**MỤC LỤC**

[MỤC LỤC 1](#_Toc362706692)

[1. Danh mục các bảng 4](#_Toc362706693)

[2. Danh mục các hình 4](#_Toc362706694)

[Chương 1 6](#_Toc362706695)

[MỞ ĐẦU 6](#_Toc362706696)

[1.1. Phạm vi và mục tiêu 6](#_Toc362706697)

[1.2. Những đóng góp chính của khóa luận 7](#_Toc362706698)

[1.3. Cấu trúc khóa luận 8](#_Toc362706699)

[1.4. Qui ước ký hiệu và viết tắt 8](#_Toc362706700)

[Chương 2 10](#_Toc362706701)

[TỔNG QUAN VỀ GIẢI THUẬT 10](#_Toc362706702)

[VÀ ĐỘ PHỨC TẠP CỦA GIẢI THUẬT 10](#_Toc362706703)

[2.1. Giới thiệu 10](#_Toc362706709)

[2.2. Giải thuật và các tính chất của giải thuật 10](#_Toc362706710)

[2.3. Độ phức tạp của giải thuật 13](#_Toc362706711)

[2.4. Một số kỹ thuật cơ bản để thiết kế giải thuật 16](#_Toc362706712)

[2.5. Cơ sở toán học hỗ trợ tính độ phức tạp của giải thuật 22](#_Toc362706715)

[2.6. Kết luận 28](#_Toc362706716)

[Chương 3 29](#_Toc362706717)

[3.1. Giới thiệu 29](#_Toc362706720)

[3.2. Định nghĩa và ví dụ giải thuật ngẫu nhiên 29](#_Toc362706721)

[3.3. Phân loại các giải thuật ngẫu nhiên 31](#_Toc362706722)

[3.4. Các lĩnh vực áp dụng của giải thuật ngẫu nhiên 32](#_Toc362706723)

[3.5. Kết luận 33](#_Toc362706724)

[Chương 4 34](#_Toc362706725)

[4.1. Giới thiệu 34](#_Toc362706727)

[4.2. Bài toán sắp xếp 34](#_Toc362706728)

[4.3. Bài toán xác định số nguyên tố 38](#_Toc362706729)

[4.4. Bài toán tìm cặp điểm gần nhất 40](#_Toc362706730)

[4.5. Bài toán kiểm tra phép nhân ma trận 44](#_Toc362706731)

[4.6. Bài toán tìm cây bao trùm nhỏ nhất của đồ thị 46](#_Toc362706732)

[4.7. Kết luận 54](#_Toc362706733)

[Chương 5 55](#_Toc362706734)

[HIỆN THỰC CÁC ỨNG DỤNG 55](#_Toc362706735)

[GIẢI THUẬT NGẪU NHIÊN 55](#_Toc362706736)

[5.1. Giới thiệu 55](#_Toc362706738)

[5.2. Bài toán sắp xếp 55](#_Toc362706739)

[5.3. Bài toán xác định số nguyên tố 55](#_Toc362706740)

[5.4. Bài toán tìm cặp điểm gần nhất 60](#_Toc362706741)

[5.5. Bài toán so trùng mẩu 60](#_Toc362706742)

[5.6. Giải thuật ngẫu nhiên tìm cây bao trùm nhỏ nhất của đồ thị 60](#_Toc362706743)

[5.7. Kết luận 72](#_Toc362706744)

[Chương 6 74](#_Toc362706745)

[TỔNG KẾT VÀ ĐỀ NGHỊ 74](#_Toc362706746)

[3. Tổng kết 74](#_Toc362706748)

[4. Đề nghị 74](#_Toc362706749)

[TÀI LIỆU THAM KHẢO 74](#_Toc362706750)

## Danh mục các bảng

## Danh mục các hình

[Hình 4.4.1: Phân vùng các điểm 40](#_Toc362706751)

[Hình 4.4.2: Phân vùng các điểm hình vuông với cạnh *δ* 41](#_Toc362706752)

[Hình 4.4.3 Các dạng mở rộng khi nhân đôi *δ* 41](#_Toc362706753)

[Hình 4.4.4: Minh họa ví dụ 4.4.1 43](#_Toc362706754)

[Hình 4.6.1: Kết quả sau khi áp dụng Boruvka lần thứ nhất 48](#_Toc362706755)

[Hình 4.6.2: Các cạnh được chọn theo bổ đề Boruvka lần hai 49](#_Toc362706756)

[Hình 4.6.3: Đồ thị kết quả hợp nhất các thành phần liên thông lần thứ hai 49](#_Toc362706757)

[Hình 4.6.4: Kết quả sau khi áp dụng Boruvka lần thứ hai 49](#_Toc362706758)

[Hình 4.6.5 Cây bao trùm nhỏ nhất của đồ thị 50](#_Toc362706759)

[Hình 5.3.1 Giao diện chính của giải thuật xác định số nguyên tố 57](#_Toc362706760)

[Hình 5.3.2 Đồ thị so sánh thời gian chạy của giải thuật cổ điển xác định số nguyên 59](#_Toc362706761)

[Hình 5.6.1 Giao diện chính của giải thuật tìm cây bao trùm nhỏ nhất 64](#_Toc362706762)

[Hình 5.6.2 Kết quả thu được khi thực hiện giải thuật tìm cây bao trùm nhỏ nhất 65](#_Toc362706763)

[Hình 5.6.5 66](#_Toc362706764)

[Hình 5.6.6 66](#_Toc362706765)

[Hình 5.6.7 67](#_Toc362706766)

[Hình 5.6.8 67](#_Toc362706767)

[Hình 5.6.9 68](#_Toc362706768)

[Hình 5.6.10 68](#_Toc362706769)

[Hình 5.6.11 69](#_Toc362706770)

[Hình 5.6.12 69](#_Toc362706771)

[Hình 5.6.13 70](#_Toc362706772)

[Hình 5.6.14 Biểu đồ thời gian chạy của giải thuật Prim 71](#_Toc362706773)

[Hình 5.6.15 Biểu đồ thời gian chạy của giải thuật ngẫu nhiên 71](#_Toc362706774)

# Chương 1

# MỞ ĐẦU

## Phạm vi và mục tiêu

Các nhà khoa học vào thế kỉ trước đã thấy rằng tính chất ngẫu nhiên là thành phần thiết yếu trong việc mô hình hóa và phân tích tự nhiên. Ngày nay, sự ngẫu nhiên đóng vai trò quan trọng trong hầu hết mọi lĩnh vực của khoa học từ di truyền học và tiến hóa trong sinh học đến mô hình biến động giá của nền kinh tế thị trường tự do. Khoa học máy tính cũng không ngoại lệ, phương pháp ngẫu nhiên và xác xuất đóng vai trò cốt lỗi trong khoa học máy tính hiện đại. Trong hai thập kỉ qua chúng ta đã chứng kiến sự phát triển to lớn của việc sử dụng lý thuyết xác xuất trong khoa học máy tính. Thêm vào đó nhiều kỹ thuật xác suất hiện đại và cải tiến tạo điều kiện cho việc phát triển các giải thuật ngẫu nhiên.

Khái niệm giải thuật ngẫu nhiên là một khái niệm tương đối mới. Trong mọi giải thuật được giới thiệu từ xưa đến nay, mỗi bước trong giải thuật đều được xác định. Với giải thuật ngẫu nhiên thì chúng ta không xác định được, trong quá trình thực hiện một giải thuật ngẫu nhiên nó sẽ đưa ra một chọn lựa tùy ý. Một số hành động được thực hiện một cách ngẫu nhiên. Ngoài đầu vào, giải thuật sử dụng một nguồn các con số ngẫu nhiên. Trong quá trình tính toán, nó sử dụng những lựa chọn ngẫu nhiên phụ thuộc vào những số ngẫu nhiên. Output có thể khác nhau nếu giải thuật chạy nhiều lần trên cùng một input.

Hơn nữa, giải thuật ngẫu nhiên thường đơn giản và dễ thực hiện. Các giải thuật này chạy nhanh hơn giải thuật cổ điển giải cùng một bài toán. Yêu cầu về không gian của một giải thuật ngẫu nhiên có thể nhỏ hơn của giải thuật nhất định mà chúng ta biết để giải cùng một bài toán . Trong vài trường hợp, giải thuật ngẫu nhiên là cách duy nhất hoặc tốt nhất để giải quyết vấn đề.(**Error! Reference source not found.**,**Error! Reference source not found.**,**Error! Reference source not found.**)

Giải thuật ngẫu nhiên và xác suất đóng vai trò quan trọng trong khoa học máy tính, với những ứng dụng từ tối ưu tổ hợp và máy tính học cách giao tiếp với mạng và giao thức bảo mật.

Hiện nay, các giải thuật được áp dụng trong lĩnh vực khoa học máy tính đã được tối ưu rất nhiều, các giải thuật này đã chạy rất nhanh. Tuy nhiên, thời gian rất quý giá, một giây cũng có thể thay đổi rất nhiều thứ. Với giải thuật ngẫu nhiên chúng ta sẽ tiết kiệm được rất nhiều thời gian so với giải thuật truyền thống cùng giải một bài toán. Thêm vào đó, giải thuật ngẫu nhiên có thể sử dụng ít tài nguyên hệ thống hơn. Với những ưu điểm mà giải thuật ngẫu nhiên mang lại, thì việc nghiên cứu các giải thuật ngẫu nhiên là hết sức có ý nghĩa. Vì vậy, chúng em đã chọn đề tài “*Nghiên cứu và ứng dụng các giải thuật ngẫu nhiên*” để làm khóa luận. Trong khóa luận này chúng em đã đầu tư nghiên cứu các giải thuật ngẫu nhiên, so sánh giải thuật ngẫu nhiên với các giải thuật cổ điển đã biết.

## Những đóng góp chính của khóa luận

Dưới đây là những đóng góp chính của khóa luận đối với lĩnh vực khoa học máy tính:

1. Trình bày khái niệm giải thuật và cách tính toán độ phực tạp của giải thuật
2. Trình bày khái niệm giải thuật ngẫu nhiên
3. Trình bày và áp dụng lý thuyết xác suất để tính toán độ phức tạp, cũng như khả năng cho ra kết quả đúng của giải thuật ngẫn nhiên.
4. Hiện thực một số giải thuật ngẫu nhiên cho các bài toán thuộc các lĩnh vực toán học, tìm kiếm, lý thuyết đồ thị.

## Cấu trúc khóa luận

Khóa luận bao gồm 6 chương. Chương 1 trình bày phạm vi, mục tiêu và ý nghĩa về lý thuyết cũng như ứng dụng của đề tài khóa luận, giới thiệu cấu trúc, các qui ước ký hiệu và viết tắt trong khóa luận. Mỗi chương tiếp theo, từ Chương 2 đến Chương 6 có một phần giới thiệu và một phần kết luận.

Chương 2 giới thiệu tổng quan về giải thuật và độ phức tạp của giải thuật. Đây là một số kiến thức cần thiết sử dụng cho việc thiết kế giải thuật. Chương này còn giới thiệu các công cụ toán học hỗ trợ phân tích, đánh giá độ phức tạp của giải thuật

Chương 3 trình bày tổng quát về giải thuật ngẫu nhiên và phân loại giải thuật ngẫu nhiên.

Chương 4 trình bày một số giải thuật ngẫu nhiên cho các bày toán thuộc nhiều lĩnh vực khác nhau như: sắp xếp, tìm kiếm, toán học, lý thuyết đồ thị….

Chương 5 là phần hiện thực các giải thuật ngẫu nhiên được nêu ra ở chương 4.

Chương 6 là phần tổng kết và đề nghị các hướng nghiên cứu trong tương lai liên quan đến giải thuật ngẫu nhiên.

## Qui ước ký hiệu và viết tắt

Trong sách này có một số qui ước và kí hiệu viết tắt như sau:

: Với mọi

: Thuộc

: Tồn tại

: -lớn

: Omega

: Theta

: Phép toán giao tập hợp

: Phép toán hợp tập hợp

BT: Bài toán.

In : Input

Out : Output

# Chương 2

# 

# TỔNG QUAN VỀ GIẢI THUẬT

# VÀ ĐỘ PHỨC TẠP CỦA GIẢI THUẬT



## Giới thiệu

Chương này giới thiệu một cách khái quát về giải thuật, độ phức tạp thời gian của giải thuật và trình bày một số công cụ toán học hỗ trợ việc phân tích và thiết kế giải thuật.

Đầu tiên Phần 2.2 trình bày khái niệm giải thuật, tính chất và ngôn ngữ biểu diễn giải thuật. Kế đến Phần 2.3 nêu khái niệm độ phức tạp của giải thuật, cách biểu diễn độ phức tạp thời gian của giải thuật và một số ví dụ minh họa. Tiếp theo, Phần 2.4 là trình bày một số chiến lược thiết kế cơ bản của giải thuật là cơ sở cho việc phát triển các giải thuật truyền thống cũng như giải thuật ngẫu nhiên. Phần 2.5 trình bày một số công cụ toán học làm cơ sở cho việc tính toán độ phức tạp của giải thuật. Cuối cùng, Phần 2.6, là các kết luận đáng lưu ý của Chương này.

## Giải thuật và các tính chất của giải thuật

Một cách không hình thức, giải thuật là cách thức để đạt được lời giải của bài toán dựa trên một số hữu hạn các bước cần phải thao tác, tính toán. Giải thuật được định nghĩa một cách hình thức ([3],[7]) như sau:

**Định nghĩa 2.2.1** *Giải thuật* là một thủ tục xác định bao gồm một dãy hữu hạn các bước cần thực hiện để thu được lời giải bài toán.

