh đã được đặt trọng lượng và việc đối sánh phải thực hiện sao cho tổng trọng lượng tìm được là tối đa; và bài toán đối sánh có trọng cho đồ thị tổng quát.

Việc đối sánh có trọng số cho các đồ thị hai phần và các tổng quát hóa tương tự có thể được giải quyết bằng cách mở rộng thuật toán cho các tổng quát hóa của bài toán dòng chảy trong mạng lưới. Tuy nhiên đối với các đồ thị tổng quát thì lại là một câu chuyện khác. Bài toán hôn nhân bền vững có thể xem như một cách để né tránh bài toán đối sánh có trọng cho đồ thị tổng quát bằng cách định nghĩa lại bài toán.

Nếu cần bàn đầy đủ hơn về vấn đề đối sánh trong các đồ thị tổng quát thì có phải cần trọn một quyển sách, đây là một trong các bài toán được nghiên cứu rộng rãi trong lý thuyết đồ thị.

BÀI TẬP

1. Tìm tất cả các bộ đối sánh năm cạnh của đồ thị hai phần trong Hình 34.2.

2. Dùng thuật toán đã cho để tìm ra các bộ đối sánh cực đại cho các đồ thị hai phần có *50* đỉnh và *100* cạnh. Có khoảng bao nhiêu cạnh trong mỗi bộ đối sánh?

3. Xây dựng một đồ thị hai phần gồm 6 nút và 8 cạnh mà có một bộ đối sánh 3 cạnh. Nếu không được hãy chứng minh không tồn tại đồ thị như thế.

4. Giả sử rằng các đỉnh trong một đồ thị hai phần biểu diễn các công việc và con người, mỗi người được giao cho hai việc. Có thể dùng thuật toán dòng chảy trong mạng lưới để giải bài toán này hay không ? Chứng minh trả lời của bạn.

5. Hãy sửa đổi chương trình dòng chảy trong mạng lưới của Chương 33 để lợi dụng cấu trúc đặc biệt của mạng lưới 0-1 xuất hiện trong bài toán đối sánh của đồ thị hai phần.

6. Hãy viết một chương trình hiệu quả để xác định xem một phương án của bài toán hôn nhân có bền vững hay không.

7. Có thể để cho hai chàng trai chọn cô gái cuối cùng của danh sách của mình trong thuật toán hôn nhân bền vững hay không? Chứng minh trả lời của bạn.

8. Xây dựng một tập các danh sách sở thích với $N=4$ cho bài toán hôn nhân bền vững trong đó mỗi người chọn được người thứ hai trong danh sách sở thích của mình hay chứng minh rằng không tồn tại tập hợp như thế.

9. Cho biết cấu hình bền vững của bài toán hôn nhân trong trường hợp các danh sách sở thích của các chàng trai và các cô gái là như nhau theo thứ tự tăng.

10. Chạy chương trình hôn nhân bền vững với $N=50$, sử dụng các hoán vị ngẫu nhiên cho các danh sách sở thích. Có khoảng bao nhiêu cuộc hôn nhân trong suốt quá trình thực hiện thuật toán?

35

SỐ NGẪU NHIÊN

Chương này giới thiệu các thuật toán dùng máy vi tính để tạo các số ngẫu nhiên. Mặc dù chúng ta đã gặp các số ngẫu nhiên trong nhiều phần của quyển sách này,nhưng ở đây chúng ta bắt đầu thử xét xem chính xác chúng là gì.

Thông thường, trong các cuộc đối thoại, người ta dùng thuật ngữ ngẫu nhiên khi muốn nói đến sự tùy ý. Một người đòi hỏi một số ngẫu nhiên, nghĩa là không chú ý đó là số mấy, số nào cũng được. Ngược lại, số ngẫu nhiên là một khái niệm toán học được định nghĩa chính xác: mọi số đều có khả năng xuất hiện tương đương nhau.

Để đáp ứng được định nghĩa số ngẫu nhiên nêu trên, chúng ta phải giới hạn các số được dùng vào một phạm vi nhất định. Không thể có một số nguyên ngẫu nhiên, chỉ có một số nguyên ngẫu nhiên trong một miền xác định nào đó.

Thông thường, trong hầu hết các trường hợp không chỉ cần một số ngẫu nhiên, mà cần đến dãy số ngẫu nhiên. Khi đó, toán học dự vào: cần chứng minh nhiều mặt về các thuộc tính của dãy số ngẫu nhiên. Ví dụ trong một chuỗi dài các số ngẫu nhiên của một phạm vi nhỏ, chúng ta có thể muốn biết giá trị về số lần xuất hiện.

Chuỗi số ngẫu nhiên tương tự như nhiều trường hợp trong tự nhiên và một số lượng đáng kể các thuộc tính của chúng đã được biết rõ. Để thích hợp với các sử dụng hiện hành, chúng ta sẽ xét đến các

số của chuỗi ngẫu nhiên như là các số ngẫu nhiên.

Không có cách nào để tạo ra các số ngẫu nhiên thực sự từ một máy vi tính. Một khi chương trình do chúng ta viết, thì chắc chắn các số nó tạo ra có thể suy luận được, vậy thì chúng có thể nào là ngẫu nhiên ? Phương pháp tốt nhất chúng ta hy vọng là viết các chương trình để tạo ra các chuỗi số có được nhiều thuộc tính giống như các số ngẫu nhiên. Các số này thường được gọi là các số giả ngẫu nhiên (pseudo-random). Chúng không thực sự ngẫu nhiên nhưng chúng có thể hữu dụng như sự xấp xỉ của các số ngẫu nhiên (cũng như các số kiểu chấm động được dùng là sự xấp xỉ của các số thực)

Trong một vài tình huống, một ít thuộc tính của các số ngẫu nhiên thì quan trọng trong khi các thuộc tính còn lại thì không cần thiết. Trong trường hợp đó, cần tạo các số gần ngẫu nhiên, chúng chắc chắn có các thuộc tính mong muốn nhưng không hứa hẹn sẽ có các thuộc tính khác. Trong một số ứng dụng, các số gần ngẫu nhiên có thể chứng minh là thích hợp hơn các số giả ngẫu nhiên.

Dễ thấy rằng việc làm cho gần đúng thuộc tính "mỗi số có khả năng xuất hiện như nhau" trong một chuỗi dài vẫn không đủ dùng. Ví dụ, mỗi số trong đoạn [1,100] xuất hiện 1 lần trong chuỗi (1,2,...100), nhưng chuỗi này không chắc là hữu dụng như một chuỗi gần tương đương chuỗi ngẫu nhiên.

Trong thực tế, trong một chuỗi ngẫu nhiên gồm 100 số thuộc miền xác định [1,100], thực ra, một vài số sẽ xuất hiện hơn một lần và một vài số khác thì không có trong chuỗi. Nếu không đạt được điều này trong một chuỗi giả ngẫu nhiên thì nghĩa là đã có điều gì sai sót trong việc tạo chuỗi. Nhiều phương pháp kiểm tra phức tạp đặt cơ sở trên các nhận xét kiểu trên đã được áp dụng cho các phương pháp tạo chuỗi ngẫu nhiên, để kiểm tra xem 1 chuỗi số giả ngẫu nhiên dài có một số thuộc tính của chuỗi ngẫu nhiên không. Cách tạo số ngẫu nhiên mà chúng ta sẽ xem xét đã vượt qua rất tốt các kiểm tra như vậy. Chúng ta cũng sẽ xem xét một trong các kiểm tra

quan trọng nhất, đó là kiểm tra Chi bình phương (chi-square).

Chúng ta sẽ bàn kỹ về các số ngẫu nhiên phân bố đều, với các giá trị xem như tương đương nhau. Cũng tương tự cho các số ngẫu nhiên tuân theo các phân phối không đồng đều, với một số giá trị có nhiều hơn một số khác. Các số giả ngẫu nhiên với phương pháp phân phối *non-uniform* (*không đồng đều*) thường bao gồm bởi thực hiện vài phân phối kiểu *uniform* tạo thành.

Đa số các ứng dụng trong quyển sách này dùng các số ngẫu nhiên phân bố đều. Như chúng ta sẽ thấy, khó mà thuyết phục rằng các số ngẫu nhiên chúng ta tạo có "tất cả" các thuộc tính của số ngẫu nhiên. Đây cũng là một vấn đề quan trọng đối với các phương pháp phân phối khác.

CÁC ỨNG DỤNG

Như chúng ta đã thấy trong quyển sách này, nhiều ứng dụng đã sử dụng các số ngẫu nhiên rất hữu hiệu. Một trong số đó là trong thuật toán mật mã, với mục đích chính là mã hóa một thông điệp để chỉ có người nhận mới đọc được chúng. Và như chúng ta đã thấy ở chương 23, một cách để làm việc này là làm cho thông điệp có vẻ như ngẫu nhiên bằng cách dùng một chuỗi giả ngẫu nhiên để mã hóa và cũng với chuỗi đó, người nhận có thể giải mã được.

Một lãnh vực khác cũng sử dụng rộng rãi các số ngẫu nhiên là mô phỏng. Một *simulation* đặc trưng bao gồm một chương trình lớn theo kiểu một số khía cạnh của thế giới thực: các số ngẫu nhiên thì rất tự nhiên, thích hợp làm dữ liệu nhập cho các chương trình như vậy. Ngay cả khi không cần các số ngẫu nhiên, simulation vẫn cần các số tùy ý dùng làm dữ liệu nhập, và điều này được cung cấp rất thuận lợi bởi các công cụ tạo số ngẫu nhiên.

Khi một số lượng lớn dữ liệu được phân tích, đôi khi chỉ cần xử lý trên một tập con nhỏ của nó ,bằng cách lựa chọn theo kiểu thử

ngẫu nhiên, là đủ. Những áp dụng như vậy rất phổ biến, đáng chú ý nhất là việc thăm dò chính kiến của dân chúng .

Chúng ta thường cần phải có một sự lựa chọn giữa các phần tử mà tất cả đều có vẻ tương đương nhau, nghĩa là phải "tạo một quyết định". Ví dụ như phương pháp áp dụng trong các trường trung học để sắp xếp phòng ngủ cho học sinh là một minh họa của việc dùng số ngẫu nhiên trong việc tạo quyết định lựa chọn. Trong trường hợp này, trách nhiệm quyết định được giao cho "số phận" (hay máy tính).

Ngay trong quyển sách này các bạn cũng có thể thấy việc sử dụng rộng rãi các số ngẫu nhiên trong việc mô phỏng: cung cấp số liệu nhập ngẫu nhiên hay tùy ý cho các chương trình. Một ứng dụng khác nữa là việc tạo thuận lợi bằng cách dùng các số ngẫu nhiên để thử hay giúp cho việc "tạo quyết định". Ví dụ chính là thuật toán Quicksort và truy tìm chuỗi Rabin-Karp (chương 19).

PHƯƠNG PHÁP ĐỒNG DƯ TUYẾN TÍNH

Là phương pháp nổi tiếng nhất để tạo số ngẫu nhiên, được sử dụng gần như độc chiếm kể từ khi D. Lehner đưa ra vào năm 1951 Đoạn chương trình sau tạo ra N số ngẫu nhiên cho mảng A:

```
a[0] : = seed;
for i: = 1 to N do a[i]:=(a[i-1]*b + 1 ) mod m;
```

Với ba hằng số *seed*, *b* và *m*.

Trong thuật toán này, để tạo một số ngẫu nhiên mới, ta dùng số trước đó nhân với *b*, cộng thêm *1*, và lấy phần dư trong phép chia (kết quả) cho *m*. Như vậy số nhận được luôn nằm trong đoạn từ 0 đến *m-1*. Điều này rất thích hợp để sử dụng trong máy vi tính, bởi vì hàm *mod* thường dễ cài đặt: nếu chúng ta không xét đến tình huống

tràn trong các phép toán số học, khi mà hầu hết các máy vi tính đều loại các bit bị tràn và vì vậy thực hiện rất hiệu quả phép toán *mod* với m bằng một số lớn hơn giá trị tối đa của một *word* trên máy. Hơn nữa, các số thì không thực sự ngẫu nhiên, chương trình chỉ tạo ra các số mà chúng ta hy vọng sẽ xuất hiện ngẫu nhiên cho vài tiến trình khác.

Trông thì có vẻ đơn giản, nhưng phương pháp tạo số ngẫu nhiên đồng dư tuyến tính đã là chủ đề của nhiều tập sách toán khó và chi tiết. Chúng cung cấp cho chúng ta một số hướng dẫn trong việc chọn các hằng số *seed*, b và m. Vài nguyên tắc tổng quát được vận dụng, nhưng trong trường hợp này sự tổng quát không đủ để bảo đảm các số ngẫu nhiên tốt.

Trước hết, m nên lớn, nó có thể là giá trị tối đa của một *word*, nhưng cũng không cần phải hoàn toàn lớn như vậy nếu không tiện. Thông thường, chọn m là một lũy thừa của *10* hay *2* là thuận lợi. Thứ hai b không nên quá lớn hay quá nhỏ: một lựa chọn an toàn là dùng một số có ít hơn m một chữ số. Thứ ba, b nên là một hằng số tùy ý, không theo một mẫu riêng nào cả, ngoại trừ nó nên kết thúc bởi ...x21, với x chẵn, là yêu cầu cuối cùng, một đặc quyền phải được thừa nhận, nhưng nó tránh được một vài sự cố có thể xảy ra mà các phân tích toán học còn để hở.

Các quy luật nêu trên được phát triển bởi D.E. Knuth. Knuth chứng minh rằng các sự lựa chọn này làm cho phương pháp đầy đủ tuyến tính tạo ra các số ngẫu nhiên tốt, thỏa mãn được nhiều kiểm tra thống kê phức tạp. Vấn đề có khả năng nghiêm trọng nhất, là tạo ra một chu kỳ nhỏ so với miền xác định của nó. Ví dụ như với $b=19$, $m=381$, $seed=0$, sẽ tạo ra chuỗi *0,1,20,0,1,20,...* một chuỗi không ngẫu nhiên trong khoảng từ *0* đến *380*.

Đáng tiếc là không phải tất cả các khó khăn như vậy đều dễ nhận ra. Vì vậy tốt nhất là nên theo các chỉ dẫn do Knuth đưa ra, để tránh được các cạm bẫy tinh tế mà ông đã khám phá được.

Giá trị khởi động bất kỳ nào cũng có thể sử dụng được trong việc tạo các số ngẫu nhiên mà không có ảnh hưởng gì đặc biệt (dĩ nhiên là ngoại trừ việc giá trị khởi động khác nhau thì tạo thành các chuỗi ngẫu nhiên khác nhau). Thường thì không cần phải lưu trữ cả chuỗi như trong chương trình nêu trên. Ngược lại, chúng ta đơn giản giữ lại trong một biến toàn cục a, khởi động với một giá trị nào đó, rồi cập nhật bằng phép tính a:=(a*b+1) mod m.

Trong Pascal (và nhiều ngôn ngữ lập trình khác), chúng ta còn một bước nữa mới có thể thực hiện được, bởi vì chúng ta không được phép bỏ qua tình trạng tràn: nó được định nghĩa là một trường hợp lỗi mà có thể tạo ra các kết quả không dự đoán được. Giả sử rằng máy vi tính của chúng ta có *word 32 bit*, và chúng ta chọn $m=100000000$, $b=31415821$, và khởi động $a=1234567$. Tất cả các giá trị này đều nhỏ hơn giá trị tối đa của một số nguyên, nhưng phép toán đầu tiên $a*b+1$ đã làm tràn. Phần của kết quả đã gây ra sự tràn thì không có quan hệ với sự tính toán của chúng ta bởi vì chúng ta chỉ quan tâm đến 8 ký số sau. Một thủ thuật để loại bỏ sự tràn là phân nhỏ phép nhân ra làm nhiều phần. Để nhân p và q, chúng ta viết: $p = 10^4 p_1 + p_0$ và $q = 10^4 + q_1 + q_0$

Do đó kết quả là:

$$pq=(10^4p_1+p_0)(10^4q_1+q_0)=10^4p_1q_1+10^4(p_1q_0+p_0q_1)+p_0q_0$$

Bây giờ, chúng ta chỉ muốn 8 ký số cho kết quả, vì thế chúng ta có thể bỏ qua số hạng đầu tiên ($10^4p_1q_1$) và bỏ qua 4 ký số đầu của số hạng thứ nhì (10^4 ($p_1q_0+p_0q_1$)) Điều này dẫn đến chương trình ở trang sau.

Hàm *mult* trong chương trình này tính *(p*q mod m)*,không bị tràn khi mà m nhỏ hơn 1/2 giá trị số nguyên tối đa. Kỹ thuật này hiển nhiên có thể đáp ứng với các giá trị m1 khác theo nguyên tắc $m = m1 * m1$.

Khi chạy chương trình với dữ liệu nhập $N=10$ và $q=1234567$,

```
program random (input,output);
   const m=100000000; m1=10000;b=3141581;
   var i,a,N:integer;
   function mult(p,q:integer):integer;
      var p1,p0,q1,q0:integer;
      begin
         p1:=p div m1; p0:=p mod m1;
         q1:=q div m1; q0:=q mod m1;
         mult:=(((p0*q1+p1*q0)mod m1)*m1 +p0q0) mod m;
      end;
   function random:integer;
      begin
         a:=(mult(a,b)+1) mod m;
         random:=a;
      end;
begin
   readln(N,a);
   for i:=1 to N do writeln(random);
end.
```

chương trình này sẽ tạo được *10* số như sau *35884508, 80001069, 63512650, 43635651, 1034472, 87181513, 6917174, 209855, 67115956, 59939877*. Trong các số này, có vài sự không ngẫu nhiên: ví dụ, các ký số sau cùng (hàng đơn vị) xoay vòng qua các ký số từ 0 đến 9. Dễ dàng chứng minh được điều này sẽ xảy ra từ công thức. Nói chung các ký số bên phải không thật sự ngẫu nhiên, thực tế đó là nguồn gốc của các lỗi thông thường và quan trọng trong việc sử dụng phương pháp tạo số ngẫu nhiên đồng dư tuyến tính.

Chương trình sau đây là một chương trình dở, tạo các số ngẫu nhiên trong giới hạn *[0,r-1]*:

```
function randombad(r:integer):integer;
begin   a:=(mult(b,a)+1) mod m;
        randombad:= a mod r;
end;
```

Các ký số không ngẫu nhiên bên phải là các ký số duy nhất được sử dụng, nên chuỗi kết quả chỉ có được ít thuộc tính mong muốn. Vấn đề này có thể dễ dàng khắc phục bằng cách dùng các ký số bên trái thay thế.

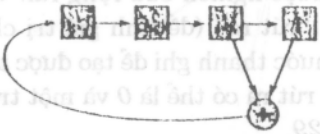
Chúng ta muốn tính một con số giữa *0* và *r-1* bằng công thức *a*r div m*, nhưng một lần nữa, lại phải tránh tình huống bị tràn, như trong phần cài đặt sau.

```
function randomint(r:integer):integer;
begin   a:=(mult(a,b)+1) mod m;
        randomint:=((a div m1)*r)div m1;
end;
```

Một kỹ thuật thông thường khác là tạo số thực ngẫu nhiên giữa *0* và *1* bằng cách xem các số tạo như trên là phần thập phân với chấm thập phân ở bên trái. Điều này có thể cài được dễ dàng bằng cách trả về giá trị thực *(real) a/m* thay vì số nguyên *a*. Khi đó người sử dụng có thể nhận được một số nguyên trong giới hạn *[0,r)* bằng cách đơn giản nhân giá trị này với *r* và cắt lấy phần nguyên của kết quả. Hay có thể nói một số thực ngẫu nhiên giữa *0* và *1* có thể mới chính là các số cần thiết.

PHƯƠNG PHÁP ĐỒNG DƯ CỘNG

Một phương pháp khác để tạo các số ngẫu nhiên đặt cơ sở trên "các thanh ghi dịch chuyển hồi tiếp tuyến tính" được sử dụng trên các



Hình 35.1 *Một thanh ghi dịch chuyển hồi tiếp tuyến tính 4 bit*

máy mã hóa trước đây. Ý tưởng bắt đầu với một thanh ghi chứa một mẫu tùy ý (không phải tất cả *0*), sau đó chuyển sang phải một bước, bỏ vào các vị trí bị trống ở bên trái (do chuyển) bằng một bit được xác định theo nội dung của thanh ghi.

Hình 35.1 mô tả thanh ghi dịch chuyển hồi tiếp tuyến tính 4 bit, với bit mới được tạo theo công thức: bằng *eXclusive OR* (XOR) của hai bit phải nhất của thanh ghi đó. Ví dụ: Nếu thanh ghi được khởi tạo là *1111* thì sau dịch chuyển, nội dung là *0111* (*0=1 XOR 1*),tiếp theo sẽ là *0011, 0001, 1000, 0100, 0010, 1001, 1100, 0110, 1011, 0101, 1010, 1101, 1110, 1111*. Khi chúng ta nhận lại mẫu khởi tạo, tiến trình hiển nhiên đã lập lại (xoay vòng). Chú ý rằng tất cả các mẫu bit có thể có (không phải *0*) đều xuất hiện: giá trị khởi đầu lập lại sau 15 bước.

Tuy nhiên, nếu chúng ta thay đổi các vị trí "rút ra" (các bit dùng để tính giá trị cho hồi tiếp bên trái) thành bit *0* và *2* thay vì *0* và *1* như trên (thứ tự tính từ phải sang trái) thì chúng ta nhận được chuỗi *1111, 0111, 0011, 1001, 1100, 1110, 1111,* không phải chu kỳ đầy đủ như trước. Chúng ta lại muốn bảo đảm rằng luôn nhận được một chu kỳ đầy đủ.

Thông thường với thanh ghi n-bit chúng ta có thể sắp xếp để chiều dài vòng lập là 2^n*-1*. Vì vậy, với giá trị *n* lớn, các thanh ghi n-bit tạo được chuỗi ngẫu nhiên khá tốt. Điển hình, chúng ta thường hay dùng *n=31 hay n=63*.

Cũng như phương pháp đồng dư tuyến tính, các đặc tính toán học của các thanh ghi này đã được nghiên cứu rộng rãi. Ví dụ, có nhiều phương án lựa chọn vị trí "rút ra" (để tính giá trị cho bit trái) áp dụng cho nhiều loại kích thước thanh ghi để tạo được tất cả các mẫu. Ví dụ, với $n=31$, các vị trí rút ra có thể là *0* và một trong các vị trí: *4, 7, 8, 14, 19, 25, 26* hay *29*.

Các giá trị liên tiếp của thanh ghi không hữu dụng như một chuỗi ngẫu nhiên, bởi vì hầu như chỉ có một bit gối lên nhau trong một cặp giá trị liên tiếp. Hơn nữa, thực hiện theo kiểu này giống như tạo một chuỗi bit ngẫu nhiên (bit trái nhất của thanh ghi), trong ví dụ nêu trên là chuỗi *1000100110110111*. Như đã đề cập ở chương 23, phương pháp này thích hợp với thao tác *cryptography* bởi vì có thể tạo ra các chuỗi bit dài từ các khóa nhỏ.

Một vấn đề cần chú ý khác là có thể tính toán mỗi lần một word thay vì mỗi lần một bit theo cùng phương pháp đệ quy như trên. Trong ví dụ trên, nếu sử dụng phép toán bit XOR cho hai *word* kế tiếp, chúng ta tạo được một word sẽ xuất hiện tại ba vị trí sau đó trong danh sách. Điều này giúp ta có được một phương pháp tạo số ngẫu nhiên có thể dễ dàng áp dụng cho máy vi tính. Dùng thanh ghi dịch chuyển hồi tiếp với hai bit lấy ra tính toán là bit thứ *b* và thứ *c*, cũng giống như dùng công thức đệ quy sau:

$a[k] = (a[k-b]+a[k-c]) \bmod m$

Để thích hợp với phương pháp thanh ghi dịch chuyển, phép toán cộng (+) trong công thức này nên được thay thế bằng phép toán trên bit *eXclusive or* (XOR). Tuy nhiên, điều này chứng tỏ rằng có thể tạo được các số ngẫu nhiên tốt ngay cả khi chỉ dùng phép cộng số nguyên thông thường mà thôi. Phương pháp này gọi là đồng dư cộng (*additive congruential*).

Bit phải nhất của các số trong phương pháp này cũng giống như các bit trong thanh ghi dịch chuyển hồi tiếp tương ứng. Do đó, số

bước đạt được trước khi bắt đầu lặp lại, ít nhất cũng bằng chiều dài của chu kỳ lặp. Ngoài ra, một số kết quả cũng đã được đưa ra về các số được tạo theo kiểu này: cơ sở để tin vào chúng chủ yếu theo kinh nghiệm (chúng vượt qua được các kiểm tra thống kê).

Để cài đặt chương trình tạo số ngẫu nhiên theo phương pháp đồng dư cộng, chúng ta cần giữ một bảng gồm c phần tử, luôn chứa c số được tạo gần nhất. Việc tính toán được tiếp tục bằng cách thay thế một trong các số trong bảng bằng tổng hai số khác trong bảng. Khởi đầu, bảng nên gồm các số không lớn quá và cũng không nhỏ quá (một cách tạo dễ dàng là dùng phương pháp đồng dư tuyến tính).

Knuth khuyên nên chọn *b=31, c=55*. Khi đó, chúng ta cần phải giữ lại 55 số được tạo gần nhất. Cấu trúc dữ liệu thích hợp là dùng hàng đợi (*queue*, xem chương 3), nhưng bởi vì kích thước của bảng cố định nên chúng ta chỉ dùng một mảng kích thước đó, với chỉ số xoay vòng như sau:

```
procedure randinit(s:integer);
   begin   a[0]:=s; j:=0;
           repeat j:=j+1; a[j]:=(mult(b,a[j-1])+1) mod m;
           until j=54;
   end;
function randomint(r:integer):integer;
   begin   j:=(j+1) mod 55;
           a[j]:=a[(j+23) mod 55]+a[(j+54) mod 55]) mod m;
           randomint:=((a[j] div m1)*r) div m1;
   end;
```

Biến toàn cục *a* được thay bằng một mảng và một con trỏ chỉ số *(j)* trên mảng. Dung lượng lớn của biến toàn cục (mảng *a*) là một khuyết điểm của phương pháp tạo này trong một số ứng dụng.

Nhưng nó cũng chính là một lợi điểm, bởi vì nó làm cho chu kỳ lặp rất dài (ít nhất là 2^{55}-*1* ngay cả khi *m* nhỏ).

Hàm *randomint* trả về một số nguyên giữa *0* và *r-1*. Và dĩ nhiên, chúng ta cũng có thể dễ dàng thay đổi như phần trước để trả về một số thực ngẫu nhiên giữa *0* và *1* (*a[j]/m*).

KIỂM TRA SỰ NGẪU NHIÊN

Thường có thể dễ dàng nhận ra một chuỗi không ngẫu nhiên, nhưng thật khó chứng minh rằng một chuỗi là ngẫu nhiên. Như đã đề cập trước đây, không có chuỗi nào được tạo bằng máy vi tính có thể thực sự ngẫu nhiên, nhưng chúng ta có thể có được một chuỗi thể hiện nhiều đặc tính của các số ngẫu nhiên. Rủi thay, thường không thể xác định chính xác thuộc tính nào của số ngẫu nhiên là quan trọng đối với một trình ứng dụng đặc biệt. Hơn nữa, lúc nào cũng nên nghĩ đến việc thực hiện một vài kiểu kiểm tra trên phương pháp tạo số ngẫu nhiên, để đảm bảo rằng không có những tình huống suy thoái xảy ra. Các phương pháp tạo số ngẫu nhiên có thể rất tốt, nhưng khi nó xấu thì sẽ rất kinh khủng.

Nhiều kiểm tra đã được triển khai để xác định một chuỗi có được nhiều thuộc tính của chuỗi ngẫu nhiên thực sự hay không. Hầu hết các kiểm tra này đều có cơ sở thực sự giá trị trong toán học và rất nên xem xét thêm chúng chi tiết ngoài phạm vi của quyển sách này.

Trong số đó, kiểm tra *chi-bình phương*, một kiểm tra thống kê, có bản chất cơ bản, dễ cài đặt và hữu dụng trong nhiều ứng dụng. Vì vậy, chúng ta sẽ xem xét nó chi tiết hơn.

Ý tưởng của phương pháp *chi-bình phương* là xét xem các số tạo ra có phân bố một cách hợp lý hay không. Nếu chúng ta tạo *N* số dương có giá trị nhỏ hơn *r*, thì chúng ta mong muốn nhận được khoảng *N/r* số cho mỗi giá trị. Nhưng, và đây cũng chính là cốt lõi của vấn đề, số lần xuất hiện của tất cả các giá trị lại không nên lúc

nào cũng bằng nhau, vì như thế thì không ngẫu nhiên !

Điều này dẫn đến việc cần tính toán xem một chuỗi các số có được phân bố giống như một chuỗi ngẫu nhiên hay không, đơn giản như chương trình sau:

```
function chisquare(N,r,s:integer):real;
  var i,t:integer; f: array[0..rmax] of integer;
  begin   raninit(s);
          for i:=0 to rmax do f[i]:=0;
          for i:=1 to N do
          begin   t:=randomint(r);   f[t]:=f[t]+1;
          end;
          t:=0;
          for i:=0 to r-1 do t:=t+f[i]*f[i];
          chisquare:=((r*t/N)-N);
  end;
```

Chúng ta đơn giản tính tổng số của bình phương tần số (số lần) xuất hiện của mỗi giá trị, chia cho tần số mong muốn (N/r), rồi trừ bớt chiều dài chuỗi (N).

Con số này, giá trị thống kê *chi-bình phương*, có thể biểu diễn theo toán như sau:

$$X^2 = \frac{\sum(f_i - N/r)^2}{N/r}$$

Với $0<i<r$. Nếu thống kê chi-bình phương gần với r thì các số đó là ngẫu nhiên, ngược lại, nếu nó xa r thì chúng không ngẫu nhiên. Khái niệm gần và xa có thể được định nghĩa chính xác hơn: các bảng hiện hữu thể hiện chính xác làm thế nào liên hệ giữa sự thống kê với các thuộc tính của chuỗi ngẫu nhiên.

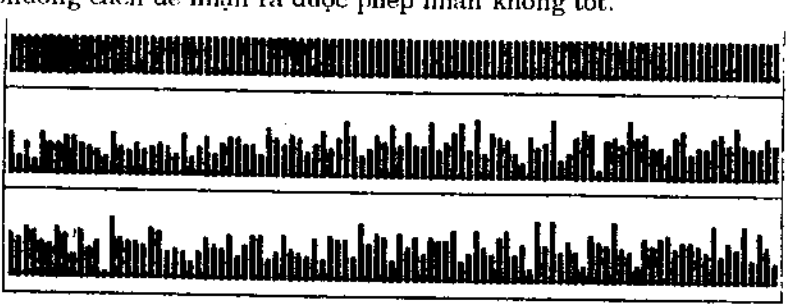
Đối với phương pháp kiểm tra đơn giản mà chúng ta đang thực

hiện, giá trị thống kê chỉ nên lệch một khoảng $2\sqrt{r}$ so với r. Điều này chỉ đúng nếu N lớn hơn $10r$. Để bảo đảm, nên thực hiện kiểm tra vài lần, bởi vì nó có thể sai *1* trong *10* lần kiểm tra.

Phương pháp kiểm tra này cài đặt đơn giản nên thích hợp cho việc kết nó vào bên trong mỗi chương trình tạo số ngẫu nhiên, chỉ để đảm bảo rằng không có điều gì ngoài dự tính có thể gây ra các vấn đề nghiêm trọng. Tất cả các thuật toán "tốt" mà chúng ta đã thảo luận đều vượt qua được các kiểm tra này, phương pháp "xấu" thì không.

Sử dụng các phương pháp tạo nêu trên cho *1000* số có giá trị nhỏ hơn *100*, chúng ta có giá trị thống kê chi-square là *100.8* với phương pháp đồng dư tuyến tính và *105.4* với phương pháp đồng dư cộng. Cả hai đều đạt trong vùng lệch *20* so với *100* (nghĩa là trong *[100-20,100+20]*). Nhưng với phương pháp "xấu" mà sử dụng các bit bên phải từ phương pháp đồng dư tuyến tính (*101011*), thì giá trị thống kê là *77.8* hiển nhiên là vượt khỏi miền xác định tốt (nêu trên).

Hình 35.2 biểu diễn tần số xuất hiện của mỗi giá trị *100* trong ba bảng của phương pháp đồng dư tuyến tính đã đề cập. Dùng các bit bên trái (biểu đồ trên cùng) thì rất xấu, còn hai biểu đồ còn lại thì không khác nhau lắm: thống kê chi-bình phương cung cấp một phương cách để nhận ra được phép nhân không tốt.



Hình 35.2 *Tần số xuất hiện của mỗi giá trị 100 trong ba bảng của phương pháp đồng dư tuyến tính*

CÁC LƯU Ý KHI CÀI ĐẶT

Một số lợi điểm thường được thêm vào để làm cho các thao tác tạo số ngẫu nhiên hữu dụng đối với nhiều loại trình ứng dụng. Người ta thường tạo một hàm, khởi động nó, rồi gọi lặp lại nhiều lần, mỗi lần trả về một số ngẫu nhiên khác. Một cách khác là chỉ gọi một lần và nó tạo ra dãy các số ngẫu nhiên cho trình ứng dụng. Trong cả hai trường hợp trên, người ta đều mong muốn nhận được các chuỗi giống nhau trong các lời gọi kế tiếp (để tìm lỗi hay so sánh các chương trình khác nhau trên cùng dữ liệu nhập) và tạo được một chuỗi tùy ý. Tất cả các điểm nêu trên đều đòi hỏi phải dùng đến trạng thái được giữ lại giữa các lời gọi của hàm tạo số ngẫu nhiên. Điều này có thể không thuận lợi trong một số môi trường lập trình. Phương pháp cộng bất lợi là có một trạng thái khá lớn (mảng giữ các word vừa tạo)nhưng lại có một thuận lợi là có một chu kỳ dài như vậy, mà có thể không cần phải khởi tạo.

Một cách thận trọng để tránh các sai lệch xảy ra trong thao tác tạo số ngẫu nhiên là liên kết hai phương pháp với nhau (ví dụ như dùng phương pháp đồng dư tuyến tính để khởi tạo mảng trong phương pháp đồng dư cộng). Một cách thực hiện phương pháp tạo kết nối dễ dàng là dùng phương pháp thứ nhất tạo một bảng và thao tác thứ hai chọn các vị trí bằng ngẫu nhiên để lấy ra các số cho kết xuất.

Khi tìm lỗi cho một chương trình, nên dùng trước một cách tạo số ngẫu nhiên bị thoái hoá (ví dụ luôn trả về *0* hay chuỗi có thứ tự).