Xét về bản chất toán học, mỗi giải thuật tương ứng với một *ánh xạ* (mapping). Nghĩa là, ánh xạ là mô hình toán học của giải thuật. Mỗi giải thuật có một tên, một tập dữ liệu *đầu vào* (input) và một tập dữ liệu *đầu ra* (output) tương ứng với yêu cầu và lời giải bài toán. Một giải thuật có một số đặc tính cơ bản sau.

1. Tính *chính xác* (precision), đảm bảo kết quả tính toán hay các thao tác giải thuật được thực hiện là chính xác.
2. Tính *hữu hạn* (finiteness), nghĩa là thuật giải phải dừng sau một số hữu hạn bước thực hiện.
3. Tính *phổ dụng* (generality), nghĩa là giải thuật có thể áp dụng cho một lớp các bài toán có cùng kiểu dữ liệu đầu vào với kích thước khác nhau.

Để có thể mô tả và biểu diễn giải thuật và từ đó có thể viết thành các chương trình máy tính, chúng ta có thể dùng *ngôn ngữ tự nhiên* (natural language), *mã giả* (pseudocode) hay *ngôn ngữ lập trình cấp cao* (high programming language) như Pascal, C/C++, v.v. Trong phạm vi khóa luận này, các giải thuật sẽ được trình bày bằng mã giả.

Ví dụ 2.2.1 Giải thuật tìm số *k* trong dãy số *a*1, *a*2, ...., *an*, trong đó n là một số tự nhiên, có thể được viết như sau.

SequentialSearch(*a*[1..*n*], *k*)

1 **for** *i* ← 1 **to** *n*

2 **do if** *a*[*i*] =*k*

3 **then** **return** true

4 **return** false

Ở đây, đầu vào là một số *k* và một dãy *n* số *a*[1], *a*[2],…, *a*[*n*], đầu ra là một giá trị logic biểu diễn có hay không số *k* trong dãy đã cho. Đây cũng là một giải thuật có mặt hầu như trong bất kỳ hệ thống thông tin nào. Chẳng hạn, trong CSDL bài toán tìm kiếm một đối tượng theo một khóa dựa trên thuật giải này.

Trong lý thuyết độ phức tạp tính toán, giải thuật được chia làm hai loại. Giải thuật *đơn định* (deterministic)và *không* *đơn định* (nondeterministic) như các định nghĩa sau.

**Định nghĩa 2.2.2** Một giải thuật được gọi là *đơn định* nếu kết quả của mỗi thao tác, tính toán luôn luôn được xác định một cách duy nhất.

Giải thuật đã nêu trong Ví dụ 2.2.1 là một giải thuật đơn định. Vì rõ ràng kết quả của mỗi tính toán trong giải thuật này là duy nhất.

**Định nghĩa 2.2.3** Một giải thuật được gọi là *không đơn định* nếu tồn tại các thao tác, tính toán mà kết quả thuộc về một tập nhưng không biết chính xác kết quả nào trong tập đó.

Ví dụ 2.2.2 Giải thuật sắp xếp sau đây là một giải thuật không đơn định (có độ phức tạp O(n))

NSort(*A*[1..*n*]) // sắp xếp *n* số nguyên dương

1 **for** *i* ← 1 **to** *n* **do** *B*[*i*] ← 0

2 **for** *i* ← 1 **to** *n*

3 **do**  *j* := **choice**(1:*n*) //thực hiện đồng thời nhiều bản sao (không đơn định)

4 **if** *B*[*j*] <> 0 **then failure**

5 **else** *B*[*j*] ← *A*[*i*]

6 **for** *i* ← 1 **to** *n*-1

7 **do if** *B*[*i*] > *B*[*i*+1] **then failure**

8 **return** *B* //*B*[1..*n*] là mảng được sắp các số trong *A*[1..*n*]

Lớp các giải thuật ngẫu nhiên được trình bày trong các chương sau là các giải thuật không đơn định.

## Độ phức tạp của giải thuật

*Độ phức tạp của giải thuậ*t (complexity of algorithm) là độ do về tính hiệu quả của giải thuật. Độ phức tạp là phương tiện, công cụ để đánh giá giải thuật tốt hay xấu, giải thuật dễ hay khó. Độ phức tạp giải thuật là một khái niệm cơ bản, trung tâm của lý thuyết phân tích và thiết kế giải thuật.

**Định nghĩa 2.3.1** *Độ phức tạp của giải thuật* là chi phí về tài nguyên của hệ thống (chủ yếu là thời gian, bộ nhớ, bộ xử lí và đường truyền) cần thiết để thực hiện giải thuật ([3] **Error! Reference source not found.**).

Để tính toán độ phức tạp của giải thuật, cần thiết phải phân tích giải thuật. Đó là kỹ thuật đánh giá tài nguyên chi phí cho giải thực hiện như định nghĩa sau.

**Định nghĩa 2.3.2** Phân tích giải thuật (analyzing of algorithm) là quá trình tìm ra những đánh giá về tài nguyên cần thiết để thực hiện giải thuật.

Trong khóa luận này, chỉ độ phức về thời gian của giải thuật được quan tâm, vì độ phức tạp thời gian của giải thuật phụ thuộc chủ yếu vào kỹ thuật thiết kế và độc lập với phần cứng của hệ thống máy tính.

Một cách cụ thể hơn, thời gian chi phí cho một giải thuật phụ thuộc vào số thao tác (lệnh) mà giải thuật thực hiện, nghĩa là phụ thuộc vào kích thước đầu vào của giải thuật. Nói các khác thời gian thực hiện của một giải thuật là một hàm *T*(*n*) của đối số n nguyên dương biểu diễn kích thức của dữ liệu bài toán mà giải thuật được thiết kế để giải nó. Trong quá trình phân tích giải thuật, thời gian được tính toán có thể được chia thành ba trường hợp như định nghĩa sau.

**Định nghĩa 2.3.3** Thời gian tối thiểu, nhiều nhất, trung bình để thực hiện giải thuật với kích thước đầu vào n gọi là thời gian chạy *tốt nhất* (best-case), *xấu nhất* (worst-case), *trung bình* (average-case) của giải thuật.

Ví dụ 2.3.1 Xét giải thuật trong Ví dụ 2.2.1, gọi α, β và γ là thời gian thực hiện của phép gán, phép so sánh và trả về của giải thuật, thì

1. Trường hợp tốt nhất xảy ra nếu *a*[1] = *k*, khi đó *T*(*n*) = α+β + γ.
2. Trường hợp xấu nhất xảy ra nếu *k* ∉ {*a*[1], *a*[2], ...., *a*[*n*]} khi đó *T*(*n*) = (*n*+1)α + *n*β + γ.
3. Trường hợp trung bình được tính toán dựa trên kỳ vọng xác suất. Gọi xác suất tìm kiếm thành công là *p* (xác không tìm thấy là 1-*p*) khi đó

*T*(*n*) = Σ*i*=1,..*n*(*i*α + *i*β + γ)*p*/*n* + ((*n*+1)α + *n*β + γ))(1-*p*)

= (α + β)(*n*+1)*p*/2 +γ*p* + ((*n*+1)α + *n*β + γ))(1-p)

Lưu ý rằng khi *p*=1 (luân luôn tìm thấy), thì *T*(*n*)= (α + β)(*n*+1)/2 +γ.

Để biểu diễn độ phức tạp giải thuật một cách đơn giản, gần đúng, chúng ta dùng khái niệm tốc độ tăng của hàm như định nghĩa sau ([3][7]).

**Định nghĩa 2.3.3** Giả sử là hàm không âm đối số nguyên dương *n*, khi đó

1. có tốc độ tăng (bậc) không quá và viết nếu có hằng số và số nguyên dương N sao cho ,
2. có tốc độ tăng ít nhất là và viết nếu có hằng số và số nguyên dương N sao cho ,
3. có tốc độ tăng là và viết nếu và

Các ký hiệu O, và giọi là ký hiệu tiệm cận (asymptotic notation). Chúng ta dễ dàng chứng minh được định lý sau.

**Định lý 2.3.1** Giả sử và *c* là một hằng số thì

Định lý có thể được phát biểu tương tự cho các ký hiệu và Bây giờ thời gian chạy của một giải thuật có thể được định nghĩa theo các ký hiệu tiệm cận như sau.

**Định nghĩa 2.3.4** Nếu giải thuật có thời gian chạy tốt nhất (trung bình, xấu nhất) là và thì ta nói thời gian chạy tốt nhất (trung bình, xấu nhất) của giải thuật có bậc (tốc độ tăng) không quá hay thời gian chạy tốt nhất (trung bình, xấu nhất) của giải thuật là .

Chúng tôi lưu ý rằng, Định nghĩa 3.2.4 có thể được phát biểu cho các ký hiệu tiệm cận. Tốc độ tăng của thời gian chạy càng lớn thì giải thuật càng chậm (chẳng hạn, giải thuật có thời gian chạy hiệu quả hơn giải thuật có thời gian chạy

**Ví dụ 2.3.2** Phân tích giải thuật tính gần đúng

Exp(x, n)

1 s ←1

2 **for** i ←1 to n

3 **do** p ←1

4 **for** j ←1 to i

5 **do** p ←p\*x/j

6 s ←s + p

7 **return** s

Gọi lần lượt là thời gian thực hiện lệnh gán, trả về, thì

Tốc độ tăng của độ phức tạp giải thuật biểu diễn theo O, , phụ thuộc vào số lần thực hiện của thao tác (lệnh) được thực thi nhiều lần nhất và có có chi phí lớn nhất trong giải thuật. Các thao tác được thực thi nhiều lần nhất và có có chi phí lớn nhất trong giải thuật gọi là *thao tác cơ bản* (basic operation). Vì vậy, có thể tính độ phức tạp đơn giản bằng cách chỉ xác định và đếm số thao tác cơ bản trong giải thuật ([3],[8]).

**Ví dụ 2.3.3** Phân tích độ phức tạp của giải thuật sau (một cải tiến của giải thuật trong Ví dụ 2.3.2)

Exp(, n)

1 s ←1

2 p ←1

3 **for** i 🡨1 to *n*

4 **do** p ← // phép toán cơ bản là phép chia

5 s ←s + p

6 **return** s

Rõ ràng lệnh cơ bản là *x*/*i*, nên , trong đó là thời gian thực hiện của thao tác *x*.

## Một số kỹ thuật cơ bản để thiết kế giải thuật



Để có thể phát triển các giải thuật hiệu quả cho các bài toán ứng dụng thực tế, cần có các phương pháp, cách thức và *chiến lược thiết kế* (design strategy) một cách khoa học. Dưới đây là một số chiến lược thiết kế cơ bản nhất, thường được ứng dụng nhiều nhất trong quá xây dựng các giải thuật.

Chiến lược trực tiếp



Chiến lược *trực tiếp* (brute force) áp dụng để thiết kế giải thuật cho một bài toán dựa trên định nghĩa và các khái niệm liên quan trực tiếp đến bài toán đó ([3][3]). Đây là chiến lược dễ dàng áp dụng và được lựa chọn đầu tiên trong quá trình phân tích thiêt kế giải thuật cho bài toán . Chiến lược trực tiếp được áp dụng cho một lớp rất rộng các bài toán. Chi phí thiết kế rẻ, nên chiến lược trực tiếp thích hợp cho các bài toán kích thước nhỏ. Tuy nhiên, chiến lược trực tiếp có thể sinh ra một số giải thuật có độ phức tạp khá lớn hoặc rất lớn). Mặc dù vậy, chiến lược trực tiếp là cơ sở để đề xuất các giải thuật mới.

**Ví dụ 2.4.1** Xét bài toán tính tổng *S*=1+2+…+ *n*, với *n* là một số nguyên dương. Giải thuật dựa trên chiến lược trực tiếp để tính tổng này như sau.

Sum(*n*)

1 *S* ←0

2 **for** *i* ←1 to *n*

3 **do** *S* ←*S*+*i*

4 **return** *S*

Rõ ràng đây là một giải thuật được thiết kế trực tiếp và tự nhiên nhất dựa trên biểu diễn trực tiếp của tổng *S*. Kỹ thuật đơn giản là cộng dồn (như trong giải thuật), hoặc đệ qui đều có thể tính *S* một cách trực tiếp. Dễ thấy độ phức tạp của giải thuật là O(*n*). Tuy nhiên nếu dùng phép biến đổi *S*=*n*(*n*+1)/2 thì sẽ có giải thuật O(1), hiệu quả hơn rất nhiều.

Chiến lược chia để trị

Chiến lược chia để trị (Divide-and-Conquer) là chiến lược thiết kế được áp dụng rất rộng rãi trong giải toán nói chung và thiết kế giải thuật nói riêng ([3]). Giải thuật giải một bài toán dựa trên chiến lược chia để trị được phát triển theo ba bước:

1. Chia bài toán thành các bài toán con, thường là cùng kiểu.
2. Giải các bài toán con (thường dùng kỹ thuật đệ qui).
3. Kết hợp lời giải các bài toán con để có lời giải bài toán ban đầu .

**Ví dụ 2.4.2** Cho một mảng các số được sắp theo thứ tự tăng và một số , tìm trong mảng số có giá trị bằng

BinarySearch(*A*[*p*..*r*], *K*)

1. **if** *p*≤*r*
2. **then** *m* ← ⎣(*p*+*r*)/2⎦
3. **if** *A[m]=K*
4. **then return** true
5. **else if** *A*[*m*]>*K*
6. **then return** BinarySearch(*A*[*p*..*m*-1], *K*)
7. **else return** BinarySearch(*A*[*m+1*..*r*], *K*)
8. **else return** false

Để tính độ phức tạp của giải thuật này, đặt *n*=*r*+*p*-1 (số phần phần tử của mảng). Khi đó *T*(*n*)= O(1), nếu *p*>*r* và *T*(*n*)=*T*(⎣*n*/2⎦)+O(1), nếu *p*≤*r*. Không mất tính tổng quát giả sử *m* là số nguyên sao cho *n*=2*m*. Khi đó *T*(*n*)≤*T*(*n*/2)+1=*T*(*n*/22)+2=…= *T*(*n*/2*m*)+*m*. Vậy *T*(*n*) ≤T(1)+*m*= O(1)+log2*n*=O(log2*n*).

Chiến lược quy hoạch động



Chiến lược qui hoạch động (dymamic programming) giải một bài toán bằng cách giải các bài toán con kích thước nhỏ hơn([3]). Nghiệm của bài toán có được bằng cách kết hợp nghiệm của các bài toán con. Nghiệm (lời giải) bài toán ban đầu và các bài toán con được biểu diễn bởi một hệ thức truy hồi. Sử dụng chiến lược qui hoạch động để giải một bài toán được chia làm hai bước:

1. Mô hình hóa lời giải bài toán bằng một hệ thức truy hồi
2. Giải hệ thức truy hồi bằng một giải thuật hiệu quả (từ dưới lên)

Chiến lược qui hoạch động phát triển giải thuật giải bài toán bằng cách giải các bài toán con từ *dưới lên* (bottom up). Kết quả các bài toán con được lưu trữ trong một bảng và được sử dụng để giải bài toán đã cho. Vì kết quả các bài toán con được lưu trữ trong một bảng và được sử dụng để giải bài toán ban đầu nên chiến lược qui hoạch động rất hiệu quả. Qui hoạch động thường được ứng dụng để giải các bài toán tối ưu.

Ví dụ 2.4.3 Cho một dãy n đồng xu có giá trị tương ứng là c1, c2, . . . , c*n*, hãy nhặt một số các đồng xu sao cho tổng giá trị lớn nhất và không có 2 đồng xu nào kề nhau.

Trước hết cần xây dựng hệ thức truy hồi biểu diễn tổng giá trị lớn nhất các đồng xu nhặt được F(*n*) xét hai trường hợp. Nếu đồng xu cuối cùng được chọn thì giá trị lớn nhất là *cn*+F(*n*-2). Nếu đồng xu cuối cùng không được chọn thì giá trị lớn nhất là F(*n-*1). Suy ra hệ thức truy hồi là

F(*n*)=*max*{*cn*+ F(*n* − 2), F(*n* − 1)} với *n* > 1,

F(0)=0, F(1)=*c*1

Giải thuật qui hoạch động được thiết kế như sau.

CoinRow(C[1..n])

1 F[0]←0; F[1]←C[1]

2 **for** *i* ←2 **to** *n*

3 **do**  F[*i*]←*max*(C[*i*]+ F[*i* − 2], F[*i* − 1])

4 **return** F[*n*]

Dễ thấy độ phức tạp của giải thuật là O(*n*). Chúng tôi lưu ý rằng, nếu dùng chiến lược trực tiếp để giải bài toán này sẽ dẫn đến giải thuật có độ phức tạp lũy thừa (rất lớn).

Chiến lược tham ăn



Giải thuật được thiết kế bằng chiến lược *tham ăn* (greedy) giải các bài toán con trước khi giải bài toán gốc. Quá trình tìm lời giải được thực hiện thông qua một dãy các bước để tìm lời giải (nghiệm) tối ưu cục bộ (lời giải các bài toán con) cho đến khi tìm thấy lời giải bài toán gốc. Mỗi bước tìm nghiệm cục bộ thỏa mãn ba tính chất sau.

1. Phải thỏa mãn ràng buộc bài toán.
2. *Tối ưu cục bộ* (locally optimal)
3. Không thay đổi nghiệm cục bộ trong các bước kế tiếp.

Chiến lược greedy luôn thực hiện một “lựa chọn” tốt nhất hiện tại khi tìm kiếm lời giải bài toán con mà không quan tâm đến bước tiếp theo. Chiến lược gready không giải tất cả các bài toán con (tối ưu) như qui hoạch động mà mỗi bước chỉ tìm lời giải tối ưu trong một tập các bài toán con ([3][3])

**Ví dụ 2.4.4** Tìm cây bao trùm nhỏ nhất của đồ thị liên thông vô hướng có trọng số.

Bài toán này được giải bằng chiến lược tham lam theo Kruskal như sau.

Kruskal(*G*, *w*) // Đồ thị *G* = (*V*, *E*) có trọng số mỗi cạnh *e* là *w*(*e*)

1 *F* ← ∅; *Q* ←*E*[*G*]; *N* ← *V*[*G*]

3 **while** **⏐***F***⏐**<**⏐***N***⏐**-1 and *Q*≠ ∅

4 **do** *e* ← ExtractMin(*Q*) // e có trọng số bé nhất

5 **if** *F*∪{*e*} not contain cycle **then** *F* ← *F*∪{*e*}

6 **if** ⏐*F*⏐<**⏐***N***⏐**-1

7 **then** *G* is not connected

8 **else** **return** *T* // T= (V, F)

Dòng 4 chọn cạnh có trọng số nhỏ nhất bằng thủ tục ExtractMin(*Q*) tại một lần lặp thể hiện chiến lượng tham lam trong giải thuật. Thời gian chạy của ExtractMin(*Q*) là O(log2 *E*). từ đó thời gian chạy của giải thuật này được xác định là O(*V* log2 *E*).

Chiến lược biến đổi để trị



Chiến lược biến đổi để trị (transform-and-conquer) gồm hai tầng (giai đoạn) ([3]):

1. Biến đổi thể hiện (instance-input) của bài toán để có thể dễ giải (hoặc giải hiệu quả) hơn.
2. Giải bài toán trên thể hiện (input) đã được biến đổi.

**Ví dụ 2.4.5** Xét bài toán tính tổng *S*=1+2+…+ *n*, với *n* là một số nguyên dương (trong Ví dụ 2.4.1). Áp dụng chiến lược biến đổi để trị biến đổi tổng *S*=1+2+…+ *n=n*(*n+*1)/2, ta có giải thuật.

Sum(*n*)

1 **return** *n*\*(*n*+1)/2

Rõ ràng độ phức tạp giải thuật là O(1), tốt hơn rất nhiều độ phức tạp O(n) của giải thuật trực tiếp trong Ví dụ 2.4.1).

Chiến lược quay lui



Giải thuật được thiết kế bằng chiến lược quay lui (backtraking) biểu diễn nghiệm bài toán như một vector với được chọn trong tập nào đó. Quá trình tìm được thực hiện bằng cách tính toán mỗi thành phần. tại một thời điểm. Tại bước thứ *i*, vector con tìm được gọi là một nghiệm bộ phận của bài toán. Để tìm thành phần thứ *k* của nghiệm (thỏa một số ràng buộcnào đó), khi đã chọn được thành phần của , giải thuật chọn (tính) trong số các khả năng có thể có của nó trong . Với mỗi khả năng j, giải thuật kiểm tra xem có chấp nhận đươc không. Nếu chấp nhận thì xác định theo j, khi thì có một lời giải, ngược thì tiếp tục xác định (thường bằng đệ qui). Nếu thử tất cả các khả năng mà không có khả năng nào được chấp nhận thì quay lui lại bước trước để xác định lại ([3]). Lược đồ tổng quát cho giải thuật được thiết kế theo chiến lược quay lui.

BackTracking(*x*[1..*k*], *n*) // xác định *x*[*k*], *k* nguyên

1 **for** *j* ←1 **to** *nk* // xét khả năng *j* trong *nk* khả năng

2 **do** **if** accepting *j*

3 **then** <computing *x*[*k*] in *Ak* that subjects to *j*>

4 **if** *k*=*n*

5 **then** < recording 1 solution >

6 **else** BackTracking(*x*[1..*k*+1], *n*)

Chiến lược nhánh cận



Chiến lược quay lui cho phép tìm được nghiệm đúng của bài toán, nhưng do phải thử mọi khả năng nên khi kích thước bài toán lớn sẽ rất kém hiệu quả. Chiến lược nhánh cận (Branch-and-Bound) khắc phục được hạn chế này bằng cách xác định nhánh cận với mục tiêu tại mỗi bước tìm kiếm, vì vậy loại bỏ được hầu hết các hướng tìm kiếm không cần thiết (trong cây không gian tìm kiếm). Chiến lược nhánh cận thường được áp dụng để giải các bài toán tối ưu. Chiến lược nhánh cận dựa trên chiến lược quay lui và một *hàm lượng giá* (estimate function) mục tiêu hướng đến các nhánh cận mục tiêu để tìm lời giải nhanh nhất có thể ([3]). Lược đồ tổng quát cho giải thuật được thiết kế theo chiến lược nhánh cận.

BranchBound(*x*[1..*k*], *n*) // developing local solution (*a*1, *a*2, …., *ak*-1)

1 **for** *ak* ∈ *Ak*

2 **do** **if** accepting *ak*

3 **then** *xk* ← *ak*

4 **if** *k* = *n*

5 **then** < update record>

6 **else if** *g*(*a*1, *a*2, …., *ak*) ≤ *f*\*

7 **then** BranchBound(*x*[1..*k*+1], *n*)

Trong lược đồ này *g*(*x*) là cận dưới (lượng giá) của hàm mục tiêu *f*(*x*) và *f*\* là giá trị nhỏ nhất của hàm mục tiêu hiện tại (gọi là một kỷ lục).

## Cơ sở toán học hỗ trợ tính độ phức tạp của giải thuật

Để tính toán độ phức tạp của giải thuật, cần có một số công cụ toán cần thiết được giới thiệu sau đây

Phần nguyên và một số hệ thức cơ bản

Khái niệm phần nguyên của một số thực được định nghĩa sau đây, thường được áp dụng trong thiết kế và phân tích giải thuật.

**Định nghĩa 2.5.1** Phần nguyên *sàn* (floor) của một số thực x, ký hiệu , là số nguyên lớn nhất nhỏ hơn *x*. Phần nguyên trần (ceiling) của số thực , ký hiệu , là các số nguyên nhỏ nhất lớn hơn *x*.

Một số hệ thức toán học cơ bản được ứng dụng để tính toán độ phức tạp của giải thuật:

1. Giả sử *x* là một số thực, thì
2. Nếu n là số tự nhiên và a, b, c là các số thực dương thì
3. Với mọi thì

( gọi là số điều hòa thứ n - Harmonic Number)

Hệ thức truy hồi

Hệ thức truy hồi **(**recurrence Relation) là cơ sở toán học để tính toán độ phức tạp của giải thuật đệ qui. Phần này giới thiệu khái niệm và một định lý cơ bản về hệ thức truy hồi.

**Định nghĩa 2.5.2** Hệ thức truy hồi đối với dãy là một phương trình có dạng

Ví dụ 2.5.1 Hệ thức truy hồi

Dãy được gọi là nghiệm của hệ thức truy hồi nếu các số hạng của nó thỏa mãn hệ thức truy hồi này. Ví dụ dãy {a*n*} với a*n* =3*n* là nghiệm của hệ thức trong Ví dụ 2.5.1.

**Định nghĩa 2.5.3** Một *hệ thức truy hồi tuyến tính thuần nhất* (linear homogeneous recurrence relation) bậc *k*>0 là một hệ thức truy hồi (**Error! Reference source not found.**) có dạng

Với là các hằng số thực,

Định lý 2.5.1 Giả sử là các hằng số, phương trình có k nghiệm phân biệt. Khi đó dãy {} là nghiệm của hệ thức truy hồi nếu và chỉ nếu với n =0,1,2,… Trong đó là các hằng số.

**Định lý 2.5.2** Giả sử *c*1, *c*2, ..., *ck* là các hằng số, phương trình *rk* - *c*1*rk*-1 - ... -*ck* = 0 có *t* nghiệm *r*1, *r*2, …, *rt* bội tương ứng *m*1, *m*2,,..., *m*t. Khi đó dãy {an} là nghiệm của hệ thức truy hồi *an*= *c*1*an*-1 + *c*2*an*-2 + ... + *ckan*-*k* nếu và chỉ nếu *an*= (α1, 0+ *α*1, 1*n*+…+ *α*1, *m*1-1*nm*1-1)*r*1*n* + (*α*2, 0+ *α*2,1*n* +…+ *α*2, *m*2-1n*m*2-1)*r*2*n* +...+ (*αt*, 0+ *αt*, 1*n*+…+ *αt*, *mt*-1n*mt*-1)*rtn* với *n*= 0, 1, 2, ... Trong đó *αi*, *j*, *i*=0,..*t*, *j*=0…, *mt* -1 là các hằng số.

**Định nghĩa 2.5.4** Một *hệ thức truy hồi tuyến tính không thuần nhất* (linear nonhomogeneous recurrence relation) bậc *k*>0 là một hệ thức truy hồi (**Error! Reference source not found.**) có dạng

với là các hằng số thực, , là một hàm chỉ phụ thuộc vào *n*

Ví dụ 2.5.2 Hệ thức là một hệ thức truy hồi tuyến tính không thuần nhất bậc 1.

Định lý 2.5.4 Nếu là một nghiệm của hệ thức truy hồi tuyến tính không thuần nhất , thì mọi nghiệm của nó có dạng , trong đó là nghiệm của hệ thức truy hồi tuyến tính .

Định lý 2.5.5 Giả sử thỏa mãn hệ thức truy hồi tuyến tính không thuần nhất , với , t là một số nguyên không âm. Khi đó

1. Nếu *s* không phải là nghiệm của phương trình thì (1) có một nghiệm dạng
2. Nếu *s* là một nghiệm bội *m* của phương trình thì (1) có một nghiệm dạng

**Định nghĩa 2.5.5** Hệ thức truy hồi dạng , trong đó a, b là các nguyên dương, được gọi là hệ thức truy hồi chia để trị (divide-and-conquer recurrence relation).

Ví dụ 2.5.3 là một hệ thức truy hồi chia để trị.