Như một quy luật, các công cụ tạo số ngẫu nhiên thì dễ hỏng, và cần phải xem xét cẩn thận. Nếu không thực hiện nhiều loại kiểm tra thống kê thì khó bảo đảm một phương pháp tạo số ngẫu nhiên là tốt. Trên nguyên tắc, để sử dụng được một công cụ tạo số ngẫu nhiên tốt thì phải làm hết sức mình, đặt cơ sở trên các phân tích toán học và các kinh nghiệm khác; chỉ để bảo đảm rằng các số trông có vẻ ngẫu nhiên. Nếu có gì sai sót, hãy xét lại các công cụ tạo số ngẫu nhiên.

BÀI TẬP

1. Viết một chương trình tạo ra các từ bốn ký tự. Ước tính xem chương trình của bạn tạo ra được bao nhiêu từ thì lặp lại.

2. Dựa theo trò chơi thảy hai con xúc xắc rồi tính tổng nhận được để tạo ra số ngẫu nhiên, làm phức tạp thêm bằng cách cho phép con xúc xắc không chuẩn (có thể có số 1,2,3,5,8 và 13).

3. Tính chuỗi các mẫu tạo ra bởi phương pháp thanh ghi dịch chuyển hồi tiếp tuyến tính như trong hình 35.1, nhưng với vị trí rút ra là bit đầu tiên và bit cuối cùng. Cho mẫu khởi tạo là 1111.

4. Tại sao không dùng OR hay AND (thay vì XOR) trong phương pháp thanh ghi chuyển hồi tiếp tuyến tính.

5. Viết chương trình tạo ra một hình ngẫu nhiên trong không gian hai chiều (ví dụ: tạo ra các bits ngẫu nhiên, vẽ "*" nếu là 1, " " nếu là 0; Ví dụ khác: dùng các số ngẫu nhiên như là toạ độ trong hệ trục Descartes hai chiều và vẽ một ký tự "*" tại điểm đó.

6. Dùng thuật toán tạo số ngẫu nhiên đồng dư cộng để tạo ra 1000 số nguyên ngẫu nhiên dương nhỏ hơn 1000. Thiết kế và áp dụng một cách kiểm tra xem chúng có ngẫu nhiên hay không.

7. Tương tự bài 6 với phương pháp đồng dư tuyến tính và các tham số bạn tự chọn.

8. Tại sao không nên sử dụng, ví dụ như $b=3$ và $c=6$, với phương pháp tạo đồng dư cộng ?

9. Giá trị của phương pháp kiểm tra *chi-square* là bao nhiêu khi thuật toán luôn tạo ra các số như nhau.

10. Bạn làm thế nào để tạo ra các số ngẫu nhiên với m lớn hơn kích thước word.

36

SỐ HỌC

Mặc dù đang có nhiều thay đổi, nhưng lý do để tồn tại của nhiều hệ thống máy tính vẫn là khả năng thực hiện nhanh, chính xác các phép tính trên những con số. Máy tính đã được chế tạo có khả năng thực hiện những tính toán số học trên các số nguyên và số thực kiểu chấm động. Ví dụ ngôn ngữ Pascal có hai kiểu số là integer và real, cùng với tất cả các phép tính thông thường trên hai kiểu đó. Phương pháp hiện nay sử dụng cho các phép tính số học là một phần trong việc thiết kế và kiến tạo máy tính và không phải là điều mà chúng ta quan tâm tới ở đây (mặc dù một thuật toán mới, nhanh dùng cho việc nhân hai số nguyên 32 bits vẫn có thể có tầm quan trọng lớn). Thay vào đó, chúng ta sẽ thảo luận về một số thuật toán áp dụng cho các phép toán trên những đối tượng toán học phức tạp hơn.

Trong chương này, chúng ta sẽ xem xét việc lập trình bằng ngôn ngữ Pascal những thuật toán cho phép tính cộng và nhân trên các đa thức, số nguyên dài và ma trận. Những thuật toán sơ cấp trên các vấn đề này đều quen thuộc, dễ hiểu và dễ làm, nhưng việc vận dụng các cấu trúc dữ liệu cơ sở trong những trường hợp đặc biệt cũng là một vấn đề đáng nghiên cứu. Chúng ta sẽ không nhấn mạnh đến những áp dụng như thường lệ, mà sẽ xem xét tính chất phức tạp trong tính toán của các vấn đề số học căn bản; bởi vì số học cho chúng ta một ví dụ tuyệt hảo về việc thế nào là vận dụng đúng đắn suy nghĩ thuật toán để tạo ra được những phương pháp tinh vi thực sự hiệu quả hơn các phương pháp sơ cấp. Nghiên cứu này thật lý thú,

bởi vì bản chất lịch sử của vấn đề: các thuật toán trong việc thực hiện những phép tính số học sơ cấp như cộng, nhân và chia có một lịch sử hết sức xa xưa, bắt nguồn từ những nghiên cứu thuật toán trong công trình của nhà toán học người Ả Rập al-Khowarizmi, với cội rễ nền tảng còn sâu xa hơn nữa từ thời những người Hy lạp và Balylonia (một đế quốc Tây Nam Á).

Chúng ta sẽ thấy, phép nhân đa thức và đặc biệt là nhân ma trận cho ta những ví dụ chuẩn mực về sức mạnh của kiểu "chia để trị". Rủi thay, những thuật toán hiệu quả này (với ngoại lệ đặc biệt quan trọng của phương pháp đã được bàn luận trong chương 41) lại khó thực hành; những phương pháp sơ cấp sử dụng những cấu trúc dữ liệu cơ sở tất nhiên được xem là tốt nhất, ngoại trừ đối với những vấn đề quá rộng.

SỐ HỌC VỀ ĐA THỨC

Giả sử chúng ta muốn viết một chương trình cộng hai đa thức, nghĩa là muốn thực hiện những phép tính như:

$(1+2x-3x^3)+(2-x)=3+x-3x^3$

Một cách tổng quát, giả sử chúng ta muốn chương trình có thể tính $r(x)=p(x)+q(x)$, trong đó p và q là các đa thức có N số hạng. Thực hiện việc đó rất đơn giản với một biểu diễn mảng (*array*). Chúng ta lưu trữ đa thức $p(x)=p_0+p_1x+...+p_{N-1}x^{N-1}$ bằng **array** *p[0..n-1]* với *p[j]=pj* ,.v.v. Sau đó, phép cộng không có gì ngoài chương trình có dòng:

---

for *i:=0* **to** *N-1* **do** *r[i]:=p[i]+q[i];*

Thông thường, trong ngôn ngữ Pascal (xem chương 3) chúng ta phải quyết định trước con số N sẽ có thể lớn tới cỡ nào để khai báo mảng p, q và r.

Chương trình trên cho thấy khi đã chọn cách biểu diễn các đa thức bằng mảng thì phép cộng thực sự đơn giản, các phép tính khác cũng có thể viết dễ dàng. Thí dụ phép nhân đa thức có thể thực hiện bằng đoạn chương trình sau:

```
for i:=0 to 2*(N-1) do r[i]:=0;
for i:=0 to N-1 do
    for j:=0 to N-1 do r[i+j]:=r[i+j]+p[i]*q[j];
```

Mảng r phải được khai báo lớn gấp hai lần số số hạng của đa thức tích. Mỗi số hạng trong N số hạng của p được nhân với từng số hạng trong N số hạng của q, vì vậy số lần lặp của thuật toán này rõ ràng là bình phương số số hạng của đa thức.

Như chúng ta đã thấy ở chương 3, việc biểu diễn một đa thức bằng một mảng chứa các hệ số của nó có lợi điểm là dễ dàng tham chiếu trực tiếp một hệ số bất kỳ, nhưng lại bất lợi ở chỗ phải khai báo dự trữ chỗ cho các con số nhiều hơn số lượng cần thiết. Chẳng hạn như chương trình trên rõ ràng không nên dùng để nhân hai đa thức sau:

$$(1+x^{10000})(1+2x^{10000})=1+3x^{10000}+2x^{20000}$$

mặc dù, dữ liệu nhập chỉ có bốn số hạng và kết quả xuất chỉ có ba số hạng.

Chúng ta có thể biểu diễn đa thức bằng cách sử dụng các danh sách liên kết (*linked lists*) và cộng chúng như chương trình *add* ở trang sau.

Các đa thức nhập được thể hiện bằng các danh sách kết nối với mỗi phần tử của danh sách tương ứng với một số hạng trong đa thức; đa thức xuất là kết quả của thủ tục cộng. Thao tác trên các liên kết hoàn toàn tương tự như chúng ta đã thấy ở các chương 3, 8, 14, 29 và những chỗ khác trong quyển sách này.

```
function add(p,q:link):link;
   var t:link;
   begin    t:=z;
            repeat
               new(t↑.next); t:=t↑.next;
               t↑.c:=p↑.c+q↑.c;
               p:=p↑.next; q:=q↑.next
            until (p=z) and (q=z);
            t↑.next:=z; add:=z↑.next;
   end;
```

Chương trình trên không cải tiến thực sự đối với việc biểu diễn mảng, ngoại trừ điểm nó khéo léo xử lý sự thiếu mảng động (*dynamic array*) trong Pascal (vì cũng hao tốn chỗ cho các kết nối trên từng số hạng). Tuy nhiên, như ví dụ trên cho thấy, chúng ta có thể lợi dụng khả năng có thể có nhiều số hạng là zero (*0*). Chúng ta có thể tạo những phần tử của danh sách (*list nodes*) chỉ biểu diễn cho các số hạng có hệ số khác *0* của đa thức bằng cách kê cả bậc của số hạng được biểu diễn trong phần tử của danh sách; nghĩa là mỗi phần tử của danh sách chứa các giá trị của *c* và *j*, để đại diện cho cx^j. Khi đó, việc tách chức năng: tạo một node và thêm node đó vào danh sách rất thuận tiện như dưới đây:

```
type link=↑node;
     node=record c:real; j:integer; next:link end;
function listadd(t:link; c:real; j:integer):link;
   begin    new(t↑.next); t:=t↑.next; t↑.c:=c; t↑.j:=j;
            listadd:=t;
   end;
```

Hàm *listadd* tạo ra một node mới, gán cho node các field đặc tả (*c*,*j*) và kết nối vào danh sách sau node *t*. Để có thể thực hiện các thao tác trên đa thức một cách có tổ chức, các danh sách có thể được sắp xếp theo thứ tự tăng dần của bậc (*j*).

Bây giờ, hàm *add* trở nên lý thú hơn, bởi vì nó phải thực hiện một phép cộng chỉ dành cho các số hạng có cùng bậc và đảm bảo rằng không tạo ra một số hạng nào có giá trị *0*.

```
function add(p,q:link):link;
  begin
    t:=z;
    repeat
      if (p↑.j=q↑.j) and (p↑.c+q↑.c<>0.0)
      then begin   t:=listadd(t,p↑.c+q↑.+c,p↑.j);
                   p:=p↑.next; q:=q↑.next
           end
      else if p↑.j<q↑.j
           then begin t:=listadd(t,p↑.c,p↑.j); p:=p↑.next
                end
           else if q↑.j<p↑.j
                then begin t:=listadd(t,q↑.c,q↑.j);
                           q:=q↑.next
                     end;
    until (p=z) and (q=z);
    t↑.next:=z; add:=z↑.next
  end;
```

Xử lý này cần những thao tác tinh tế để giải quyết những đa thức bị kéo giãn do có nhiều phần tử *0*. Bởi vì, không gian và thời gian cần cho việc xử lý các đa thức thì tỷ lệ với số lượng số hạng, chứ không phải bậc của đa thức. Cũng có những tiết kiệm tương tự như vậy cho các phép tính khác trên đa thức, ví dụ như phép nhân, thế nhưng phải đề phòng vì các đa thức có thể bị "suy thoái" sau một số lượng các phép toán như vậy được thực hiện. Việc biểu diễn bằng mảng sẽ tốt hơn nếu đa thức có ít thành phần có hệ số *0*, hoặc nếu bậc không cao. Theo chúng tôi biểu diễn này là để đơn giản hóa việc miêu tả nhiều thuật toán trên các đa thức sẽ được trình bày dưới đây.

Một đa thức có thể không chỉ có một mà nhiều biến, thí dụ cần phải xử lý các đa thức như:

$1+wx^2+y^6z+w^{25}x^{50}y^{99}z^{38}+x^{1000}z^{1000}$

Trong những trường hợp này, dứt khoát phải cần đến cách biểu diễn bằng danh sách liên kết: việc lựa chọn (các mảng nhiều chiều) có thể đòi hỏi rất nhiều chỗ. Mở rộng chương trình *listadd* ở trên để xử lý các đa thức như vậy không khó khăn gì.

NỘI SUY VÀ TÍNH GIÁ TRỊ ĐA THỨC

Chúng ta hãy xem tính giá trị của một đa thức tại một điểm xác định như thế nào. Thí dụ để tính:

$p(x)=x^4+3x^3-6x^2+2x+1$

cho một giá trị x bất kỳ, ta có thể tính x^4, rồi cộng với $3x^3$... Phương pháp này đòi hỏi phải lập lại việc tính các lũy thừa của x, chúng ta có thể lưu lại kết quả mỗi lần tính lũy thừa của x, nhưng như thế lại cần lưu trữ quá nhiều.

Một phương pháp đơn giản tránh được tình trạng lập lại nhiều lần một con tính và không sử dụng quá nhiều chỗ lưu trữ được gọi là quy tắc Horner: lần lượt thực hiện các phép tính nhân và cộng tương ứng, một đa thức bậc N có thể tính được mà chỉ sử dụng $N-1$ phép nhân và N phép cộng. Phương pháp đặt thừa số chung:

$p(x)=x(x(x(x+3)-6)+2)+1$

làm cho thứ tự tính toán rõ ràng:

```
y:=p[N];
for i:=N-1 downto 0 do y:=x*y+p[i];
```

Chúng tôi đã sử dụng một dạng của phương pháp này trong một ứng dụng thực hành hết sức quan trọng: tính các hàm băm của các

khóa dài (*long key*, xem chương 16).

Một vấn đề phức tạp hơn là tính một đa thức ở nhiều điểm khác nhau. Nhiều thuật toán khác nhau phù hợp tùy thuộc vào việc cần phải tính bao nhiêu giá trị và có phải thực hiện cùng một lúc hay không. Nếu phải tính một số lượng lớn các giá trị thì nên thực hiện một vài "cái tính trước", chúng có thể làm giảm phần nào hao tốn phiền toái của các phép tính định trị về sau. Lưu ý rằng phương pháp của Horner đòi hỏi khoảng N bình phương phép tính nhân để tính giá trị một đa thức bậc N ở N điểm khác nhau. Còn có nhiều phương pháp rắc rối, phức tạp hơn nữa đã được xây dựng để giải quyết vấn đề này trong $N(\log N)$ bước và trong chương 41 chúng ta sẽ thấy một phương pháp chỉ sử dụng $N\log N$ phép nhân cho một tập hợp N điểm xác định.

Nếu đa thức chỉ có một phần tử thì vấn đề tính đa thức này chỉ còn là vấn đề lũy thừa: tính x lũy thừa N. Quy tắc Horner trong trường hợp này giảm cấp xuống một thuật toán đơn giản chỉ cần $N-1$ phép nhân. Để thấy được chúng ta có thể làm tốt hơn nhiều, hãy xét chuỗi sau đây để tính x lũy thừa 32:

$$x, x^2, x^4, x^8, x^{16}, x^{32}$$

Mỗi một số hạng có được bằng cách bình phương số hạng trước, cho nên chỉ cần 5 phép nhân thay vì 31 phép nhân.

Phương pháp bình phương liên tiếp có thể được mở rộng một cách dễ dàng cho một N tổng quát nếu như các giá trị tính được được lưu giữ lại. Thí dụ x lũy thừa 55 có thể tính được từ các giá trị nêu trên, và thêm 4 phép nhân nữa:

$$x^{55}=x^{32}.x^{16}.x^4.x^2.x^1$$

Thông thường, cách biểu diễn nhị phân có thể dùng cho N để chọn giá trị nào đã được tính để dùng lại. (Trong ví dụ này, bởi vì $55=(110111)$ nên ngoại trừ x^8, còn lại tất cả sẽ được sử dụng). Các

phép toán bình phương liên tiếp có thể được tính và các bits của N được kiểm tra trong cùng một vòng lặp (*loop*). Có hai phương pháp thực hiện được điều này mà chỉ sử dụng một *accumulator* (biến tích lũy) giống như phương pháp Horner. Một là thuật toán quét biểu diễn nhị phân của N từ trái sang phải, khởi động biến lưu giữ bằng *1*. Tại mỗi bước, bình phương *accumulator* và đồng thời nhân với x nếu giá trị bit tương ứng trong biểu diễn nhị phân của N bằng *1*. Chuỗi giá trị sau đây được tính bằng phương pháp này với $N=55$:

$$1.1\,x.x^2.x^3.x^6.x^{12}.x^{13}.x^{26}.x^{27}.x^{54}.x^{55}$$

Một thuật toán nổi tiếng khác thao tác tương tự như vậy, nhưng quét N từ phải sang trái. Đây là một bài tập nhập môn lập trình chuẩn. Mặc dù khả năng tính toán trên những con số lớn như vậy khó có ích lợi thiết thực, nhưng khi bàn luận về các số nguyên lớn, chúng ta sẽ thấy phương pháp này đóng một vai trò trong việc cài đặt mật mã khóa công khai của chương 23.

Bài toán "ngược" với thao tác tính một đa thức bậc N ở N điểm là việc "nội suy đa thức": cho một dãy N điểm $x^1, x^2, ..., x^N$ và các giá trị tương ứng là $y^1, y^2, ..., y^N$, hãy tìm đa thức bậc $N-1$ duy nhất thỏa $p(x_1)=y_1$, $p(x_2)=y_2$, ..., $p(x_N)=y_N$.

Bài toán nội suy là tìm một đa thức nhờ vào một dãy các điểm và các giá trị đã cho. Bài toán định trị là tìm các giá trị khi cho trước đa thức và các điểm (Bài toán tìm các điểm khi cho biết đa thức và các giá trị được gọi là tìm nghiệm).

Giải pháp chuẩn của vấn đề nội suy dựa trên công thức của Lagrange, thường được sử dụng để chứng tỏ rằng một đa thức bậc $N-1$ hoàn toàn được xác định bởi N điểm:

$$p(x)=\sum_{1\le i\le N} y_i \prod_{1\le i\le N} \frac{x-x_i}{x_j-x_i}$$

Công thức này thoạt nhìn có vẻ ghê gớm nhưng thực tế thật đơn giản. Ví dụ, đa thức bậc 2 có $p(1)=3$, $p(2)=7$ và $p(3)=13$ được cho bởi:

$$p(x)=3\frac{x-2}{1-2}\frac{x-3}{1-3}+7\frac{x-1}{2-1}\frac{x-3}{2-3}+13\frac{x-1}{3-1}\frac{x-2}{3-2}$$

và rút gọn thành:

$$x^2+x+1$$

Với mỗi x từ $x_1, x_2, ..., x_N$, công thức được xây dựng thế nào để có $p(x)=y$ với $1\le x\le N$, bởi vì tích số đều là 0 trừ khi $j=k$, khi tích là 1. Trong thí dụ này, hai số hạng cuối là 0 khi $x=1$, số hạng đầu và cuối là 0 khi $x=2$ và hai số hạng đầu là 0 khi $x=3$.

Chuyển một công thức từ dạng được mô tả theo công thức Lagrange sang cách biểu diễn số hạng chuẩn của chúng ta không dễ làm chút nào. Ít nhất phải có N^2 phép tính, khi có N phần tử trong tổng số. Mỗi số hạng là một tích với N thừa số. Thực sự, để thực hiện một thuật toán bậc hai, cần có ít nhiều linh động, bởi vì các thừa số không chỉ là những con số mà còn là những đa thức bậc N. Hơn nữa, mỗi phần tử rất giống phần tử đi trước. Người đọc hẳn thích thú khám phá rằng có thể lợi dụng điểm này như thế nào để thực hiện thuật toán *quadratic*. Bài tập này còn một vấn đề là sự đánh giá bản chất đặc biệt của việc viết một chương trình thực hiện tính toán theo một công thức toán.

Cũng giống như việc định trị đa thức, có nhiều phương pháp phức tạp có thể giải quyết bài toán trong $N(\log N)$ bước, và trong chương 41, chúng ta sẽ thấy một phương pháp chỉ dùng $N\log N$ phép nhân cho một tập hợp N điểm.

NHÂN ĐA THỨC

Thuật toán số học phức tạp đầu tiên của chúng ta dành cho bài toán nhân đa thức: cho hai đa thức $p(x)$ và $q(x)$, hãy tính $p(x)q(x)$. Như đã

lưu ý ở đầu chương này, các đa thức bậc *N-1* có thể có *N* phần tử (tính luôn hằng số), và kết quả cần tính có bậc *2N-2*, với *2N-1* phần tử. Thí dụ:

$(1+x+3x^2-4x^3)(1+2x-5x^2-3x^3)=(1+3x-6x^3-26x^4+11x^5+12x^6)$

Thuật toán chân phương, giản dị giải quyết bài toán này trình bày ở đầu chương đòi hỏi *N* phép nhân cho các đa thức bậc *N-1*: mỗi một phần tử trong *N* phần tử của *p(x)* được nhân với từng phần tử trong *N* phần tử của *q(x)*.

Để cải tiến thuật toán chân phương này, chúng ta sẽ "chia để trị". Một phương pháp để phân một đa thức ra làm hai là chia đôi các số hạng: cho một đa thức bậc *N-1* (với *N* số hạng), chúng ta có thể phân nó ra thành hai đa thức với *N/2* số hạng (giả sử rằng *N* chia chẵn cho 2): một đa thức sẽ gồm *N/2* số hạng bậc thấp và một đa thức khác sẽ gồm *N/2* số hạng bậc cao. Với $p(x)=p_0+p_1x+...+p_{N-1}x^{N-1}$ ta định nghĩa

$$p_l(x)=p_0+p_1x+...+p_{N/2-1}x^{N/2-1}$$

$$p_h(x)=p_{N/2}+p_{N/2+1}x+...+p_{N-1}x^{N/2-1}$$

Sau đó, cũng dùng cách đó để tách q(x) ra, chúng ta có:

$$p(x)=p_l(x)+x^{N/2}p_h(x)$$

$$q(x)=q_l(x)+x^{N/2}q_h(x)$$

Bây giờ, dựa vào những đa thức nhỏ hơn, kết quả được tính bởi:

$$p(x)q(x)=p_l(x)q_l(x)+(p_l(x)q_h(x)+q_l(x)p_h(x))x^{N/2}+p_h(x)q_h(x)x^N$$

(Chúng ta đã sử dụng cách chia này trong chương 35 để tránh vấn đề tràn.)

Bây giờ, điểm mấu chốt của việc làm này là chỉ cần có ba phép nhân để tính được các kết quả (thay vì 4 như theo công thức trên) bởi vì nếu chúng ta tính

$r_l(x)=p_l(x)q_l(x)$, $r_h(x)=p_h(x)q_h(x)$ và

$r_m(x)=(p_l(x)+p_h(x))(q_l(x)+q_h(x))$ chúng ta có thể đạt được kết quả bằng cách tính:

$$p(x)q(x)=r_l(x)+(r_m(x)-r_l(x)-r_h(x))x^{N/2}+r_h(x)x^N$$

Những tiết kiệm có được có lẽ không thể thấy rõ từ ví dụ nhỏ này. Phương pháp dựa trên cơ sở phép cộng đa thức đòi hỏi một thuật toán tuyến tính, và phép nhân đa thức *brute-force* là bình phương, vì vậy nên thực hiện một ít tính cộng (dễ) để giảm được một phép tính nhân (khó). Chúng ta sẽ xem xét cặn kẽ hơn về những tiết kiệm mà phương pháp này có thể tận dụng.

Trong ví dụ nêu trên, với $p(x)=1+x+3x^2-4x^3$ và $q(x)=1+2x-5x^2-3x^3$ chúng ta có:

$$r_l(x)=(1+x)(1+2x)=1+3x+2x^2$$

$$r_h(x)=(3-4x)(-5-3x)=-15+11x+12x^2$$

$$r_m(x)=(4-3x)(-4-x)=-16+8x+3x^2$$

Do đó, ta có $r_m(x)-r_l(x)=-2-6x-11x^2$, và kết quả được tính như là tổng của ba số hạng theo công thức trên:

$$\begin{aligned}p(x)q(x)&=(1+3x+2x^2)\\&\quad+(-16+8x+3x^2)\\&\quad+(-15+11x+12x^2)\\&=1+3x-6x^3-26x^4+11x^5+12x^6\end{aligned}$$

Phương pháp chia-để-trị này giải quyết vấn đề nhân một đa thức cỡ N bằng cách giải quyết ba bài toán con cỡ $N/2$, dùng vài phép tính cộng đa thức để giải quyết các bài toán con và kết hợp các kết quả lại. (Nếu $N=1$ thì kết quả tính được chỉ là một sản phẩm vô hướng của hai số hạng là hằng). Vì vậy, thủ tục này có thể thực hiện dễ dàng bằng một chương trình đệ quy như ở trang sau.

```
function   mult(p,q:array[0..N-1] of real; N:integer)
           : array [0..2*N-2] of real;
var pl,ql,ph,qh,t1,t2: array [0..(N div 2)-1] of real;
    rl,rm,rh: array [0..N-1] of real;
    i,N2: integer;
begin
if N=1
then mult[0]:=p[0]*q[0]
else begin  N2:=N div 2;
            for i:=0 to N2-1 do
               begin pl[i]:=p[i]; ql[i]:=q[i] end;
            for i:=N2  to N-1 do
               begin ph[i-N2]:=p[i]; qh[i-N2]:=q[i]; end;
            for i:=0 to N2-1 do t1[i]:=pl[i]+ph[i];
            for i:=0 to N2-1 do t2[i]:=ql[i]+qh[i];
            rm:=mult(t1,t2,N2);
            rl:=mult(pl,ql,N2);
            rh:=mult(ph,qh,N2);
            for i:=0 to N-2 do mult[N+i]:=rh[i];
            mult[N-1]:=0;
            for i:=0 to N-2 do mult[Ni]:=rh[i];
            for i:=0 to N-2 do
               mult[N2+1]:=mult[N2+i]+rm[i]-(rl[i]+rh[i]);
     end;
end;
```

Mặc dù đoạn chương trình trên mô tả rõ ràng và ngắn gọn phương pháp này, nhưng không may mắn, nó lại không là một chương trình hợp lệ của ngôn ngữ Pascal bởi vì các hàm không thể khai báo mảng kiểu như vậy. Vấn đề này có thể chuyển sang ngôn ngữ Pascal bằng cách biểu diễn các đa thức bằng các danh sách liên kết; chúng tôi để lại công việc này như một bài tập cho người đọc. Chương trình trên giả thiết N là một bội số của 2, mặc dù các chi tiết đối với N tổng quát có thể giải quyết một cách dễ dàng. Những phức

tạp chủ yếu là đảm bảo được sự đệ qui kết thúc đúng lúc và các đa thức được phân chia thỏa đáng khi N không chia chẵn cho 2.

Tại sao phương pháp chia-để-trị này lại là một cải tiến ? Để tìm được câu trả lời, chúng ta cần giải quyết một công thức đệ quy căn bản, chỉ hơi phức tạp hơn một chút so với công thức trong chương 6.

Tính chất 36.1 *Hai đa thức bậc N có thể nhân với nhau bằng cách sử dụng khoảng N phép nhân.*

Từ chương trình đệ quy, rõ ràng rằng số lượng phép tính nhân số nguyên cần để nhân hai đa thức cấp N bằng với số lượng các phép tính nhân để nhân ba cặp đa thức cỡ $N/2$ (Lưu ý rằng, đối với ví dụ, để tính $r_h(x)x^N$ không cần một phép tính nhân nào, chỉ cần chuyển dữ liệu) Nếu M_N là số lượng phép tính nhân cần làm để nhân hai đa thức cấp N, chúng ta có:

$M_N=3M_{N/2}+1$ cho các $N\geq 2$ với $M_1=1$

Vì vậy, $M(2)=3$, $M(4)=9$, $M(8)=27$,... Như trong chương 6, nếu chúng ta cho $N=2^n$ thì chúng ta có thể áp dụng một lần nữa sự lặp lại của chính nó để tìm ra kết quả:

$$M(2^n)=3M(2^{n-1})=3^2M(2^{n-2})=3^3M(2^{n-3}) = \ldots = 3^nM(1) = 3^n$$

Nếu $N=2^n$ thì $3^n=2^{(\lg 3)n}=2^{n\lg 3}=N^{\lg 3}$ Mặc dù lời giải này chỉ chính xác với $N=2^n$, nhưng nói chung trường hợp tổng quát ta vẫn có: $M_N \approx N^{\lg 3}\approx N^{1.58}$

rõ ràng tiết kiệm hơn so với phương pháp cũ (độ phức tạp N^2).

Lưu ý rằng nếu chúng ta đã sử dụng hết bốn phép tính nhân trong phương pháp chia để trị đơn giản này, thì sẽ có thao tác thực hiện y như phương pháp sơ cấp bởi vì số lần lặp sẽ là $M(N)=4M(N/2)$ cho giải đáp $M(2^n)=4^n=N^2$.

Phương pháp này minh họa rất tốt cho kỹ thuật chia-để-trị, nhưng hiếm khi được sử dụng trong thực tế bởi vì có một phương

pháp chia và trị tốt hơn nhiều mà chúng ta sẽ nghiên cứu tới trong chương 41. Phương pháp này thực hiện bằng cách chỉ chia bài toán gốc ra làm 2 bài toán nhỏ, với một ít xử lý thêm. Nó dẫn tới số lượng phép nhân cần có tính theo công thức đệ quy của chia và trị chuẩn là $M_N=2M_{N/2}+N$ và cho kết quả M_N gần bằng $NlgN$.

CÁC PHÉP TÍNH SỐ HỌC VỚI CÁC SỐ NGUYÊN LỚN

Một số nguyên lớn có thể xử lý như một đa thức, với những ràng buộc hạn chế trên những phần tử. Thí dụ số nguyên 28 chữ số
0120200103110001200004012314
có thể tương ứng với đa thức:

$$x^{26}+2x^{25}+2x^{23}+x^{20}+3x^{18}+x^{17}+x^{16}+x^{12}+2x^{11}+4x^6+x^4+2x^3+3x^2+x+4$$

Khi đó, số nguyên chính là giá trị của đa thức với $x=10$. Ngược lại, bất kỳ đa thức bậc $N-1$ nào với các hệ số dương nhỏ hơn 10 đều tương ứng chính xác với một số nguyên N chữ số.

Do đó, chúng ta có thể sử dụng các phép tính trên đa thức để xử lý các số nguyên lớn. Khi đó, chúng ta có thể biểu diễn các số nguyên một cách đơn giản bằng các mảng, rồi sử dụng các thao tác tính toán thông thường, chỉ nâng cao các thao tác này khi nào mảng lưu trữ đa thức. Thí dụ, để nhân hai số nguyên 100 chữ số, chúng ta có thể sử dụng thuật toán trên để tính được một kết quả 200 chữ số. Khuyết điểm của cách tính này là các hệ số trong kết quả tính được không phải lúc nào cũng nhỏ hơn 10. Có thể điều chỉnh điều này trong một tình huống cá biệt: bắt đầu với $i=0$, cộng $p[i]$ **div** 10 cho $p[i+1]$, thay thế $p[i]$ bằng $p[i]$ **mod** 10, rồi tăng i lên, tiếp tục cho đến khi không còn hệ số nào khác 0.

Có thể sử dụng một cơ số lớn hơn 10. Thí dụ, con số 28 chữ số nói trên cũng có thể tương ứng với đa thức :

$120x^6+2001x^5+311x^4+x^3+2000x^2+401x+2314$

Tính trị đa thức này với $x=10000$ sẽ được số nguyên nói trên. Điều này cho phép số nguyên được biểu diễn với ít bộ nhớ hơn (trong ví dụ này là 1/4), thế nhưng các hệ số lại có khả năng tràn trong các tính toán xử lý trung gian. Toán học đã nghiên cứu cẩn thận vấn đề có thể sử dụng một cơ số lớn đến cỡ nào, nhưng trong thực tế vẫn còn chút ít thiệt hại khi vì cẩn trọng mà dùng một cơ số nhỏ.

Đối với mật mã RSA (Chương 23), chúng ta không chỉ cần nhân các số nguyên lớn, mà còn phải lũy thừa và chia. Đặc biệt, chúng ta cần tính $M^P \bmod N$ khi M, p, và N đều là những số nguyên lớn. Đó không phải là một tính toán đơn giản để thực hiện, nhưng chúng ta có thể phác thảo một phương pháp. Đầu tiên, phép tính lũy thừa có thể thực hiện bằng những phép nhân liên tiếp như đã trình bày ở trên, do đó chỉ cần xem làm thế nào để tính $M_1M_2 \bmod N$ với M_1M_2 và N đều là những số nguyên lớn. Điểm then chốt trong việc thực hiện tính toán phép mod là tính $10^P \bmod N$ cho mọi số 10^P nhỏ hơn số nguyên lớn nhất có thể gặp. Sau đó thì một phép toán mod đặc biệt nào cũng là một liên kết tuyến tính của những giá trị này. Một cơ số lớn hơn sẽ giảm được tổng số những tính toán cần thiết. Trong các thuật ngữ toán học, phương pháp này phù hợp với việc tính, chẳng hạn:

$0120200103110001200004012314 \bmod N$

bằng cách tính:

$$120(x^6 \bmod N) + 200(x^5 \bmod N) + 311(x^4 \bmod N) + (x^3 \bmod N) + 2000(x^2 \bmod N) + 401(x \bmod N) + 2314$$

với $x=10000$. Các giá trị $10000^P \bmod N$ có thể được tính trước và lưu vào trong một bảng, hoặc tính thêm khi cần, như trong thuật toán tìm chuỗi Rabin-Karp trong chương 19.

SỐ HỌC MA TRẬN

Cài đặt các phép toán trên ma trận cũng có những vấn đề tương tự như đối với các đa thức đã nêu trên. Ví dụ như cộng hai ma trận cũng đơn giản bởi vì cộng ma trận chính là cộng từng phần tử một, giống như các đa thức.