Định lý 2.5.6 Giả sử là một hàm tăng thỏa mãn hệ thức truy hồi khi n là số nguyên lớn hơn 1, c là một số thực dương , thì

Định lý 2.5.7 Giả sử là một hàm tăng thỏa mãn hệ thức truy hồi . khi n= là số nguyên lớn hơn 1, c và d là một số thực dương, thì

Các Định lý 2.5.6 và 2.5.7 được áp dụng để tính độ phức tạp của các giải thuật chia để trị.



Lý thuyết xác suất

Phần này giới thiệu một số khái niệm và hệ thức cơ bản nhất về xác suất được ứng dụng để thiết kế và phân tích giải thuật ngẫu nhiên trong các chương sau.



**Định nghĩa 2.5.6** Nếu *S* là một tập khác rỗng (gọi là không gian mẩu) của các *kết cục đồng khả năng* (equally likely outcome) và *E* là một sự kiện (một tập con của *S*), thì xác suất của *E* là *P*(*E*)*=* .

Chúng tôi lưu ý rằng một tập con *E* của *S* gọi là một sự kiện, một phần tử *s*∈*S*, gọi là một sự kiện cơ bản.

**Định nghĩa 2.5.3** Một phân bố xác suất (probability distribution) là một ánh xạ từ không gian mẩu đến tập các số thực *R*, Pr: *S*→ *R*, thỏa các tiên đề sau:

1. Pr(*A*)≥0, ∀*A*⊂*S*.
2. Pr(*S*)=1.
3. Pr(*A*∪*B*)=Pr(*A*)+Pr(*B*) với hai sự kiện bất kỳ *A*, *B* xung khắc, loại trừ lẫn nhau (mutually exclusive).

Từ các tiên đề ta suy ra:

1. Pr(∅)=0
2. Pr

**Định nghĩa 2.5.4** Một phân bố xác suất được gọi là *phân bố rời rạc* (discrete distribution) nếu nó được định nghĩa trên không gian mẩu rời rạc. Nếu *S* là không gian mẩu, thì Pr(*A*)=Σ*s*∈*A* Pr({*s*}), ∀*A*⊂*S*. Nếu Pr({*s*})=1/|*S*| thì ta có *phân bố đều* (uniform probability distribution) trên *S*

Để đơn giản khi sử dụng ký hiệu Pr({*s*}) thường được viết Pr(*s*).

**Ví dụ 2.5.4** Tung một đồng xu như nhau *n* lần, xác suất để đồng xu sấp (S) hoặc ngữa (N) mỗi lần tung là ½. Khi đó chúng ta có một phân bố đều trên không gian *S*={S, N}*n* với 2*n* phần tử. Mỗi sự kiện xác suất là một xâu *n* ký hiệu trong {S, N}. Mỗi xâu xuất hiện với xác suất là 1/2*n*. Xem sự kiện *A* là biến có (xâu) có đúng *k* mặt sấp xuất hiện. Khi đó |*A*|=, nên Pr(*A*)= /|*S*|

**Định nghĩa 2.5.4** Xác suất có điều kiện của sự kiện A khi có sự kiện B được định nghĩa là

Từ định nghĩa trên ta có các hệ thức sau:

1. , nếu A và B độc lập

Một hệ các sự kiện được gọi là đầy đủ nếu chúng đôi một xung khắc và =S.

**Định lý 2.5.8** Nếu các biến cố là một hệ đầy đủ các biến cố. H là một biến cố trong không gian mẩu. Khi đó

Ta có:

**Định lý 2.5.8** (Định lý Bayes)Giả sử A và B là 2 sự kiện trong không gian mẩu, ta có

**Định nghĩa 2.5.5** Một *biến ngẫu nhiên* (random variable) là một hàm *X*: *S* → *R* từ không gian mẩu *S* đến tập các số thực *R*.

Với mỗi biến ngẫu nhiên *X* và số thực *x*, sự kiện *X*=*x* được định nghĩa là {*s*∈*S*: *X*(*s*)=*x*}. Khi đó Pr(*X*=*x*)=Σ*s*∈*S*: *X*(*s*)=*x* Pr(*s*).

**Ví dụ 2.5.5** Nếu ta tung 2 con xúc sắc 6 mặt thì không gian xác suất *S* gồm 36 sự kiện cơ bản. Giả sử phân bố xác suất trên *S* là đều, biến ngẫu nhiên *X* được định nghĩa là gía trị lớn nhất của số trên mặt của 2 con xúc sắc xuất hiện khi tung chúng. Từ định nghĩa ta có Pr(*X*=3) =5/36. Sự kiện *X*=3 tương ứng với {(1; 3), (2; 3), (3; 3), (3; 2), (3; 1)}.

**Định nghĩa 2.5.5** *Giá trị kỳ vọng* (expected value) của biến ngẫu nhiên *X* được định nghĩa là

*E*(*X*)=Σ*x*∈*R* *x*.Pr(*X*=*x*)

Từ định nghĩa chúng ta suy ra rằng *E*(*X*)=Σ*s*∈*S* Pr(*s*).*X*(*s*). Ngoài ra, chúng ta cũng nhận xét rằng giá trị kỳ vọng của biến *X* cũng là giá trị trung bình của nó. Vì vậy, kỳ vọng là công cụ toán học giúp tính toán độ phức tạp trung bình của giải thuật.

**Ví dụ 2.5.6** Giả sử *X* là biến ngẫu nhiên nhận giá trị là số trên mặt của con xúc sắc xuất hiện khi chúng ta tung nó. Tính kỳ vọng của *X*.

Có 6 khả năng xảy ra, biến *X* nhận giá trị tương ứng là 1, 2,…, 6 với xác suất là 1/6 cho mỗi khả năng. Vì vậy *E*(*X*)=1/6(1+2+…+6)= 21/6 là số trung bình trên mặt con xúc sắc xuất hiện khi tung nó.

## Kết luận

Trong chương 2 này, các khái niệm cơ bản nhất về giải thuật, độ phức tạp, cũng như các chiến lược thiết kế giải thuật thông dụng nhất đã được trình bày làm cơ sở để xây dựng và phát triển các giải thuật hiệu quả. Một số công cụ toán học như hệ thức truy hồi làm phương tiện để tính độ phức tạp các giải thuật đệ qui. Các công cụ về lý thuyết xác suất là cơ sở để xây dựng và phân tích các giải thuật ngẫu nhiên trong chương 3 tiếp theo.

# Chương 3

**GIẢI THUẬT NGẪU NHIÊN**



## Giới thiệu

Chương này trình về giải thuật ngẫu nhiên, phân loại và các lĩnh vực áp dụng của giải thuật ngẫu nhiên.

Đầu tiên, Phần 3.2 trình bày định nghĩa của giải thuật ngẫu nhiên. Kế đến, Phần 3.3 trình bày về hai loại của giải thuật ngẫu nhiên đó là giải thuật Monte Carlo và giải thuật Las Vegas. Tiếp đó, Phần 3.4 trình bày các lĩnh vực áp dụng của giải thuật ngẫu nhiên như lý thuyết số, cấu trúc dữ liệu,…Cuối cùng, Phần 3.5 là kết luận của Chương này.

## Định nghĩa và ví dụ giải thuật ngẫu nhiên

Trong lịch sử, giải thuật ngẫu nhiên đầu tiên là một phương pháp được phát triển bởi Rabin năm ? [] được dùng cho bài toán tìm cặp điểm gần nhất trong hình học. Việc nghiên cứu các giải thuật ngẫu nhiên được thúc đẩy bởi việc khám phá kiểm tra số nguyên tố ngẫu nhiên vào năm 1977 (tức là xác định xem số đó là số nguyên tố hay hợp số) bởi Solovay and Strassen năm ? [] . Ngay sau đó, Rabin năm [] đã chứng minh rằng việc kiểm tra số nguyên tố của Miller năm 1976 có thể được chuyển thành một giải thuật ngẫu nhiên. ([8])

Trong các giải thuật truyền thống, mỗi bước trong giải thuật đều được xác định. Với giải thuật ngẫu nhiên thì chúng ta có thể không xác định được bước tiếp theo, vì trong quá trình thực hiện một giải thuật ngẫu nhiên nó sẽ đưa ra một chọn lựa tùy ý. Một số hành động được thực hiện một cách ngẫu nhiên. Hình 3.2.1 và 3.2.2 dưới đây là sự mô tả sự khác biệt giữa giải thuật xác định và giải thuật ngẫu nhiên.

**INPUT**

**ALGORITHM**

**OUTPUT**

**Hình 3.2.1 Sơ đồ giải thuật xác định**

**INPUT**

**ALGORITHM**

**OUTPUT**

**RANDOM NUMBERS**

**Hình 3.2.2 Sơ đồ giải thuật ngẫu nhiên**

Ngoài dữ liệu đầu vào, giải thuật còn dùng một nguồn số ngẫu nhiên và thực hiện các chọn lựa ngẫu nhiên trong quá trình thực hiện. Kết quả có thể khác nhau ngay cả khi thực hiện trên một đầu vào nhất định.

Định nghĩa 3.2.1 *Giải thuật ngẫu nhiên* (randomized algorithm) là một giải thuật mà ngoài dữ liệu đầu vào nó còn nhận một dãy số ngẫu nhiên nhằm mục đích đưa ra các lựa chọn ngẫu nhiên.

Ngay cả đối với đầu vào là cố định, những lần chạy khác nhau của giải thuật ngẫu nhiên thì đưa ra các kết quả khác nhau. Ví dụ, với đầu vào dữ liệu cố định, thời gian thực hiện của một giải thuật ngẫu nhiên có thể là một giá trị ngẫu nhiên ([1]).

Ví dụ 3.2.1 Bài toán tìm ‘a’ trong mảng. Cho một dãy phần tử, trong đó một nửa là *a* và nửa còn lại là *b*. Tìm *a* trong mảng.

Có thể phát triển hai giải thuật ngẫu nhiên để giải bài toán này như dưới đây ([8]).

Findinga\_lv(Array *a*,*n,k*)

1. **for** i=1 **to** k **do**
2. Randomly select one element out of *n* elements
3. **if** ‘a’ is found
4. **then return**

Nếu *a* được tìm thấy, giải thuật thành công, không thì giải thuật thất bại. Sau khi lặp lại *k* lần, xác suất tìm thấy *a* là:

Giải thuật này không bảo đảm thành công nhưng thời gian chạy là cố định. Sự chọn lựa thì được thực hiện chính xác *k* lần, do dó thời gian chạy là

Findinga\_lv(Array *a*,*n*)

* 1. **while** (‘a’ is not found)
  2. Randomly select one element out of *n* elements.

Giải thuật này thành công với xác suất là 1. Thời gian chạy là ngẫu nhiên (có thể lớn tùy ý) nhưng kỳ vọng (thời gian chạy trung bình) của nó bị chặn trên bởi .

## Phân loại các giải thuật ngẫu nhiên

Như đã nêu trong Ví dụ 3.2.1, chúng ta có thể giải một bài toán bằng hai giải thuật ngẫu nhiên. Một giải thuật có thời chạy luôn luôn cố định nhưng chỉ thành công với một xác suất nào đó. Giải thuật còn lại chắc chắn thành công nhưng thời gian chạy không xác định. Các giải thuật này tương ứng thuộc về lớp giải thuật Monte Carlo và Las Vegas. Một giải thuật ngẫu nhiên thuộc về một về một trong hai lớp này. Các lớp giải thuật Monte Carlo và Las Vegas lần lượt được định nghĩa như sau.

Định nghĩa 3.3.1 Một Giải thuật Monte Carlo là một giải thuật ngẫu nhiên có thời gian chạy luôn luôn xác định nhưng kết quả có thể sai với một xác suất nào đó.

Trong lĩnh vực tính toán, một giải thuật Monte Carlo là một giải thuật ngẫu nhiên mà có độ phức tạp xác định, nhưng đầu ra của nó có thể không chính xác với xác suất nhất định (thường là nhỏ). Đối với các bài toán đưa ra quyết định, một giải thuật có thể được phân loại *lệch về false* (false-biased) hoặc *lệch về true* (true-biased). Một giải thuật Monte Carlo lệch về *false* thì luôn đưa ra kết quả chính xác khi nó trả về giá trị *false*, một giải thuật lệch về *true* thì luôn chính xác khi nó trả về *true,* và được gọi chung là có lỗi một chiều. Những giải thuật khác không có thiên hướng bị lệch, chúng được gọi là giải thuật có các lỗi hai chiều. Câu trả lời chúng đưa ra (cả *true* và *false*) sẽ là không chính xác, hoặc chính xác với một xác suất có giới hạn.

Định nghĩa 3.2.2 Một giải thuật Las Vegas là một giải thuật ngẫu nhiên luôn đưa ra kết quả đúng, nhưng thời gian chạy là một giá trị ngẫu nhiên ở mỗi lần thực hiện khác nhau.

Trong lĩnh vực tính toán, một giải thuật Las Vegas là một giải thuật ngẫu nhiên, mà nó luôn đưa ra các kết quả chính xác. Nghĩa là, luôn sinh ra kết quả chính xác hoặc nó thông báo cho biết thất bại. Nói cách khác giải thuật Las Vegas không đánh cược với tính chân thực của kết quả, nó đánh cược với các tài nguyên (thời gian, bộ nhớ, bộ xử lí…) được dùng để tính toán. Một ví dụ đơn giản là giải thuật *randomized quicksort*, có chốt được chọn ngẫu nhiên, nhưng kết quả là dãy luôn là được sắp xếp.

Một giải thuật Monte Carlo có thể được chuyển thành giải thuật Las Vegas khi mà tồn tại một thủ tục để xác định output được sinh ra bởi giải thuật chắc chắn là chính xác.