Nhân ma trận cũng không có gì phức tạp. Nếu gọi *r* là ma trận tích của hai ma trận *p* và *q*, thì *r[i,j]* (phần tử có dòng là i, cột là j) chỉ đơn giản là tổng của các tích từng cặp phần tử *p[i,0]*q[0,j]+p[i,1]*q[1,j]+...+p[i,N-1]*q[N-1,j]*, như trong chương trình sau:

```
for i:=0 to N-1 do
   for j:0 to N-1 do
      begin    t:=0.0;
               for k:=0 to N-1 do t:=t+p[i,k]*q[k,j];
               r[i,j]:=t
      end;
```

Người đọc có thể thử qua ví dụ sau để kiểm tra lại chương trình nêu trên với trường hợp:

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} 8 & 3 & 0 \\ 3 & 10 & 2 \\ 0 & 2 & 6 \end{pmatrix} = \begin{pmatrix} 17 & 25 & -18 \\ 11 & 9 & -10 \\ -14 & -13 & 26 \end{pmatrix}$$

Mỗi phần tử trong số N^2 phần tử của ma trận tích được tính bởi N phép tính nhân, vì thế cần khoảng N^3 phép toán để nhân hai ma trận N dòng N cột.

Cũng như trong các đa thức, các ma trận thưa (nghĩa là có nhiều phần tử bằng 0) có thể xử lý năng động hơn bằng cách dùng các danh

sách liên kết. Để giữ nguyên cấu trúc hai chiều, mỗi phần tử khác *0* của ma trận được lưu trong một nút của danh sách chứa giá trị và hai liên kết, một trỏ tới phần tử khác *0* kế tiếp trên cùng dòng và một trỏ tới phần tử khác 0 kế tiếp trên cùng cột. Thực hiện trên các ma trận thưa theo cách này cũng tương tự như đối với đa thức thưa, nhưng bị phức tạp ở chỗ mỗi nút xuất hiện trong hai danh sách.

Áp dụng kỹ thuật chia để trị trong bài toán số học nổi tiếng nhất là phương pháp của Strassen trong vấn đề nhân ma trận. Chúng ta sẽ không đề cập chi tiết ở đây, nhưng chúng ta có thể phác thảo phương pháp, bởi vì nó rất giống về cơ bản với phương pháp nhân đa thức vừa nghiên cứu.

Phương pháp bình thường để nhân hai ma trận N dòng N cột đòi hỏi N^3 phép nhân vô hướng vì mỗi phần tử trong N^2 phần tử của ma trận tích kết quả cần N phép nhân. Phương pháp của Strassen là chia bài toán ra làm hai, ứng với việc chia ma trận ra làm bốn ma trận $N/2$ dòng $N/2$ cột. Bài toán còn lại cũng bằng với việc nhân hai ma trận 2x2 cũng giống như chúng ta có thể giảm số phép nhân từ bốn xuống ba bằng cách liên hệ các phần tử với nhau trong bài toán nhân đa thức, Strassen cũng tìm được một cách liên kết các phần tử với nhau để giảm số phép nhân khi nhân hai ma trận 2x2 từ tám xuống bảy. Việc sắp xếp lại và các thành phần yêu cầu khá phức tạp.

Tính chất 36.2 *Hai ma trận NxN có thể nhân với nhau bằng cách sử dụng khoảng* $N^{2.81}$ *phép nhân.*

Theo trên, chúng ta có số phép nhân cần thiết để nhân hai ma trận bằng phương pháp Strassen được tính theo công thức đệ quy chia để trị : $M(N)=7\,M(N/2)$

Nó có kết quả là: $M(N)=N^{\lg 7}\approx N^{2.81}$

Kết quả này thật sự bất ngờ khi nó xuất hiện lần đầu tiên vào năm 1968, bởi vì trước đó, tất cả đều cho rằng để nhân hai ma trận NxN, tất nhiên phải dùng N^3 phép nhân. Vấn đề đã được nghiên cứu

sâu trong những năm gần đây, và các phương pháp hơi tốt hơn phương pháp của Strassen đã được tìm ra. Nhưng thuật toán "tốt nhất" cho phép nhân ma trận vẫn chưa tìm được, và đây là một trong các bài toán nổi tiếng nhất còn để lại của Tin học

Một chú ý quan trọng là chúng ta chỉ đếm số phép nhân. Trước khi lựa chọn một thuật toán cho một áp dụng trong thực hành, còn phải quan tâm đến hao phí do các phép cộng và trừ thêm vào để liên kết các phần tử và do các lời gọi đệ quy. Các hao phí này phụ thuộc nặng vào máy và cách cài đặt riêng. Nhưng tổng phí này lại làm cho thuật giải Strassen kém hiệu quả hơn phương pháp chuẩn đối với các ma trận nhỏ. Ngay cả trong các ma trận lớn, tùy theo số thành phần của dữ liệu nhập, phương pháp Strassen cũng chỉ thực sự cải thiện trong khoảng từ $N^{1.5}$ đến $N^{1.41}$ Độ cải tiến này không đáng kể, trừ khi N thật lớn. Ví dụ như, N phải lớn hơn 1 triệu thì phương pháp Strassen chỉ dùng 1 phần 4 số phép nhân so với phương pháp chuẩn. Vì thế thuật giải này chỉ là một sự đóng góp về mặt lý thuyết chứ không phải thực hành.

BÀI TẬP

1. Có thể biểu diễn đa thức dưới dạng $r_0(x-r_1)(x-r_2)...(x-r_N)$. Làm thế nào để nhân hai đa thức trong biểu diễn này ?

2. Viết chương trình Pascal để nhân hai đa thức thưa (hầu hết các hệ số đều bằng 0) bằng cách dùng biểu diễn xâu liên kết không chứa các hệ số 0.

3. Viết chương trình Pascal để gán giá trị *v* cho phần tử tại dòng *i* và cột *j* trong một ma trận thưa. Giả sử ma trận được biểu diễn bằng xâu liên kết chứa các phần tử *0* của ma trận.

4. Hãy đưa ra một phương pháp tính giá trị một đa thức đã biết trước các nghiệm $r_1, r_2, ..., r_N$ và so sánh với phương pháp Horner.

5. Viết một chương trình để tính giá trị đa thức bằng phương pháp Horner, trong đó đa thức được biểu diễn bằng xâu liên kết. Phải bảo đảm chương trình của bạn hoạt động hiệu quả đối với các đa thức thưa.

6. Viết một chương trình cài đặt phương pháp nội suy Lagrange có thời gian chạy N^2.

7. Có thể tính x^{55} mà chỉ dùng ít hơn chín phép nhân hay không ?

8. Liệt kê tất cả các phép nhân đa thức được thực hiện khi dùng thủ tục *mult* trong chương này để bình phương đa thức $1+x+x^2+x^3+x^4+x^5+x^6+x^7+x^8$.

9. Cần khoảng bao nhiêu phép nhân khi áp dụng thủ tục *mult* để bình phương đa thức $1+x^N$?

10. Cài đặt một phiên bản của thủ tục mult, trong đó dùng biểu diễn xâu liên kết cho các đa thức.

37

PHÉP KHỬ GAUSS

Một trong những tính toán cơ bản nhất là giải hệ phương trình. Phép khử Gauss, là thuật toán cơ bản để giải hệ phương trình này, thì tương đối đơn giản và có thay đổi chút ít kể từ khi nó được khám phá ra cách đây 150 năm. Thuật toán này trở nên dễ hiểu đặc biệt trong 20 năm qua, do đó người ta rất yên tâm khi dùng nó.

Phép khử Gauss là một ví dụ về một thuật toán được cài đặt sẵn trong hầu hết các thư viện chương trình; thật vậy, nó là một thành tố cơ bản trong một số ngôn ngữ máy tính, ví dụ APL và Basic. Thuật toán cơ bản thì dễ hiểu và dễ cài đặt, tuy nhiên, một số tình huống đặc biệt đã phát sinh khi người ta muốn cài đặt một phiên bản có thể sửa đổi được của thuật toán chớ không phải là thủ tục chuẩn bình thường. Phương pháp này cũng đáng để nghiên cứu như một trong những phương pháp số quan trọng nhất được sử dụng thường xuyên.

Cũng như các tài liệu toán mà ta đã nghiên cứu từ trước tới giờ, chúng ta sẽ chỉ tập trung giải quyết các nguyên lý cơ bản. Độc giả quen với đại số tuyến tính không đòi hỏi phải hiểu phương pháp cơ bản. Chúng tôi sẽ xây dựng một bản cài đặt đơn giản bằng Pascal, cách này có thể dễ hiểu hơn là một thủ tục trong thư viện cho các ứng dụng đơn giản. Tuy nhiên chúng tôi cũng có trưng ra các ví dụ về các khó khăn khi gặp phải. Dĩ nhiên là đối với những ứng dụng lớn hay quan trọng cần phải có bản cài đặt chuyên sâu hơn.

VÍ DỤ ĐƠN GIẢN

Giả sử có ba biến x, y và z và ba phương trình sau:

$x+3y-4z=8$ $x+y-2z=2$ $-x-2y+5z=-1$

Mục tiêu của ta là tính giá trị của các biến sao cho chúng thỏa đồng thời ba phương trình trên. Tùy từng hệ phương trình mà bài toán có thể không có lời giải (ví dụ với hệ có hai phương trình mâu thuẫn nhau, như $x+y=1, x+y=2$) hoặc có nhiều lời giải (ví dụ, có hai phương trình giống nhau hay tổng số biến nhiều hơn tổng số phương trình). Giả sử rằng số biến và số phương trình bằng nhau, ta sẽ thấy rằng lời giải nếu có thì duy nhất.

Để dễ mở rộng công thức, ta đặt tên biến có kèm chỉ số (đổi tên biến x, y, z thành x_1, x_2, x_3):

$x_1+3x_2-4x_3=8$ $x_1+x_2-2x_3=2$ $-x1-2x_2+5x_3=-1$

Để tránh viết lặp lại các tên biến, ta dùng ký hiệu ma trận để biểu diễn hệ phương trình:

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}$$

Có thể áp đặt nhiều thao tác lên hệ phương trình trên mà không làm thay đổi nghiệm:

Đổi chỗ lẫn nhau giữa các phương trình: rõ ràng rằng thứ tự viết các phương trình thì không ảnh hưởng đến nghiệm. Trong phép biểu diễn ma trận, thao tác này tương đương với biến đổi theo dòng trong ma trận (và trong vector bên phải).

Đổi tên biến: tương đương với biến đổi theo cột trong phép biểu diễn ma trận. (Nếu cột i và j đổi chỗ nhau thì các biến x_i, x_j cũng phải đổi chỗ nhau).

Nhân phương trình với một hằng số: tương đương với nhân một hàng trong ma trận (và phần tử của vector bên phải) với một hằng số.

Cộng hai phương trình: Thay thế một phương trình bằng tổng của chúng.

Cần phải suy nghĩ một chút để tự thuyết phục rằng các toán tử này, đặc biệt là toán tử cuối cùng, không ảnh hưởng đến lời giải. Ví dụ: Lấy một hệ phương trình tương đương với hệ trên bằng cách thay thế phương trình thứ hai bằng hiệu của hai phương trình đầu tiên:

$$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ -1 \end{pmatrix}$$

Lưu ý rằng điều này đã khử x1 ra khỏi phương trình thứ hai. Tương tự vậy, có thể khử x1 ra khỏi phương trình thứ ba bằng cách thay thế phương trình đó bằng tổng của phương trình thứ nhất và thứ ba:

$$\begin{pmatrix} 1 & 3 & -4 \\ 0 & 2 & -2 \\ 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 8 \\ 6 \\ 7 \end{pmatrix}$$

Biến x1 đã bị khử ở phương trình thứ hai và ba. Theo cách này ta có thể biến đổi hệ phương trình gốc thành hệ phương trình đơn giản hơn mà có cùng lời giải. Với ví dụ trên, chỉ cần một vài bước kết hợp hai trong những toán tử trên: thay thế phương trình thứ ba bằng cách trừ cái thứ hai với hai lần cái thứ ba, điều này khiến tất cả các phần tử nằm dưới đường chéo chính trở thành zero, lúc này hệ phương trình trở nên rất dễ giải:

$$x_1+3x_2-4x_3=8$$
$$2x_2-2x_3=6$$
$$-4x_3=-8$$

Phương trình thứ ba có nghiệm là $x_3=2$, thay giá trị này vào phương trình thứ hai, tính được x^2 là:

$$2x_2-4=6$$
$$x_2=5$$

Tương tự, thay hai giá trị của x2, x3 vào phương trình thứ nhất để tính x1

$$x_1+15-8=8$$
$$x_1=1$$

và hệ phương trình được giải xong.

Ví dụ này minh họa hai pha cơ bản trong phép khử Gauss. Pha thứ nhất là *pha khử-tiến*, trong đó hệ phương trình ban đầu được biến đổi, một cách có hệ thống để khử các biến ra khỏi phương trình, thành hệ có toàn zero dưới đường chéo. Tiến trình này gọi là tam giác hóa. Pha thứ hai là pha thay-ngược, dùng ma trận tam giác vừa được tạo ở pha một để tính các giá trị biến.

TÓM TẮT PHƯƠNG PHÁP

Tổng quát, giải hệ N phương trình với N ẩn số:

$$a_{11}x_1+a_{12}x_2+\cdots+a_{1N}x_N=b_1$$
$$a_{21}x_1+a_{22}x_2+\cdots+a_{2N}x_N=b_2$$

$$a_{N1}x_1+a_{N2}x_2+\cdots+a_{NN}x_N=b_N$$

Hệ phương trình này được viết dưới dạng ma trận như một phương trình duy nhất:

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ & & & \\ a_{N1} & a_{N2} & \cdots & a_{NN} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \\ x_3 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ \\ b_N \end{pmatrix}$$

Hay viết đơn giản $Ax=b$, trong đó A là đại diện cho ma trận, x đại diện cho các biến và b đại diện vế phải của phương trình. Vì các dòng của A được thao tác dọc theo các phần tử của b, để tiện lợi xem b như là cột thứ $N+1$ của và dùng mảng $Nx(N+1)$ để chứa chúng.

Pha *khử-tiến* được tóm tắt như sau: trước hết khử biến đầu tiên trong mọi phương trình trừ phương trình thứ nhất bằng cách cộng phương trình thứ nhất (đã nhân với một hằng số thích hợp) với từng phương trình còn lại, kế đến khử biến thứ hai trong mọi phương trình trừ hai phương trình đầu tiên bằng cách cộng phương trình thứ hai (đã nhân với một hằng số thích hợp) với từng phương trình kể từ phương trình thứ ba đến phương trình thứ N, kế đến khử biến thứ ba trong mọi phương trình trừ ba cái đầu tiên, v..v.. Để khử biến thứ i trong phương trình thứ j (với j nằm giữa $i+1$ và N) ta nhân phương trình thứ i với a_{ji}/a_{ii} và trừ nó ra khỏi phương trình thứ j. Tiến trình này được mô tả ngắn gọn hơn qua các dòng lệnh sau:

```
for i:=1 to N do
   for j:=i+1 to N do
      for k:=N+1 downto i do
         a[j,k]:=a[j,k]-a[i,k] * a[j,i]/a[i,i];
```

Đoạn mã có ba vòng lặp, thời gian thực hiện tỉ lệ với N^3. Vòng lặp thứ ba truy ngược lên để tránh phá hủy nội dung của *a[j,i]* trước khi

dùng nó để điều chỉnh giá trị của các phần tử khác trong cùng một dòng. Đoạn mã trên quá đơn giản để mà có thể đúng hoàn toàn: *a[i,i]* có thể là *0*, vì vậy có thể xảy ra trường hợp chia cho zero. Tuy nhiên, điều này dễ sửa vì có thể đổi chỗ bất kỳ dòng nào (từ *i+1* đến *N*) với dòng thứ *i* để *a[i,i]* khác *0* ở vòng lặp ngoài cùng. Nếu không có dòng nào như vậy, thì ma trận này là kỳ dị: không có nghiệm duy nhất. (chương trình nên thông báo tường minh trường hợp này, hoặc cứ để lỗi chia cho zero xảy ra.) Cần viết thêm một đoạn mã để tìm dòng thấp hơn có phần tử ở cột *i* khác *0* và đổi chỗ dòng này với dòng i. Phần tử *a[i,i]*, cuối cùng được dùng để khử các phần tử khác *0* dưới đường chéo trong cột thứ *i*, được gọi là *phần tử trụ*.

Thật sự, tốt nhất nên dùng dòng (từ *i+1* đến *N*) mà phần tử ở cột *i* có có giá trị tuyệt đối lớn nhất. Lý do là có thể xảy ra lỗi sai nếu giá trị *pivot* dùng để chia quá nhỏ. Nếu *a[i,i]* quá nhỏ thì kết quả của phép chia *a[j,i]/a[i,i]* được dùng để khử biến *i* ra khỏi phương trình *j* (với *j* từ *i+1* đến *N*) sẽ rất lớn. Thật sự, nó có thể lớn như vậy là để kéo các hệ số *a[j,k]* về một điểm mà tại đó giá trị *a[j,k]* trở nên méo mó do "lỗi sai".

Nói cách khác, các số khác biệt nhiều về độ lớn không thể được cộng hay trừ một cách chính xác theo số dấu chấm động, hệ thống thường dùng để biểu diễn các số thực, và việc dùng một phần tử trụ nhỏ làm tăng đáng kể khả năng những phép toán được thực hiện. Dùng giá trị lớn nhất trong cột *i* từ dòng *i+1* đến *N* sẽ chắc chắn rằng kết quả của phép chia trên luôn luôn nhỏ hơn *1* và sẽ tránh được lỗi sai này. Có thể nhắm đến việc nhìn trước cột *i* để tìm phần tử lớn nhất. Người ta đã chứng minh rằng có thể đạt được câu trả lời đúng đắn mà không cần dùng biện pháp phức tạp này.

Đoạn mã sau minh họa pha khử-tiến. Với mỗi *i* từ 1 đến *N*, rà xuống cột *i* để tìm phần tử lớn nhất (trong những dòng thứ *i+1* trở lên). Dòng có chứa phần tử này được đổi chỗ với dòng *i*, và biến thứ *i* bị khử khỏi trong các phương trình từ *i+1* đến N:

```
procedure eliminate;
   var i,j,k:integer; t:real;
   begin
      for i:=1 to N do
      begin
         max:=i;
         for j:=i+1 to N do
            if abs(a[j,i])>abs(a[max,i]) then max:=j;
         for k:=i to N+1 do
         begin t:=a[i,k]; a[i,k]=a[max,k]; a[max,k]:=t end;
         for j:=i+1 to N do
            for k:=N+1 downto i do
               a[j,k]:=a[j,k]-a[i,k]*a[j,i]/a[i,i];
      end
   end;
```

Một số thuật giải có yêu cầu phần tử trụ *a[i,i]* phải được dùng để khử biến thứ *i* ra khỏi mọi phương trình ngoại trừ phương trình thứ *i* (không chỉ là thứ *i+1* đến thứ *N*).

Sau khi thực hiện xong pha khử-tiến, mảng *a* có những phần tử nằm dưới đường chéo là *0*, kế đến thực hiện *pha thay-ngược*. Sau đây là đoạn mã của pha này:

```
procedure substitute;
   var j,k:integer; t:real;
   begin
      for j:=N downto 1 do
         begin
            t:=0.0;
            for k:=j+1 to N do t:=t+a[j,k]*x[k];
            x[j]:=a[j,N+1]-t)/a[j,j];
         end;
   end;
```

Gọi lệnh khử, tiếp theo gọi lệnh thay-ngược để tính nghiệm của mảng x có N phần tử. Đối với ma trận kỳ dị (*singular*) việc chia cho 0 vẫn có thể xảy ra- phải có một hàm thư viện kiểm tra ngoại lệ này.

Tính chất 37.1 *Một hệ N phương trình đồng thời có N biến phải dùng $N^3/3$ phép nhân và cộng để giải nghiệm.*

Thời gian thực hiện thủ tục thay-ngược là $O(N^2)$, vì thế hầu hết các công việc được thực hiện là trong thủ tục khử. Duyệt lại thủ tục đó cho thấy rằng mỗi giá trị của i, thì k được lặp $N-i+2$ lần và j lặp $N-i$ lần: điều này nghĩa là vòng lặp trong cùng được thực hiện tổng các số hạng $(N-i+2)(N-i)$ với 1, giá trị của tổng này là $N^3/3 + O(N^2)$. Giá trị của $-a[j,i]/a[i,i]$ có thể tính ở ngoài vòng lặp k, vì thế vòng lặp trong cùng chỉ còn một phép nhân và một phép cộng.

Một cách khác, sau khi thủ tục khử tạo các số 0 dưới đường chéo thì dùng đúng thủ tục này để tạo các số 0 trên đường chéo: trước hết tạo cột cuối cùng thành 0 ngoại trừ phần tử $a[N,N]$ bằng cách cộng với $a[N,N]$ (sau khi đã nhân $a[N,N]$ với một hằng số thích hợp), kế đó thực hiện y như vậy cho cột kế cột cuối cùng, ... Nghĩa là ta xoay từng phần một lần nữa, nhưng trên phần khác của mỗi cột. thực hiện ngược lên qua các cột. Sau khi tiến trình này (tên là *Rút gọn Gauss-Jordan*) hoàn tất, ta sẽ chỉ có những phần tử ở đường chéo là khác không, lúc hệ phương trình có lời giải tầm thường. Tuy nhiên số phép toán được dùng trong tiến trình này lớn hơn nhiều so với tiến trình thay-ngược.

Lỗi do tính toán là nguồn quan tâm chính trong phép khử Gauss. Như đã nói ở trên, ta nên thận trọng trong những tình huống độ lớn của các hệ số khác biệt nhau nhiều. Dùng phần tử lớn nhất có sẵn trong cột làm phần tử trụ để chắc chắn rằng các hệ số lớn sẽ không được tạo ra một cách tùy ý trong tiến trình xoay, nhưng có thể sẽ không luôn luôn tránh được lỗi nghiêm trọng này. Ví dụ: những hệ số rất bé sẽ trở thành lớn khi hai phương trình khác nhau có hệ số gần nhau. Tuy nhiên, người ta có thể xác định trước khi nào thì vấn

đề như vậy sẽ gây ra những lời giải không chính xác. Mỗi ma trận có một số lượng số được kết hợp gọi là số điều kiện, số này có thể được dùng để đánh giá tính đúng đắn của lời giải của máy tính. Một thủ tục thư viện tốt cho phép khử Gauss sẽ tính được số điều kiện của ma trận cũng như tính được lời giải sao cho tính đúng đắn của lời giải có thể được biết. Một giải thích cặn kẽ cho vấn đề đó vượt quá phạm vi của sách này.

Phép khử Gauss với việc xoay từng phần sử dụng phần tử trụ lớn nhất có sẵn là "bảo đảm" sản sinh các kết quả có lỗi sai về tính toán nhỏ nhất. Có những kết quả toán học được tính toán cẩn thận cho thấy rằng lời giải hoàn toàn đúng, ngoại trừ những ma trận có điều kiện xấu (có thể là do hệ phương trình hơn là do phương pháp giải).

Thuật giải này đã từng là chủ đề của nhiều công cuộc nghiên cứu lý thuyết khá chi tiết và có thể được giới thiệu như là một thủ tục tính toán của ứng dụng rộng

NHỮNG BIẾN ĐỔI VÀ MỞ RỘNG

Phương pháp vừa mô tả thì thích hợp nhất với ma trận NxN với N*N phần tử khác 0. Như ta đã thấy đối với những bài toán khác, những kỹ thuật đặc biệt thì thích hợp với những ma trận thưa trong đó hầu hết các phần tử là 0. Tình trạng này tương ứng hệ phương trình mà mỗi phương trình chỉ có ít số hạng.

Nếu các phần tử khác 0 không có cấu trúc đặc biệt, thì biểu diễn xâu liên kết đã bàn ở chương 36 rất thích hợp, với mỗi nút là một phần tử khác không của ma trận, được xâu lại với nhau theo dòng và cột. Phương pháp chuẩn có thể được cài đặt bằng biểu diễn này, với những phức tạp do nhu cầu tạo và hủy các phần tử khác không. Kỹ thuật này có thể không đáng bỏ công làm nếu có đủ bộ nhớ để chứa toàn bộ ma trận, vì nó phức tạp hơn nhiều so với phương pháp chuẩn. Cũng vậy, ma trận thưa trở nên ít thưa hơn trong tiến trình khử Gauss.

Một số ma trận không những chỉ có vài phần tử khác không mà còn có một cấu trúc đơn giản, do vậy xâu liên kết là không cần thiết. Ví dụ phổ biến nhất của loại ma trận này là ma trận *band*, trong đó các phần tử khác không hầu hết ở gần đường chéo. Trong trường hợp này, vòng lặp bên trong của thuật giải khử Gauss chỉ cần lặp vài lần, vì thế tổng thời gian thực hiện (và bộ nhớ cần thiết) thì tỉ lệ với N, chứ không phải N^3.

Một trường hợp đặc biệt lý thú của ma trận "dãy" là ma trận "ba đường chéo", trong đó chỉ những phần tử trực tiếp ngay ở trên, trực tiếp ở bên trên hay trực tiếp ở dưới đường chéo là khác không. Ví dụ, dạng tổng quát của ma trận "ba đường chéo" với $N=5$ là

$$\begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{pmatrix}$$

Với những ma trận như vậy, pháp khử-tiến và thay-ngược chỉ còn một vòng lặp:

```
for i:=1 to N-1 do
   begin
   a[i+1,N+1]:=a[i+1,N+1]-a[i,N+1]*a[i+1,i]/a[i,i];
   a[i+1,i+1]:=a[i+1,i+1]-a[i,i+1]*a[i+1,i]/a[i,i];
   end;
for j:=N downto 1 do
   x[j]:=(a[j,N+1]-a[j,j+1]*x[j+1])/a[j,j];
```

Với phép khử-ngược, chỉ trường hợp $j:=i+1$ và $k:=i+1$ cần được gộp vào vì $a[i,k]=0$ với $k>i+1$. (Trường hợp $k=i$ có thể được bỏ qua

vì nó gán zero cho một phần tử mảng mà phần tử này không bao giờ được xét lại - sự thay đổi tương tự này có thể được tạo thẳng trong phép khử Gauss.)

Tính chất 37.2 *Một hệ "ba đường chéo" các phương trình đồng thời có thể được giải theo thời gian tuyến tính.*

Dĩ nhiên, một mảng hai chiều có kích thước *NnN* sẽ không được dùng để chứa ma trận "ba đường chéo". Bộ nhớ cần cho chương trình trên có thể là tuyến tính theo N bằng cách dùng bốn mảng thay vì một ma trận: mỗi mảng chứa một trong ba đường chéo khác không, mảng cuối cùng chứa cột thứ $N+1$. Chú ý rằng chương trình này không cần thiết phải xoay phần tử có sẵn lớn nhất, vì không có bảo đảm nào cho việc chia cho 0 hay sự tích lũy các lỗi sai tính toán. Đối với một vài loại ma trận ba đường chéo điều này xảy ra thường xuyên, tuy nhiên, người ta đã chứng minh rằng nó không phải là lý do để quan tâm.

Phép rút gọn Gauss-Jordan có thể được cài đặt với tiến trình xoay đầy đủ thay thế một ma trận bởi nghịch đảo của nó. Nghịch đảo của ma trận A viết A^{-1}, có tính chất sau: hệ phương trình $Ax=b$ có thể giải được bằng cách thực hiện phép nhân $x=A^{-1}b$. Cũng vậy, cần N^3 phép toán để tính x với b cho trước. Tuy nhiên, có cách tiền xử lý ma trận và phân rã nó thành các phần sao cho có thể giải hệ phương trình tương đương với bất kỳ vế phải nào trong thời gian tỉ lệ với N^2, nhờ việc lưu thừa số N qua mỗi lần dùng phép khử Gauss. Đại khái, điều này liên hệ đến sự nhớ các tác tử thực hiện trên cột thứ $(N+1)$ trong pha khử-tiến, vì thế kết quả của khử-tiến trên cột thứ $(N+1)$ mới có thể được tính một cách có hiệu quả, kế đến thực hiện pha thay-ngược như bình thường.

Việc giải hệ phương trình tuyến tính cho thấy về mặt tính toán tương đương với phép nhân ma trận, đã có những thuật giải (ví dụ phép nhân ma trận của Strassen) có thể giải hệ N phương trình có N ẩn số trong thời gian xấp xỉ $N^{2.81}$. Như phép nhân ma trận, dùng

phương pháp như vậy không đáng trừ khi hệ phương trình rất lớn được xử lý như lệ thường (nếu có). Như đã nói thời gian thực hiện phép khử Gauss liên hệ đến số dữ liệu liệu nhập là $N^3/2$, thời gian này khó cải thiện hơn nữa.

BÀI TẬP

1. Tìm ma trận được tính bởi pha khử-tiến của phép khử Gauss khi dùng để giải các phương trình $x+y+z=6$, $2x+y+3z=12$ và $3x+y+33z=14$.

2. Tìm hệ ba phương trình với ba ẩn số sao cho ba vòng lặp **for** của pha khử-tiến sai, ngay cả trường hợp có lời giải.

3. Cần bao nhiêu dung lượng bộ nhớ cho phép khử Gauss trong trường hợp ma trận NxN có $3N$ phần tử khác không ?

4. Mô tả điều gì xảy ra khi phép khử thực hiện trên ma trận có một dòng toàn bằng không ?

5. Mô tả điều gì xảy ra khi phép khử thực hiện trên ma trận có một cột toàn bằng không ?

6. Có bao nhiêu tác tử toán học dùng trong phép thu gọn Gauss-Jordan?

7. Nếu hoán vị các cột trong ma trận, có ảnh hưởng gì đế hệ phương trình tương đương ?

8. Làm thế nào kiểm tra những phương trình mâu thuẫn khi dùng phép khử ? Còn các phương trình đồng nhất thì sao ?

9. Nếu dùng phép khử Gauss cho hệ M phương trình với N ẩn số khi $M<N$ hay $M>N$ thì sao ?

10. Cho ví dụ chứng tỏ cần phải pivot trên phần tử có sẵn lớn nhất, dùng nguyên lý thần thoại của máy tính nghĩa là các số có thể biểu diễn dưới dạng hai ký số (mọi số phải ở dạng $x.y * 10^z$, trong đó x, y và z là các số nguyên một ký số).

38

KHỚP ĐƯỜNG CONG

Thuật ngữ khớp đường cong (curve fitting) hay điều chỉnh dữ liệu được dùng để mô tả bài toán tổng quát của việc tìm các hàm khớp với một tập các giá trị đang được quan sát ứng với một tập điểm. đặc biệt, cho các điểm

$x_1, x_2, ..., x_N$

và các giá trị tương ứng

$y_1, y_2, ... y_N$

Mục đích là tìm các hàm sao cho

$f(x_1)=y_1, f(x_2)=y_2, ... , f(x_N)=y_N$

và sao cho $f(x)$ được giả sử "hợp lý" ở những điểm dữ liệu khác. Có thể rằng một x và y được liên hệ bởi hàm $f(x)$ ẩn danh nào đó và mục tiêu của ta là tìm ra hàm đó, nhưng, định nghĩa của từ "hợp lý" phụ thuộc từng ứng dụng. Ta sẽ thấy rằng thường thì xác định hàm "không hợp lý" thì dễ hơn.

Khớp đường cong có ứng dụng hiển nhiên trong sự phân tích các dữ liệu thuộc thí nghiệm và còn nhiều ứng dụng khác nữa... Ví dụ, nó có thể được dùng trong đồ họa máy tính để sản sinh ra đường cong "coi được" mà không cần phải lưu một số lượng lớn các điểm vẽ. Một ứng dụng có liên hệ là dùng chỉnh đường cong để cho ra một thuật giải nhanh trong tính toán giá trị của hàm chưa biết ở một

điểm bất kỳ: Giữ một bảng nhỏ chứa các giá trị chính xác, sự hiệu chỉnh đường cong sẽ suy ra các điểm khác.

Có hai phương pháp cơ bản được dùng để tiếp cận bài toán này. Phương pháp thứ nhất là phép nội suy: tìm một hàm liên tục khớp với các giá trị đã cho. Phương pháp thứ hai, điều chỉnh dữ liệu bình phương nhỏ nhất (least-squares data fitting), được dùng khi các giá trị có thể không chính xác và hàm tìm được khớp càng nhiều càng tốt.

PHÉP NỘI SUY

Ta đã thấy phương pháp giải bài toán nối dữ liệu: nếu biết *f* là đa thức bậc *N-1*, ta có bài toán nội suy đa thức ở Chương 36. Cho dù ta không có hiểu biết đặc biệt gì về *f*, ta cũng có thể giải bài toán nối dữ liệu bằng cách cho *f(x)* là đa thức nội suy bậc *N-1* trên những điểm vá gía trị đã cho. Điều này có thể được tính toán bằng cách dùng các phương pháp được phác thảo ở Chương 36, nhưng có nhiều lý do để không dùng phép nội suy đa thức cho nối dữ liệu. Vì một lượng lớn các phép tính có liên hệ. Ví dụ, tính một đa thức bậc 100 dường như phá hủy phép nội suy đường cong qua 100 điểm.

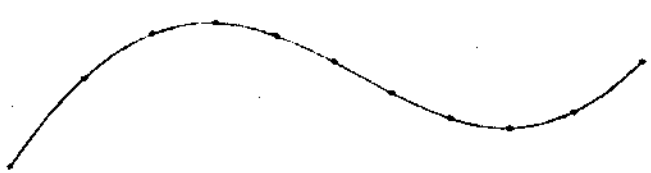
Hạn chế chính của phép nội suy đa thức là do các đa thức bậc cao là những hàm tương đối phức tạp có những tính chất không đoán trước được, không thích hợp cho hàm nối. Một kết quả từ toán cổ điển (lý thuyết xấp xỉ Weierstrass) nói rằng có thể xấp xỉ bất kỳ hàm bằng đa thức (bậc đủ cao). Không may, các đa thức bậc khá cao có khuynh hướng dao động nhiều. Cho dù hầu hết các hàm được xấp xỉ hầu hết khắp nơi trên một khoảng đóng bởi một đa thức nội suy nhưng luôn luôn có những chỗ mà sự xấp xỉ trở nên không chấp nhận được. Hơn nữa, lý thuyết này giả sử rằng các giá trị dữ liệu là các giá trị chính xác lấy từ một hàm ẩn nào đó, nhưng trong thực tế các giá trị đó thường là chỉ xấp xỉ. Nếu những y là những giá trị được

xấp xỉ từ một số đa thức ẩn bậc thấp, ta hy vọng các hệ số cho số hạng bậc cao trong đa thức nội suy là 0. Cách này ít được dùng, thay vào đó, đa thức nội suy thử dùng các số hạng bậc cao để giúp đạt được sự nối chính xác. Những ảnh hưởng này khiến các đa thức nội suy không thích hợp trong nhiều ứng dụng nối đường cong.