Giải thuật Las Vegas được giới thiệu bởi Babai năm 1979, trong bài toán đẳng cấu đồ thị. Giải thuật Las Vegas có thể được dùng trong các tình huống ở đó số lượng các lời giải khả thi tương đối hạn chế, việc kiểm tra tính đúng đắn của các lời giải được đề cử tương đối dễ dàng trong khi việc tính toán các lời giải lại thực sự phức tạp.

## Các lĩnh vực áp dụng của giải thuật ngẫu nhiên

Giải thuật ngẫu nhiên được ứng dụng trong hầu hết các lĩnh vực của khoa học tính toán. Giải thuật ngẫu nhiên và xác suất đóng vai trò quan trọng trong khoa học máy tính, với những ứng dụng từ tối ưu tổ hợp, lý thuyết số, hình học, lý thuyết đồ thị,.., cho đến máy học, mạng máy tính và các giao thức bảo mật. Cụ thể về một số bài toán có thể giải bằng các giải thuật ngẫu nhiên trong một số lĩnh vực như sau ([5]):

* Lý thuyết số: bài toán kiểm tra số nguyên tố, tìm nghiệm phương trình đồng dư.
* Đại số: bài toán xác định tính đồng nhất của các ma trận và đa thức, hệ thống chứng minh tương tác.
* Lập trình toán: bài toán lập trình tuyến tính, làm tròn số.
* Tìm kiếm và sắp xếp: bài toán kiểm tra tính bằng nhau của hai xâu ký tự có độ dài lớn, so trùng mẩu, bài toán sắp xếp một dãy số (Quicksort).
* Lý thuyết đồ thị: bài toán tìm cây bao trùm nhỏ nhất, đường đi ngắn nhất, lát cắt nhỏ nhất.
* Hình học: bài toán tìm cặp điểm gần nhau nhất.
* Tối ưu tổ hợp: bài toán liệt kê tổ hợp, tính số phần tử của hợp các tập hợp, bài toán gép cặp cực đại.

## Kết luận

Trong Chương 3 này, các khái niệm cơ bản về giải thuật ngẫu nhiên và sự phân loại của chúng đã được giới thiệu. Đó là các giải thuật có thể không xác định kết quả duy nhất tại mỗi bước thực thi và kết quả tính toán mà giải thuật trả về cũng có thể không đúng với một xác suất nào đó. Tuy nhiên, giải thuật ngẫu nhiên thường có độ phức tạp về thời gian nhỏ hơn giải thuật truyền thống, đặc biệt là độ phức tạp trung bình. Vì vậy, giải thuật ngẫu nhiên được ứng dụng nhiều trong thực tế để cải thiện hiệu suất tính toán. Chương 4 kế tiếp, một số giải thuật ngẫu nhiên tiêu biểu sẽ được giới thiệu cho thấy khả năng ứng dụng hiệu quả của chúng để giải các bài toán thực tế.

# Chương 4

**ỨNG DỤNG** [**GIẢI THUẬT**](#_Toc358754416)

**NGẪU NHIÊN**



## Giới thiệu

Chương này trình bày một số giải thuật ngẫu nhiên được ứng dụng để giải một số bài toán tiêu biểu, thường gặp với độ phức tạp về thời gian nhỏ hơn các giải thuật cổ điển. Đầu tiên, Phần 4.2 là giải thuật ngẫu nhiên (Random-Quicksort) để giải bài toán sắp xếp. Kế đến, Phần 4.3 là một giải thuật giải quyết bài toán xác định số nguyên tố. Tiếp theo, Phần 4.4 là giải thuật ngẫu nhiên để giải bài toán tìm cặp điểm gần nhất trong không gian hai chiều. Phần 4.5 giới thiệu giải thuật ngẫu nhiên kiểm tra phép nhân ma trận. Phần 4.6 là giải thuật ngẫu nhiên tìm cây bao trùm nhỏ nhất của đồ thị liên thông có trọng số. Cuối cùng, Phần 4.7 là một số kết luận của chương này.

## Bài toán sắp xếp

Giải thuật Quicksort (cổ điển) áp dụng chiến lược chia để trị để sắp xếp và có độ phức tạp trung bình là O(*n*log2 *n*), tuy nhiên độ phức tạp trong trường hợp xấu nhất của giải thuật này là O(*n*2). Hạn chế này được khắc phục bằng giải thuật Quicksort ngẫu nhiên.

Trước khi trình bày giải thuật Quicksort ngẫu nhiên, chúng tôi giới thiệu giải thuật Quicksort cổ điển ([7]) làm cơ sở để thiết kế giải thuật ngẫu nhiên này như sau.

Quicksort(*A,p,r*)

1. **if** *p<r*
2. q=Partition(*A,p,r*)
3. Quicksort(*A,p,q-1*)
4. Quicksort(*A,q+1,r*)

Partition(*A,p,r*)

1. **for** **to**
2. **if**
3. *i* =*i*+1
4. exchange *A*[*i*] with *A*[*j*]
5. exchange *A*[*i+1*] with *A*[*r*]
6. **return** *i+1*

Giải thuật Quicksort cổ điển luôn luôn chọn phần tử *chốt* (pivot) tại vị trí cố định (trong phiên bản này là phần tử cuối mảng). Như trong ([7]) đã phân tích, độ phức tạp xấu nhất là O(*n*2), trường hợp này xảy ra khi trong tất cả các lời gọi đệ qui, mảng các phần tử cần sắp luôn luôn được phân hoạch thành hai phần có kích thước là *n*-1 và 0.

Trong phiên bản giải thuật ngẫu nhiên [8], chúng ta giả sử các phần tử của mảng là phân biệt và sẽ chọn ngẫu nhiên một phần tử trong *A*[*p…r*] để làm phần tử chốt thay vì dùng *A*[*r*] như ở Quicksort cổ điển. Bằng cách chọn giá trị phần tử chốt ngẫu nhiên, chúng ta có thể kỳ vọng sự phân hoặch của mảng input được cân bằng khá tốt để hầu như không rơi vào trường hợp xấu nhất của giải thuật Quicksort cổ điển. Giải thuật Quicksort ngẫu nhiên [8] được thiết kế bằng cách chọn *ngẫu nhiên* (random) phần tử chốt như sau.

Mã giả

Randomized-Partition(*A,p,r*)

1. *i*= Random(*p,r*)
2. exchange *A*[*r*] with *A*[*i*]
3. **return** Partition(*A,p,r*)

Hàm Randomized-Partition được thay cho hàm Partition trong giải thuật Quicksort

Randomized-Quicksort(*A,p,r*)

1. **if** *p*<*r*
2. *q*= Randomized-Partition(*A,p,r*)
3. Randomized-Quicksort(*A,p,q-1*)
4. Randomized-Quicksort(*A,q+1,r*)

Phân tích độ phức tạp

Để phân tích độ phức tạp của giải thuật Randomized-Quicksort, trước hết chúng ta chứng minh bổ đề về độ phức tạp của Quicksort sau đây,

Bổ đề 4.2.1 Giả sử *X* là số phép so sánh thực hiện ở dòng 4 của hàm Partition trong quá trình thực thi của Quicksort trên một mảng *n* phần tử. Thì thời gian chạy của Quicksort là.

**Chứng minh:** Hàm Partition được gọi nhiều nhất là *n* lần trong Quicksort, chi phí thời gian mỗi lần thực hiện là một hằng số cộng thêm chi phí thời gian của vòng lặp **for***,* thao tác hoán vị dòng 7 và trả về dòng 8. Trong mỗi lần lặp của vòng lặp for câu lệnh ở dòng 4 luôn luôn được thực hiện. Vì vậy, tổng thời gian của Quicksort là *c*(n+*X*) trong đó *c* là một hằng số, *X* là tổng số lần so sánh trong dòng 4.

Để dễ phân tích, chúng ta đổi tên các phần tử của mảng *A* thành *z1,z2,…,zn* với *zi* là phần tử nhỏ nhất thứ *i*. Gọi *Zij=*{*zi, zi+1,…,zj*} là tập các phần tử giữa *zi* và *zj* và bao gồm cả *zi, zj*.

Giả sử *Xij=*I{*zi* được so sánh với *zj*}, là *biến ngẫu nhiên chỉ định* (indicator random variable) kết hợp với sự kiện “*zi* được so sánh với *zj*”. Lưu ý trong Partition mỗi cặp phần tử chỉ được so sánh một lần, khi đó tổng số lần so sánh *X* được xác định

và

Do bất cứ phần tử nào trong *Zij* cũng có thể là phần tử đầu tiên được chọn làm chốt. Bởi vì tập *Zij* có phần tử, và phần tử chốt được chọn ngẫu nhiên và độc lập, xác suất để một phần tử trở thành phần tử đầu tiên được chọn làm chốt là . Do đó, chúng ta có:

Nên (vì )

Bằng cách thay biến () và sử dụng Harmonic Number (xem phần 2.5), chúng ta có thể đánh giá tổng trên như sau:

Do đó, chúng ta có thể kết luận rằng, khi sử dụng Randomized-Partition, thời gian chạy kỳ vọng của Quicksortlà

Trong giải thuật Random-Quicksort, sử dụng Randomized-Partition, phần tử chốt được chọn ngẫu nhiên nên xác suất xẩy ra trường hợp xấu nhất như trong Quicksort là rất nhỏ. Khi đó thời gian chạy của giải thuật Random-Quicksort là thời gian chạy kỳ vọng của giải thuật Quicksort như đã nêu ở trên. Vì vậy, chúng ta có định lý sau.

**Định lý 4.2.1** Thời gian chạy của giải thuật Random-Quicksort là .

Giải thuật Random-Quicksort là loại giải thuật Las Vegas.

## Bài toán xác định số nguyên tố

Bài toán xác định một nguyên dương *n* có là nguyên tố hay không là bài toán không có giải thuật đa thức cho đến năm 2004 ([2]). Giải thuật kinh điển để giải bài toán này dựa trên việc kiểm tra nó có chia hết cho các số từ 2 đến hay không. Giải thuật ngẫu nhiên sẽ giảm thời gian tính toán bằng cách kiểm tra một cách ngẫu nhiên một số lần để xác định tính nguyên tố của *n* dựa trên định lý Fermat ([5],[8]) sau đây.

**Định lý 43.1** Nếu *n* là một số nguyên tố, thì với mọi phải thỏa mãn ( tương đương )

Trong giải thuật ngẫu nhiên sau đây, số lần kiểm tra tính nguyên tố của n được chọn là *m* sao cho với ε đủ bé. Giải thuật sẽ cho kết quả đúng n là số nguyên tố với xác suất đủ lớn.

**Mã giả**

**Input**: Một số nguyên *n* >1, và một tham số *m*, với .

**Output**: *n* là số nguyên tố hay hợp số, với xác suất đúng là

PrimeNumberTest(*n,m*)

1. **if** *n* mod 2 = 0
2. **then** **return** false
3. **else** *k*=(*n*-1)/2
4. **for** *i*=0 **to** *m* **do**
5. *b*= Random(2*,n-*1)
6. *kq*= Power(*a*,*n*) mod *n*
7. **if** *kq≠a*
8. **then return** false
9. **return** true

Trong đó Power là hàm tính . Khi thủ tục trên trả về false thì số *n* là hợp số, ngược lại nếu trả về true thì *n* là số nguyên tố.

**Ví dụ 4.3.1**

1. *n*=12, *m*=3, lần kiểm tra thứ nhất *a*1=9, tiếp tục kiểm tra lần thứ 2. Lần kiểm tra thứ hai a2=2, nên 12 là hợp số.
2. *n*=13, *m*=3, lần kiểm tra thứ nhất *a*1=9,  tiếp tục kiểm tra lần thứ 2. Lần kiểm tra thứ hai *a*1=10,  tiếp tục kiểm tra lần thứ 3. Lần kiểm tra thứ ba *a*1=8, . Cả ba lần kiểm tra đều thất bại nến ta kết luận 13 là số nguyên tố.

**Phân tích độ phức tạp**

Gọi *t* là độ phức tạp của hàm Power(*a,n*) thì độ phức tạp của toàn bộ giải thuật là O(*k.t*). Hàm Power(*a,n*) được viết theo phương pháp chia để trị như sau:

Power(*a,n*)

1. **if** *n* =0
2. **then return** 1;
3. *u* = Power(a, );
4. **if** *n* mod 2 = 1
5. **then** **return** *u \* u \* a*;
6. **else** **return** *u* *\* u*;

Độ phức tạp của hàm Power là

Áp dụng định lý 2.5.6 ta có

Như vậy độ phức tạp của toàn bộ giải thuật là , so với độ phức tạp của giải thuật cổ điển kiểm tra số nguyên tố là . Với *k* =10 thì giải thuật ngẫu nhiên cho kết quả đúng với xác xuất cao, và có độ phức tạp thấp hơn nhiều so với giải thuật cổ điển khi *n* lớn. Giải thuật này là loại giải thuật Monte Carlo.

## Bài toán tìm cặp điểm gần nhất

Bài toán tìm cặp điểm gần nhất có thể giải quyết bằng giải thuật chia để trị cổ điển với độ phức tạp là . Trong phần này, chúng tôi sẽ chỉ ra một giải thuật ngẫu nhiên và độ phức tạp của giải thuật để giải quyết bài toán tìm cặp điểm gần nhất là .

Gọi *x1,x2,….xn* là *n* điểm trên mặt phẳng. Yêu cầu của bài toán là tìm cặp điểm *xi* và *xj* mà khoảng cách giữa *xi* và *xj*là nhỏ nhất trong toàn bộ các cặp điểm. Với phương pháp tiếp cận trực tiếp dùng để giải cần tính *n(n-*1*)/*2 khoảng cách sau đó tìm con số nhỏ nhất trong tất cả khoảng cách này, thời gian thực hiện là O(*n*2).