NỘI SUY SPLINE

Cũng vậy, các đa thức bậc thấp là những đường cong đơn giản được sử dụng rộng rãi trong nối đường cong. Mánh là hủy bỏ tư tưởng thử tạo một đa thức đi qua tất cả các điểm và thay vì dùng các đa thức khác nhau để nối các điểm kề nhau, nối các đoạn sao cho thật mịn. Một trường hợp đặc biệt khá tế nhị liên hệ sự tính toán tương đối trực tiếp, được gọi là nội suy spline.

Spline là một thiết bị cơ học được các người vẽ sơ đồ thiết kế dùng để vẽ các đường cong đẹp, có thẩm mỹ: người vẽ xác định tập hợp các điểm (nút) rồi bẻ cong một giải plastic hay miếng gỗ linh hoạt (spline) quanh chúng và lấy vết chúng để tạo thành một đường cong. Nội suy spline thì tương đương về mặt toán học với tiến trình này và cho ra cùng một kết quả. Hình 38.1 minh họa một spline qua 10 điểm.



Hình 38.1 *Một spline qua 10 nút*

Có thể thấy rằng hình dạng của một đường cong tạo bởi spline giữa hai nút kề nhau là một đa thức bậc ba. Trở lại bài toán nối dữ liệu, điều này có nghĩa là ta nên xem đường cong là *N-1* đa thức khác nhau có bậc ba.

$$s_i(x)=a_ix^3+b_ix^2+c_ix+d_i \qquad i = 1,2...N-1$$

Với $s_i(x)$ là đa thức bậc ba xác định giữa khoảng x_i và x_{i+1}. Spline có thể được biểu diễn trong một mảng bốn chiều (hay trong *4x(N-1)* mảng hai chiều). Việc tạo một spline gồm việc tính các hệ số *a, b, c, d* từ các điểm *x* và các giá trị *y* đã cho. Việc bẻ cong một spline về mặt vật lý tương ứng với việc giải hệ phương trình với nghiệm là các hệ số. Ví dụ, Hiển nhiên ta phải có: $s_i(x_i)=y_i$ và $s_{i+1}(x_{i+1})=y_{i+1}$ với $i=1, 2,..., N-1$ vì spline phải chạm các nút. Không những nó phải chạm các nút mà đường cong đi qua các nút đó phải mịn, không gất. Về mặt toán học điều này nghĩa là các đạo hàm bậc nhất cuả các đa thức spline phải bằng nhau ở mỗi nút ($s'_{i-1}(x_i)=s'_i(x_i)$ với $i=2, 3, ... N-1$). Thật sự thì các đạo hàm bậc hai của các đa thức cũng phải bằng nhau ở mỗi nút. Các điều liện này cho ra *4N-6* phương trình với *4(N-1)* hệ số là ẩn. Cần xác định thêm hai điều kiện nữa để mô tả tình trạng ở hai điểm cuối của spline. Có nhiều cách: dùng cái gọi là spline 'tự nhiên' được rút từ $s''_i(x_i)=0$ và $s''_{N-1}(x_N)=0$. Các điều kiện này cho ra một hệ đầy đủ là *4N-4* phương trình với *4N-4* ẩn số, hệ phương trình này giải được bằng phép khử Gauss với ẩn là các hệ số.

Tuy nhiên, cũng cùng một spline nhưng có thể tính toán khá hiệu quả hơn vì thật sự chỉ có *N-2* ẩn: hầu hết các điều kiệc của spline là thừa. Ví dụ, giả sử p_i là đạo hàm bậc hai của spline tại điểm x_i, vì $s''_{i-1}(x_i)=s''_i(x_i)=p_i$ với $i=2, 3, ..., N-1$ và $p_1=p_N=0$ nếu biết trước các giá trị p_i và p_N, thì tất cả hệ số *a, b, c, d* có thể được tính trên các đoạn spline, vì ta có bốn phương trình với bốn ẩn số trên các đoạn: với $i=1, 2, ..., N-1$ ta phải có :

$$s_i(x_i) = y_i$$

$$s_i(x_{i+1}) = y_{i+1}$$

$$s''_i(x_i) = p_i$$

$$s''_i(x_{i+1}) = p_{i+1}$$

Các giá trị x, y đã cho trước, để xác định đầy đủ spline chỉ cần tính các giá trị $p_2,..., p_N$. Để tính được, dùng điều kiện đạo hàm bậc nhất phải bằng nhau: có $N-2$ điều kiện ứng với $N-2$ phương trình cần để giải $N-2$ ẩn, theo p_i Để diễn tả các hệ số a, b, c, d theo p, các giá trị đạo hàm bậc hai, ta thay các biểu thức vào bốn phương trình trên trên mỗi đoạn spline, điều này dẫn đến một số biểu thức phức tạp không cần thiết. Thay vì diễn tả các phương trình trên từng đoạn spline ở dạng chuẩn liên hệ đến ít ẩn số hơn. Nếu ta thay các biến là $t=(x-x_i)/(x_{i+1}-x_i)$ thì spline được biểu diễn như sau

$$s_i(t)=ty_{i+1}+(1-t)y_i + (x_{i+1}-x_i)^2((t^3-t)p_{i+1} - ((1-t)^3 - (1-t))p_i)/6$$

bây giờ mỗi spline được xác định trên soạn [0,1]. phương trình này đỡ "ghê hơn" là hình thức bên ngoài của nó, vì ta chỉ quan tâm đến hai điểm mút 0 và 1 và hoặc t hoặc $(1-t)$ bằng 0 ở những điểm mút này. Cách biểu diễn này dễ kiểm spline được nội suy và liên tục vì $s_{i-1}(1)=s_i(0)=y_i$ với $i=2, ..., N-1$, nó chỉ hơi khó chứng minh đạo hàm bậc hai liên tục vì $s''_i(1)=s''_{i+1}(0)=p_{i+1}$. Có các đa thức bậc ba thỏa những điều kiện này tại các điểm mút, vì thế chúng tương đương với các đoạn spline mô tả ở trên. Nếu ta thay thành t và tìm hệ số của x^3,... thì ta sẽ có cùng biểu thức ẩn là a, b, c, d theo x, y và p giống như ta dùng phương pháp được mô tả ở đoạn trước. Nhưng không có lý do gì làm như vậy, vì ta đã kiểm rằng các đoạn spline này thỏa các điều kiện cuối, và có thể lượng giá từng spline ở bất kỳ điểm nào trong đoạn của nó bằng cách tính t và dùng công thức trên (khi đã biết p).

Để tính p, ta gán các đạo hàm cấp một của các đoạn spline bằng

các điểm cuối. Đạo hàm cấp 1 (theo x) của các phương trình trên là:

$s'_i(t)=z_i+ (x_{i+1}-x_i) ((3t^2 -1)p_{i+1} + (3(1-t)^2 -1) p_i)/6$

Trong đó $z_i=(y_{i+1}-y_i)/(x_{i+1}-x_i)$. Kế đến gán s'i-1(1)=s'i(0) với $i=2...N-1$, ta sẽ có hệ $N-2$ phương trình:

$(x_i-x_{i-1})p_{i-1} + 2(x_{i+1}-x_{i-1})p_i + (x_{i+1} -x_i)p_{i+1} =6(z_i-z_{i-1})$

Hệ phương trình này thuộc dạng tam giác đơn giản dùng phép khử Gauss ở Chương 37 để giải. Ví dụ, nếu đặt $u_i=x_{i+1}-x_i$, $d_i=2(x_{i+1}-x_{i-1})$, và $wi=6(z_i-z_{i+1})$, ta có hệ phương trình với $N=7$:

$$\begin{pmatrix} d_2 & u_2 & 0 & 0 & 0 \\ u_2 & d_3 & u_3 & 0 & 0 \\ 0 & u_3 & d_4 & u_4 & 0 \\ 0 & 0 & u_4 & d_5 & u_5 \\ 0 & 0 & 0 & u_5 & d_6 \end{pmatrix} \begin{pmatrix} p_2 \\ p_3 \\ p_4 \\ p_5 \\ p_6 \end{pmatrix} = \begin{pmatrix} w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \end{pmatrix}$$

Thật sự, hệ tam giác đối xứng này có nửa dưới bằng với nửa trên của đường chéo chính. Nó được xoay trên phần tử có sẵn lớn nhất nếu không cần tìm lời giải đúng cho hệ phương trình này.

Phương pháp được mô tả ở đoạn trên để tính spline bậc ba được dễ dàng viết dưới dạng Pascal như thủ tục *makespline* ở trang sau:

Các mảng d và u là đại diện ma trận tam giác mà dùng chương trình ở Chương 37 để giải. Trong chương trình đó thay *a[i,i]* bằng *d[i]*, *a[i+1,i]* hay *a[i,i+1]* bằng *u[i]* và *a[i,N+1]* bằng *z[i]*.

Tính chất 38.1 *Một spline bậc ba trên N điểm có thể thực hiện theo thời gian tuyến tính.*

Điều này hiển nhiên có được từ chương trình

Ví dụ xây dựng một spline bậc ba từ năm điểm:

```
procedure makespline;
  var i:integer;
  begin
    readln(N);
    for i:=1 to N do readln(x[i],y[i]);
    for i:=2 to N-1 do d[i]:=2+(x[i+1]-x[i-1]);
    for i:=1 to N-1 do u[i]:=x[i+1]-x[i];
    for i:=2 to N-1 do
      w[i]:=6.0*((y[i+1]-y[i])/u[i] - (y[i]-y[i-1])/u[i-1] );
    p[1]:=0.0; p[N]:=0.0;
    for i:=2 to N-2 do
    begin
      w[i+1]:=w[i+1]-w[i]*u[i]/d[i];
      d[i+1]:=d[i+1]-u[i]*u[i]/d[i];
    end;
    for i:=N-1 downto N-1 do
      p[i]:=(w[i]-u[i]*p[i+1])/d[i];
  end;
```

(1.0,2.0), (2.0,1.5), (4.0,1.25), (5.0,1.2), (8.0,1.125), (10.0,1.1)

(Các điểm trên lấy từ hàm $1+1/x$). Các tham số spline tìm được bằng cách giải hệ phương trình:

$$\begin{pmatrix} 6 & 2 & 0 & 0 \\ 2 & 6 & 1 & 0 \\ 0 & 1 & 8 & 3 \\ 0 & 0 & 3 & 10 \end{pmatrix} \begin{pmatrix} p_2 \\ p_3 \\ p_4 \\ p_5 \end{pmatrix} = \begin{pmatrix} 2.250 \\ .450 \\ .150 \\ .075 \end{pmatrix}$$

với kết quả

p_2=0.39541, p_3=-0.06123, p_4=0.02658, p_5=-0.00047

Để lượng giá *spline* cho bất kỳ giá trị x nào trong khoảng $[x_1,x_N]$, đơn giản ta tìm khoảng $[x_i,x_{i+1}]$ có chứa x, rồi tính t và dùng công

thức trên cho $s_i(x)$ (rồi lại dùng hàm này để tính p_i và p_{i+1}).

```
function eval(v:real):real;
   var t:real; i:integer;
   function f(x:real):real;
      begin f:=x*x*x-x; end;
begin
i:=0;
repeat i:=i+1 until v[i+1];
t:=(v-x[i])/u[i];
eval:=t*y[i+1]+(1-t)*y[i]+u[i]*u[i]*(f(t)*p[i+1]+f(t-1)*p[i])/6.0;
end;
```

Chương trình này không kiểm tra khi v không nằm giữa *x[1]* và *x[N]*. Nếu số đoạn spline nhiều (nghĩa là N lớn) có một số phương pháp tìm có hiệu quả hơn ở Chương 14 được dùng để tìm khoảng chứa v.

Có nhiều thay đổi về tư tưởng điều chỉnh đường cong bằng cách ráp các đoạn đa thức sao cho thật mịn. Việc tính toán các spline thuộc lĩnh vực nghiên cứu đã phát triển rất tốt. Các kiểu spline khác liên hệ đến các chuẩn làm mịn cũng như các thay đổi khác như nới lỏng điều kiện các spline phải chạm đúng vào mỗi điểm dữ liệu. Về mặt tính toán, chúng liên hệ cùng một số bước xác định các hệ số cho mỗi đoạn spline bằng cách giải hệ phương trình tuyến tính rút từ các giới hạn được đưa ra trên sự kiện chúng được nối với nhau như thế nào.

PHƯƠNG PHÁP BÌNH PHƯƠNG NHỎ NHẤT

Khi những giá trị dữ liệu không chính xác, thường ta cần phải tưởng tượng ra hình dạng của hàm khớp với dữ liệu. Hàm này có thể phụ thuộc một số tham số

$$f(x)=f(c_1, c_2, ..., c_M, x)$$

và thủ tục điều chỉnh đường cong là tìm cách chọn các tham số nào khớp nhất với các giá trị quan sát được ở các điểm đã cho. Nếu hàm là một đa thức (với tham số là những hệ số) và các giá trị là chính xác thì đây chính là phép nội suy. Nhưng ta đang xét những hàm tổng quát hơn và dữ liệu không đúng. Để đơn giản hoá vấn đề, ta tập trung vào việc nối những hàm là tổ hợp tuyến tính của hàm đơn giản hơn, với các tham số ẩn là các hệ số:

$$f(x)=c_1f_1(x) + c_2f_2(x) + ... + c_Mf_M(x)$$

Hàm này bao hàm hầu hết các hàm mà ta quan tâm. Sau khi nghiên cứu xong trường hợp này, ta sẽ xét đến những hàm tổng quát hơn

Một cách phổ thông để đo mức độ tốt của hàm nối là tiêu chuẩn bình phương tổng quát. Ở đây sai số được tính bằng cách thêm vào bình phương sai số ở mỗi điểm quan sát được:

$$E = \sum_{1\le j\le N} (f(x_j)-y_j)^2$$

Đây là một phép đo rất tự nhiên: bình phương được thực hiện để khử các ước lược giữa các sai số với các dấu hiệu khác nhau. Hiển nhiên, điều người ta mong muốn nhất là tìm ra một phép chọn các tham số sao cho tối tiểu hóa E. Điều này khiến phép chọn có thể được tính có hiệu quả; đây là phương pháp bình phương nhỏ nhất, phương pháp này đúng như định nghĩa của nó. Để đơn giản hóa đạo hàm, xét trường hợp $M=2$, $N=3$. Giả sử có ba điểm x_1, x_2, x_3 và các giá trị tương ứng y_1, y_2, y_3 thỏa hàm có dạng : $f(x)=c_1f_1(x))+ c_2f_2(x)$. Ta phải tìm một phép chọn các hệ số c_1, c_2 sao cho tối tiểu hóa sai số bình phương nhỏ nhất.

$$E=(c_1f_1(x_1) +c_2f_2(x_1) - y_1)^2 +(c_1f_1(x2) +c_2f_2(x_2) - y_2)^2 \\ +(c_1f_1(x_3) +c_2f_2(x_3) - y_3)^2$$

Để tìm các phép chọn của c_1 và c_2 sao cho tối tiểu hóa sao số này, đơn giản chỉ cần gán zero các đạo hàm dE/dc_1 và dE/dc_2 Với c_1 ta có:

$$\begin{aligned} dE/dc_1 = {} & 2(c_1f_1(x_1) + c_2f_2(x_1) - y_1)f_1(x_1) \\ & +2(c_1f_1(x_2) + c_2f_2(x_2) - y_2)f_1(x_2) \\ & +2(c_1f_1(x_3) + c_2f_2(x_3) - y_3)f_1(x_3). \end{aligned}$$

Việc gán đạo hàm bằng zero cho ra phương trình có biến là c_1 và c_2 phải thỏa ($f_1(x_1)$, ... là các "hằng số" có trị biết trước)

$$\begin{aligned} & c_1(f_1(x_1)f_1(x_1) + f_1(x_2)f_1(x_2) + f_1(x_3)f_1(x_3)) \\ & +c_2(f_2(x_1)f_1(x_1) + f_2(x_2)f_1(x_2) + f_2(x_3)f_1(x_3)) \\ & =y_1f_1(x_1) + y_2f_1(x_2) + y_3f_1(x_3) \end{aligned}$$

Ta có phương trình tương tự, khi gán đạo hàm dE/dc2 về zero. Những phương trình này trông có vẻ ghê gớm nhưng có thể đơn giản chúng bằng cách dùng ký hiệu vector và toán tử tích vô hướng Nếu định nghĩa vector $x=(x_1,x_2,x_3)$ và $y=(y1,y2,y3)$ thì tích vô hướng của x và y là một số thực được định nghĩa bởi:

$$x.y=x_1y_1+x_2y_2+x_3y_3$$

Bây giờ, nếu ta định nghĩa các vector $f_1=(f_1(x_1),f_1(x_2),f_1(x_3))$ và $f_2=(f_2(x_1),f_2(x_2),f_2(x_3))$, thì phương trình với hệ số c1, c2 có dạng đơn giản như sau:

$$c_1f_1.f_1+c_2f_1.f_2=y.f_1 \qquad c_1f_2.f_1+c_2f_2.f_2=y.f_2$$

Các phương trình này có thể giải bằng phép khử Gaus để tìm các hệ số. Ví dụ, giả sử có các điểm dữ liệu:

(1.0,2.05) (2.0,1.53) (4.0,1.26) (5.0,1.21) (8.0,1.13) (10.0,1.1)

khớp với hàm có dạng c_1+c_2/x (các điểm này hơi không chính xác khớp với hàm $1+1/x$.) Trong trường hợp này, f1 là hằng

$f_1=(1.0,1.0,1.0,1.0,1.0,1.0)$ và $f_2=(1.0,0.5,0.25,0.2,0.125,0.1)$,

vì vậy ta phải giải hệ phương trình sau

$$\begin{pmatrix} 6.000 & 2.175 \\ 2.175 & 1.378 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} 8.280 \\ 3.623 \end{pmatrix}$$

và có đáp số là $c_1=1.054$ và $c_2=0.998$ (cả hai đều xấp xỉ bằng 1 như mong muốn).

Phương pháp trên dễ dàng được tổng quát hoá cho trường hợp nhiều hơn hai hệ số. Để tìm các hằng $c_1,c_2,...,c_M$ trong

$$f(x)=c_1f_1(x)+c_2f_2(x)+...+c_Mf_M(x)$$

làm tối tiểu hóa sai số bình phương nhỏ nhất cho các điểm và các vector

$$x=(x_1,x_2, ..., x_N) \qquad y=(y_1,y_2, ..., y_N)$$

trước hết chúng ta tính các vector thành phần của hàm

$f_1=(f_1(x_1), f_1(x_2), ..., f_1(x_N))$
$f_2=(f_2(x_1), f_2(x_2), ..., f_2(x_N))$
. . .
$f_M=(f_M(x_1), f_M(x_2), ..., f_M(x_N))$

Và có hệ phương trình tuyến tính MxM dạng $Ac=b$ với $a_{ij}=f_i.f_j$, $b_j=f_j.y$, lời giải của hệ phương trình này là các hệ số cần tìm.

Phương pháp này dễ dàng cài đặt bằng cách dùng mảng hai chiều cho các vector f_i và coi y như vector thứ $(M+1)$. thì mảng $a[1..M,1..M+1]$ có thể được điền đầy bởi đoạn chương trình:

```
for i:=1 to M do
   for j:=1 to M+1 do
   begin   t:=0.0;
           for k:=1 to N do t:=t+f[i,k]*f[j,k];
           a[i,j]:=t;
   end;
```

và giải bằng cách dùng phép khử Gauss ở Chương 37.

Phương pháp bình phương nhỏ nhất có thể được mở rộng cho những hàm phi tuyến (ví dụ như hàm $f(x)=c_1e^{-c_2x}\ sinc_4x$) và nó thường được dùng cho loại ứng dụng này. Về cơ bản thì tư tưởng giống nhau, nhưng có thể khó tính đạo hàm. Người ta thường dùng phương pháp lặp: dùng một số ước lượng cho các hệ số, rồi sử dụng các ước lượng trong phương pháp bình phương nhỏ nhất để tính các đạo hàm, do vậy cần phép ước lượng các hệ số tốt hơn. Ngày nay phương pháp cơ bản này đang được dùng rộng rãi, một phương pháp được phác thảo bởi Gauss ngay từ thập niên 1820.

BÀI TẬP

1. Xấp xỉ hàm *lgx* bằng đa thức nội suy bậc bốn ở các điểm 1,2,3,4 và

5. Ước lượng chất lượng khớp bằng cách tính tổng bình phương của các sai số tại 1.5, 2.5, 3.5 và 4.5

2. Giải bài toán trên với hàm *sinx*. Vẽ đồ thị của hàm và xấp xỉ trên máy tính của bạn, nếu có thể.

3. Giải các bài toán trên dùng spline bậc ba thay vì một đa thức nội suy

4. Xấp xỉ hàm *lgx* bằng spline bậc ba ở các nút 2^N với $1 \le N \le 10$ Thử nghiệm ở những vị trí khác nhau của các nút trong cùng miền xác định để lấy những lời giải tốt hơn.

5. Điều gì xảy ra trong phép nối bình phương nhỏ nhất nếu một trong những hàm là hàm $f_i(x)=0$ với i nào đó?

6. Dùng phép nối đường cong bình phương nhỏ nhất tìm các giá trị của a và b sao cho cho ra công thức tốt nhất ở dạng $a_N lnN+b_N$ để mô tả tổng số các chỉ thị được thực hiện khi Quicksort đang thực hiện trên tập tin ngẫu nhiên.

7. Giá trị nào của a, b, c tối tiểu hóa sai số bình phương nhỏ nhất trong việc sử dụng hàm $f(x)=axlogx + bx + c$ để xấp xỉ $f(1)=0$, $f(4)=13$, $f(8)=41$?

8. Có bao nhiêu phép nhân (gồm cả pha khử Gauss) liên quan trong việc dùng phương pháp bình phương nhỏ nhất tìm M hệ số dựa trên N điểm quan sát được?

9. Trong tình huống nào một ma trận trong nối đường cong bình phương nhỏ nhất là kỳ dị ?

10. Phương pháp bình phương nhỏ nhất có dùng được không nếu hai điểm quan sát khác nhau được mô tả bởi cùng một điểm ?

39

TÍNH TÍCH PHÂN

Tích phân là phép toán giải tích căn bản thường tiến hành trên các hàm được xử lý trên máy tính. Chúng ta cần tìm "diện tích giới hạn bởi một đường cong" một cách hiệu quả và với một độ chính xác hợp lý. Trong chương này, chúng ta sẽ xem xét một số thuật toán cổ điển dùng để giải bài toán tính số căn bản này.

Trước hết, chúng ta sẽ xét trường hợp các hàm số được cho sẵn. Khi ấy, có thể thực hiện phép *tích phân hình thức* (symbolic integration) để chuyển sự biểu diễn cho một hàm thành một biểu diễn tương tự cho tích phân. Điều này thích hợp khi hàm số được xử lý nằm trong một lớp hẹp các hàm số mà tích phân của nó cho sẵn ở dạng giải tích, hoặc nằm trong phạm vi của các hệ thống mà nó xử lý các biểu diễn như vậy của các hàm số.

Mặt khác, hàm số có thể xác định bởi một bảng, khi giá trị của hàm số chỉ có thể xác định tại một số ít các điểm. Trong trường hợp như vậy, người ta chỉ có thể cho được một giá trị xấp xỉ của tích phân, dựa trên những giả định về dáng điệu của hàm số giữa các điểm. Tính chính xác của tích phân hầu như phụ thuộc hoàn toàn vào tính đúng đắn của các giả định.

Hầu hết các trường hợp thông thường nằm giữa hai loại hàm số được nói ở trên: Hàm số được tích phân có thể tính được giá trị tại bất kỳ điểm nào. Cũng như trước, tính chính xác của tích phân phụ

thuộc vào các giả định về dáng điệu của hàm số ở giữa các điểm mà ta lựa chọn để tính toán. Mục đích của nó là để tính toán một xấp xỉ tích phân hợp lý mà không phải tính quá mức số lượng giá trị hàm số tại các điểm. Sự tính toán này thường gọi là *phép cầu phương* (quadrature) bằng giải tích số.

Trong chương này chúng ta sẽ xem xét nhiều phép cầu phương. Các phương pháp này rất sơ cấp, mục đích là để thu thập một số kinh nghiệm như là các phương pháp số cơ sở. Nhiều áp dụng thực sự có thể rút ra từ các kỹ thuật cơ bản mà ta xét, tuy nhiên các phương pháp giải các bài toán khó hơn, đặc biệt là giải số các phương trình vi phân, lại quan trọng hơn trong ứng dụng.

TÍCH PHÂN HÌNH THỨC

Một ví dụ đơn giản là tích phân các đa thức. Trong chương 36, chúng ta đã xem xét các phương pháp "hình thức hóa" (symbolically) cách tính tổng và tích của các đa thức, sử dụng một chương trình tác động lên biểu diễn của đa thức (và vi phân) của các đa thức có thể làm bằng cách này. Nếu một đa thức:

$$p(x) = p_0 + p_1 x + p_2 x^2 + \ldots + p_{N-1} x^{N-1}$$

được biểu diễn đơn giản bằng cách đưa các hệ số vào trong một mảng (*array*) p thì tích phân có thể tính toán dễ dàng như sau:

```
for i:=N downto 1 do p[i]:=p[i-1]/i; p(o):= 0
```

Chương trình trên áp dụng công thức quen thuộc nếu i >> 0. Một lớp hàm rộng hơn lớp hàm đa thức cũng có thể tính toán hình thức nếu ta thêm vào một số quy tắc. Chẳng hạn, quy tắc *tích phân từng phần*.

$$\int u\,dv = uv - \int v\,du$$

nếu được sử dụng sẽ mở rộng khá nhiều tập hợp các hàm số có thể lấy tích phân.

Tuy nhiên, do số quy tắc có sẵn có thể dùng để tính tích phân cho một hàm đặc biệt rất nhiều, nên việc tính toán hình thức thực sự gặp khó khăn khi phải lựa chọn các quy tắc, người ta chỉ vừa mới phát hiện ra một thuật toán giải quyết vấn đề này: hoặc tính được tích phân (hình thức) hoặc trả lời được rằng kết quả tích phân không thể biểu diễn bằng các số hạng của những hàm sơ cấp. Sự mô tả thuật toán đó vượt quá khuôn khổ cuốn sách này.

Dĩ nhiên, kỹ thuật hình thức (symbolic techmiques) có những giới hạn căn bản do nhiều tích phân (trong các ứng dụng) không thể tính được bằng kỹ thuật này. Trong phần sau, ta sẽ xét qua một số kỹ thuật khác.

CÁC PHƯƠNG PHÁP CẦU PHƯƠNG ĐƠN GIẢN

Có lẽ phương pháp hiển nhiên để tính xấp xỉ một tích phân là phương pháp hình chữ nhật. Việc tính một tích phân giống cách tính phần diện tích giới hạn bởi một đường cong, và ta có thể ước lượng phần diện tích này bằng cách cộng diện tích của các hình chữ nhật nhỏ mà nó nằm sát ngay dưới đường cong, như hình 39.1.

Chính xác hơn, giả sử chúng ta cần tính $\int_a^b f(x)dx$ và giả sử khoảng $[a, b]$ được chia thành N phần, giới hạn bởi các điểm $x_1,..., x_{N+1}$. Khi đó ta có N hình chữ nhật với chiều rộng của hình chữ nhật thứ i $(1 \le i \le N)$ là $(x_{i+1} - x_i)$. Chiều cao của hình chữ nhật này có thể chọn là $f(x_i)$ hay $f(x_{i+1})$, tuy nhiên, hình như kết quả sẽ chính xác hơn nếu ta dùng giá trị của f tại điểm giữa các khoảng

Ta có công thức cầu phương:

$$r = \sum_{1 \le i \le N} (x_{i+1} - x)\, f\left(\frac{x_i + x_{i+1}}{2}\right)$$

Trường hợp tất cả các khoảng chia có độ dài bằng nhau, đặt $x_{i+1} - x_i = \omega$, ta có $x_{i+1} + x_i = (2i + 1)\,\omega$, vậy xấp xỉ r tính được dễ dàng.

```
function intrect(a,b:real; N:integer): real;
  var i : integer; w, r : real;
  begin   r:=0; w:=(b-a)/N;
          for i:=1 to N do r:=r+w*f(a-w/2+i*w);
          intrect:=r;
  end;
```

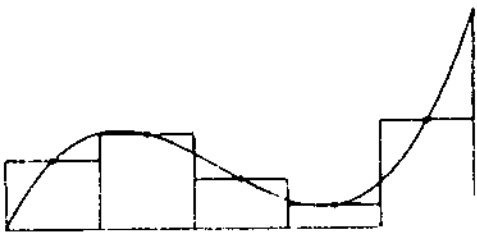
Dĩ nhiên là khi N lớn, lời giải sẽ chính xác hơn. Hình 39.2 cho thấy kết quả của việc xử dụng các khoảng có độ dài nhỏ hơn so với các khoảng trong hình 39.1

Dưới đây là một ví dụ định lượng hơn để tính giá trị của (mà ta biết là ln2=0.6931471805599 ...) khi dùng *intrect (1.0, 2.0, N)* với *N=10, 100, 1000:*

10	0,6928353694100
100	0,6931440556283
1000	0,6931471493100

Khi $N=1000$, lời giải của ta chính xác đến bảy chữ số sau dấu phẩy.

Sự xem xét cách đánh giá số thường gợi ý cho các phương pháp chính xác hơn. Khai triển Taylor hàm f trong mỗi khoảng (x_{i+1}, x_i) tại điểm giữa của nó, tích phân, rồi cộng tất cả các khoảng lại. Ta



Hình 39.1 *Phương pháp hình chữ nhật*

không đi vào chi tiết, kết quả cuối cùng cho thấy:

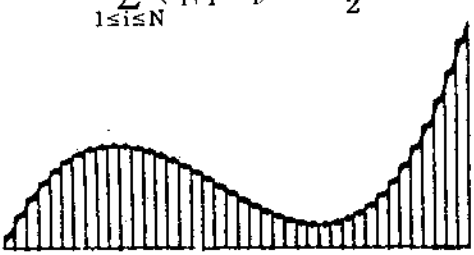
$$\int_a^b f(x)dx = r + w^3e_3 + w^5e_5 + \ldots$$

Với $w=(b - a)/N$ và e_3 phụ thuộc vào giá trị của đạo hàm bậc ba của f tại trung điểm của khoảng (x_{i+1}, x_i)... (thông thường thì đó là một xấp xỉ tốt vì hầu hết các hàm thông thường đều có các đạo hàm bậc cao không lớn lắm). Ví dụ, nếu ta chọn $w=0,01$ (tương ứng với $N=200$ trong ví dụ trên), công thức này cho thấy sự tính toán tích phân chính xác đến sáu chữ số sau dấu phẩy.

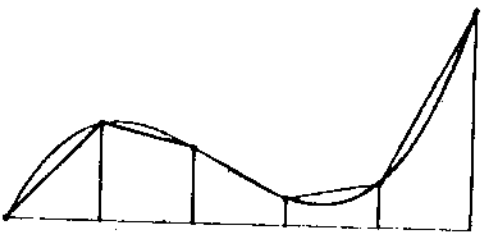
Một phương pháp xấp xỉ tích phân khác là chia phần diện tích cong thành những hình thang (xem hình 39.3). Nhắc lại rằng diện tích hình thang là tích của chiều cao với trung bình cộng của hai đáy.

Phương pháp *hình thang* dẫn đến công thức cầu phương.

$$t = \sum_{1\le i\le N} (x_{i+1}-x_i)\,\frac{f(x_i)+f(x_{i+1})}{2}$$



Hình 39.2 *Phương pháp hình chữ nhật với khoảng chia nhỏ nhơn*



Hình 39.3 *Phương pháp hình thang*

Trong trường hợp độ dài của tất cả các khoảng (x_i, x_{i+1}) là như nhau, ta có chương trình sau:

```
function inttrap(a,b:real; N:integer):real;
  var i:integer; w,t:real;
  begin  t:=0; w:=(b-a)/N;
         for i:=1 to N do t:=t+w*(f(a+(i-1)*w)+f(a+i*w))/2;
         inttrap:=t;
  end;
```

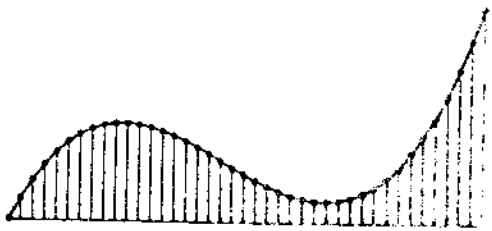
Sai số của phương trình này có thể tính tương tự trường hợp phương pháp hình chữ nhật.

$$\int_a^b f(x)dx = t - 2w^3e_3 - 4w^5e_5 + \ldots$$

Vậy sai số trong phương pháp hình thang nhiều gấp hai sai số trong phương pháp hình chữ nhật. Tính tích phân $\int_1^2 dx/x$ ta có thể có kết quả sau:

10	0.6937714031754
100	0.6931534304818
1000	0.6931472430599

Một phương pháp khác là cầu phương spline: phép nội suy spline đã xét trong các chương trước và bây giờ tích phân được tính bằng cách áp dụng trên từng đoạn kỹ thuật lấy tích phân hình thức cho các hàm đa thức đã được mô tả ở trên. Phương pháp này rất gần với



Hình 39.4 *Phương pháp hình thang với khoảng chia nhỏ nhơn*

phương pháp hình thang và phương pháp hình chữ nhật, như ta sẽ thấy dưới đây.

PHƯƠNG PHÁP PHỐI HỢP

Xem xét các công thức dùng để tính sai số trong phương pháp hình chữ nhật và phương pháp hình thang, ta tìm được một phương pháp đơn giản có độ chính xác cao hơn, phương pháp Simpson. Ý tưởng chính dựa trên sự khử số hạng đầu tiên trong phần sai số bằng cách kết hợp hai phương pháp. Nhân công thức dùng để tính sai số của phương pháp hình chữ nhật với 2 rồi cộng với công thức tương ứng của phương pháp hình thang đem kết quả chia 3 ta có công thức:

$$\int_a^b f(x)dx = \frac{1}{3}(2r + t - 2w^5e_5 + ...)$$

Số hạng w_3 bị khử, vậy công thức cho ta một phương pháp đạt được kết quả với sai số cỡ bằng kết hợp các công thức cầu phương tương ứng:

$$s = \sum_{1 \le i \le N} \frac{x_{i+1} - x_i}{6} (f(x_i) + 4f(\frac{x_i + x_{i+1}}{2}) + f(x_{i+1}))$$

Nếu chia đoạn *[a, b]* thành các khoảng cỡ 0.01, thì phương pháp Simpson cho kết quả chính xác vào cỡ mười chữ số sau dấu phẩy.

Chương trình tương ứng ở trang sau này đòi hỏi tính ba giá trị của hàm số trong mỗi vòng lặp trong, nhưng kết quả thì chính xác

hơn kết quả của hai phương pháp trước:

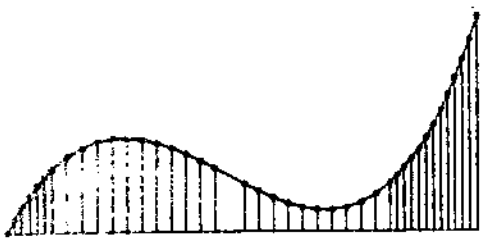
```
function intsimp(a,b:real; N:integer):real;
  var i:integer; w,s:real;
  begin   s:=0; w:=(b-a)/N;
          for i:=1 to N do
          s:=s+u*(f(a+(i-1)*w+4*f(a-w/2+i*w)+f(a+i*w))/6;
          intsimp:=s;
  end;
```

10	0.6931473746651
100	0.6931471805795
1000	0.6931471805599

phương pháp Simpson thực ra chính là phương pháp tích phân các hàm nội suy bậc hai từng mảnh. Điều thú vị là bốn phương pháp chúng ta vừa xem xét đều có thể xem như phương pháp tích phân các hàm nội suy: phương pháp hình chữ nhật (đa thức nội suy cấp 0), phương pháp hình thang (đa thức nội suy cấp 1), phương pháp Sympson (đa thức nội suy cấp 2) và cầu phương spline với đa thức bậc ba.

PHƯƠNG PHÁP THÍCH ỨNG

Một phần lớn thiếu sót trong các phương pháp chúng ta đã xem xét cho đến giờ là các sai số phụ thuộc không chỉ trên độ dài các khoảng con (của phép phân hoạch) mà còn phụ thuộc vào độ lớn của đạo hàm cấp cao của hàm số lấy tích phân. Các phương pháp này sẽ không tốt cho vài hàm số (có đạo hàm cấp cao lớn). Tuy nhiên, chỉ có một số ít hàm có đạo hàm cấp cao lớn ở mọi điểm. Điều hợp lý là hàm lớn và các khoảng (của phân hoạch) nhỏ ở chỗ các đạo hàm nhỏ. Một phương pháp thực hiện điều này một cách có hệ thống gọi là một phương pháp *cầu phương thích ứng*.



Hình 39.5 *Phương pháp cầu phương thích ứng*

Phương pháp tổng quát là sử dụng hai phương pháp cầu phương khác nhau trên mỗi khoảng con, so sánh kết quả, và chia nhỏ khoảng này ra nếu sự sai khác quá lớn. Dĩ nhiên ta cần phải chú ý một tí vì nếu hai phương pháp sử dụng đều kém, kết quả (của hai phương pháp) có thể không chênh lệch nhưng lại là một kết quả kém. Một cách để tránh điều đó là chọn một phương pháp luôn luôn cho kết quả lớn hơn giá trị chính xác của tích phân (trên khoảng con đang xét) và phương pháp kia lại cho kết quả bé hơn giá trị chính xác này. Một cách khác để tránh điều đó là bảo đảm cho một phương pháp chính xác hơn phương pháp kia. Cách thứ hai này sẽ được mô tả trong đoạn trình dưới đây.

```
function adapt(a,b:real):real;
  begin
    if abs(intsimp(a,b,10)-intsim(a,b,5)))<tolerance
    then adapt:=intsimp(a,b,10)
    else adapt:=adapt(a,(a+b)/2)+adapt((a+b),2,b)
  end;
```

Cả hai cách tính tích phân đều là phương pháp Simpson, tuy nhiên một phân hoạch có số đoạn nhiều gấp đôi số đoạn của phân hoạch kia, chủ yếu, một phương pháp của chương trình dùng để kiểm tra độ chính xác của phương pháp Simpson trên một khoảng, nếu kết quả chưa tốt thì chia nhỏ khoảng ra nữa.

Trong các phương pháp khác, ta quyết định số lượng các bước cần phải thực hiện và nhận kết quả (dù có chính xác hay không). Tro ng phương pháp cầu phương thích ứng thì độ chính xác được quan tâm tới, còn số lượng các bước thực hiện lại không định trước. Do đó, tolerance cần phải chọn cẩn thận để chương trình không bị lặp mà không thoát ra được. Số các bước cần thiết phụ thuộc rất nhiều trên dạng của hàmg được lấy tích phân. Một hàm số biến thiên nhiều thì số lượng các bước sẽ nhiều, tuy nhiên kết quả sẽ chính xác hơn nhiều so với các phương pháp trước. Hình 39.5 cho thấy sự phân hoạch của đoạn khi dùng cầu phương thích ứng (dựa trên công thức hình thang) cho hàm số trong các hình 39.1 - 39.4. Chú ý : các khoảng lớn khi hàm số thẳng và trơn, các khoảng nhỏ hơn khi hàm dốc nhanh.

Một hàm số như trong ví dụ của ta có thể được tính với một số bước vừa phải. Bảng sau cho thấy các giá trị của *t*, kết quả tích phân tương ứng và số lần sử dụng đến chương trình con trên để tính toán

0.00001000000	0.6931473746651	1
0.00000010000	0.6931471829695	5
0.00000000100	0.6931471806413	13
0.00000000001	0.6931471805623	33

Chương trình này có thể cải tiến bằng nhiều cách. Trước hết, chắc chắn là không cần gọi *intsimp (a, b, 10)* hai lần. Thực ra, các giá trị hàm cho chương trình con này đã được tính một phần trong *intsimp (a, b, 5)*. Thứ hai, giá trị *tolerance* có thể liên quan chặt chẽ hơn với độ chính xác của lời giải nếu nó được xét theo tỉ lệ độ dài khoảng đang xét với độ dài toàn khoảng. Cuối cùng, một chương trình con tốt hơn có thể được thực hiện bằng một phương pháp tốt hơn phương pháp cầu phương thích ứng bằng quy tắc *simpson* (tuy nhiên, theo một quy luật cơ bản của đệ quy (recursion), một chương trình con thích ứng khác không phải là một ý hay). Một phương pháp cầu phương thích ứng tinh vi có thể cho ra kết quả chính xác cho các bài toán không giải được bằng cách khác, tuy nhiên nên chú ý đến loại hàm số đang xử lý để tìm phương pháp thích hợp.

BÀI TẬP

1. Viết một chương trình để tích phân hình thức (và đạo hàm hình thức) các đa thức x và $\ln x$. Sử dụng đệ quy trên cơ sở tích phân từng phần.

2. Phương pháp cầu phương nào cho kết quả tốt nhất khi tích phân các hàm sau: $f(x) = 5x$, $f(x) = (3 - x)(4 + x)$, $f(x) = sin(x)$?

3. Sử dụng bốn phương pháp cầu phương (hình chữ nhật, hình thang, simpson, spline) để tính tích phân hàm $y = 1/x$ trong khoảng [0.1, 10].

4. Làm bài tập trên với $y = sin\ x$.

5. Điều gì xảy ra khi phương pháp thích ứng được sử dụng để tính tích phân hàm $y = 1/x$ trên khoảng [-1; 2].

6. Làm bài tập trên với bốn phương pháp cầu phương căn bản.

7. Tìm các điểm phân hoạch khi cầu phương thích ứng hàm $y = 1/x$ trên khoảng [0.1; 10] với tolerance là 0.1.

8. So sánh độ chính xác của kết quả tích phân bài trước khi dùng phương pháp cầu phương thích ứng dựa trên quy tắc simpson và khi dùng phương pháp cầu phương thích ứng dựa trên quy tắc hình chữ nhật.

9. Giải bài toán trên với $y = sin\ x$.

10. Tìm một ví dụ cho thấy rõ là phương pháp cầu phương thích nghi cho kết quả chính xác hơn hẳn ở phương pháp khác.

40

CÁC THUẬT TOÁN SONG SONG

Hầu hết các thuật toán chúng ta đã khảo sát qua đều có khả năng ứng dụng rất thiết thực. Phần lớn các phương pháp này đã tồn tại khoảng một thập niên hoặc lâu hơn, trải qua nhiều thay đổi quan trọng về phần cứng và phần mềm trên máy tính. Các thiết kế phần cứng mới cũng như các khả năng mới của phần mềm chắc chắn có thể tác động rất nhiều đến những thuật toán đã có, nhưng hầu hết các thuật toán tốt trên những máy cũ, cũng là những thuật toán tốt trên các máy mới.

Một lý do của hiện tượng này là thiết kế nền tảng của các máy tính ít thay đổi trong những năm qua. Thiết kế của rất nhiều hệ thống máy tính đều dựa trên cùng một nguyên lý cơ bản được phát triển bởi nhà toán học Von Neumann được xây dựng từ quan điểm lưu trữ dữ liệu và chỉ thị trong cùng một vùng nhớ, và một bộ xử lý lấy từng chỉ thị từ bộ nhớ, xử lý tuần tự mỗi lần một chỉ thị. Nhiều cơ chế đã được xây dựng một cách công phu tỉ mỉ nhằm làm cho máy tính trở nên rẻ tiền hơn, xử lý nhanh hơn, kích thước vật lý nhỏ hơn mà khả năng lưu trữ lại lớn hơn, nhưng kiến trúc của hầu hết máy tính chỉ có thể xem là những biến thể từ mô hình Von Neumann.

Tuy nhiên thời gian gần đây, các tiến bộ về kỹ nghệ phần cứng làm cho giá thành của các bộ phận máy tính trở nên thấp hơn, cho

phép chúng ta có thể nghĩ đến khả năng phát triển những mô hình máy khác, một trong số đó là máy tính có khả năng xử lý một số lượng lớn các chỉ thị trong cùng một thời điểm; hay các chỉ thị được cứng hóa (*wired in*) để tạo thành máy chuyên dụng có khả năng giải quyết một vấn đề; hay máy tính gồm các máy nhỏ hơn hợp tác với nhau để giải quyết cùng một vấn đề. Nói ngắn gọn, chúng ta có thể nghĩ đến việc cho máy tính thực hiện đồng thời nhiều thao tác thay vì mỗi lúc chỉ làm một thao tác. Trong chương này, chúng ta sẽ xem xét hiệu quả tiềm tàng trong các ý tưởng như thế, trên một số vấn đề và thuật toán ta đã nghiên cứu. Một cách riêng, chúng ta cũng xem xét hai cách tiếp cận thiết kế máy tính có thể thích hợp với sự phát triển của các thuật toán song song: *perfect shuffle* và *systolic array*.

CÁC CÁCH TIẾP CẬN TỔNG QUÁT

Các thuật toán cơ sở rất thường được sử dụng, và khi dùng chúng trong các bài toán lớn thường có khuynh hướng thực hiện các thuật toán này trên các máy tính lớn hơn và mạnh hơn. Kết quả của khuynh hướng này là một loạt các "siêu máy tính" ra đời biểu hiện cho kỹ thuật mới nhất, chúng ta chấp nhận một số nhượng bộ đối với khái niệm cơ sở Von Neumann nhưng vẫn được thiết kế thành một máy đa năng và hữu dụng đối với tất cả các chương trình. Cách tiếp cận chung để sử dụng loại máy này cho các vấn đề thuộc loại chúng ta đang quan tâm bắt đầu từ những thuật toán tốt nhất trên các máy quy ước và sửa đổi chúng cho phù hợp với những đặc tính nào đó của một máy mới. Rõ ràng hướng tiếp cận này khuyến khích việc duy trì các thuật toán và kiến trúc cũ trong máy mới.

Các bộ vi xử lý với tiềm năng tính toán đáng chú ý gần đây đã có giá thành hạ hơn. Một cách tiếp cận hiển nhiên là thử sử dụng nhiều bộ vi xử lý này để giải quyết cùng một vấn đề. Một số thuật toán có thể thích ứng tốt với cách xử lý phân tán này, số còn lại đơn giản là

không thích hợp với cách thức cài đặt này.

Sự phát triển của các bộ xử lý tương đối hiệu quả mà giá rẻ đã dẫn đến sự xuất hiện của các công cụ đa năng dùng trong việc thiết kế và xây dựng các bộ xử lý mới. Điều này, đến lượt nó lại dẫn đến các hoạt động gia tăng trong việc phát triển các máy chuyên dụng để xử lý các vấn đề riêng biệt. Nếu không có sẵn một máy tính thích hợp cho sự thi hành một thuật toán nào đó, ta có thể thiết kế một máy như thế! Vậy các máy tính thích hợp với những bài toán khác nhau có thể được thiết kế và xây dựng vừa vặn trong một vi mạch.

Một ý tưởng chung trong tất cả các cách tiếp cận này là cơ chế song song: chúng ta cố gắng tiết kiệm thời gian bằng cách để nhiều việc khác nhau được xảy ra cùng một lúc. Ý tưởng này có thể dẫn đến sự hỗn độn nếu như nó không được thực hiện một cách trật tự. Dưới đây, chúng ta sẽ xem xét hai ví dụ minh họa cho một số kỹ thuật nhằm đạt được mức độ cao của cơ chế song song trong một số các lớp bài toán. Ý chính là giả sử chúng ta không chỉ có một mà là M bộ xử lý trên đó có thể thực hiện chương trình của chúng ta. Vì thế nếu mọi điều làm việc tốt, ta có thể hy vọng chương trình của chúng ta chạy nhanh hơn trước đó M lần.

Các vấn đề trước mắt liên quan đến việc tổ chức cho M bộ xử lý làm việc cùng nhau như thế nào để giải cùng bài toán. Điều quan trọng nhất là các bộ xử lý này phải liên lạc được với nhau theo một cách nào đó: phải có các mạch nối liền chúng và một cơ chế được đặc tả để truyền dữ liệu trên các mạch này. Hơn nữa, có những giới hạn vật lý trên kiểu kết nối được phép. Ví dụ giả sử các bộ xử lý của chúng ta là các mạch thích hợp 32 đường kết nối. Ngay cả chúng ta có đến 1000 bộ xử lý như thế, thì mỗi bộ xử lý chỉ có thể nối liền 32 bộ xử lý khác. Việc chọn lựa cách kết nối các bộ xử lý với nhau là điều cơ bản nhất trong xử lý song song. Hơn nữa, cần phải nhớ là lựa chọn đó phải được quyết định từ trước: một chương trình có thể thay đổi hoạt động của nó tùy từng thể hiện của bài toán, nhưng một cái máy

thì không thể thay đổi cách kết nối giữa các phần của nó với nhau được.

Trong mỗi một ý tưởng về các kiểu máy mới vừa được nhắc đến ở trên, quan điểm tổng quát về xử lý song song theo nghĩa một số bộ xử lý độc lập cùng với một số cách kết nối cố định được áp dụng như sau: một siêu máy tính có nhiều bộ xử lý rất chuyên dụng và các mẫu kết nối có tính nguyên vẹn đối với kiến trúc của nó (và ảnh hưởng đến nhiều khía cạnh trong việc vận hành của nó); các bộ vi xử lý được nối liền với nhau bao gồm một số tương đối nhỏ các bộ xử lý mạnh và sự kết nối đơn giản; và một mạch tích hợp (*VLSI*) bao gồm một số khá lớn các bộ xử lý đơn giản với những kết nối phức tạp.

Nhiều quan điểm khác về xử lý song song đã được nghiên cứu rộng rãi từ thời Von Neumann, đang được quan tâm trở lại vì đã có thể sản xuất các bộ xử lý với giá thành thấp. Việc đề cập đến tất cả các quan điểm đó nằm ngoài mục tiêu của cuốn sách này. Thay vào đó, chúng ta sẽ xem xét hai loại máy được đặc tả cho một số bài toán quen thuộc. Khi khảo sát các máy này, chúng ta sẽ thấy được hiệu quả của cấu trúc máy tính trên việc thiết kế thuật toán. Ở đây có đôi chút phụ thuộc lẫn nhau: chắc chắn không ai thiết kế một máy tính mới mà không nghĩ đến dùng nó vào mục đích gì, và ai cũng muốn chọn loại máy tính tốt nhất có thể để thực hiện các thuật toán quan trọng nhất.

SỰ XÁO TRỘN HOÀN TOÀN

(PERFECT SHUFFLE)

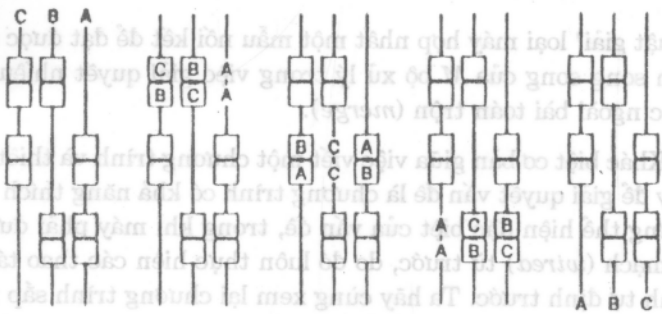
Để làm sáng tỏ một số vấn đề đang gây tranh cãi liên quan đến việc thực hiện các thuật toán như một máy thay vì một chương trình, chúng ta sẽ xét đến một thuật toán trộn thích hợp cho việc thực hiện bằng phần cứng. Như chúng ta sẽ thấy về sau, có thể phát triển một phương pháp tổng quát giống như thế vào trong bản đồ án về máy

"thuật giải" loại máy hợp nhất một mẫu nối kết để đạt được sự thực hiện song song của *M* bộ xử lý trong việc giải quyết nhiều vấn đề khác ngoài bài toán trộn (*merge*).

Khác biệt cơ bản giữa việc viết một chương trình và thiết kế một máy để giải quyết vấn đề là chương trình có khả năng thích ứng với những thể hiện đặc biệt của vấn đề, trong khi máy phải được thiết kế mạch (*wired*) từ trước, do đó luôn thực hiện các thao tác trong trình tự định trước. Ta hãy cùng xem lại chương trình sắp xếp đầu tiên *sort3* trong chương 8 để thấy rõ sự khác biệt này. Dù dữ liệu là ba số nguyên xuất hiện trong thứ tự nào, chương trình *sort3* vẫn thực hiện lần lượt ba thao tác "so sánh - hoán đổi" đúng trình tự đã định trong chương trình. Không có thuật toán nào khác trong các thuật toán sắp xếp chúng ta đã gặp có tính chất này. Dãy các thao tác so sánh chúng thực hiện luôn phụ thuộc vào kết quả của lần so sánh trước, vì thế thể hiện nhiều vấn đề trong việc cứng hóa.

Trường hợp đặc biệt, nếu chúng ta có một thiết bị phần cứng gồm hai mạch nhập và hai mạch xuất có khả năng so sánh hai số nhập vào, và hoán vị chúng khi cần thiết, thì chúng ta có thể mắc nối ba thiết bị như thế với nhau như hình 40.1 để tạo ra một máy sắp xếp với ba dữ liệu nhập (phía trên của hình vẽ) và ba dữ liệu xuất (phía dưới của hình vẽ). Ở đây, hộp đầu tiên hoán vị C và B, sau đó hộp thứ hai hoán vị B và A, sau cùng hộp thứ ba hoán vị C và B để cho ra kết quả cuối cùng. Máy này sắp xếp được bất kỳ hoán vị nào được nhập nào (như chương trình *sort3* làm).

Dĩ nhiên còn nhiều chi tiết cần phải được bàn luận kỹ lưỡng hơn trước khi có thể xây dựng được một sắp xếp thật sự dựa trên sơ đồ chúng ta vừa thiết kế ở trên. Ví dụ, phương pháp mã hóa dữ liệu nhập vẫn còn chưa được đặc tả. Để giải quyết vấn đề, ta có thể nghĩ đến việc xem mỗi mạch trong sơ đồ trên như một kênh với đủ đường truyền để có thể truyền một bit trên một đường truyền. Cũng có thể nghĩ đến phương án để cho bộ so sánh chuyển đổi đọc từng bit dữ

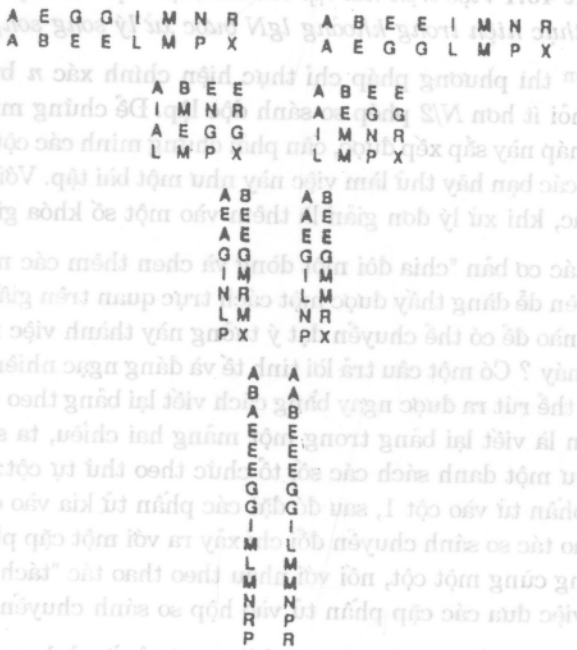


Hình 40.1 *Máy sắp xếp ba phần tử*

liệu từ một đường truyền đơn (bit dấu trước tiên). Vấn đề thời điểm cũng cần phải bàn thêm: phải có một cơ chế để đảm bảo không có thao tác so sánh hoán chuyển nào được thực hiện trước khi có dữ liệu nhập sẵn sàng. Chúng ta không có khả năng đi sâu hơn vào chi tiết của những vấn đề về thiết kế mạch như thế, mà sẽ tập trung vào các vấn đề cấp cao về việc nối liền các bộ xử lý đơn giản thành bộ xử lý so sánh hoán chuyển để giải các bài toán rộng hơn.

Để bắt đầu, ta quan tâm đến bài toán trộn hai tập tin đã được sắp, sử dụng một dãy các thao tác so sánh hoán chuyển độc lập với dữ liệu cần trộn, vì thế thích hợp cho việc thực hiện bằng phần cứng. Hình 40.2 cho thấy thao tác của phương pháp này trong việc trộn hai tập tin đã sắp có tám khóa thành một tập tin đã sắp.

Trước tiên, chúng ta viết nội dung hai tập tin theo hai hàng ngang, mỗi tập tin nằm trên một hàng, và thẳng hàng với tập tin ở trên. Trên từng cột của hai hàng dữ liệu này, hoán vị hai số trên hai tập tin nếu cần, sao cho số lớn hơn nằm dưới số nhỏ hơn trong cùng cột. Tiếp đó, chia đôi từng hàng, chèn thêm hai nửa mới vào bảng và thực hiện các thao tác so sánhhoán chuyển như trên cho các cặp số ở hàng hai và hàng ba (lưu ý rằng việc so sánh các cặp số trên những hàng khác là không cần thiết vì lần sắp trước đã làm). Theo cách này các dòng và cột của bảng đều được sắp. Đây là một tính chất cơ bản của phương pháp, bạn đọc có thể muốn kiểm chứng nhưng để đưa ra



Hình 40.2 *Trộn có tính chất tách và chen*

một bằng chứng chính xác là một bài tập đòi hỏi nhiều "mánh khóe" hơn là chúng ta có thể nghĩ đến.

Hóa ra đặc tính này được bảo tồn bởi cùng một thao tác chia đôi mỗi dòng và chèn các nửa dòng mới này vào bảng, và thực hiện so sánh hoán đổi từng cặp phần tử kề nhau trên cùng một cột mà trước đó các phần tử này thuộc về hai dòng khác nhau. Sau mỗi bước số dòng sẽ tăng đôi, số cột giảm đi một nửa và các dòng, cột này vẫn còn được sắp. Đầu tiên chúng ta có 16 cột và 1 dòng, sau đó là 8 cột và 2 dòng, sau đó là 4 dòng và 4 cột, 2 dòng và 8 cột, và cuối cùng là 16 dòng và 1 cột đã được sắp.

Tính chất 40.1 *Việc trộn hai tập tin đã được sắp với N phần tử có thể được thực hiện trong khoảng lgN bước xử lý song song.*

Nếu $N=2^n$ thì phương pháp chỉ thực hiện chính xác n bước, mỗi bước đòi hỏi ít hơn $N/2$ phép so sánh độc lập. Để chứng minh rằng phương pháp này sắp xếp được, cần phải chứng minh các cột vẫn còn được sắp, các bạn hãy thử làm việc này như một bài tập. Với các kích thước khác, khi xử lý đơn giản là thêm vào một số khóa giả.

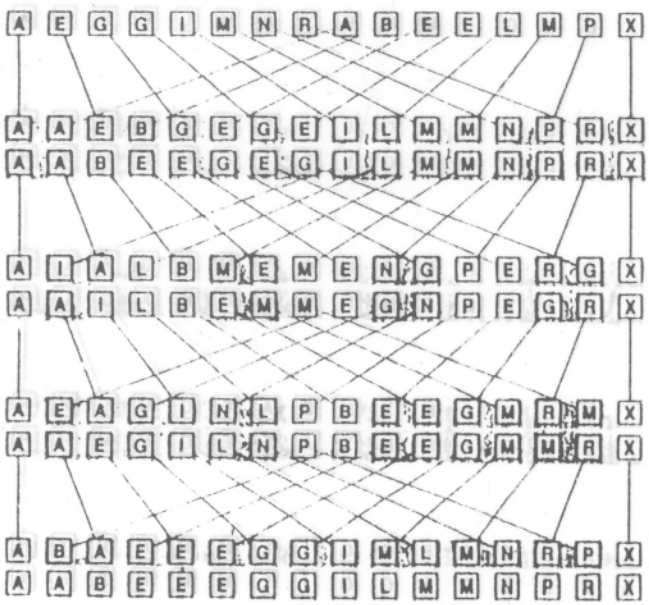
Thao tác cơ bản "chia đôi một dòng và chen thêm các nửa" được mô tả ở trên dễ dàng thấy được một cách trực quan trên giấy, nhưng làm cách nào để có thể chuyển đạt ý tưởng này thành việc nối mạch cho một máy ? Có một câu trả lời tinh tế và đáng ngạc nhiên cho vấn đề này có thể rút ra được ngay bằng cách viết lại bảng theo một cách khác. Hơn là viết lại bảng trong một mảng hai chiều, ta sẽ viết lại chúng như một danh sách các số, tổ chức theo thứ tự cột: đầu tiên đặt một phần tử vào cột 1, sau đó đặt các phần tử kia vào các cột 2, 3... Vì thao tác so sánh chuyển đổi chỉ xảy ra với một cặp phần tử kề nhau trong cùng một cột, nối với nhau theo thao tác "tách và chen" cần cho việc đưa các cặp phần tử vào hộp so sánh chuyển đổi.

Hình 40.3 thể hiện tương ứng những mô tả về sử dụng các bảng ở trên, ngoại trừ tất cả các bảng được viết theo thứ tự cột (bao gồm một bảng khởi động kích thước 1x16 với một tập tin, và bảng khác cho tập tin còn lại). Độc giả có thể kiểm chứng sự tương ứng giữa lưu đồ này và các bảng đã cho trước đó. Các hộp so sánh hoán đổi được vẽ rõ, và các dòng được vẽ cho thấy các phần tử dịch chuyển trong thao tác "chia đôi và chen thêm". Đáng ngạc nhiên là khi biểu diễn theo cách này, mỗi thao tác "chia đôi và chen thêm" được biến đổi một cách chính xác và đơn giản thành một mẫu kết nối. Mẫu kết nối này được gọi là *perfect shuffle* vì các mạch thật sự được chen vào theo kiểu người ta xáo trộn các quân bài.

Phương pháp này được đặt tên là *trộn chẵn lẻ* do K.E.Bather phát minh ra vào năm 1968. Nét đặc trưng cơ bản của phương pháp

này là tất cả các thao tác so sánh hoán đổi trong cùng một giai đoạn được thực hiện song song. Như đã đề cập trong tính chất 40.1, nó đáng được chú ý vì rõ ràng nó chứng tỏ hai tập tin N phần tử có thể được trộn trong $logN$ bước song song, sử dụng ít hơn $NlogN$ hộp so sánh hoán đổi. Từ mô tả ở trên thì nhận xét này có vẻ như được rút ra dễ dàng, nhưng thực tế vấn đề tìm kiếm một máy như thế đã thách thức các nhà khoa học trong một thời gian.

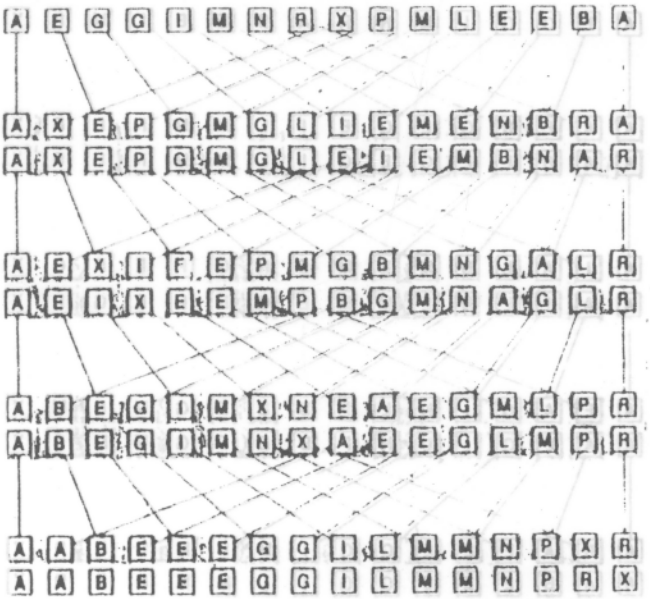
Bather cũng phát triển một thuật toán trộn khác gần như vậy (nhưng khó hiểu hơn), *trộn đôi một* dẫn đến một loại máy khác còn đơn giản hơn như được thể hiện trong hình 40.4. Phương pháp này có thể được mô tả chính xác bằng các thao tác "chia đôi và chèn thêm" trên các bảng như ở trên, ngoại trừ việc bắt đầu với tập tin



Hình 40.3 *Trộn chẵn lẻ bằng phương pháp xáo trộn hoàn toàn*

thứ hai trong thứ tự sắp ngược và luôn luôn làm động tác so sánh hoán đổi giữa hai cặp phần tử kề nhau trong một cột mà các phần tử này trước đó ở cùng một dòng. Chúng ta sẽ không chứng minh phương pháp này làm việc tốt, mà chú trọng đến khía cạnh khác: phương pháp *bitonic merge* đã khắc phục được sự rầy rà trong phương pháp *trộn chẵn lẻ* là các hộp so sánh hoán đổi trong ở giải đoạn đầu sẽ bị dời đi một vị trí so với gốc khi ở các giai đoạn sau. Như hình 40.4 cho thấy, mỗi giai đoạn của phương pháp *binotic merge* đều có cùng số bộ so sánh và cùng một vị trí.

Bây giờ không những có quy luật trong sự kết nối mà còn có quy luật trong các vị trí của các bộ so sánh hoán đổi. Có nhiều bộ so sánh hoán đổi hơn so với phương pháp trộn chẵn lẻ, nhưng đó không phải

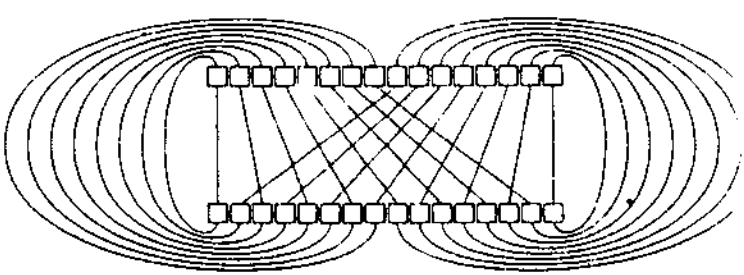


Hình 40.4 *Trộn đôi một bằng phương pháp xáo trộn hoàn toàn*

là vấn đề, vì chỉ cần cùng một số bước song song như ở phương pháp cũ. Điểm quan trọng của phương pháp là nó trực tiếp đưa đến cách thức thực hiện phép trộn mà sử dụng N bộ so sánh hoán đổi. ý tưởng là đơn giản mô phỏng các dòng trong bảng trên bằng hai dòng, vì thế tạo ra một máy chu kỳ (*cycling*) được mắc nối cùng nhau như trong hình vẽ 40.5. Một máy như vậy có thể thực hiện $\log N$ chu kỳ so sánh hoán đổixáo trộn, mỗi thao tác cho một giai đoạn trong hình vẽ.

Chú ý cẩn thận rằng đây không phải là cách thực hiện song song lý tưởng: vì chúng ta có thể trộn hai tập tin N phần tử với nhau chỉ sử dụng một bộ xử lý với số bước tỉ lệ N, chúng ta có thể hy vọng thực hiện phép trộn đó với một số bước cố định sử dụng N bộ xử lý. Tuy nhiên, trong trường hợp này chứng minh được là không thể đạt được mức độ lý tưởng và các máy mô tả ở trên đã đạt được cách thực hiện song song tốt nhất cho việc trộn sử dụng các bộ so sánh hoán đổi.

Kiểu kết nối *perfect shuffle* thích hợp cho nhiều vấn đề khác nhau. Ví dụ, nếu một ma trận $2^n x 2^n$ được giữ trong thứ tự dòng, thì *perfect shuffle* sẽ chuyển vị nó (chuyển nó về thứ tự cột). Một số ví dụ quan trọng hơn bao gồm phép biến đổi *Fourier nhanh* (sẽ xét ở chương sau), sắp xếp, ước lượng đa thức, và một số vấn đề khác. Mỗi bài toán trên có thể giải quyết bằng cách sử dụng một máy *cycling*



Hình 40.5 *Một máy xáo trộn hoàn toàn*

perfect shuffle với cách thức kết nối giống như sơ đồ ở trên nhưng các bộ xử lý phức tạp hơn. Một vài nhà nghiên cứu còn tính đến việc áp dụng kiểu kết nối *perfect shuffle* cho các máy đa năng.

CÁC MẢNG SYSTOLIC

(SYSTOLIC ARRAYS)

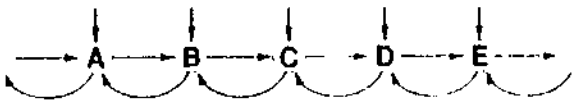
Một vấn đề của mô hình *perfect shuffle* là cần các dây nối dài. Hơn nữa có nhiều dây nối giao nhau: một shuffle n dây nối có số điểm giao nhau tỷ lệ với N^2 hai đặc điểm này làm cho việc xây dựng một máy *perfect shuffle* thật sự trở nên khó khăn hơn: dây nối dài dẫn đến việc trì hoãn thời gian, các điểm giao nhau làm cho chi phí kết nối cao và không thuận tiện.