Ý tưởng chính của giải thuật ngẫu nhiên cho bài toán này là dựa trên sự quan sát sau: Nếu hai điểm *xi* và *x­j* có khoảng cách giữa chúng được dự đoán không là ngắn nhất thì có thể bỏ qua, chỉ quan tâm đến các cặp gần nhau hơn. Với ý tưởng này, giải thuật ngẫu nhiên trước hết phân vùng các điểm thành một vài cụm theo một cách nào đó mà ở mỗi cụm có các điểm gần nhau. Sau đó chúng ta tính toán khoảng cách giữa các điểm trong cùng một cụm.

Xét các điểm trong Hình 4.4.1:

Hình 4.4.1: Phân vùng các điểm

Nếu chúng ta chia vùng cho sáu điểm ở Hình 4.4.1 thành ba cụm *S1=*{*x1,x2*}*, S2=*{*x3,x4*}*, S1=*{*x5,x6*}, sau đó chúng ta chỉ tính toán ba khoảng cách đặt tên là *d(x1,x2)*, *d(x2,x4), d(x5,x6)*, cuối cùng chúng ta tìm giá trị nhỏ nhất trong ba khoảng cách đó. Ngược lại, nếu chúng ta không chia các điểm thành các cụm, số khoảng cách phải tính là khoảng cách.

Dĩ nhiên, mô tả này có thể không thực tế bởi không có gì đảm bảo giải thuật này hoạt động. Trong thực tế, giải thuật này có vẻ giống giải thuật chia để trị. Tuy nhiên, nó có một quá trình phân chia nhưng không có quá trình hợp nhất. Xét Hình 4.4.2

Hình 4.4.2: Phân vùng các điểm hình vuông với cạnh *δ*

Chúng ta có thể thấy cặp điểm gần nhất trong Hình 4.4.2 là {*x1,x3*}. Nhưng, chúng ta đã chia hai điểm này vào hai cụm khác nhau. Vì thế, nếu chúng ta chia không gian này thành các hình vuông với độ dài cạnh *δ* không nhỏ hơn khoảng cách ngắn nhất, thì sau khi chúng ta tính hết các khoảng cách ở từng vùng, chúng ta phải nhân đôi độ dài cạnh và tạo ra các hình vuông lớn hơn. Các trường hợp mở rộng điển hình được minh họa ở Hình 4.4.3

Hình 4.4.3 Các dạng mở rộng khi nhân đôi *δ*

Cuối cùng, câu hỏi quan trọng là tìm ra mạng lưới kích thước *δ* thích hợp. Nếu *δ* quá lớn, hình vuông ban đầu sẽ quá lớn và số lượng khoảng cách cần tính cũng lớn. Trong thực tế, nếu *δ* quá lớn, hầu như không có việc chia ra và bài toán của chúng ta trở về bài toán ban đầu. Mặt khác, *δ* cũng không thể quá nhỏ bởi vì nó không thể nhỏ hơn khoảng cách ngắn nhất. Trong giải thuật ngẫu nhiên, chúng ta chọn lựa ngẫu nhiên một tập con các điểm và tìm khoảng cách ngắn nhắn trong tập đó. Khoảng cách ngắn nhất này sẽ trở thành *δ*.

Giải thuật

**Input:** Một tập *S* gồm *n* điểm *x1, x2,…, xn* với .

**Output:** Cặp điểm gần nhất trong tập R

Bước 1: Chọn ngẫu nhiên một tập điểm *S1=*{*xi1, xi2,…, xim*}với . Tìm cặp điểm gần nhất trong tập *S1* và gán giá trị khoảng cách giữa cặp điểm đó cho *δ*.

Bước 2: Xây dựng tập hình vuông *T* có độ dài cạnh là *δ*.

Bước 3: Xây dựng bốn tập hình vuông *T1, T2, T3 và T­­­4* bắt nguồn từ *T* bằng cách nhân đôi kích thước cạnh hình vuông thành 2*δ*.

Bước 4: Đối với mỗi *Ti*, xác định các tập điểm là giao giữa tập điểm S và các hình vuông của , sao cho

Bước 5: Đối với mỗi *x­p,xq Sj(i)*, tính *d(xp, xq)*. Đặt *xa* và *xb* là cặp điểm có khoảng cách ngắn nhất trong tất cả các cặp điểm. Trả về *xa* và *xb* là cặp điểm gần nhất.

Ví dụ 4.4.1 Cho tập điểm S có 27 điểm, được mô tả trong Hình 4.4.4. Bước 1, chúng ta chọn ngẫu nhiên điểm, được đánh dấu là *x1, x2,…, x9*. Nhìn vào *Hình 4.4.4*, ta có thể thấy rằng cặp điểm gần nhau nhất là *(x1, x2)*. Gán khoảng cách giữa *x1* và *x2* cho *δ* và xây dựng tập hình vuông *T*, ở ví dụ này sẽ gồm 36 hình vuông, theo yêu cầu của bước 2. Sau đó xây dựng các tập hình vuông *T1, T2, T3, T4* như yêu cầu ở bước 3:

.

.

.

.

Hình 4.4.4: Minh họa ví dụ 4.4.1

Tổng số khoảng cách giữa các cặp điểm cần phải tính lúc này là:

.

.

.

.

Trong cặp điểm trên, cặp điểm gần nhất được tìm thấy tại ô vuông .

Phân tích độ phức tạp giải thuật

Bước 1 trong giải thuật là tìm cặp điểm gần nhất trong điểm. Khoảng các ngắn nhất này có thể được tìm bằng cách gọi đệ quy giải thuật. Nghĩa là, chọn ngẫu nhiên từ điểm ban đầu, sau đó dùng chiến lược trực tiếp để tìm cặp điểm gần nhất trong điểm, với số phép tính khoảng cách cần thực hiện là .

Bước 2 và Bước 3 có thể hoàn thành với O*(n)*. Trong Bước 4, có thể xác định nhanh hình vuông mà một điểm thuộc về nó bằng kỹ thuật băm (hashing). Từ đó Bước 4 có thể thực hiện với độ phức tạp O*(n)*.

Số phép tính khoảng cách ở Bước 5 là O(*n*) với xác suất , trong đó *c* là một hằng số, *e* là số Napier ([2]).

Do đó, chúng ta có thể kết luận rằng thời gian chạy của giải thuật ngẫu nhiên tìm cặp điểm gần nhất là O*(n)* với xác suất . Như vậy, giải thuật ngẫu nhiên tốt hơn nhiều so giải thuật cổ điển tốt nhất (chia để trị) với độ phức tạp . Giải thuật ngẫu nhiên tìm cặp điểm gần nhau nhất là loại giải thuật Las Vegas.

## Bài toán kiểm tra phép nhân ma trận

Giải thuật Freivalds (đặt tên theo Rusins Freivalds) ([8]) là một giải thuật xác suất ngẫu nhiên được sử dụng kiểm tra phép nhân ma trận. Cho ba ma trận *A*, *B* và *C* kích thước , kiểm tra liệu . Một giải thuật thông thường có thể tính sau đó so sánh kết quả với *C*. Tuy nhiên, ngay cả khi dùng một giải thuật nhân ma trận tốt nhất, thời gian chạy cũng không thể nhỏ hơn . Giải thuật Freivalds sử dụng sự ngẫu nhiên để giảm thời gian xuống với một xác suất cao.

Giải thuật

**Input**: Matrix *A* ∈ *Rm* × *p*, *B* ∈ *Rp* × *n*, *C* ∈ *Rm* × *n* and *k* ∈N

**Output**: True if *C* = *A* · *B*; false if *C* ≠ *A* · *B*

MatrixEqualityTest(*A*, *B*, *C*, *k*)

1 *for* *i*=1 to *k* do

2 Choose *r*=(*r*1,...,*rn*) ∈ {0,1}*n* at random

3 Compute *C* · *r* and *A* · (*B* · *r*)

4 **if** *C* · *r* ≠ *A* · (*B* · *r*)

5 t**hen** **return** false

6 *i* = *i* + 1

7 **return** true

**Định lý 4.5.1** Giải thuật chạy cho kết quả chính xác (correct) với xác suất 1-(1/2)*k*.

Thật vậy nếu *A*.*B* ≠ *C* thì *D* =*A*.*B*-*C* ≠ 0. Không mất tính tổng quát, giả sử *d*11 ≠ 0. Mặt khác, Pr(*A*.*B*.*r* = *C*.*r*) = Pr(*A*.*B*-*C*).*r* = 0) = Pr(*D*.*r* = 0). Nếu *D*.*r* = 0 thì phần tử đầu tiên của *D*.*r* bằng 0. Nghĩa là Σ*j*=1, *n* *d*1*jrj* = 0. Vì *d*11 ≠ 0 nên ta có

Nếu cố định tất cả *rj* trừ *r*1, thì đẳng thức trên thỏa nhiều nhất là hai lựa chọn *r*1∈{0, 1}. Vì vậy Pr(*Abr* = *Cr*)≤1/2.

Vòng lặp thực hiện *k* lần. Nếu *C*=*A*.*B* thì giải thuật luôn luôn cho kết quả đúng. Nếu *C*≠*A.B*, giải thuật cho kết quả đúng với xác suất để ít nhất là 1-(1/2)*k*. Vậy định lý được chứng minh.

Phân tích độ phức tạp giải thuật

Chi phí dòng 2 là O(*n*), dòng O(*n*2), từ đó thời gian chạy của giải thuật là O(*kn*2).

Giải thuật ngẫu nhiên kiểm tra phép nhân của ma trận là giải thuật Monte Carlo.

**Ví dụ 4.5.1** Giả sử có một yêu cầu xác định đẳng thức ma trận

.

Chọn ngẫu nhiên một ma trận *r* , tính toán theo dòng 3 của giải thuật. Ta có

Ta được kết quả là ma trận 0. Tuy nhiên, nếu trong lần thử thứ hai với *r* kết quả được tính toán là

Kết quả khác 0, nên (phù hợp với tế ).

Có tất cả bốn ma trận 0-1, hai trong số đó cho kết quả vectơ 0 (), nên giải thuật trả lời với xác suất là . Nghĩa là, giải thuật trả lời kết quả sai với xác suất nhiều nhất là ¼. Giải thuật này là loại giải thuật Monte Carlo.

## Bài toán tìm cây bao trùm nhỏ nhất của đồ thị

Trong phần này, chúng tôi sẽ giới thiệu một giải thuật ngẫu nhiên tìm cây bao trùm nhỏ nhất với độ phức tạp là Giải thuật này tốt hơn rất nhiều so với các giải thuật truyền thống tìm cây bao trùm nhỏ nhất (giải thuật Kruskal, độ phức tạp hay giải thuật Prim, độ phức tạp ) ([2]).

Giải thuật này dựa trên khái niệm Boruvka step, được đề xuất bởi Boruvka vào năm 1926 ([2]). Bổ đề sau minh họa ý tưởng bên trong Boruvka step:

Bổ đề 4.6.1 Giả sử *G* = (*V*, *E*) là đồ thị vô hướng liên thông có trọng số, *V*1 và *V*2 là các tập con khác rỗng của *V* sao cho *V*1∪*V*2 = *V* và *V*1∩*V*2 *=*∅, cạnh (u,v) là cạnh có trọng số nhỏ nhất với một điểm kết thúc trong *V*1và một điểm kết thúc trong *V*2. Thì, (u,v) là một trong các cạnh trong cây bao trùm nhỏ nhất của G.



Hình 4.6.1: Một đồ thị vô hướng liên thông có trọng số

Bổ đề 4.6.1 có thể được phát biểu dưới dạng khác như sau: trong một đồ thị G vô hướng liên thông có trọng số, với mỗi đỉnh u, trong số các cạnh liên thuộc u, nếu cạnh (u,v) có trọng số nhỏ nhất thì (u,v) phải là cạnh nằm trong cây bao trùm nhỏ nhất của G.

**Ví dụ 4.6.1** Xem đồ thị ở Hình 4.6.1, với đỉnh 2, trong số tất cả cạnh liên thuộc với đỉnh 2, cạnh (2,4) có trọng số nhỏ nhất. Vì vậy, cạnh (2,4) phải nằm trong cây bao trùm nhỏ nhất của G. Tương tự, cạnh (5,6) cũng phải nằm trong cây bao trùm nhỏ nhất.

**Ví dụ 4.6.2** Áp dụng bổ đề 4.6.1, các cạnh phải thuộc cây bao trùm nhỏ nhất của đồ thị trong Hình 4.6.1, được chỉ ra trong Hình 4.6.2. Các cạnh này gom thành các nhóm tạo nên các thành phần liên thông của đồ thị ban đầu.



Hình 4.6.2: Các cạnh được chọn của đồ thị theo bổ đề Boruka

Do cây bao trùm nhỏ nhất của một đồ thị là liên thông chứa tất cả các đỉnh của đồ thị và không có chu trình, nên để tìm các cạnh còn lại của cây bao trùm nhỏ nhất ta thu tất cả các đỉnh của một thành phần liên thông về chỉ một đỉnh duy nhất như Hình 4.6.3.



Hình 4.6.3: Đồ thị hợp nhất các thành phần liên thông về một nút

Sau khi loại bỏ các cạnh bội có trọng số lớn và cạnh khuyên chúng ta được kết quả như Hình 4.6.4.



Hình 4.6.4: Kết quả sau khi áp dụng Boruvka lần thứ nhất

Từ đồ thị trong Hình 4.6.4, áp dụng chọn các cạnh theo bồ đề Boruvka lần thứ hai ta có kết quả như Hình 4.6.5.



Hình 4.6.5: Các cạnh được chọn theo bổ đề Boruvka lần hai

Sau khi thu các nút ở mỗi thành phần liên thông về một nút đơn, chúng ta có hai đỉnh như Hình 4.6.6.