Một cách tự nhiên để tránh khỏi hai nhược điểm trên là chỉ nối các bộ xử lý với các bộ xử lý có vị trí vật lý kề nó. Như đã nói ở trên, chúng ta thao tác đồng thời các bộ xử lý: tại mỗi bước, mỗi bộ xử lý đọc dữ liệu từ bộ xử lý "hàng xóm" của nó, thực hiện các xử lý và xuất kết quả vào các "hàng xóm". Hóa ra điều này cũng không cần thiết phải giới hạn, và vào năm 1978, H.T.Kung chỉ ra rằng một mảng các bộ xử lý như thế, mà ông gọi là *systolic array* (vì cách thức dữ liệu di chuyển trong chúng gợi nhớ đến nhịp tim đập), cho phép sử dụng các bộ xử lý rất hiệu quả cho nhiều vấn đề cơ bản.

Chúng ta sẽ xem xét việc sử dụng các systolic array cho bài toán nhân ma trận với vectơ như một ứng dụng điển hình. Giả sử có thao tác nhân ma trận vector sau

$$\begin{pmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{pmatrix} \begin{pmatrix} 1 \\ 5 \\ 2 \end{pmatrix} = \begin{pmatrix} 8 \\ 2 \\ -1 \end{pmatrix}$$

Việc tính toán xử lý này sẽ được thực thi trên một dòng của các bộ xử lý đơn mà mỗi bộ xử lý có 3 dòng nhập và 2 dòng xuất như



Hình 40.6 *Một mảng systolic*

trong hình 40.6. Chúng ta sử dụng 5 bộ xử lý vì đang biểu diễn dữ liệu nhập và đọc dữ liệu xuất trong cách thức tính toán thời gian kỹ lưỡng như mô tả dưới đây.

Suốt mỗi bước, mỗi bộ xử lý đọc một dữ liệu nhập từ bên *trái*, một từ *trên* và một bên *phải*; thực hiện một thao tác tính toán đơn giản; và ghi một dữ liệu xuất ra bên *trái*, và ghi một dữ liệu xuất ra bên *phải*. Theo đặc tả, dữ liệu xuất bên *phải* nhận bất cứ nội dung gì của dữ liệu nhập bên *trái*, và dữ liệu xuất bên *phải* nhận kết quả của phép cộng tích của phép nhân dữ liệu nhập trái và trên, với dữ liệu nhập *phải*. Một đặc tính quan trọng của các bộ xử lý này là chúng luôn thực hiện một sự biến đổi động từ dữ liệu nhập sang dữ liệu xuất; chúng không bao giờ phải ghi nhớ các giá trị đã tính toán (Điều này cũng đúng cho các bộ xử lý trong máy perfect shuffle). Đây là một luật nền tảng do sự khống chế ở mức thấp thiết kế phần cứng, vì việc bổ sung thêm bộ nhớ có thể rất đắt.

Đoạn vừa rồi đã đưa ra một "chương trình" cho máy systolic, để hoàn tất việc mô tả thuật toán, chúng ta cũng cần mô tả chính xác cách thức đưa dữ liệu nhập nào. Việc lập kế hoạch này là một đặc trưng cần thiết cho máy systolic, nó nhấn mạnh sự tương phản với máy *perfect shuffle*, là máy mà tất cả các dữ liệu nhập đầu được đưa vào cùng một lúc, và các dữ liệu xuất đều có thể sử dụng được ở một thời điểm sau đó.

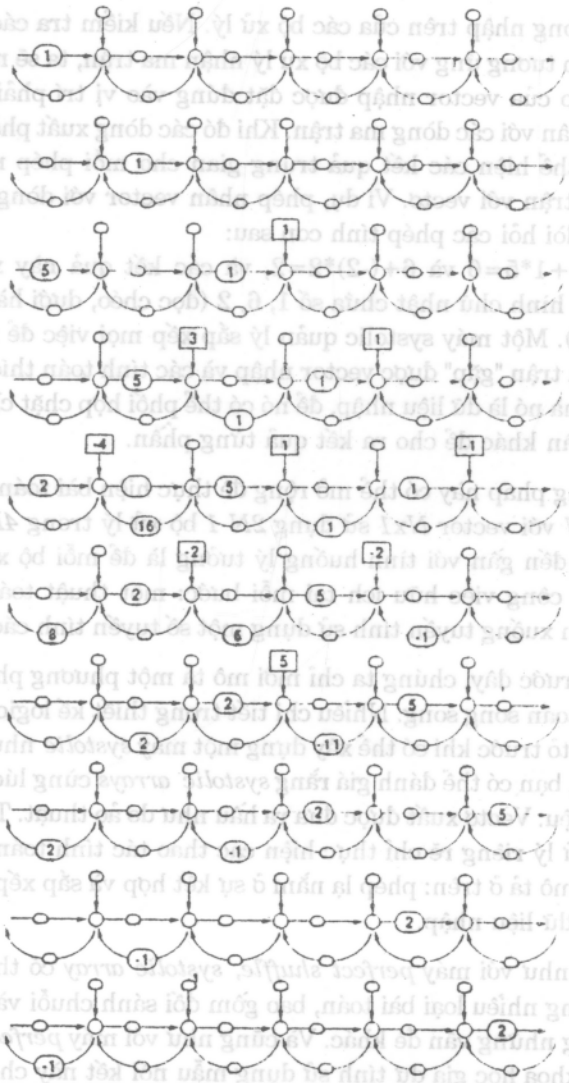
Kế hoạch tổng quát ở đây là đưa ma trận vào từ các dòng nhập phía trên của các bộ xử lý, phản chiếu (reflected) xung quanh đường chéo chính và quay 45°, và vector được đưa vào từ dòng nhập bên *trái* của bộ xử lý A, sau đó truyền sang các bộ xử lý khác. Các kết quả

trung gian được truyền từ *phải* sang *trái* của mảng, với output eventually xuất hiện bên dòng xuất phía trái của bộ xử lý A. Hình 40.7 cho thấy từng bước thực hiện trong ví dụ ban đầu.

Vector nhập được đưa vào qua dòng nhập *trái* của bộ xử lý A tại các bước 1,3,5, và sau đó truyền sang *phải* đến các bộ xử lý khác ở các bước sau. Ma trận nhập được đưa vào cho các dòng nhập *trên* ở bước 3, đối xứng lệch để các đường chéo *phải* sang *trái* của ma trận có thể được đưa vào ở các bước kế tiếp. Vector xuất hiện như dữ liệu xuất *trái* của bộ xử lý A ở các bước 6, 8, 10. (trong lưu đồ, vectơ này xuất hiện như một dữ liệu nhập phải của một bộ xử lý tưởng tượng ở bên trái của A, bộ xử lý này thu nhận đáp án).

Có thể theo vết việc tính toán thật sự bằng cách theo các dữ liệu nhập *phải* (dữ liệu xuất *trái*), chúng di chuyển từ *phải* sang *trái* của mảng. Tất cả việc tính toán đều cho ra kết quả 0 cho đến tận bước 3; sau bước 3, bộ xử lý C có 1 cho dòng nhập *trái* của nó và 2 cho dòng nhập *trên*, vì thế nó tính toán được kết quả 1, kết quả này được truyền qua cho bộ xử lý B như là dữ liệu nhập *phải* cho bước 4. Ở bước 4, bộ xử lý B có ba giá trị nhập đều khác 0, và nó tính toán được giá trị 16, để truyền qua cho bộ xử lý A ở bước 5. Đang lúc ấy, bộ xử lý D tính toán giá trị 1 cho bộ xử lý C sử dụng ở bước 5. Sau đó ở bước 5, bộ xử lý A tính toán được giá trị 8, giá trị này được xem như dữ liệu xuất đầu tiên ở bước 6; C tính toán giá trị 6 cho B sử dụng ở bước 6, và E tính toán giá trị đầu tiên khác 0 (-1) cho D sử dụng ở bước 6. Việc tính toán giá trị dữ liệu nhập thứ nhì được hoàn tất bởi B ở bước 6 và truyền qua cho A để xuất ở bước 8, và việc tính toán giá trị dữ liệu nhập thứ nhì được hoàn tất bởi C ở bước 7 và truyền qua cho B và A để xuất ở bước 10.

Một khi quá trình xử lý đã được mô tả ở mức độ chi tiết như trên, phương pháp trở nên dễ hiểu hơn ở mức cao hơn. Các số trong phần giữa của hình 40.7 (trong hình chữ nhật) chỉ đơn giản là bản sao của ma trận nhập, được quay và phản chiếu như đã đòi hỏi để thể hiện



Hình 40.7 *Nhân ma trận với vectơ bằng mảng systolic*

cho các dòng nhập trên của các bộ xử lý. Nếu kiểm tra các số ở các dòng nhập tương ứng với các bộ xử lý nhận ma trận, ta sẽ nhận thấy ba bản sao của vector nhập được đặt đúng vào vị trí phải và đúng lượt để nhân với các dòng ma trận. Khi đó các dòng xuất phải của các bộ xử lý thể hiện các kết quả trung gian cho mỗi phép nhân một dòng ma trận với vectơ. Ví dụ, phép nhân vector với dòng giữa của ma trận đòi hỏi các phép tính con sau:
1*1=1, 1+1*5=6 và 6+(-2)*2=2, và các kết quả này xuất hiện trong các hình chữ nhật chứa số 1, 6, 2 (đọc chéo, dưới hàng 1 1 -2 một bước). Một máy systolic quản lý sắp xếp mọi việc để mỗi phần tử của ma trận "gặp" được vector nhập và các tính toán thích hợp tại bộ xử lý mà nó là dữ liệu nhập, để nó có thể phối hợp chặt chẽ với các thành phần khác để cho ra kết quả từng phần.

Phương pháp này có thể mở rộng để thực hiện bài toán nhân ma trận *NxN* với vector *Nx1* sử dụng *2N-1* bộ xử lý trong *4N-2* bước. Điều này đến gần với tình huống lý tưởng là để mỗi bộ xử lý thực hiện một công việc hữu ích tại mỗi bước: một thuật toán bậc hai được giảm xuống tuyến tính sử dụng một số tuyến tính các bộ xử lý.

Như trước đây, chúng ta chỉ mới mô tả một phương pháp chung của tính toán song song. Nhiều chi tiết trong thiết kế logic cần được làm sáng tỏ trước khi có thể xây dựng một máy *systolic* như trên. Từ ví dụ này, bạn có thể đánh giá rằng *systolic arrays* cùng lúc đơn giản và hữu hiệu. Vectơ xuất được đưa ra hầu như do ảo thuật. Tuy nhiên, mỗi bộ xử lý riêng rẽ chỉ thực hiện các thao tác tính toán đơn giản như vừa mô tả ở trên: phép lạ nằm ở sự kết hợp và sắp xếp thời gian thể hiện dữ liệu nhập.

Cũng như với máy *perfect shuffle, systolic array* có thể được sử dụng trong nhiều loại bài toán, bao gồm đối sánh chuỗi và nhân ma trận cùng những vấn đề khác. Và cũng như với máy *perfect shuffle,* một vài khoa học gia dự tính sử dụng mẫu nối kết này cho các máy song song đa dụng.

TRIỂN VỌNG

Qua việc nghiên cứu máy *perfect shuffle* và máy *systolic array* ta đã thấy được thiết kế phần cứng có thể tác động đáng chú ý trên việc thiết kế thuật toán, và một dự đoán rõ ràng là các thay đổi có thể đem đến những thuật toán mới thú vị và một hứng khởi mới cho những người thiết kế thuật toán.

Trong khi đây đang còn là một lĩnh vực nghiên cứu lý thú và hứa hẹn nhiều kết quả trong tương lai, ta phải đưa ra một vài ghi chú tỉnh táo. Trước tiên, cần phải có nhiều cố gắng về kỹ thuật để chuyển đổi các sơ đồ tổng quát được phác thảo ở trên thành các máy thật sự có khả năng vận hành tốt. Đối với một số ứng dụng, phí tổn tài nguyên không thể biện minh được vì một máy thuật giải với bộ vi xử lý truyền thống thực hiện trên một máy truyền thống vẫn có thể giải quyết đủ tốt vấn đề. Ví dụ, nếu một người có nhiều thể hiện của cùng một vấn đề và có nhiều bộ vi xử lý để giải quyết vấn đề đó, thì ý tưởng xử lý song song có thể đạt được đơn giản bằng cách để mỗi bộ vi xử lý (sử dụng thuật toán truyền thống) thao tác trên một phiên bản, mà không cần nối kết các bộ xử lý này với nhau. Nếu có N tập tin cần sắp xếp và N bộ xử lý để sắp xếp chúng, tại sao không để đơn giản dùng mỗi bộ xử lý cho một việc sắp xếp hơn là ép chúng cùng làm việc với nhau ?

Rất khó lượng giá hiệu quả của các chiến lược xử lý song song khác nhau trên việc thực hiện thuật toán. Tất cả các vấn đề đã bàn cãi ở chương 6 và 7 cần được xem xét lại, với một sự phức tạp bổ sung, một máy tự nó trở thành một biến. Cách dễ nhất và phổ biến nhất để so sánh hai máy là cho thực hiện cùng một chương trình trên cả hai máy, lại có thể đánh lừa chúng ta hoàn toàn nếu thuật toán cơ sở của chương trình liên quan chặt chẽ đến kiến trúc của một trong hai máy. Lúc đó máy có kiến trúc phù hợp với thuật toán sẽ thực hiện chương trình tốt hơn nhiều so với máy kia. Để so sánh chính xác hai máy, cần tập trung vào vấn đề cần được giải quyết và

xem xét thuật toán tốt nhất cho vấn đề đó trên từng máy. Liệu kiến trúc máy mới có cho phép chúng ta giải quyết các vấn đề không thể giải quyết trên kiến trúc máy cũ ?

Các kỹ thuật mà chúng ta vừa bàn luận đến trong chương này hiện nay chỉ lý giải được ưu điểm cho các ứng dụng với yêu cầu thời gian hay không gian đặc biệt. Trong việc nghiên cứu nhiều sơ đồ khác nhau của xử lý song song và hiệu quả của chúng trên sự thực hiện các thuật toán, chúng ta có thể trông chờ sự phát triển một máy song song đa dụng có khả năng thực hiện được cải tiến cho nhiều lớp thuật toán hơn.

BÀI TẬP

1. Phác họa phương pháp song song cho thuật toán Quicksort.

2. Chứng minh phương pháp *tách - và - chèn* bảo đảm các cột được sắp xếp.

3. Viết chương trình Pascal để trộn tệp dựa vào phương pháp của Batcher

4. Viết chương trình Pascal để trộn tệp dựa vào phương pháp của Batcher nhưng không thực hiện bất kỳ xáo trộn nào.

5. Cần bao nhiêu perfect shuffle để mang tất cả các phần tử trong mảng kích thước 2^n trở về các vị trí gốc của chúng ?

6. Hãy vẽ một bảng giống như Hình 40.7 để minh họa thao tác của phép nhân ma trận và vector systolic cho bài tập sau

$$\begin{pmatrix} 2 & 1 & 4 \\ 3 & 0 & 1 \\ 1 & -1 & 3 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ -1 \end{pmatrix} = \begin{pmatrix} 3 \\ 8 \\ -1 \end{pmatrix}$$

7. Viết một chương trình Pascal giả lập thao tác mảng systolic dùng để nhân ma trận NxN và vectơ Nx1.

8. Cho biết làm thế nào để chuyển vị ma trận bằng phương pháp mảng systolic.

9. Cần bao nhiêu bộ xử lý và bao nhiêu bước khi dùng máy systolic để nhân ma trận MxN và vector Nx1.

10. Hãy trình bày một sơ đồ song song đơn giản để nhân ma trận và vectơ bằng cách dùng các bộ xử lý có khả năng "nhớ" các giá trị tính được.

41

BIẾN ĐỔI FOURIER NHANH

Một trong những thuật toán số học được sử dụng rộng rãi là *biến đổi Fourier nhanh*, một phương pháp hữu hiệu để tiến hành các tính toán căn bản. Biến đổi Fourier nhanh có tầm quan trọng lớn trong giải tích toán học và là đối tượng của nhiều sách nghiên cứu. Sự xuất hiện của phương pháp tính toán hiệu quả là một bước ngoặc trong lịch sử tính toán.

Biến đổi Fourier có rất nhiều ứng dụng. Nó là cơ sở của nhiều thao tác cơ bản trong xử lý tín hiệu. Hơn nữa, như ta sẽ thấy, nó cung cấp một phương pháp thuận tiện để cải thiện hiệu quả của các thuật toán số học thông thường. Mục đích của phần này là phác qua cơ sở toán học của biến đổi Fourier và nói tổng quát về những áp dụng của nó. Chúng ta sẽ xem xét đặc trưng của thuật toán căn bản này trong khuôn khổ các thuật toán mà ta đã khảo sát.

Đặc biệt, chúng ta sẽ xem xét phương pháp sử dụng thuật toán này để rút gọn thực sự thời gian cần thiết để nhân đa thức, bài toán đã khảo sát ở chương 36. Chỉ cần một số ít kiến thức về giải tích phức, ta có thể cho thấy bằng cách nào biến đổi Fourier có thể sử dụng để nhân đa thức và có thể đánh giá được thuật toán biến đổi Fourier nhanh.

TÍNH GIÁ TRỊ - NHÂN - NỘI SUY

Chiến lược chung của phương pháp cải tiến việc nhân đa thức mà ta đã xem xét nhờ điểm thuận lợi là một đa thức bậc *N-1* được xác định hoàn toàn bởi giá trị của nó tại *N* điểm khác nhau. Khi ta nhân hai đa thức cấp *N-1*, ta được một đa thức bậc *2N-1*; nếu ta có thể tìm được giá trị của đa thức tích tại *2N-1* điểm thì nó hoàn toàn được xác định. Mặt khác, giá trị của kết quả tại một điểm bất kỳ được tính bằng tích số hai giá trị của hai đa thức ban đầu tại điểm đó.

Do trên, ta có sơ đồ tổng quát sau để nhân hai đa thức bậc N-1.

Tính giá trị hai đa thức ban đầu tại *2N-1* điểm khác nhau.

Nhân hai giá trị tìm được tại mỗi điểm.

Nội suy để tìm kết quả mà các giá trị đã tính tại các điểm đã cho.

Ví dụ, để tính $r(x)=p(x)q(x)$ với $p(x)=1+x+x^2$, $q(x)=2-x+x^2$ ta có thể tính $p(x)$, $q(x)$ tại năm điểm bất kỳ, chẳng hạn -2,-1,0,1,2 để được:

$$[p(-2), p(-1), p(0), p(1), p(2)] = [3, 1, 1, 3, 7]$$

$$[q(-2), q(-1), q(0), q(1), q(2)] = [8, 4, 2, 2, 4]$$

Nhân từng số hạng tương ứng ta có:

$$[r(-2), r(-1), r(0), r(1), r(2)] = [24, 4, 2, 6, 28]$$

Theo công thức nội suy Lagrange:

$$\begin{aligned} r(x) = {} & 24 \frac{x+1}{-2+1}\frac{x-0}{-2-0}\frac{x-1}{-2-1}\frac{x-2}{-2-2} \\ & + 4 \frac{x+2}{-1+2}\frac{x-0}{-1-0}\frac{x-1}{-1-1}\frac{x-2}{-1-2} \\ & + 2 \frac{x+2}{0+2}\frac{x+1}{0+1}\frac{x-1}{0-1}\frac{x-2}{0-2} \\ & + 6 \frac{x+2}{1+2}\frac{x+1}{1+1}\frac{x-0}{1-0}\frac{x-2}{1-2} \\ & + 28 \frac{x+2}{2+2}\frac{x+1}{2+1}\frac{x-0}{2-0}\frac{x-1}{2-1} \end{aligned}$$

Rút gọn ta được:

$$r(x)=2+x+2x^2+x^4$$

Như đã nói từ trước, phương pháp này không phải là một thuật toán hấp dẫn vì các thuật toán tốt nhất cho hai công việc tính giá trị (áp dụng phương pháp Horner) và nội suy (công thức Lagrange) đòi hỏi N^2 phép toán. Tuy nhiên, vẫn có hy vọng tìm được một thuật toán tốt hơn nếu ta tìm được $2N-1$ điểm để tính giá trị và nội duy dễ nhất.

CÁC CĂN SỐ PHỨC CỦA ĐƠN VỊ

Ta nhắc lại một số kiến tức về giải tích phức. Số $i=\sqrt{-1}$ là một *số ảo*. Một số phức bao gồm hai phần thực và ảo, thường được viết là $a + bi$ với a, b là hai số thực. Để nhân hai số thực, ta tiến hành bình thường và thay $i^2=-1$. Ví dụ:

$$(a + bi)(c + di) = (ac - bd) + (ad + bc)i$$

Trong khi nhân các số phức, đôi khi phần thực (hay phần ảo) có thể mất đi. Ví dụ:

$$(1 - i)(1 - i) = -2i$$
$$(1 + i)^4 = -4$$
$$(1 + i)^8 = 16$$

Chia phương trình cuối cho $16=\sqrt{2}^8$, ta có:

$$\left(\frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}\right)^8 = 1$$

Nói chung, có nhiều số phức khi lũy thừa lên ta được số 1. Những số như vậy gọi là căn của đơn vị. Thực ra, với mỗi N, có đúng N số phức Z sao cho $Z^N=1$. Một trong chúng, đặt tên là w_N, gọi là căn bậc N chính của đơn vị, các bậc N khác nhận được bằng cách lũy thừa lên bậc k, $k=0, 1, 2,..., N-1$. Chẳng hạn, ta có thể liệt kê ra các căn bậc tám của đơn vị như sau:

$$w_8^0, w_8^1, w_8^2, w_8^3, w_8^4, w_8^5, w_8^6, w_8^7$$

Căn số đầu tiên, w_{80} là số 1 và căn số thứ hai $w8^1$, là căn bậc N chính. Với N chẵn, ta cũng có là -1 (vì không quan trọng. Chúng ta sẽ chỉ dùng các tính chất căn bản của căn bậc N dựa trên định nghĩa của căn bậc N.

TÍNH GIÁ TRỊ TẠI CÁC CĂN CỦA ĐƠN VỊ

Điểm khó khăn của ta là một chương trình để tính giá trị của một đa thức bậc *N-1* tại các căn bậc N của đơn vị. Chương trình này đổi N hệ số xác định đa thức thành trị tính tại các căn bậc N của đơn vị.

Bước đầu tiên của quá trình nhân đa thức tức là tính giá trị của một đa thức bậc *N-1* tại *2N-1* điểm. Tuy nhiên, nếu ta xem đa thức bậc *N-1* như là một đa thức bậc *2N-2* với *N-1* hệ số (của các số hạng có bậc cao nhất) là zero.

Thuật toán được sử dụng để tính giá trị của đa thức bậc *N-1* tại N điểm sẽ dựa trên một chiến lược xử lý "chia để trị" (divide-and-conquer). Không giống như việc chia đa thức ở giữa (như thuật toán nhân ở chương 4), chúng ta sẽ chia đa thức thành hai phần: một phần gồm các số hạng bậc lẻ, phần còn lại gồm các số hạng bậc chẵn. Ví dụ với $N=8$, sự sắp xếp các số hạng được thực hiện như sau:

$$\begin{aligned} p(x) &= p_0+p_1x^1x+p_2x^2+p_3x^3+p_4x^4+p_5x^5+p_6x^6+p_7x^7 \\ &= (p_0+p_2x^2+p_4x^4+p_6^6) + x\,(p_1+p_3x^2+p_5x^4+p_7x^6) \\ &\equiv p_e(x^2)+x\,p_0(x^2) \end{aligned}$$

Căn bậc N của đơn vị rất thích hợp với kiểu phân tích này vì nếu ta bình phương một căn của đơn vị, ta được một căn khác của đơn vị. Tốt hơn nữa, với N chẵn, nếu ta bình phương một căn bậc N của đơn vị, ta sẽ được một căn bậc $\frac{1}{2}N$ của đơn vị. Để tính giá trị của đa thức với N hệ số tại N điểm, ta chia chúng thành hai đa thức có

$\frac{1}{2}N$ hệ số. Các đa thức này chỉ cần tính tại $\frac{1}{2}N$ điểm (tại các căn bậc $\frac{1}{2}N$ của đơn vị).

Để minh họa, ta hãy xét việc tính một đa thức bậc 7 p(x) tại các căn bậc 8 của đơn vị.

$$W_8{:}w_8{}^0,\ w_8{}^1,\ w_8{}^2,\ w_8{}^3,\ w_8{}^4,\ w_8{}^5,\ w_8{}^6,\ w_8{}^7$$

Vì $w_8{}^0$=-1, dãy số trên là:

$$W_8{:}w_8{}^0,\ w_8{}^1,\ w_8{}^2,\ w_8{}^3,\ -w_8{}^0,\ -w_8{}^1,\ -w_8{}^2,\ -w_8{}^3$$

Bình phương mỗi số hạng của dãy trên ta được một dãy $\{W_4\}$ các căn bậc 4 của đơn vị.

$$W_8{:}w_4{}^0,\ w_4{}^1,\ w_4{}^2,\ w_4{}^3,\ w_4{}^0,\ w_4{}^1,\ w_4{}^2,\ w_4{}^3$$

Bây giờ, đẳng thức

$$p(x)=p_e(x^2)+xp_0(x^2)$$

cho ta ngay cách tính $p(x)$ tại tám giá trị căn bậc tám của đơn vị. Đầu tiên, ta tính $p_e(x)$ và $p_0(x)$ tại các căn bậc 4 của đơn vị. Sau đó ta có:

$$\begin{aligned}
p(w_8{}^0)&=p_e(w_4{}^0)+w_8{}^0p_0(w_4{}^0)\\
p(w_8{}^1)&=p_e(w_4{}^1)+w_8{}^1p_0(w_4{}^1)\\
p(w_8{}^2)&=p_e(w_4{}^2)+w_8{}^2p_0(w_4{}^2)\\
p(w_8{}^3)&=p_e(w_4{}^3)+w_8{}^3p_0(w_4{}^3)\\
p(w_8{}^4)&=p_e(w_4{}^4)+w_8{}^4p_0(w_4{}^4)\\
p(w_8{}^5)&=p_e(w_4{}^{5)+w}{}_8{}^5p_0(w_4{}^5)\\
p(w_8{}^6)&=p_e(w_4{}^6)+w_8{}^6p_0(w_4{}^6)\\
p(w_8{}^7)&=p_e(w_4{}^7)+w_8{}^7p_0(w_4{}^7)
\end{aligned}$$

Nói chung, để đánh giá $p(x)$ tại căn bậc N của đơn vị, ta tính một cách đệ quy trên căn bậc $p_e(x)$ và $p_0(x)$ tại căn bậc $\frac{1}{2}N$ của đơn vị và tiến hành N phép nhân như trên. Cách này chỉ tiến hành được khi N chẵn, vậy để làm bằng đệ quy được, ta giả sử N là một lũy thừa của 2. Quá trình đệ quy dừng khi $N=2$ và ta có p_0+p_1x cần được tính tại

1 và -1 với các kết quả p_0+p_1 và p_0-p_1.

Tính chất 41.1 *Một đa thức bậc N-1 có thể tính giá trị tại các căn bậc N của đơn vị với khoảng NlgN phép nhân.*

Tính chất trên có thể thấy được nếu ta dùng công thức $M(N)=2M(N/2)+N(M(N)$ là số phép nhân cần thiết để tính giá trị của các đa thức bậc N tại các căn bậc N của đơn vị). Dùng công thức 4 ở chương 6, ta có $M(N)=NlgN$.

NỘI SUY TỪ CÁC CĂN ĐƠN VỊ

Bây giờ ta đã có một cách tính giá trị của đa thức khá nhanh trên một tập hợp đặc biệt các điểm. Ta cần một phương pháp nhanh để nội suy một đa thức trên cùng một tập hợp. Từ đó ta sẽ có một cách tính phép nhân đa thức nhanh. Điều ngạc nhiên là, việc chạy chương trình tính toán các giá trị đa thức tại các căn đơn vị sẽ cho ta phép nội suy. Đó chỉ là một ví dụ của phép biến đổi Fourier ngược.

Chẳng hạn, với $N=8$, bài toán nội suy là tìm một đa thức:

$$r(x)=r_0+r_1x+r_2x^2+r_3x^3+r_4x^4+r_5x^5+r_6x^6+r7x^7$$

với các giá trị

$$r(w_8^0)=s_0\ r(w_8^1)=s_1\ r(w_8^2)=s_2\ r(w_8^3)=s_3$$
$$r(w_8^4)=s_4\ r(w_8^5)=s_5\ r(w_8^6)=s_6\ r(w_8^7)=s_7$$

Khi các điểm được xét là các căn của đơn vị, th ực sự bài toán nội suy là bài toán "ngược" của bài toán tính toán. Nếu ta đặt:

$$s(x)=s_0+s_1x+s_2x^2+s_3x^3+s_4x^4+s_5x^5+s_6x^6+s7x^7$$

thì ta có thể nhận được các hệ số r_i *(i = 0, 1, ..7)* bằng cách tính giá trị của đa thức $s(x)$ tại nghịch đảo của căn bậc N của đơn vị.

$$W_8^{-1}:w_8^0,w_8^{-1},w_8^{-2},w_8^{-3},w_8^{-4},w_8^{-5},w_8^{-6},w_8^{-7}$$

nhưng đó chính là dãy với thứ tự đổi khác:

$$W_8^{-1}: w_8^0, w_8^7, w_8^6, w_8^5, w_8^4, w_8^3, w_8^2, w_8^1$$

Nói khác đi, ta có thể sử dụng cùng một chương trình cho phép nội suy và phép tính giá trị: chỉ đơn giản là sắp xếp lại các điểm dùng để tính giá trị.

Chứng minh các lý luận trên cần vài tính toán cho các tổng hữu hạn; các tính toán được thực hiện như sau:

$$\begin{aligned} s(w_N^{-1}) &= \sum_{0\le j\le N} s_j (w_N^{-1})^j \\ &= \sum_{0\le j\le N} r(w_N^j)(w_N^{-t})^j \\ &= \sum_{0\le j\le N} \sum_{0\le i\le N} r_i (w_N^j)^i (w_N^{-t})^j \\ &= \sum_{0\le j\le N} \sum_{0\le i\le N} r_i w_N^{ij(i-t)} \\ &= \sum_{0\le i\le N} r_i \sum_{0\le j\le N} w_N^{ij(i-t)} = N r_t \end{aligned}$$

Trong tính toán trên, ta sử dụng công thức:

$$\sum_{0\le j<N} w_N^{ij(i-t)} = \frac{w_N^{(i-t)N}-1}{w_N^{(i-t)}-1} = 0$$

Chú ý rằng một thừa số được thêm vào là N. Đây chính là "định lý ngược" cho biến đổi Fourier rời rạc. Từ đây ta có:

Tính chất 41.2 *Một đa thức bậc N - 1 có thể nội suy tại các căn bậc N của đơn vị với khoảng chừng NlgN phép nhân.*

Các lý luận toán học nói trên có vẻ phức tạp. Tuy nhiên kết quả lại rất dễ áp dụng, để nội suy một đa thức theo các điểm căn bậc N của đơn vị, ta sử dụng cùng một chương trình như khi tính giá trị,

sắp xếp lại kết quả ta sẽ có được hệ số của đa thức.

CHƯƠNG TRÌNH

Hiện tại ta đã có từng phần thuật toán nhân hai đa thức chỉ sử dụng khoảng *NlgN* phép toán. Sơ đồ tổng quát là:

Tính giá trị của đa thức nhập vào tại các căn bậc *(2N - 1)* của đơn vị.

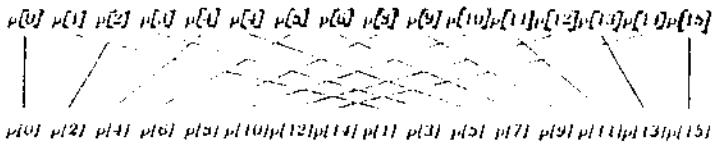
Nhân hai giá trị tìm được tại mỗi điểm.

Nội suy để tìm kết quả bằng cách tính giá trị của đa thức xác định chỉ bởi các số được tính tại các căn bậc *(2N - 1)* của đơn vị.

Sự mô tả trên có thể chuyển trực tiếp sang thành một chương trình trong đó sử dụng một thủ tục (*procedure*) để tính giá trị của đa thức bậc *N-1* tại các căn bậc *N* của đơn vị. Tuy nhiên, các phép toán này thực hiện trên số phức mà trong Pascal lại không có xây dựng kiểu số phức. Do đó, ta cần có một thủ tục để xác định kiểu số phức cũng như các phép toán trên các số này. Với giả định là kiểu số phức đã có, ta có chương trình tính giá trị sau:

```
eval(p,outN,0);
eval(q,outN,0);
for i:=0 to outN do r[i]:=p[i]*q[i];
eval(r,outN,0);
for i:=1 to N do
   begin t:=r[i]; r[i]:=r[outN+1-i]; r[outN+1-i]:=t; end;
for i:=0 to outN do r[i]:=r[i]/(outN+1);
```

Trong chương trình này, ta giả sử biến toàn cục *outN* là *2N-1* và *p, q, r* là các mảng số phức đánh từ *0* tới *2N-1*. Hai đa thức được nhân *p. q* có cấp *N-1* và các hệ số thêm vào để được mảng *2N-1* phần tử



Hình 41.1 *Sơ đồ "không xáo trộn lý tưởng" cho FFT*

là các số không. Thủ tục *eval* thay các hệ số đã cho như là biến thứ nhất của đa thức bởi các giá trị của đa thức tính tại các căn của đơn vị. Biến thứ hai xác định bậc của đa thức và biến thứ ba sẽ mô tả dưới đây. Chương trình trên tính tích của *p*, *q* và để kết quả ở mảng *r*.

Như ta đã thấy từ trước, chương trình đệ quy có các mảng có thể gây khó khăn khi cài đặt. Ngoài ra, còn một bài toán thông thường là quản lý vùng chứa (storage) bằng cách dùng lại nó một cách thông minh. Điều ta cần ở đây là có một thủ tục đệ quy đưa vào một mảng $N+1$ hệ số và cho ra $N+1$ giá trị trong cùng một mảng. Tuy nhiên, quá trình đệ quy lại bao gồm việc xử lý hai mảng rời nhau: các hệ số lẻ và chẵn. Sự xáo trộn lý tưởng (perfect shuffle) của chương trước là cái mà ta cần. Ta có thể đưa các hệ số lẻ vào trong một mảng con (nửa đầu) và các hệ số chẵn vào một mảng con (nửa sau) bằng cách thực hiện sự "không xáo trộn lý tưởng" (perfect unshuffle) của dữ liệu nhập, như sơ đồ trong hình 41.1 với $N=15$.

Dĩ nhiên các giá trị căn số phức cũng cần cài đặt. Ta có.

$$w_N^j = \cos(\frac{2\pi j}{N+1}) + i\sin(\frac{2\pi j}{N+1})$$

Sử dụng các hàm lượng giác quy ước ta có thể tính dễ dàng giá trị w_{Nj}. Trong chương trình dưới đây, mảng w được giả định là chứa các căn bậc ($autN+1$) của đơn vị.

Ta có chương trình sau của FFT (Fast Fourier Transform : *biến đổi Fourier nhanh*).

```
procedure eval(var p:poly; N,k:integer);
   var i,j:integer;
   begin
   if N=1
   then begin t:=p[k]; p1:=p[k+1]; p[k]:=t+p1; p[k+1]:=t-p1;
        end
   else  begin
         for i:=0 to N div 2 do
            begin j:=k+2*i; t[i]:=p[j]; t[i+1+N div 2]:=p[j]+1;
            end;
         for i:=0 to N do p[k+i]:=t[i];
         eval(p,N div 2,k);
         eval(p,N div 2,)k+1+N div 2);
         j:=(outN+1) div (N+1);
         for i:=0 to N div 2 do
            begin    t:=w[i*j]*p[k+(N div 2)+1+i];
                     t[i]:=p[k+i]+t; t[i+(N div 2)+1]:=p[k+i]*t
            end;
         for i:=0 to N do p[k+i]:=t[i]
         end;
   end;
```

Chương trình này chuyển đa thức bậc N vào mảng con $p[k... k + N]$ bằng cách dùng phương pháp đệ quy đã nói qua ở trên. (Để đơn giản, mã này giả sử rằng $N+1$ là một lũy thừa của 2, mặc dầu điều này bỏ đi dễ dàng). Nếu $N=1$, ta dễ dàng tính giá trị tại 1 và -1. Với $N \neq 1$ thủ tục này đầu tiên sẽ xáo trộn (shuffles), rồi gọi đệ quy chính nó để chuyển sang bài toán cho $N/2$, sau đó kết hợp các kết quả tính toán như đã mô tả ở trên. Để nhận được các căn vị cần thiết, chương trình chọn từ mảng tại một khoảng xác định bởi biến i. Ví dụ, nếu $outN=15$, các căn bậc 4 của đơn vị tìm thấy trong $w[0], w[4], w[8], w[12]$. Điều này làm giảm bớt số tính toán các căn của đơn vị sử dụng.

Tính chất 41.3 *Hai đa thức cấp N có thể được nhân với 2NlgN+0(N) phép nhân phức.*

Như đã nói, sự áp dụng của FFT rộng hơn nhiều so với phép toán nhân đa thức chỉ ra ở đây; và FFT đã được sử dụng mạnh mẽ và khảo sát trong nhiều lĩnh vực khác nhau. Tuy nhiên, các nguyên tắc chính trong các áp dụng cũng tương tự như trong việc nhân đa thức được xem xét ở đây. FFT là một ví dụ cổ điển về phương pháp "chia-để-trị" (divide-and-conquer).

BÀI TẬP

1. Có thể cải tiến như thế nào thuật toán tính giá trị - nhân - nội suy để nhân hai đa thức $p(x)$, $q(x)$ với các nghiệm đã biết $p_0, p_1, \ldots p_N$ và $q_0, q_1 \ldots p_N$?

2. Tìm một tập N số thực mà giá trị của đa thức bậc N tính tại đó có số phép toán ít hơn N^2.

3. Tìm một tập N số thực mà phép nội duy đa thức bậc N trên tập đó dùng ít hơn N^2 phép toán.

4. Giá trị của $w_N{}^M$ với $M>N$ là gì ?

5. Có nên sử dụng FFT để nhân các đa thức có nhiều hệ số bằng 0.

6. Chương trình FFT ba lần gọi *eval*, tương tự như thuật toán nhân ở chương 36 ba lần gọi *mult*. Vì sao cài đặt FFT hiệu quả nhiều hơn ?

7. Tìm một cách nhân hai số phức sử dụng ít hơn 4 phép nhân.

8. Vùng chứa bao nhiêu sẽ được sử dụng bởi FFT nếu ta không dùng bài toán quản lý vùng chứa với sự xáo trộn lý tưởng (*perfect shuffle*).

9. Vì sao không thể dùng các kỹ thuật như sự xáo trộn lý tưởng cho các mảng khai báo động (*dynamically declared arrays*) trong thủ tục nhân đa thức ở chương 36 ?

10. Viết một chương trình hiệu quả để nhân một đa thức cấp N với một đa thức cấp M (không nhất thiết là các lũy thừa của 2).

42

QUY HOẠCH ĐỘNG

Đối với nhiều thuật toán chúng ta đã tìm hiểu ở các chương trước, nguyên lý "chia để trị" thường đóng vai trò chủ đạo trong việc thiết kế thuật toán: để giải quyết một bài toán lớn, chúng ta chia nó thành nhiều bài toán con có thể được giải quyết độc lập. Trong phương pháp quy hoạch động, việc thể hiện nguyên lý này được đẩy đến cực độ: khi không biết chắc cần giải quyết bài toán con nào, chúng ta giải quyết tất cả các bài toán con và lưu trữ những lời giải này với mục đích sử dụng lại chúng theo một sự phối hợp nào đó để giải quyết các bài toán tổng quát hơn. Cách tiếp cận này được sử dụng rộng rãi trong lĩnh vực vận trù học. Thuật ngữ "quy hoạch" mà chúng tôi sử dụng ở đây ngụ ý nói đến quá trình đưa bài toán ban đầu về một dạng nào đó có thể áp dụng phương pháp này để giải. Đó là cả một nghệ thuật, do vậy chúng tôi sẽ không đi sâu vào chi tiết mà chỉ xem xét một vài ví dụ để qua đó giới thiệu cùng các bạn cách tiếp cận này.

Khi sử dụng phương pháp quy hoạch động để giải quyết vấn đề, ta có thể gặp phải hai khó khăn sau. Một là, không phải lúc nào sự kết hợp lời giải của các bài toán con cũng cho ra lời giải của bài toán lớn hơn. Hai là, số lượng các bài toán con cần giải quyết và lưu trữ đáp án có thể rất lớn, không thể chấp nhận được. Cho đến nay, chưa ai xác định được một cách chính xác những bài toán nào có thể được giải quyết hiệu quả bằng phương pháp quy hoạch động. Có những vấn đề quá phức tạp và khó khăn mà xem ra không thể ứng dụng

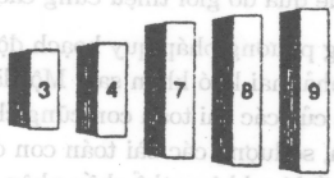
quy hoạch động để giải quyết được, trong khi cũng có những bài toán quá đơn giản khiến cho việc sử dụng quy hoạch động để giải quyết lại kém hiệu quả hơn so với dùng các thuật toán kinh điển.

Trong chương này chúng ta sẽ khảo sát một số bài toán có thể dùng quy hoạch động giải quyết một cách hiệu quả. Những vấn đề này đều liên quan đến bài toán tìm phương án tối ưu để thực hiện một công việc nào đó, và chúng có chung một tính chất là đáp án tốt nhất cho một bài toán con vẫn được duy trì khi bài toán con đó trở thành một phần trong bài toán lớn hơn.

BÀI TOÁN CHIẾC TÚI XÁCH

Một tên trộm sau khi mở được một két sắt thì nhận thấy trong đó chứa đựng N loại đồ vật có kích thước và giá trị khác nhau. Nhưng hắn ta chỉ có một chiếc túi xách có dung lượng M (có thể chứa được một số đồ vật sao cho tổng kích thước của các đồ vật này nhỏ hơn hay đúng bằng M). Vậy vấn đề đặt ra cho tên trộm là hắn phải chọn lựa một danh sách các đồ vật sẽ mang đi như thế nào để cho tổng giá trị lấy cắp được là lớn nhất.

Ví dụ tên trộm có một túi xách dung lượng 17, và trong két sắt chứa đựng các đồ vật thuộc 5 loại khác nhau mà kích thước và giá trị được liệt kê trong hình 42.1. Khi đó tên trộm có thể mang đi 5 đồ vật



Hình 42.1 *Một ví dụ về "bài toán chiếc túi xách"*

	k	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
j=1																		
	cost[k]	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
	best[k]			A	A	A	A	A	A	A	A	A	A	A	A	A	A	A
j=2																		
	cost[k]	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
	best[k]			A	B	B	A	B	B	A	B	B	A	B	B	A	B	B
j=3																		
	cost[k]	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
	best[k]			A	B	B	A	C	B	A	C	C	A	C	C	A	C	C
j=4																		
	cost[k]	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
	best[k]			A	B	B	A	C	D	A	C	C	A	C	C	D	C	C
j=5																		
	cost[k]	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
	best[k]			A	B	B	A	C	D	E	C	C	E	C	C	D	E	C

Hình 42.2 *Lời giải cho "bài toán chiếc túi xách"*

loại A (không thể 6) với tổng giá trị là 20, hay hắn có thể chất đầy túi xách với một đồ vật loại D và một đồ vật loại E với tổng lượng i có thể đạt được là *(cost[i-size[j]]+val[j]* (vì đồ vật *j* thêm vào sẽ làm đầy phần còn lại của túi xách có dung lượng *I*). Nếu giá trị mới này vượt quá *cost[i]* (giá trị lớn nhất túi xách dung lượng *i* có thể đạt được khi không có đồ vật *j*) thì chúng ta sẽ cập nhật *cost[i]* và *best[i]*.

Hình 42.2 minh họa từng bước qúa trình tính toán trên ví dụ của chúng ta.

Nội dung thật sự của phương án tối ưu (danh sách các đồ vật cần bỏ vào túi xách) có thể tìm lại được dựa vào các giá trị trong mảng *best*. Theo định nghĩa, *best[M]* cho biết đồ vật cuối cùng phải thêm vào túi cách, vậy đồ vật vừa được bỏ vào túi xách trước đó là *best*

[M-size[best[M]]], cứ lần ngược như vậy ta sẽ có danh sách các đồ vật trong túi xách với tổng giá trị lớn nhất. Đối với ví dụ của chúng ta ở trên, *best[17]=C*, sau đó ta có *best[10]=C* và *best[3]=A*.

Tính chất 42.1 *Thời gian giải quyết bài toán túi xách bằng phương pháp quy hoạch động tỷ lệ với NM*

Do vậy bài toán túi xách có thể giải quyết dễ dàng khi M không qúa lớn, nhưng thời gian thi hành chương trình sẽ trở nên khó chấp nhận khi M lớn. Hơn nữa, có một điểm cốt yếu không thể bỏ qua được đó là phương pháp không thể thực hiện được nếu M và những kích thước hay giá trị của các đồ vật là số thực thay vì số nguyên. Đây không phải là một rắc rối nhỏ, mà là một khó khăn chính. Chưa có một giải pháp tốt nào cho vấn đề này được biết đến, và chúng ta sẽ thấy ở Chương 45 nhiều người tin rằng một giải pháp như vậy không tồn tại. Nhận thức rõ khó khăn của vấn đề, bạn đọc có thể sẽ muốn thử nghiệm phương pháp với trường hợp tất cả các loại đồ vật đều có giá trị 1, kích thước j và $M=N/2$.

Nhưng nếu dung lượng, kích thước và giá trị của các đồ vật đều là số nguyên, chúng ta có được một nguyên lý cơ bản là quyết định tối ưu, khi đã được chọn thì không cần thay đổi. Một khi đã biết cách tốt nhất để bỏ đồ vào các túi xách có kích thước bất kỳ với j đồ vật ban đầu, ta không cần xem xét lại những vấn đề đó nữa, bất kể các đồ vật sẽ chọn tiếp theo là gì. Khi nào nguyên lý tổng quát này còn hoạt động được, thì phương pháp quy hoạch động còn ứng dụng được. Trong trường hợp này, nguyên lý hoạt động được vì với các giá trị nguyên ta có thể chọn được một quyết định chính xác, tối ưu.

PHÉP NHÂN TỔ HỢP NHIỀU MA TRẬN

Một ứng dụng cổ điển của phương pháp quy hoạch động là tìm cách cực tiểu hóa số lượng các phép tính phải thực hiện khi cần nhân một loạt các ma trận khác kích thước với nhau.

Giả sử có 6 ma trận cần được nhân với nhau.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \begin{pmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \end{pmatrix} \begin{pmatrix} c_{11} \\ c_{21} \\ c_{31} \end{pmatrix} \begin{pmatrix} d_{11} & d_{12} \end{pmatrix} \begin{pmatrix} e_{11} & e_{12} \\ e_{21} & e_{22} \end{pmatrix} \begin{pmatrix} f_{11} & f_{12} & f_{13} \\ f_{21} & f_{22} & f_{23} \end{pmatrix}$$

Dĩ nhiên, để phép nhân hợp lệ, số cột của một ma trận phải bằng với số dòng của ma trận kế tiếp. Nhưng tổng số các phép nhân vô hướng phải làm thì phụ thuộc vào thứ tự thực hiện phép nhân ban đầu. Ví dụ, chúng ta có thực hiện phép nhân theo thứ tự từ trái qua phải: nhân A với B ta có một ma trận kết quả R1 *4x3* sau 24 phép nhân vô hướng, sau đó lại lấy R1 nhân tiếp với ma trận C ta được ma trận kết qủa R2 *4x1* sau 12 phép nhân vô hướng. Nhân tiếp tục R2 với D ta có ma trận kết quả R3 *4x2* sau 8 phép nhân vô hướng nữa, tiếp tục như thế này cho đến hết dãy ma trận, cuối cùng ta nhận được ma trận kết quả R *4x3* sau tổng cộng 84 phép nhân vô hướng. Nhưng nếu cũng thực hiện phép nhân trên theo trình tự từ phải qua trái, ta sẽ có cùng kết quả chỉ sau 69 phép nhân vô hướng.

Rõ ràng là còn nhiều thứ tự khác để thực hiện bài toán nhân ban đầu. Thứ tự thực hiện các phép nhân có thể thể hiện bằng cặp dấu đóng, mở ngoặc (): ví dụ ta có thể thực hiện theo trình tự ((((A*B)*C)*D)*E)*), hay trình tự (A*(B*(C*(D*(E*))))). Bất kỳ một trình tự hợp lệ nào cũng cho ra kết quả đúng, nhưng trình tự nào sẽ cần ít phép nhân vô hướng nhất để cho ra kết quả?

Nếu tìm được trình tự tốt nhất để thực hiện phép nhân, ta sẽ tiết kiệm được rất nhiều: nếu các ma trận B, C và trong ví dụ ở trên, mỗi ma trận có kích thước 300 thay vì 3, thì thực hiện theo trình tự trái qua phải đòi hỏi 6.024 phép nhân vô hướng, trong khi trình tự phải qua trái cần đến 247.200 phép nhân vô hướng (ở đây chúng tôi xem như chỉ sử dụng phép nhân ma trận thông thường, mỗi phép nhân một ma trận *pxq* với ma trận *qxr* sẽ cho ra một ma trận *pxr*, và tổng

các phép nhân vô hướng phải thực hiện là *pqr*)

Một cách tổng quát, giả sử N ma trận được nhân với nhau:

$$M_1M_2M_3...M_N$$

trong đó ma trận M_i thỏa điều kiện có kích thước r_i dòng và r_{i+1} cột, với $1<i<N$. Mục tiêu của chúng ta là tìm ra trình tự thực hiện phép nhân các ma trận trên sao cho dùng ít phép nhân vô hướng nhất. Rõ ràng là cách thử nghiệm tất cả các trình tự để rút ra trình tự tốt nhất là không thực tế (Số lượng các trình tự có thể thực hiện được là một hàm tổ hợp được gọi là số Catalan: Số cách kết hợp N biến bằng các dấu đóng, mở ngoặc là vào khoảng $4^{N-1}/N\sqrt{\kappa N}$). Nhưng thật ra cũng đáng mở rộng vài nỗ lực để tìm phương án tốt nhất vì N nói chung thường đủ nhỏ so với số lượng các phép nhân cần thực hiện.

Như đã nói ở phần trên, áp dụng phương pháp lập trình động nghĩa là chúng ta sẽ giải quyết vấn đề theo kiểu "bottom up": lưu trữ lời giải cho từng phần nhỏ của vấn đề để tránh phải tính toán lại khi cần giải quyết cả bài toán. Trước hết ta nhận thấy chỉ có một cách để thực hiện các phép nhân từng cặp hai ma trận M_1*M_2, M_2*M_3,... $M_{N-1}*M_N$, chúng ta sẽ tính toán số phép toán cần thực hiện cho mỗi phép nhân (phí tổn) hai ma trận này và lưu trữ các kết quả trong mảng *cost*. Kế tiếp, ta lại tính toán cách tốt nhất để nhân các bộ ba, sử dụng lại những thông tin về phí tổn nhân hai ma trận đã được giữ lại ở bước trên. Ví dụ để tìm thứ tự tốt nhất thực hiện việc nhân bộ ba $M_1M_2M_3$, trước tiên ta thử tính phí tổn để nhân $(M_1*M_2)*M_3$, phí tổn này bằng tổng phí tổn nhân M_1*M_2 (đã được lưu trữ) và phí tổn nhân kết quả thu được với M_3. Kế đến lại tính toán tương tự phí tổn cho phép nhân $M_1*(M_2*M_3)$, so sánh hai phí tổn vừa tính được và lưu giữ lại phí tổn nhỏ hơn. Tiếp tục tiến hành như vậy với tất cả các bộ ba hợp lệ, ta sẽ có tất cả các phương án tốt nhất để thực hiện phép nhân các bộ ba. Sau đó lại tính toán tiếp cho các bộ bốn ma trận... cuối cùng chúng ta sẽ ghi nhận được trình tự tốt nhất để nhân bộ N

ma trận với nhau. Áp dụng thuật toán ta có chương trình

```
for i:=1 to N do
   for j:=i+1 to N do cost[i,j]:=maxint;
for i:=1 to N do cost[i,j]=0;
for j:=1 to N-1 do
   for i:=1 to N-j do
      for k:=i+1 to i+j do
         begin
         t:=cost[i,k-1] + cost[k,i+j] + r[i]*r[k]*r[i+j+1];
         if t<cost[i,i=j] then
            begin cost[i,i+j]:=t; best[i,i+j]:=k;
            end;
         end;
```

Mục đích của chúng ta là tính phí tổn thấp nhất cho việc thực hiện phép nhân các bộ j ma trận

$$M_i M_{i+1} \ldots M_{i+j}$$

với $1 \le j \le N\text{-}j$. Cũng như trong bài toán túi xách, ta cần đến hai mảng để lưu trữ đáp án cho từng vấn đề con. *cost[i,i+j]* cho biết phí tổn nhỏ nhất để thực hiện phép nhân j ma trận từ ma trận thứ i đến ma trận thứ $i+j$ trong dãy ban đầu.

Một bộ j ma trận có thể được nhân theo k trình tự khác nhau, nếu viết lại phép nhân trên theo cách:

$$M_i M_{i+1} \ldots M_{i+j} = M_i M_{i+1} \ldots M_{k-1} * M_k M_{k+1} \ldots M_{i+j} \quad i+1 \; k \; i+j$$

thì phí tổn để thực hiện một phép nhân như vậy là tổng phí tổn thực hiện phép nhân bộ ma trận $M_i M_{i+1} \ldots M_{k-1}$ (ma trận kết quả có kích thước $r_i * r_k$), và phí tổn thực hiện phép nhân bộ ma trận $M_k M_{k+1} \ldots M_i$ (ma trận kết quả có kích thước rk*r i+j+1), cộng với phí tổn nhân hai ma trận kết quả với nhau. Gọi t là phí tổn nhân bộ j ma

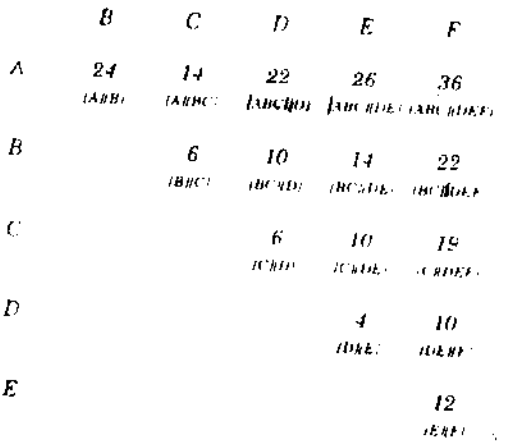
trận với dấu mở ngoặc đặt ở vị trí k, ta có

$t = cost[i,k-1] + cost[k,i+j] + r[i]*r[i+j+1]$

trong đó *cost[i,k-1]* và *cost[k,i+j]* là phí tổn ít nhất để thực hiện phép nhân các bộ *k-1-i* và *i+j-k* ma trận đã được tính toán và lưu trữ lại ở bước trước. Như đã nói qua ở phần trước ta biết rằng phí tổn để nhân một ma trận $r_i * r_k$ với một ma trận $r_k * r_{i+j+1}$ là *r[i]*r[k]*r[i+j+1]*. Nếu giá trị *t* vừa tính được nhỏ hơn giá trị *cost[i,i+j]* hiện thời thì ta cập nhật lại *cost[i,i+j]*, đồng thời cập nhật *best[i,i+j]* lưu trữ vị trí *k* cần nhóm các ma trận. Vậy khi *j=N-1* (*i=1*) thì ta tính được số phép nhân vô hướng ít nhất phải làm để nhân bộ *N* ma trận, lúc đó dựa vào nội dung mảng *best* và sử dụng một thủ tục đệ quy nhỏ dưới đây chúng ta sẽ biết được trình tự thật sự cần tiến hành phép nhân.

```
procedure order(i,j:integer);
   begin
   if i=j
   then write(name(i))
   else begin
         write('(');
         order(i, best[i,j]-1); write('*'); order(best[i,j],);
         write(')');
         end;
   end;
```

Hình 42.3 cho thấy các kết quả trong quá trình tính toán với ví dụ của chúng ta. Ví dụ dòng A cột F cho biết có 36 phép nhân vô hướng cần thực hiện khi nhận bộ 6 ma trận từ A...F. Để có thể đạt được kết quả đó cần tiến hành nhân theo trình tự (A*B*C)*(D*E*F). Mà để nhân (A*B*C) theo trình tự tốt nhất ta xét cột A dòng C và có được thứ tự A*(B*C), tương tự có (D*E)*F.



Hình 42.3 *Lời giải cho bài toán nhân ma trận*

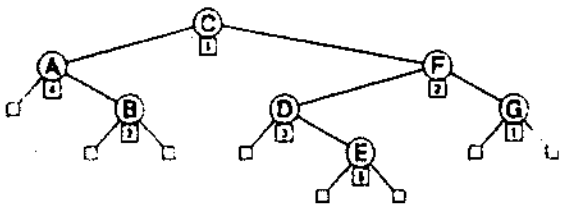
Vậy thứ tự cuối cùng là A*(B*C)* (D*E)*F

Tính chất 42.2 *Thời gian giải quyết bài toán túi xách bằng phương pháp quy hoạch động tỷ lệ với N^3 và phí tổn không gian tỷ lệ N^2.*

Một lần nữa, điều này có thể rút ra trực tiếp từ chương trình. Phí tổn không gian lớn hơn so với bài toán túi xách ở trên, nhưng so ra vẫn không đáng kể so với số lượng phép toán tiết kiệm được.

CÂY NHỊ PHÂN TÌM KIẾM TỐI ƯU

Trong nhiều ứng dụng sử dụng thao tác tìm kiếm, các khóa có thể được truy xuất với những tần số khác nhau rất nhiều. Ví dụ, một chương trình kiểm tra chính tả một văn bản tiếng Anh có thể thường xuyên gặp các từ "and" và "the" hơn là các từ "dynamic" và "programing". Nếu dữ liệu được lưu giữ trong một cây nhị phân tìm kiếm, thì rõ ràng sẽ tăng tốc độ tìm kiếm khi đưa các khóa có tần số



Hình 42.4 *Một cây nhị phân tìm kiếm kèm tần suất*

xuất hiện lớn lên phía ngọn của cây. Ta có thể áp dụng một thuật toán lập trình động để xác định một phương cách bố trí các khóa trên cây sao cho tổng chi phí tìm kiếm là thấp nhất.

Trong hình 42.4, mỗi nút của cây được gán cho một con số được giả thiết là tỷ lệ với tần suất khóa có thể được truy xuất. Nghĩa là, với 18 lần tìm kiếm trên cây, ta chờ đợi sẽ có 4 lần truy xuất đến A, 2 lần cho B, 1 lần cho C... Mỗi lần truy xuất đến A ta phải duyệt qua hai nút trên cây, truy xuất đến B phải mất ba lần truy xuất đến các nút trước B..., như đã định nghĩa trong chương 4, ta gọi số nút phải truy xuất trước khi đến được nút X là khoảng cách từ gốc đến X. Ta có thể dễ dàng tính được chi phí của một cây theo công thức:

$$cost = \sum_{1 \le i \le N} f[i]*d[i]$$

trong đó *f[i]* là tần suất của nút *i*, và *d[i]* là khoảng cách từ gốc đến nút *I*.

Giá trị cost này được gọi là độ dài đường đi trong có trọng số của cây. Đối với cây trong hình 42.4 ta có

cost= 4*2+2*3+1*1+3*3+5*4+2*2+1*3=51

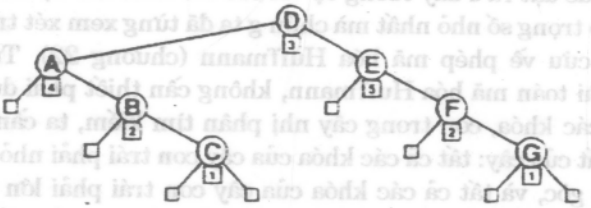
Với tập các khóa được cho ban đầu, chúng ta cần tìm một cây nhị phân tìm kiếm có các khóa được phân bố sao cho độ dài đường đi trong có trọng số của cây là nhỏ nhất trong tất cả các cây có cùng tập khóa.

Vấn đề đặt ra ở đây tương tự với các bài toán tìm độ dài đường đi ngoài có trọng số nhỏ nhất mà chúng ta đã từng xem xét trong phần nghiên cứu về phép mã hóa Huffmann (chương 22). Tuy nhiên, trong bài toán mã hóa Huffmann, không cần thiết phải duy trì thứ tự của các khóa, còn trong cây nhị phân tìm kiếm, ta cần bảo đảm tính chất của cây: tất cả các khóa của cây con trái phải nhỏ hơn khóa của nút gốc, và tất cả các khóa của cây con trái phải lớn hơn khóa của nút gốc. Yêu cầu này làm cho vấn đề trở nên rất giống với bài toán nhân dãy ma trận mà ta vừa xem xét ở trên.

Giả sử xét trường hợp đặc biệt có N khóa $K_1 < K_2 < ... < K_N$ với các tần suất tương ứng là $r_0, r_1, r_2, ..., r_N$. Chúng ta cần tìm một cây nhị phân tìm kiếm sao cho chi phí truy xuất đến các nút của cây là thấp nhất.

Tương tự những bài toán trước, cách tiếp cận quy hoạch động để giải quyết bài toán này là tìm kiếm phương án tối ưu để xây dựng các cây con j khóa $K_i, K_{i+1}, ..., K_{i+j}$ và lưu trữ lại đáp án như mô tả trong chương trình sau:

```
for i:=1 to N do
   for j:=i+1 to N+1 do cost[i,j]:=maxint;
for i:=1 to N do cost[i,i]:=f[i];
for i:=1 to N-1 do cost[i,i-1]:=0;
for j:=1 to N-1 do
   for i:=1 to N-j do
   begin    for k:=i+1 to i+j do
            begin    t:= cost[i,k-1] + cost[k+1,i+j];
                     if t<cost[i,i+j] then
                     begin cost[i,i+j]:=t;bcst[i,i+j]:=k; end;
            end;
            t:=0;
            for k:=i to i+j do t:=t+f[k];
            cost[i,i+j] + t;
   end;
```



Hình 42.5 *Cây nhị phân tìm kiếm tối ưu*

Với mỗi giá trị j, ta sẽ thử chọn những nút khác nhau làm gốc và sử dụng lại các kết quả ở bước trước để xác định cách tốt nhất xây dựng cây con j khóa. Trong tập các cây con j khóa $K_i,K_{i+1},...,K_{i+j}$ như là nhánh phải của nút gốc K_k. Độ dài đường đi trong đó có trọng số của cây con mới được tính là tổng độ dài đường đi trong có trọng số của hai nhánh cộng với tổng tần suất của các nút trên cây (vì mỗi nút trong cây mới sẽ xa gốc thêm một bước).

Để ý rằng tổng tần suất của các nút được cộng vào tất cả các giá trị *cost*, vì vậy không cần để ý đến chúng khi tìm kiếm giá trị *cost* nhỏ nhất. Hơn nữa, ta phải gán cho *cost[i,i-1] = 0* để khống chế khả năng một nút có thể chỉ có một con (không có tình huống tương tự trong bài toán nhân ma trận).

Cũng trong bài toán trước, cần đến một chương trình đệ quy ngắn để phục hồi nội dung thật sự của phương án từ các giá trị được lưu trữ trong mảng *best*. Hình 42.5 cho thấy cây tối ưu xây dựng được tự tập khóa trong cây ở hình 42.4. Độ dài đường đi trong có trọng số của cây là 41.

Tính chất 42.3 *Thời gian giải quyết bài toán túi xách bằng phương pháp quy hoạch động tỷ lệ với N^3 và phí tổn không gian tỷ lệ N^2.*

Một lần nữa, thuật toán làm việc với một ma trận kích thước N^2 và tiêu tốn N thời gian cho mỗi entry (?). Thật sự có thể giảm thời gian xuống N^2 nếu để ý rằng vị trí rối ưu của gốc không thể quá xa

vị trí tối ưu của gốc trong cây con ít khóa hơn, vì thế k không cần lấy đủ giá trị trong miền $[i,i+j]$.

YÊU CẦU THỜI GIAN VÀ KHÔNG GIAN

Từ các ví dụ ở trên ta có thể nhận xét được là các ứng dụng của phương pháp lập trình động có thể đòi hỏi về không gian và thời gian rất khác nhau, tùy thuộc vào lượng thông tin về các bài toán con. Với mỗi vấn đề, có thể thấy rằng chi phí thời gian thường lớn hơn một thừa số N so với chi phí không gian.

Khả năng ứng dụng phương pháp lập trình động trong thực tế lớn hơn nhiều so với các ví dụ minh họa của chúng tôi đưa ra ở đây. Theo quan điểm của quy hoạch động, thì đệ quy "chia-để-trị" có thể xem như một trường hợp đặc biệt mà một lượng tối thiểu các bài toán con được xác định cần phải giải quyết và lưu trữ đáp án, và tìm kiếm toàn diện (ta sẽ xem xét đến trong chương 44) có thể xem như một trường hợp đặc biệt mà một lượng tối đa các bài toán con được xác định cần phải giải quyết là lưu trữ đáp án. Quy hoạch động là một kỹ thuật thiết kế tự nhiên xuất hiện dưới nhiều hình thức để giải quyết các bài toán trong miền ứng dụng này.

BÀI TẬP

1. Trong ví dụ đã trình bày về bài toán chiếc túi xách, các phần tử được sắp xếp theo kích thước. Thuật toán còn đúng hay không nếu chúng xuất hiện theo thứ tự tùy ý.
2. Sửa đổi chương trình chiếc túi xách (Knapsack) bằng cách đưa vào một mảng *num[1..N]* chứa số phần tử có sẵn trong mỗi loại.
3. Chương trình Knapsack sẽ như thế nào nếu một trong các giá trị bị âm ?
4. Cho biết điều khẳng định sau đúng hay sai: Nếu một dãy các ma trận có chứa phép nhân ma trận *1xk* và *kx1* thì lời giải tối ưu là phép nhân này được thực hiện sau cùng.
5. Viết một chương trình nhân *N* ma trận bằng một phương pháp tối ưu. Giả sử các ma trận được lưu trữ trong một mảng ba chiều *matrices[1..Nmax,1..Dmax,1..Dmax]* , trong đó *Dmax* là chiều lớn nhất và ma trận thứ *i* được lưu trong *matrices[i,1..r[i],1..r[i+1]*.
6. Vẽ cây tìm kiếm nhị phân tối ưu cho ví dụ trong chương này nhưng tất cả các tần số đều tăng lên 1.
7. Viết chương trình xây dựng cây tìm kiếm nhị phân tối ưu.
8. Giả sử chúng ta đã tính cây tìm kiếm nhị phân tối ưu cho một tập khóa và tần số và sau đó có một tần số nào đó tăng lên 1. Hãy viết một chương trình để tính cây tối ưu mới.
9. Tại sao chúng ta không giải bài toán chiếc túi xách bằng phương pháp như dãy ma trận và cây tìm kiếm nhị phân ? Nghĩa là tìm cực tiểu của tổng giá trị tốt nhất của túi xách kích thước *k* và giá trị tốt nhất của túi xách kích thước *M-k*, trong đó *k* lấy giá trị từ 1 đến *M*.
10. Mở rộng chương trình của bài toán đường đi ngắn nhất bằng cách thêm vào thủ tục *paths(i,j: integer)* để đặt các giá trị trong mảng *path* đường ngắn nhất từ *i* tới *j*. Thủ tục này cần thời gian xấp xỉ chiều dài đường đi mỗi khi nó được gọi, dùng một cấu trúc dữ liệu trợ giúp để xây dựng một phiên bản hiệu chỉnh của chương trình trong Chương 32.

43

QUY HOẠCH TUYẾN TÍNH

Nhiều bài toán thực tiễn có những mối liên quan phức tạp và có rất nhiều đại lượng. Một trong các ví dụ loại đó là bài toán mạng dòng chảy (network flow) đã xét ở chương 33: dòng chảy trong các ống khác nhau cần phải tuân theo các định luật vật lý của mạng. Một ví dụ khác là lập lịch biểu cho nhiều công việc của một quá trình sản xuất theo mức độ ưu tiên, hạn chót (của công việc)... Cần phải đưa các bài toán như vậy về một dạng chặt chẽ hơn và cần phải phát triển các công thức toán học chính xác để giải quyết các vấn đề này. Quá trình tìm các phương trình mà nghiệm của nó giải các bài toán thực tiễn gọi là quy hoạch toán học (mathematical programming). Trong chương này ta xét một quy hoạch toán học đặc biệt, quy hoạch tuyến tính, và một phương pháp hiệu quả để giải, phương pháp đơn hình.

Quy hoạch tuyến tính và phương pháp đơn hình có tầm quan trọng căn bản vì một lớp rất rộng các bài toán giải được bằng phương pháp này. Với một số bài toán đặc biệt, người ta còn có các thuật toán tốt hơn, tuy nhiên chỉ một số ít kỹ thuật có thể áp dụng rộng rãi như kỹ thuật quy hoạch tuyến tính. Một thư viện các chương trình (routine) cho phương pháp đơn hình là một công cụ cần thiết để giải quyết các bài toán phức tạp.