Hình 4.6.6: Đồ thị kết quả hợp nhất các thành phần liên thông lần thứ hai

Từ đồ thị trong Hình 4.6.6, loại bỏ cạnh bội có trọng số lớn (cây bao trùm không chứa chu trình) và cạnh khuyên, chúng ta có đồ thị chỉ còn hai đỉnh và một cạnh như Hình 4.6.7.



Hình 4.6.7: Kết quả sau khi áp dụng Boruvka lần thứ hai

Quá trình áp dụng bổ đề Boruvka hoàn thành, cây bao trùm nhỏ nhất được xây dựng (suy dẫn từ kết quả trong Hình 4.6.7) như Hình 4.6.8.



Hình 4.6.8 Cây bao trùm nhỏ nhất của đồ thị

Giải thuật Boruvka để tìm cây bao trùm nhỏ nhất áp dụng một số bước dựa trên bổ đề Boruvka một cách đệ quy cho đến khi đồ thị kết quả chỉ còn một cạnh. Gọi đầu vào là đồ thị *G* = (*V*, *E*) và đầu ra là đồ thị *G*’=(*V’*, *E*’). Mỗi bước Boruvka (gồm hai giai đoạn) được phát biểu như sau:

1. Với mỗi đỉnh *u*, tìm cạnh (*u*,*v*) với trọng số nhỏ nhất liên thuộc với nó. Tìm tất cả cách thành phần liên thông bằng cánh đánh dấu cạnh.
2. Thu nhỏ mỗi thành phần liên thông được xác định bởi cạnh đánh dấu thành một đỉnh duy nhất. Gọi đồ thị kết quả là *G*’=(*V*’,*E*’). Loại bỏ các cạnh bội và cạnh khuyên.

Độ phức tạp của một bước Boruvka là O(*n*+*m*) với *n* là số đỉnh và *m* là số cạnh. Vì *G* liên thông nên  *m*>*n*. Do đó O(*n*+*m*)=O(*m*). Vì mỗi thành phần liên thông được xác định bởi một cạnh đánh dấu chứa ích nhất 2 đỉnh, sau khi thi hành mỗi bước Boruvka số cạnh của đồ thị thì nhỏ hơn một nữa so với ban đầu. Do đó, tổng các bước Boruvka là O(log2 *n*). Từ đó suy ra độ phức tạp của giải thuật Boruvka là O(*m* log2 *n*).

Để sử dụng các bước Boruvka hiệu quả,chúng ta phải sử dụng khái niệm cạnh nặng, cạnh nhẹ của đồ thị theo một *rừng bao trùm nhỏ nhất* (minimum spanning forest) như trong định nghĩa sau.

**Định nghĩa 4.6.1** Giảsử *Gs* là một đồ thị con của *G,* *F* là rừng bao trùm nhỏ nhất của *Gs* và *wF*(*x*, *y*) là trọng số cạnh lớn nhất trong các cạnh trên đường đi từ *x* đến *y* trong *F* (*wF*(*x*, *y*)=∞, nếu không có được đi từ *x* đến *y*). Cạnh (*x*, *y*) được gọi là cạnh nặng hay cạnh nhẹ (*F*-light) theo *F* nếu *w*(*x*, *y*)> *wF*(*x*, *y*) hay *w*(*x*, *y*)≤ *wF*(*x*, *y*)

**Ví dụ 4.6.3** Xét rừng bao trùm nhỏ nhất *F* của đồ thị *G* trong Hình 4.6.9(coi *Gs*≡*G*). Rừng *F* bao gồm các cạnh vẽ đậm liền nét. Trọng số lớn nhất của các cạnh từ đỉnh 4→3→6→5 là 3. Theo Định nghĩa 4.6.1, suy ra cạnh (4, 5) có trọng số bằng 7>3 là cạnh nặng theo *F*.



Hình 4.6.9: Cạnh nặng của đồ thị theo rừng bao trùm nhỏ nhất

Chúng ta dễ dàng nhận thấy rằng các cạnh nặng không thể thuộc bất kỳ cây bao trùm nhỏ nhất nào như bổ đề sau.

Bổ đề 4.6.2 Gọi Gs là một đồ thị con của G, F là một rừng bao trùm nhỏ nhất của Gs . cạnh nặng trong G theo F không thể là cạnh nằm trong cây bao trùm nhỏ nhất.

Cuối cùng, để có thể phát biểu được các bước của giải thuật ngẫu nhiên tìm cây bao trùm nhỏ nhất của đồ thị G, cần tìm các cạnh nhẹ của đồ thị dựa trên bổ đề sau.

Bổ đề 4.6.3 Gọi H là đồ thị con có được từ đồ thị G bằng cách chọn mỗi cạnh của *G* một cách độc lập với xác xuất p, và gọi F là rừng bao trùm nhỏ nhất của H. Số trung bình của các cạnh nhẹ trong G nhiều nhất có thể đạt được là n/p, với n là số đỉnh của G.

Bây giờ giải thuật tìm cây bao trùm nhỏ nhất được thiết kế gồm ba bước như dưới đây,

**Giải thuật** ([2])

**Input:** Một đồ thị vô hướng liên thông có trọng số G

**Output**: Cây bao trùm nhỏ nhất của G

Bước 1: Thực hiện các bước Boruvka ba lần. Kết quả đạt được là đồ thị G1=(V1, E1). Nếu G1 chứa một đỉnh, trả về bộ các cạnh đánh dấu ở bước 1 và kết thúc.

Bước 2: Phát sinh một đồ thị con H của G1 bằng cách chọn mỗi cạnh độc lập với xác xuất ½. Áp dụng đệ quy giải thuật cho H để tạo một rừng bao trùm nhỏ nhất F của H. Đồ thị con G2*=*(V2, E2) đạt được bằng cách xóa toàn bộ cạnh nặng theo F trong G1.

Bước 3: Áp dụng đệ quy giải thuật cho G2.

Độ phức tạp của giải thuật

Gọi *T*(|*V*|,|*E*|) là thời gian chạy kỳ vọng của giải thuật ứng với đồ thị Mỗi lần thực thi bước 1 chi phí là . Sau bước 1, ta có và . Với bước 2, thời gian để tính *H* là , thời gian để tính *F* là T. Thời gian cần thiết để loại bỏ những cạnh nặng là . Sử dụng Bổ đề , ta có giá trị kỳ vọng cho lớn nhất là . Từ đó, ta có thời gian kỳ vọng cần để thực hiện bước 3 là T. Nếu và . Chúng ta có hệ thức truy hồi sau:

, với *c* là hằng số.

Từ đây suy ra ([2])

Như vậy, thời gian chạy kỳ vọng của giải thuật là

Mã giả

1. Randomized-MST (G)
2. G1 = BoruvkaStep(G); // các bước Boruvka lần thứ I
3. **if** G1 has edges
4. **if** G1 has one edg //nếu G1 chỉ có một cạnh thì thêm vào đồ thị kết quả (solvedGraph)
5. Add this edg to solvedGraph
6. **else**
7. G2 = BoruvkaStep(G1); // các bước Boruvka lần thứ II
8. **if** G2 has edges
9. **if** G2 has one edg //nếu G2 chỉ có một cạnh thì thêm vào đồ thị kết quả
10. Add this edg to solvedGraph
11. **else**
12. G3 = BoruvkaStep(G2); // các bước Boruvka lần thứ III
13. **if** G3 has edges
14. **if** G3 has one edg //nếu G3 chỉ có một cạnh thì thêm vào đồ thị kết quả
15. Add this edg to solvedGraph // kết thúc bước 1
16. **else**

// phát sinh một đồ thị con H bằng cách chọn các cạnh từ đồ thị G3 với xác xuất 1/2

1. Obtain a subgraph H of G3 by selecting each edge with probability ½
2. F=Randomized-MST(H)
3. **for** each edg in G3 and not in F
4. **if** this edg is not a F-heavy edge
5. Add this edg to G4 // kết thúc bước 2
6. Randomized-MST(G4); // bước 3
7. **return** solvedGraph

## Kết luận

Trong chương này, chúng tôi đã giới thiệu và phân tích độ phức tạp của một số giải thuật ngẫu nhiên để giải một số bài toán thuộc các lĩnh vực khác nhau như lý thuyết số, lý thuyết đồ thị, hình học tính toán, lý thuyết giải thuật. Các giải thuật ngẫu nhiên được giới thiệu đều hiệu quả hơn các giải thuật truyền thống tương ứng. Đặc biệt giải thuật ngẫu nhiên giải bài toán tìm cặp điểm gần nhau nhất và tìm cây bao trùm nhỏ nhất đều có độ phức tạp tuyến tính tốt hơn nhiều so với các giải thuật truyền thống có độ phức tạp O(*n*log2 *n*). Điều đó chứng tỏ, tính hiệu quả của các giải thuật ngẫu nhiên so với giải thuật truyền thống. Đó cũng là động lực để các giải thuật ngẫu nhiên được tiếp tục nghiên cứu để giải các bài toán thực tế. Trong Chương 5 tiếp theo, chúng tôi sẽ hiện thực các giải thuật đã được giới thiệu ở các phần trên thành một hệ thống để giải các bài toán đã nêu một cách nhanh chóng và hiệu quả.

# Chương 5

# HIỆN THỰC CÁC ỨNG DỤNG

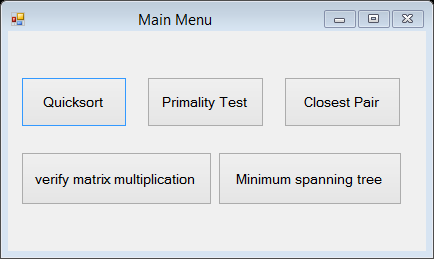
# GIẢI THUẬT NGẪU NHIÊN



## Giới thiệu

Chương này trình bày phần hiện thực của một số giải thuật ngẫu nhiên đã đề cập ở chương 4, đây là chứng minh mang tính thuyết phục cho việc khẳng định giải thuật ngẫu nhiên chạy nhanh hơn giải thuật cổ điển. Đầu tiên, Phần 5.2 là phần hiện thực giải thuật ngẫu nhiên (Random-Quicksort) để giải bài toán sắp xếp. Kế đến, Phần 5.3 là một giải thuật giải quyết bài toán xác định số nguyên tố. Tiếp theo, Phần 5.4 là giải thuật ngẫu nhiên để giải bài toán tìm cặp điểm gần nhất trong không gian hai chiều. Phần 5.5 trình bày phần hiện thực cho giải thuật ngẫu nhiên kiểm tra phép nhân ma trận. Phần 5.6 là giải thuật ngẫu nhiên tìm cây bao trùm nhỏ nhất của đồ thị liên thông có trọng số. Cuối cùng, Phần 5.7 là một số kết luận của chương này.

Trước khi vào phần hiện thực chi tiết từng giải thuật, chúng ta hãy xem qua giao diện chính của toàn bộ phần hiện thực như Hình 5.1.1. Khi muốn chọn giải thuật để thử nghiệm chúng ta chỉ cần click chọn nút tương ứng của giải thuật đó.



Hình 5.1.1 Giao diện của toàn bộ phần hiện thực

## Bài toán sắp xếp

Cấu trúc dữ liệu

Trong chương trình này, danh sách các số cần sắp xếp được tổ chức theo kiểu mảng. Vì giải thuật ngẫu nhiên được dùng để khắc phục hạn chế trong trường hợp xấu nhất của giải thuật *quicksort* nên mảng số sẽ được sắp xếp giảm dần.

Hàm chính hiện thực giải thuật

Giải thuật *quicksort* ngẫu nhiên chỉ khác với giải thuật *quicksort* ở chỗ có thêm hàm randomizedPartition(): chọn một phần tử ngẫu nhiên trong mảng và đưa nó về cuối mảng để làm phần tử chốt. Hàm randomizedQuicksort() sẽ gọi hàm randomizedPartition() để chọn chốt, sau đó thực hiện hàm Parttition() để tiến hành xếp các phần tử nhỏ hơn chốt về bên trái và các phần tử lớn hơn chốt về bên phải, cuối cùng tiếp tục gọi đệ qui trên hai mảnh đó cho đến khi toàn mảng được sắp xếp.

|  |
| --- |
| private void randomizedQuicksort(int[] a, int p, int r)  {  if (p < r)  {  int q=randomizedPartition(a,p,r);  randomizedQuicksort(a, p, q - 1);  randomizedQuicksort(a, q + 1, r);  }  }  //Chọn chốt  private int randomizedPartition(int[] a, int p, int r)  {  //Chọn chỉ mục i ngẫu nhiên trong mảng.  int i = random.Next(p, r+1);  //Hoán vị trí của phần tử thứ i với phần tử cuối mảng.  permutation(ref a[i],ref a[r]);  return Parttition(a, p, r);  }  // Sắp xếp mảng: phần tử nhỏ hơn bằng chốt về bên trái chốt và phần tử lớn hơn chốt về bên phải chốt  private int Parttition(int[] a, int p, int r)  {  int x = a[r];  int i = p - 1;  for (int j = p; j < r; j++)  {  if (a[j] <= x)  {  i++;  permutation(ref a[i],ref a[j]);  }  }  permutation(ref a[i + 1], ref a[r]);  return i + 1;  } |

Giao diện và cách sử dụng chương trình

Hình 5.2.1 Giao diện chính của giải thuật Quicksort

Giao diện được chia thành 2 tab: *Algorithm* (1)-hiển thị chương trình sắp xếp sử dụng giải thuật Quicksort và Quicksort ngẫu nhiên và *Running time chart* (2)-hiển thị đồ thị thời gian chạy.

Nút *Randomly Generate* (3) có chức năng phát sinh dãy số ngẫu nhiên và được hiển thị tại textbox (4) , kích thước của mảng sẽ do người dùng nhập tại hộp thoại xuất hiện sau khi bấm nút *Randomly Generate*.