Sự nghiên cứu về quy hoạch tuyến tính rất mạnh mẽ, và một sự hiểu biết đầy đủ tất cả các vấn đề đòi hỏi nhiều kiến thức vượt quá

khôn khổ cuốn sách này. Tuy thế, những ý tưởng căn bản rất dễ hiểu, còn phương pháp đơn hình thì không quá khó khi cài đặt, như ta sẽ thấy dưới đây.

Cũng giống như phép biến đổi Fourier nhanh ở chương 41, ta không có ý định cho một chương trình đầy đủ mà chỉ nghiên cứu một số tính chất căn bản của thuật toán và quan hệ của nó với các thuật toán khác mà ta đã khảo sát.

QUY HOẠCH TUYẾN TÍNH

Bài toán này gồm một tập hợp các biến liên hệ với nhau bởi các ràng buộc (phương trình hay bất phương trình) và một hàm mục tiêu (objective function) cần phải cực đại hóa.

Bài toán quy hoạch tuyến tính sau tương ứng với bài toán mạng dòng chảy ở chương 33.

Cực đại hóa $x_{AB}+x_{AD}$ theo các ràng buộc:

$$\begin{gathered}
x_{AB}\le 6 \quad x_{CD}\le 3\\
x_{AC}\le 8 \quad x_{CE}\le 3\\
x_{BD}\le 6 \quad x_{DF}\le 8\\
x_{BE}\le 3 \quad x_{EF}\le 6\\
x_{AB}+x_{BE}=x_{AB}\\
x_{CD}+x_{CE}=x_{AC}\\
x_{BD}+x_{CD}=x_{DF}\\
x_{BE}+x_{CE}=x_{EF}\\
x_{AB},x_{AC},x_{BD},x_{BE},x_{CD},x_{CE},x_{DF},x_{EF}\ge 0
\end{gathered}$$

Có một biến trong bài toán quy hoạch này tương ứng với mỗi dòng trong ống. Các biến này thỏa hai loại quan hệ: bất đẳng thức, tương ứng với ràng buộc dung tích của ống, và đẳng thức, tương ứng với ràng buộc của dòng chảy tại các nút.

Vậy, chẳng hạn, bất đẳng thức $x_{AB}\le 6$ cho biết ống AB có dung

tích 6, và phương trình $x_{BD}+x_{BE}=x_{AB}$ cho biết dòng ra phải bằng dòng chảy vào tại nút B.

Rõ ràng đây là một phát biểu toán học của bài toán mạng dòng chảy: nghiệm của bài toán này sẽ là nghiệm của bài toán dòng chảy mạng. Ta sẽ thấy là kỹ thuật quy hoạch tuyến tính không chỉ dùng để giải bài toán trên mà còn dùng để giải nhiều bài toán khác. Chẳng hạn, nếu ta tổng quát bài toán mạng dòng chảy để bao gồm việc tính toán giá cả thì dùng bài toán cũng sẽ không khác nhiều.

Bài toán quy hoạch tuyến tính có một ý nghĩa lớn, đồng thời có một phương pháp hiệu quả (thuật toán đơn hình) để giải bài toán này. Với vài bài toán (chẳng hạn bài toán mạng dòng chảy), có thể tìm được một thuật toán đặc biệt dành riêng cho nó, thuật toán này có thể xử lý tốt hơn phương pháp đơn hình. Với một số bài toán khác (bao gồm cả các nới rộng của bài toán mạng dong chảy), phương pháp đơn hình lại tốt hơn. Tuy nhiên, ngay cả khi có một thuật toán tốt hơn, nó có thể phức tạp hay khó cài đặt, trong khi quá trình xử lý với một thư viện chương trình con đơn hình (simplex library routine) thường nhanh hơn. Tính linh hoạt về phương diện sử dụng rộng rãi của nó. Tuy nhiên phương pháp này cũng không phải là vạn năng, không nên lạm dụng nó, đối với bài toán (nhiều bài toán như vậy đã được khảo sát trong sách này), việc sử dụng phương pháp này không mang lại hiểu quả cần thiết.

BIỂU DIỄN HÌNH HỌC

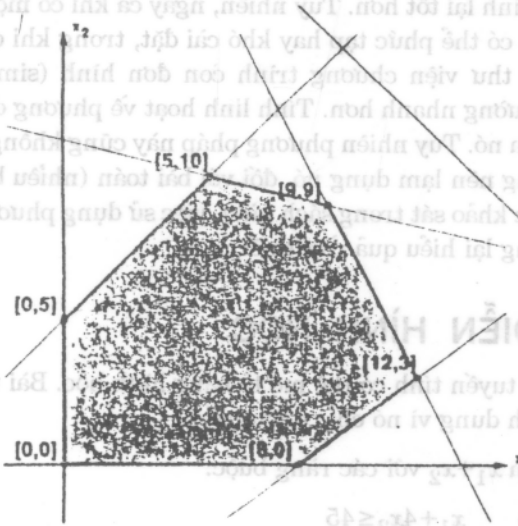
Quy hoạch tuyến tính có thể mô tả bằng hình học. Bài toán sau đặc biệt dễ hình dung vì nó chỉ có hai biến:

Cực đại hóa x_1+x_2 với các ràng buộc:

$$-x_1+x_2\le 5 \qquad x_1+4x_2\le 45$$
$$2x_1+x_2\ge 27 \qquad 3x_1-4x_2\le 24 \qquad x_1,x_2\ge 0$$

Bài toán này tương ứng với trường hợp hình vẽ trong hình 43.1. Mỗi bất phương trình xác định một nửa mặt phẳng trong đó chứa các nghiệm của bài toán quy hoạch. Ví dụ $x_1 \geq 0$ nghĩa là bất kỳ nghiệm nào của bài toán quy hoạch đều nằm bên phải trục x_2, và $-x_1+x_2 \leq 5$ nghĩa là các nghiệm cần nằm phía dưới của đường $-x_1+x_2=5$ (nối từ (0,5)) đến (5, 10). Nghiệm của bài toán phải thỏa tất cả các ràng buộc này, vậy nó nằm trong phần giao của các nửa mặt phẳng này (xem hình). Để giải bài toán, ta cần tìm điểm ở trong miền này để hàm mục tiêu x_1+x_2 đạt cực đại.

Một miền xác định bởi giao của các nửa mặt phẳng luôn luôn lồi (điều này đã được nói tới ở chương 25). Miền lồi này được gọi là một đơn hình, làm cơ sở cho một thuật toán tìm nghiệm của bài toán quy hoạch tuyến tính.



Hình 43.1 *Một đơn hình hai chiều*

Một tính chất cơ bản của đơn hình được khai thác bởi thuật toán là: hàm mục tiêu đạt cực tiểu chỉ tại các đỉnh của đơn hình; vậy ta chỉ tính các đỉnh chứ không tính tại các điểm trong. Để giải thích điều này, ta hãy xem hình 43.1. Hàm mục tiêu có thể được xem như một đường thẳng đã biết độ dốc (trường hợp này là -1) nhưng chưa biết vị trí. Ta chú ý tới điểm mà đường thẳng (độ dốc -1) đụng vào đơn hình khi chúng được dời song song từ vô cực tới. Điểm này là nghiệm của bài toán: nó thỏa tất cả các ràng buộc và nó là cực đại của hàm mục tiêu vì không có giá trị nào lớn hơn. Trong ví dụ của ta, đường thẳng gặp đơn hình tại (9, 9) và hàm mục tiêu có giá trị cực đại là 18.

Các hàm mục tiêu khác sẽ tương ứng với các đường thẳng với độ dốc khác nhưng luôn luôn có cực trị ở các đỉnh của đơn hình. Ta sẽ tìm một thuật toán để có thể thử các đỉnh của đơn hình để tìm ra giá trị nhỏ nhất. Trong trường hợp hai chiều, có rất ít cách chọn, tuy nhiên, trong trường hợp nhiều chiều hơn, bài toán phức tạp hơn nhiều.

Từ hình 43.1 ta cũng thấy rằng nếu hàm mục tiêu là phi tuyến, thì đường cong biểu diễn nó có thể đụng vào đơn hình dọc theo một cạnh chứ không ở đỉnh đơn hình. Nếu hơn nữa nếu các ràng buộc là phi tuyến, khi đó ta không vẽ được một đơn hình như hình 43.1. Do đó quy hoạch toán học trong trường hợp phi tuyến cực kỳ phức tạp.

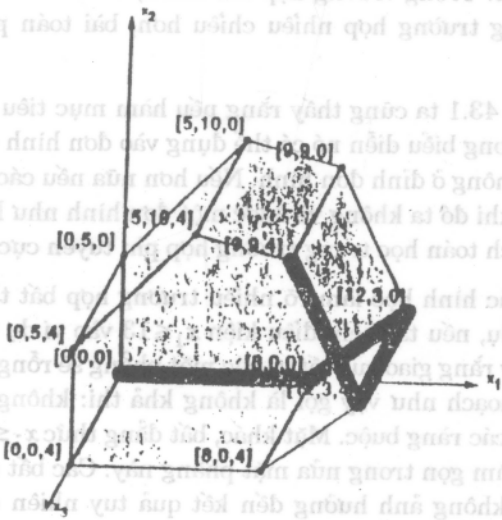
Trực giác hình học làm rõ nhiều trường hợp bất thường có thể xảy ra. Ví dụ, nếu ta thêm điều kiện $x_1 \leq 13$ vào ví dụ trên, từ hình 43.1 ta thấy rằng giao của tất cả các mặt phẳng sẽ rỗng. Một chương trình quy hoạch như vậy gọi là không khả thi: không có điểm nào thỏa tất cả các ràng buộc. Mặt khác, bất đẳng thức $x_1 \leq 13$ là thừa vì đơn hình nằm gọn trong nửa mặt phẳng này. Các bất đẳng thức dư nói chung không ảnh hưởng đến kết quả tuy nhiên chúng có thể được xét đến trong quá trình tìm lời giải.

Một vấn đề nghiêm trọng khác là đơn hình có thể không bị giới

hạn, trong trường hợp như vậy lời giải có thể không định được. Trong ví dụ của ta, nếu bỏ bất đẳng thức thứ hai và thứ ba ta sẽ gặp trường hợp này. Nếu đơn hình là không bị chận, vẫn có thể có các hàm mục tiêu có lời giải, tuy nhiên một thuật toán để tìm kết quả này sẽ gặp những trở ngại lớn do tính không bị chận của đơn hình.

Cần phải nhấn mạnh rằng, mặc dầu bài toán có vẻ dễ dàng trong ví dụ của ta, bài toán tổng quát với nhiều biến và nhiều ràng buộc thực sự phức tạp. Khi đó, sự loại trừ các trường hợp bất thường trên là một việc có ý nghĩa trong tính toán.

Trường hợp số biến nhiều hơn hai, trực giác hình học cũng tương tự. Trong trường hợp ba chiều, đơn hình là một khối lồi xác định bởi giao của các nửa không gian. Ví dụ nếu ta thêm bất đẳng thức $x_3 \geq 0$ vào ví dụ trên, đơn hình trở thành khối như hình 43.2



Hình 43.2 *Một đơn hình ba chiều*

Bây giờ ta lấy một hàm mục tiêu: $x_1+x_2+x_3$. Nó xác định một mặt phẳng vuông góc với đường $x_1=x_2=x_3$. Nếu ta dời mặt phẳng từ vô cực dọc theo đường này, nó sẽ đụng đơn hình tại (9, 9, 4), nó là nghiệm của bài toán.

Trong trường hợp n chiều, ta giao các $(n-1)$-siêu phẳng để được một đơn hình n chiều, rồi ta mang một $(n-1)$-siêu phẳng từ vô cực tới đụng đơn hình tại một điểm (nghiệm của bài toán). Các lý luận trên không thể chính xác hóa vì nó đòi hỏi nhiều kiến thức về đại số tuyến tính vượt quá khuôn khổ cuốn sách này. Tuy thế, trực giác hình học vẫn giúp chúng ta hiểu được các đặc điểm căn bản của phương pháp đơn hình.

PHƯƠNG PHÁP ĐƠN HÌNH

Phương pháp đơn hình là tên chung để chỉ cho các phương pháp tổng quát giải bài toán quy hoạch tuyến tính bằng phương pháp xoay trụ (pivoting), giống như phép khử Gauss (Gaussian elimination). Nó tương ứng với thao tác hình học chuyển từ đỉnh này tới đỉnh khác của đơn hình để tìm lời giải. Các thuật toán chỉ khác nhau ở thứ tự các đỉnh được xét đến. Từ "thuật toán" trong bài toán này có một ý nghĩa cụ thể hơn là một phương pháp chung mà có thể tiến hành theo nhiều cách khác nhau. Chúng ta đã gặp trường hợp này, ví dụ phép khử Gauss (chương 37) hay thuật toán Ford-Fulkenson (chương 33).

Trước hết, rõ ràng là bài toán quy hoạch tuyến tính có thể có nhiều dạng. Ví dụ, bài toán mạng dòng chảy có các đẳng thức lẫn các bất đẳng thức, những ví dụ hình học trên chỉ sử dụng các bất đẳng thức. Để làm giảm bớt số dạng có thể có của bài toán, ta sẽ xét một dạng chuẩn, trong đó tất cả các ràng buộc là đẳng thức, ngoại trừ một số bất đẳng th ức biểu thị cho các biến đều là số không âm. Dạng chuẩn này dường như quá chặt chẽ, tuy nhiên ta có thể chuyển bài

toán tổng quát về dạng này. Bài toán sau là dạng chuẩn của ví dụ 3 chiều trong hình 43.2.

Cực đại hóa $x_1+x_2+x_3$ với các điều kiện.

$-x_1+x_2+y_1=5 \quad x_1+4x_2+y_2=45$
$2x_1+x_2+y_3=27 \quad 3x_1-4x_2+y_4=24$
$x_3+y_5=4 \quad x_1x_2x_3,y_1,y_2,y_3,y_4,y_5\geq 0$

Mỗi bất đẳng thức có hai biến trở lên chuyển thành đẳng thức bằng cách thêm vào một biến. Các biến y gọi là các biến phụ thêm (slack variable). Ngoài ra, bằng cách đổi biến ta có thể chuyển các bất đẳng thức một biến thành loại ràng buộc không âm. Ví dụ một ràng buộc như $x_3<=-1$ chuyển thành dạng chuẩn $x'_3\geq 0$ với $x'_3=-1-x_3$.

Dạng chuẩn tạo ra một tương ứng giữa bài toán quy hoạch tuyến tính và giải hệ phương trình tuyến tính. Ta có N phương trình với N ẩn (số dương). Trong trường hợp này, chú ý là ta có N biến phụ thêm, mỗi phương trình một biến (vì ta bắt đầu với một hệ toàn bất đẳng thức). Nếu $M>N$ thì hệ phương trình có nhiều nghiệm: vấn đề là tìm một cái làm cho hàm mục tiêu cực đại.

Trong ví dụ của ta, các phương trình ràng buộc có một nghiệm tầm thường $x_1=x_2=x_3=0$, còn các biến phụ thêm y_1,y_2,y_3,y_4,y_5 thì lấy các giá trị thích hợp. Điều này xảy ra vì (0, 0, 0) là một điểm của đơn hình trong ví dụ này. Trong trường hợp tổng quát, điều này không đúng, tuy nhiên ta sẽ giới hạn sự chú ý vào các bài toán như vậy cũng còn rất lớn: ví dụ, nếu tất cả các số ở vế phải của các bất đẳng thức là dương và tất cả các biến phụ thêm có hệ số dương (như trong ví dụ của ta) thì rõ ràng ta có nghiệm tầm thường (như trong ví dụ của ta) thì trở lại trường hợp tổng quát sau.

Cho một nghiệm với M-N biến là 0, ta có thể tìm các nghiệm khác với cùng tính chất bằnag cách sử dụng một phép toán quen thuộc, lặp (pivoting). Nó giống phép khử Gauss: một phần tử $a[p,q]$ được

chọn trong ma trận hệ số, rồi lấy hàng thứ p nhân với một số thích hợp và cộng với tất cả các dòng khác để làm cho cột q trở thành 0, ngoài trừ giá trị tại giao điểm của hàng p và cột q là 1. Hãy xét ma trận sau (biểu diễn bài toán quy hoạch trên).

$$\begin{pmatrix} -1.00 & -5.00 & -1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 \\ -1.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 5.00 \\ 1.00 & 4.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 45.00 \\ 2.00 & 1.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 27.00 \\ 3.00 & -4.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 24.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

Ma trận $(N+1)x(M+1)$ này chứa các hệ số của bài toán dạng chuẩn, với cột thứ $(M+1)$ chứa các số ở vế tay phải của các phương trình (như trong phép khử Gauss) và hàng zêrô chứa các hệ số của hàm mục tiêu, với dấu đảo ngược. Ý nghĩa của hàng zêrô sẽ được xét sau; còn bây giờ chúng được xử lý như các hàng khác.

Trong ví dụ này, ta chỉ thực hiện các tính toán chính xác đến hai chữ số sau dấu phẩy. Như vậy, ta bỏ qua các vấn đề độ chính xác của tính toán và sai số tích lũy (accumulated error). Giống như trong phép khử Gauss, các vấn đề này rất quan trọng.

Các biến tương ứng với một lời giải gọi là biến cơ sở (basic variables) và các biến được chuyển thành không thể giải gọi là biến không cơ sở (non-basic variables). Trong ma trận, các cột tương ứng với biến không cơ sở tương ứng với các cột có nhiều phần tử khác không hơn.

Giả sử ta muốn trụ (pivot) ma trận này với $p=4$, $q=1$. Ta cộng dòng thứ tư (nhân với các số thích hợp) với các dòng khác để tạo thành cột thứ nhất có tất cả là 0, ngoại trừ số 1 ở hàng 4. Quá trình này cho kết quả ở bảng số trang sau.

Phép toán này bỏ cột thứ 7 khỏi cơ sở và thêm cột 1 vào cơ sở. Một cách chính xác, một cột cơ sở đã được chuyển.

$$\begin{pmatrix} 0.00 & -2.33 & -1.00 & 0.00 & 0.00 & 0.00 & 0.33 & 0.00 & 8.00 \\ 0.00 & -0.33 & 0.00 & 1.00 & 0.00 & 0.00 & 0.33 & 0.00 & 13.00 \\ 0.00 & 5.33 & 0.00 & 0.00 & 1.00 & 0.00 & -0.33 & 0.00 & 37.00 \\ 0.00 & 3.67 & 0.00 & 0.00 & 0.00 & 1.00 & -0.67 & 0.00 & 11.00 \\ 1.00 & -1.33 & 0.00 & 0.00 & 0.00 & 0.00 & 0.33 & 0.00 & 8.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

Theo định nghĩa, ta có thể nhận được một nghiệm của bài toán bằng cách cho tất cả các biến không cơ sở thành 0, rồi dùng nghiệm tầm thường cho trong cơ sở. Trong ví dụ trên, ta cho x_2 và x_3 là 0 vì chúng là biến không cơ sở và $x_1=8$, vậy ma trận tương ứng với điểm (8, 0, 0) trong đơn hình (chúng ta không chú ý đến giá trị của các biến phụ thêm). Chú ý rằng góc trên bên phải của ma trận (hàng 0, cột $M+1$) chứa giá trị của hàm mục tiêu tại điểm này.

Bây giờ, giả sử ta thực hiện phép toán xoay trụ với $p=3$, $q=2$

$$\begin{pmatrix} 0.00 & 0.00 & -1.00 & 0.00 & 0.00 & 0.64 & -0.09 & 0.00 & 15.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.09 & 0.27 & 0.00 & 14.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & -1.45 & 0.64 & 0.00 & 21.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.27 & 0.18 & 0.00 & 3.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.36 & 0.09 & 0.00 & 12.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

Nó xóa cột 6 khỏi cơ sở và thêm cột 2 vào. Bằng cách cho các biến không cơ sở là 0 và giải các biến cơ sở như trước ta thấy ma trận này tương ứng với điểm (12, 3, 0) của đơn hình, giá trị của hàm mục tiêu tại đó là 15. Chú ý là giá trị của hàm mục tiêu tăng nghiêm ngặt. Điều này sẽ được giải thích.

Ta làm cách nào để xác định p và q để thực hiện phép toán mà giá trị hàm mục tiêu sẽ tăng lên ? Hàng số 0 sẽ được dùng để xác định p, q. Ta sẽ chọn cột q ở vị trí mà giao điểm của cột q và hàng 0 là một số âm. Sau đó hàng p sẽ được xác định sao cho phần tử của hàng ở cột q và cột $M+1$ là các số không âm. Trong thủ tục tìm p, q này, có

hai khó khăn. Thứ nhất, điều gì xảy ra nếu không có phần tử nào của cột q là số dương ? đó là một trường hợp mâu thuẫn: phần tử âm ở cột q, hàng 0 cho thấy hàm mục tiêu có thể tăng lên được, tuy nhiên lại không có cách để tăng lên, trường hợp này xảy ra khi và chỉ khi đơn hình là không bị chậm, vậy thuật toán có thể kết thúc và báo kết quả. Một khó khăn tinh tế khác là phần tử ở cột $(M+1)$ của hàng đã chọn (với phần tử dương ở cột q) là 0. Hàng này có thể sử dụng, nhưng giá trị của hàm mục tiêu không tăng lên. Nếu có hai hàng như vậy ta sẽ gặp trường hợp thuật toán lặp vô hạn. Chúng ta có thể tránh những khó khăn như vậy nếu mô tả phương pháp thật rõ ràng, nhưng cần nhấn mạnh rằng các trường hợp suy yếu như vậy có thể xảy ra trong thực tế. Có nhiều phương pháp để tránh bị lặp. Một phương pháp là ngừng một cách ngẫu nhiên. Một phương pháp khác chống bị lặp sẽ mô tả dưới đây.

Trong ví dụ của ta, ta có thể làm một lần nữa với $q=3$ (vì giá trị ở cột 3, hàng 0 là $-1<0$) và $p=5$ (vì chỉ có 1 là giá trị dương ở cột 3). Ta được ma trận sau:

$$\begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.64 & -0.09 & 1.00 & 19.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 0.09 & 0.27 & 0.00 & 14.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & -1.45 & 0.64 & 0.00 & 21.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.27 & -0.18 & 0.00 & 3.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.36 & 0.09 & 0.00 & 12.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

Nó tương ứng với điểm (12, 3, 4) trên đơn hình, giá trị hàm mục tiêu ở trường hợp này là 19..

Nói chung, có nhiều cách chọn phần tử âm ở hàng 0. Chúng ta đang thực hiện theo phương pháp phổ thông nhất, phương pháp "tăng nhanh nhất" (greatest-increment): luôn luôn chọn cột có phần tử nhỏ nhất ở hàng 0 (giá trị lớn nhất theo trị tuyệt đối). Nếu cách chọn cột này kết hợp với sự chọn các hàng (tính từ sự thay đổi của

cột có chỉ số nhỏ nhất của cơ sở) thì sự lặp sẽ không xảy ra (phương pháp này do R. G. Bland tìm ra). Một khả năng chọn cột khác là tính các giá trị của hàm mục tiêu ứng với mỗi cột rồi sử dụng cột nào cho kết quả lớn nhất. Phương pháp này gọi là phương pháp "dốc xuống nhất" (steepest-descend).

Cuối cùng, thực hiện một lần nữa với $p=2$, $q=7$, ta được nghiệm:

$$\begin{pmatrix} 0.00 & 0.00 & 0.00 & 0.00 & 0.14 & 0.43 & 0.00 & 1.00 & 22.00 \\ 0.00 & 0.00 & 0.00 & 1.00 & -0.43 & 0.71 & 0.00 & 0.00 & 5.00 \\ 0.00 & 0.00 & 0.00 & 0.00 & 1.57 & -2.29 & 1.00 & 0.00 & 33.00 \\ 0.00 & 1.00 & 0.00 & 0.00 & 0.29 & -0.14 & 0.00 & 0.00 & 9.00 \\ 1.00 & 0.00 & 0.00 & 0.00 & -0.14 & 0.57 & 0.00 & 0.00 & 9.00 \\ 0.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 4.00 \end{pmatrix}$$

Nó tương ứng với điểm (9, 9, 4) trên đơn hình, hàm mục tiêu đạt cực trị tại giá trị 22. Tất cả các phần tử của hàng 0 bây giờ là không âm, vậy sự tiếp tục chỉ làm giảm giá trị của hàm mục tiêu.

Ví dụ trên cho thấy sơ qua phương pháp đơn hình. Nói tóm tắt, nếu ta bắt đầu một ma trận hệ số tương ứng với một điểm của đơn hình, ta có thể thực hiện các bước liên tiếp trên các đỉnh của đơn hình để tăng dần giá trị của hàm mục tiêu cho tới lúc cực đại đạt được.

CÀI ĐẶT

Việc cài đặt phương pháp đơn hình trong trường hợp đã mô tả là hoàn toàn trực tiếp. Đầu tiên ta cần một thủ tục dùng làm phép khử Gauss như ở chương 37.

Chương trình pivot ở trang sau cộng bội số của hàng p với mỗi hàng để được một cột q có tất cả các phần tử là 0 trừ số 1 ở hàng p. Giống như trong chương 37, cần phải giữ các giá trị của *a[p,q]* không thay đổi trước khi ta kết thúc việc sử dụng nó.

```
procedure pivot(p,q:integer);
  var j,k:integer;
  begin
    for j:=0 to N do
      for k:=M+1 downto 1 do
        if (j<>p) and (k<>q) then
          a[j,k]:=a[j,k]-a[p,k]*a[j,q]/a[p,q];
    for j:=0 to N do if j<>p then a[j,q]:=0;
    for k:=1 to M+1 do if k<>q then a[p,k]:=a[p,k]/a[p,q];
    a[p,q]:=1;
  end;
```

Khác với phép khử Gauss dùng giải hệ tuyến tính, thuật toán đơn hình chỉ bao gồm việc tìm giá trị của p và q như đã mô tả rồi gọi pivot, lặp lại quá trình này tới khi đạt được cực trị hay là tới lúc đơn hình được xác định là không bị chận.

```
repeat
  q:=0; repeat q:=q+1 until (q=M+1)or(a[0,q]<0);
  p:=0; repeat p:=p+1 until (p=N+1)or(a[p,q]>0);
  for i:=p+1 to N do
    if a[i,q]>0 then
      if (a[i,M+1]/a[i,q])<(a[p,M+1]/a[p,q]) then p:=i;
  if (q<M+1) and (p<N+1) then pivot(p,q);
until (q=M+1) or (p=N+1);
```

Nếu chương trình kết thúc với $q=M+1$ thì một nghiệm tối ưu đã được xác định, cực đại của hàm mục tiêu sẽ là $a[0,M+1]$ và giá trị các biến có thể tìm từ cơ sở. Nếu chương trình kết thúc với $p=N+1$ thì miền đơn hình là không bị chận.

Chương trình này bỏ qua các vòng lặp vô hạn. Để cài đặt phương pháp Bland, cần phải ghi nhớ các cột sẽ rời khỏi cơ sở. Điều này thực

hiện dễ dàng bằng cách đặt *outb[p]:=q* sau mỗi vòng lặp. Sau đó, vòng lặp để tính toán *p* có thể được thay đổi để đặt *p:=i* nếu đẳng thức xảy ra trong tiêu chuẩn tỉ số và *outb[p]<outb[q]*. Một cách khác, một phần tử ngẫu nhiên có thể dùng để sinh ra một biến ngẫu nhiên và thay thế mảng *a[p,q]* (hay *a[i,q]*) bằng *a[(p+x) mod (N+1),q]* (hay *a[(i+x) mod (N+1),q]*). Mục đích của phép này là cũng tìm kiếm ở cột *q* như trước, nhưng bắt đầu bằng một điểm ngẫu nhiên thay cho theo thứ tự. Ta có thể sử dụng cùng kỹ thuật để chọn một cột ngẫu nhiên (với phần tử âm ở hàng 0) để tính toán.

Chương trình và ví dụ trên chỉ dùng để mô tả các nguyên lý chính của thuật toán đơn hình, các trường hợp phức tạp có thể nảy sinh trong áp dụng được bỏ qua. Chương trình yêu cầu ma trận phải có một cơ sở thích hợp. Chương trình bắt đầu với giả thiết: có 1 nghiệm với *M-N* biến trong hàm mục tiêu là 0 còn *NxN* ma trận con gồm các biến thêm vào có thể chuyển thành ma trận đơn vị. Nói chung điều này không đúng cho các trường hợp tổng quát. Nếu ta tìm được một nghiệm, ta có thể dùng một phép biến đổi thích hợp (chuyển điểm này, tức nghiệm này, thành gốc tọa độ) để chuyển ma trận về dạng yêu cầu, tuy nhiên, thông thường ta chẳng biết là bài toán như vậy có một nghiệm cần thiết hay không.

Thực ra, việc chỉ ra rằng bài toán có nghiệm hay không cũng khó khăn như việc tìm ra được nghiệm tối ưu. Vì thế, ta không nên ngạc nhiên khi kỹ thuật dùng để chỉ ra sự tồn tại nghiệm cũng chính là phương pháp đơn hình ! Đặc biệt, ta thêm vào một số biến s_1, s_2, .., *sN* và thêm biến s_i vào phương trình thứ *i*. Nó được thực hiện bằng cách thêm vào ma trận *N* cột lấp đầy với ma trận đơn vị. Nó cho tức khắc một cơ sở thích hợp cho một chương trình quy hoạch tuyến tính mới. Điều khôn khéo ở đây là chạy thuật toán trên với hàm mục tiêu $-s_1-s_2- \ldots -s_N$. Nếu bài toán đầu có một nghiệm thì hàm mục tiêu này có thể cực đại hóa tại 0. Nếu cực đại đạt được tại một điểm khác 0 thì bài toán quy hoạch tuyến tính ban đầu là không thích hợp. Nếu

cực đại là 0 thì trường hợp bình thường là $s_1, .., s_N$ trở thành các biến không cơ sở, vậy ta có thể tính được một cơ sở thích hợp cho bài toán ban đầu. Trong trường hợp suy biến, vài biến thêm vào có thể còn trong cơ sở, vậy ta cần thiết phải thực hiện tiếp quá trình lặp để loại các biến này.

Tóm lại, một quá trình gồm hai bước thường được sử dụng để giải bài toán tuyến tính tổng quát. Thứ nhất, ta giải bài toán quy hoạch tuyến tính với các biến s thêm vào để nhận được một điểm trên đơn hình của bài toán ban đầu của ta. Sau đó, ta bỏ các biến này rồi đưa vào hàm mục tiêu của bài toán ban đầu để tìm lời giải.

Việc phân tích đánh giá thời gian xử lý của phương pháp đơn hình cực kỳ phức tạp và thu được rất ít kết quả. Không ai biết được một phương pháp lặp tốt nhất vì không có một kết quả nào để ta có thể tính được bao nhiêu bước cần làm trong một lớp bài toán có ý nghĩa. Có thể xây dựng các ví dụ cho thấy thời gian xử lý có thể rất lớn. Tuy nhiên, tất cả những người đã sử dụng thuật toán này trong ứng dụng đều nhất trí coi tiêu chuẩn kiểm nghiệm là hiệu quả trong các bài toán thực tế.

Dạng đơn giản của thuật toán đơn hình đã được xét là một phần của một công cụ toán đẹp và tổng quát, nó cung cấp một công cụ đầy đủ để có thể dùng để giải các bài toán phong phú trong ứng dụng.

BÀI TẬP

1. Vẽ một đơn hình xác định bởi các bất đẳng thức $x_1 \geq 0, x_2 \geq 0, x_3 \geq 0, x_1 + 2x_2 \leq 20$ và $x_1 + x_2 + x_3 \leq 10$.

2. Tính toán lại ví dụ trong lý thuyết nếu cột q được chọn theo nguyên tắc: q là số lớn nhất trong các cột mà $a[0,q]$ âm.

3. Làm lại ví dụ trong lý thuyết với hàm mục tiêu $x_1 + 5x_2 + x_3$

4. Điều gì sẽ xảy ra nếu thuật toán đơn hình chạy trên một ma trận với một cột gồm các số 0.

5. Thuật toán đơn hình có sử dụng cùng số bước nếu các hàng của ma trận đưa vào hoán vị không ?

6. Cho một phát biểu dạng quy hoạch của ví dụ bài toán túi xách (knapsack problem) trong chương 42.

7. Bao nhiêu bước lặp là cần thiết để giải bài toán: "cực đại hóa $x_1 + ... + x_M$ với các ràng buộc $x_1, .., x_M \leq 1$ và $x_1, .., x_M \geq 0$"?

8. Xây dựng một bài toán gồm N bất đẳng thức hai biến mà thuật toán đơn hình cần ít nhất $N/2$ bước lặp.

9. Cho một ví dụ về bài toán quy hoạch ba chiều để cho thấy sự khác nhau giữa phương pháp "tăng nhanh nhất" và phương pháp "dốc xuống nhất".

10. Thay đổi chương trình cài đặt của lý thuyết để viết ra các tọa độ của điểm tối ưu.

MỤC LỤC

TẬP 2 - CÁC THUẬT TOÁN CHUYÊN DỤNG
PHẦN 5 - CÁC THUẬT TOÁN HÌNH HỌC

PHẦN 6 - CÁC THUẬT TOÁN ĐỒ THỊ

PHẦN 8 - CÁC ĐẶC TRƯNG CẤP CAO

ROBERT SEDGEWICK

CẨM NANG
THUẬT TOÁN

(Tập 2 : CÁC THUẬT TOÁN CHUYÊN DỤNG)

Chịu trách nhiệm xuất bản	Pts. TÔ ĐĂNG HẢI
Người dịch	TRẦN ĐAN THƯ BÙI THỊ NGỌC NGA NGUYỄN HIỆP ĐOÀN ĐẶNG ĐỨC TRỌNG TRẦN HẠNH NHI
Hiệu đính	GS.TS. HOÀNG KIẾM
Biên tập	TUẤN NGHĨA
Trình bày bìa	THU THỦY

6T7.3
KHKT-2006 72-2005/CXB/75-39/KHKT

NHÀ XUẤT BẢN KHOA HỌC KỸ THUẬT
70 Trần Hưng Đạo - Hà Nội
Chi nhánh 28 Đồng Khởi Q.1 TP.HCM

In 1000 cuốn, khổ 14,5 x 20,5 cm tại Xưởng in Ban TT - VH Thành ủy TP.HCM. Giấy QĐXB số: 72-2005/CXB/75-39/KHKT cấp ngày 07/11/2005 NXB KHKT.

In xong và nộp lưu chiểu tháng 1/2006.

Niklaus Wirth said that

Data Structures

\+

Algorithms

\=

Programs

Giá : 32.000đ

ALGORITHMS ALGORITHMS ALGORITHMS ALGORITHMS