Nút *Quicksort* (5) và nút *Randomized Quicksort* (6) có chức năng thực thi giải thuật Quicksort và giải thuật Quicksort ngẫu nhiên trên dãy số ở textbox (4) và kết quả quá trình sắp xếp được hiển thị lần lượt tại textbox(7) và textbox(8). Thời gian chạy của giải thuật cũng được hiển thị tại vùng (9) và (10).

Trình tự thao tác trên chương trình: đầu tiên, người dùng phải nhập một dãy số vào textbox (4) hoặc sử dụng nút (3) để phát sinh dãy các số ngẫu nhiên (dãy số này đã được xếp giảm dần để có được đầu vào thuộc trường hợp xấu nhất của giải thuật Quicksort cổ điển). Kế đó, người dùng chọn lựa giải thuật để tiến hành sắp xếp bằng hai nút (5) và (6). Kết quả trả về được hiển thị tương ứng tại textbox (7) và (8) cùng thời gian chạy tương ứng ở vùng (9) và (10). 

Hình 5.2.2 Kết quả chạy chương trình sắp xếp 60 số có giá trị giảm dần

So sánh giữa giải thuật Quicksort và giải thuật Quicksort ngẫu nhiên

Ở Phần 4.2, chúng tôi đã phân tích độ phức tạp theo thời gian của giải thuật Quicksort ngẫu nhiên là *O(n log n)* và độ phức tạp của giải thuật Quicksort trong trường hợp xấu nhất là *O(n)*. Sau đây, chúng tôi xin trình bày kết quả khảo sát thời gian chạy thực (đơn vị: microsecond) của hai giải thuật.

Bảng 5.2‑1 So sánh thời gian chạy thực của Quicksort và Quicksort ngẫu nhiên

|  |  |  |
| --- | --- | --- |
| N | Quicksort | Quicksort ngẫu nhiên |
| 20 | 6 | 6 |
| 30 | 14 | 8 |
| 40 | 17 | 10 |
| 50 | 35 | 18 |
| 60 | 48 | 16 |

Sử dụng số liệu từ bảng trên để vẽ đồ thị minh họa, ta được đồ thị như trong Hình 5.2.3. 

Hình 5.2.3 Đồ thị thời gian chạy thực của hai giải thuật sắp xếp

## Bài toán xác định số nguyên tố

Cấu trúc dữ liệu

Xem xét Chương 4, chúng ta thấy phần mã giả của giải thuật này khá là đơn giản. Tuy nhiên có một vấn đề lớn trong phần hiên thực của giải thuật này, đó là , hàm tính lũy thừa sẽ trả về một kết quả là một số rất lớn (khi *n* lớn), mà trong C# không có một kiểu dữ liệu nào có thể mô tả được. Vì vậy chúng tôi phải kế thừa một *lớp số nguyên lớn* (BigInteger class) ([10]) , để làm việc với các số này.

Khi sữ dụng lớp BigInteger chúng ta có thể thực hiện các phép toán +, -, \*, /, %, >>, <<, ==, !=, >, <, >=, <=, &, |, ^, ++, -- và ~ trên số nguyên lớn.

Hàm chính hiện thực giải thuật

Ở đây chúng tôi sẽ trình bày phần code của hàm xác định số nguyên tố theo định lý Fermat.

|  |
| --- |
| public bool FermatLittleTest(int confidence)  {  BigInteger thisVal;  if(thisVal.dataLength == 1)  {  // đối với số nguyên tố nhỏ  if(thisVal.data[0] == 0 || thisVal.data[0] == 1)  return false;  else if(thisVal.data[0] == 2 || thisVal.data[0] == 3)  return true;  }  if((thisVal.data[0] & 0x1) == 0) // số chẵn  return false;  int bits = thisVal.bitCount();  BigInteger a = new BigInteger();  BigInteger p\_sub1 = thisVal - (new BigInteger(1));  Random rand = new Random();  for(int round = 0; round < confidence; round++)  {  bool done = false;  while(!done) // phát sinh số a, a < n  {  int testBits = 0;  // đảm bảo a ít nhất 2 bit  while(testBits < 2)  testBits = (int)(rand.NextDouble() \* bits);  a.genRandomBits(testBits, rand);  int byteLen = a.dataLength;  // đảm bảo a khác 0  if(byteLen > 1 || (byteLen == 1 && a.data[0] != 1))  done = true;  }  // tính a^(p-1) mod p  BigInteger expResult = a.modPow(p\_sub1, thisVal);  int resultLen = expResult.dataLength;  // nếu a^(p-1) mod p != 1 thì n là hợp số  if(resultLen > 1 || (resultLen == 1 && expResult.data[0] != 1))  {  return false;  }  }  return true;  } |

Giao diện và cách sử dụng chương trình

Giao diện chính của phần hiện thực cho giải thuật xác định số nguyên tố



Hình 5.3.1 Giao diện chính của giải thuật xác định số nguyên tố

Đầu tiên chúng ta sẽ nhập *Input number* vào textbox (1), sau đó click nút *Classical* (2)(để xác định số nguyên tố theo phương pháp cổ điển)hoặc *Fermat* (3)(để xác định số nguyên tố theo giải thuật ngẫu nhiên). Nếu kiểm tra theo giải thuật ngẫu nhiên chương trình sẽ yêu cầu chúng ta nhập tham số *m* ( *m* là số lần kiểm tra)

Tiếp theo chương trình sẽ thực hiện việc kiểm tra và hiển thị kết quả (gồm N là số nguyên tố hay hợp số - ở vị trí (4); thời gian chạy theo microsecond- vị trí (5) và vẽ đồ thị ở vị trí (6) ). Khi chúng ta muốn xóa hết dữ liệu thời gian chạy và đổ thị để vẽ lại đồ thị khác thì click nút *Clear all* (7).

Một số dữ liệu thử

Bảng 5.3‑1 dữ liệu thử cho giải thuật xác định số nguyên tố.

|  |  |  |
| --- | --- | --- |
| Số N | Kết quả khi thực hiện bằng giải thuật cổ điển | Kết quả khi thực hiện bằng giải thuật ngẫu nhiên |
| 100 | Hợp số | Hợp số |
| 68718952447 | Nguyên tố | Nguyên tố |
| 68718952449 | Hợp số | Hợp số |
| 274876858367 | Nguyên tố | Nguyên tố |
| 4398042316799 | Nguyên tố | Nguyên tố |
| 4398042316801 | Hợp số | Hợp số |

So sánh giữa giải thuật truyền thống và giải thuật Fermat để kiểm tra số nguyên tố.

Như đã trình bày ở Phần 4.3 độ phức tạp của giải thuật Fermat là , độ phức tạp của giải thuật cổ điển kiểm tra số nguyên tố là . Bây giờ chúng ta sẽ so sánh độ phức tạp dựa vào thời gian chạy của giải thuật.

Bảng 5.3‑2 Bảng thời gian chạy của giải thuật xác định số nguyên tố (đơn vị tính microsecond)

|  |  |  |
| --- | --- | --- |
| Số N | Kết quả khi thực hiện bằng giải thuật cổ điển | Kết quả khi thực hiện bằng giải thuật ngẫu nhiên |
| 68718952447 | 11613 | 5878 |
| 274876858367 | 23080 | 2436 |
| 4398042316799 | 92118 | 3716 |

Trong bảng trên chúng ta thấy rõ ràng giải thuật ngẫu nhiên chạy nhanh hơn giải thuật cổ điển. Điều này càng thể hiện rõ hơn ở đồ thị thời gian chạy của hai giải thuật như Hình 5.3.2 bên dưới. Trong hình, Đường màu xanh là đường biểu diễn thời gian chạy của giải thuật cổ điển, đường màu đỏ là đường biểu diễn thời gian chạy của giải thuật ngẫu nhiên.



Hình 5.3.2 Đồ thị so sánh thời gian chạy của hai giải thuật xác định số nguyên tố

## Bài toán tìm cặp điểm gần nhất

Cấu trúc dữ liệu

Trong bài toán tìm cặp điểm gần nhất, các điểm trong đầu vào và đầu ra được tổ chức theo kiểu mảng. Các ô vuông chứa các điểm cũng được tổ chức theo kiểu mảng.

Hàm chính hiện thực giải thuật

Chúng tôi xin trình bày hai hàm chính của giải thuật này.

Hàm RandomizedAl(): có chức năng hiện thực Bước 1, Bước 2 và Bước 5 của giải thuật được đề cập ở Phần 4.4

|  |
| --- |
| bool recursive = true,inrecurisve=false;  private IPoint[] RandomizedAl(IPoint[] P, out double distantRA)  {  distantRA = double.MaxValue;  //Chọn ngẫu nhiên n^2/3 điểm từ input  //Tạo thành mảng S[]  int rp = (int)Math.Round(Math.Pow(P.Length, (2 \* 1.0 / 3)), 0);  IPoint[] S = new IPoint[rp];  for (int i = 0; i < S.Length; i++)  {  bool isSame = false;  int r = random.Next(0, P.Length - 1);  for (int j = 0; j < i; j++)  {  if (S[j] == P[r])  isSame = true;  }  if (!isSame)  S[i] = P[r];  else i--;  }  //Tính khoảng cách ngắn nhất trong mảng S[] bằng cách đệ qui gọi hàm RandomizedAl 1 lần  //Đặt làm denta  double denta=double.MaxValue;  //Trong lần đệ qui đó dùng chiến lược trực tiếp để tìm khoảng cách ngắn nhất  if (inrecurisve)  {  inrecurisve = false;  IPoint[]tmp= bruteForce(S, ref denta);  }  if (recursive)  {  recursive = false;  inrecurisve = true;  IPoint[] resultS = RandomizedAl(S, out denta);  }  //Xác định số lượng ô vuông có cạnh bằng denta:  //Xác định hoành độ lớn nhất và tung độ lớn nhất trong các điểm  //Chọn giá trị lớn và đem chia cho denta  int maxX, maxY;  int SquareInRow = 0;  FindMaxXY(P, out maxX, out maxY);  int maxValue = maxX >= maxY ? maxX : maxY;  SquareInRow =(int) Math.Round(maxValue / denta);  if (SquareInRow \* denta <= maxValue)  SquareInRow++;  //Tạo bộ T gồm các ô vuông có cạnh= denta  Square[] T = new Square[SquareInRow\*SquareInRow];  for (int i = 0; i < SquareInRow\*SquareInRow; i++)  T[i] = new Square();  for (int i = 0; i < P.Length; i++)  {  double a =P[i].\_X / denta;  double b = P[i].\_Y / denta;  //Xác định ô vuông chứa điểm đang xét  T[(int)a \* SquareInRow + (int)b].lstPoints.Add(P[i]);  }  //Tạo các bộ hình vuông mở rộng  List<Square> T1, T2;  List<Square> T3, T4;  createAllT(T, SquareInRow, out T1, out T2, out T3, out T4);  IPoint[] resultP = new IPoint[2];  //Tìm khoảng cách ngắn nhất ở từng ô vuông ở tập T1, T2, T3, T4 bằng chiến lược trực tiếp  foreach (var item in T1)  {  if (item.lstPoints.Count > 1)  {  IPoint[] resultTmp = bruteForce(item.lstPoints.ToArray(), ref distantRA);  if (resultTmp != null)  {  resultP = resultTmp;  }  }  }  foreach (var item in T2)  {  if (item.lstPoints.Count > 1)  {  IPoint[] resultTmp = bruteForce(item.lstPoints.ToArray(), ref distantRA);  if (resultTmp != null)  resultP = resultTmp;  }  }  foreach (var item in T3)  {  if (item.lstPoints.Count > 1)  {  IPoint[] resultTmp = bruteForce(item.lstPoints.ToArray(), ref distantRA);  if (resultTmp != null)  resultP = resultTmp;  }  }  foreach (var item in T4)  {  if (item.lstPoints.Count > 1)  {  IPoint[] resultTmp = bruteForce(item.lstPoints.ToArray(), ref distantRA);  if (resultTmp != null)  resultP = resultTmp;  }  }  return resultP;  } |

Hàm createAllT() là hàm hiện thực Bước 3 và Bước 4 của giải thuật được đề cập ở Phần 4.4 .

|  |
| --- |
| private void createAllT(Square[] T, int SquareRow, out List<Square> T1, out List<Square> T2, out List<Square> T3, out List<Square> T4)  {  T1 = new List<Square>();  T2 = new List<Square>();  T3 = new List<Square>();  T4 = new List<Square>();  bool isAddT4 = true;  List<IPoint> tmp = new List<IPoint>();  int h = 0;  while(h<SquareRow\*(SquareRow-1))  {  //h thuộc hàng chẵn thực hiện trên T1 và T2  if ((h / SquareRow) % 2 == 0)  {  if (h % SquareRow == 0)//h ở ô vuông đầu mỗi hàng  {  List<IPoint> l1 = AddListP(T, SquareRow, h);  List<IPoint> l2 = AddListP(T, SquareRow, h + 1);  Square sG1 = new Square(l1, l2);  T1.Add(sG1);  tmp = l2;  h++;  continue;  }  if (h + 1 < SquareRow \* (h / SquareRow + 1))  {  List<IPoint> l2 = AddListP(T, SquareRow, h + 1);  Square s = new Square(tmp, l2);  //h lẻ thì thêm vào T2  if (h % 2 != 0 && s.lstPoints.Count != 0)  T2.Add(s);  else if (h % 2 == 0 && s.lstPoints.Count != 0)  T1.Add(s);  tmp = l2;  }  }  //h thuộc hàng lẻ thực hiện trên T3 T4  else  {  if (h % SquareRow == 0)  {  List<IPoint> l1 = AddListP(T, SquareRow, h);  List<IPoint> l2 = AddListP(T, SquareRow, h + 1);  Square sG1 = new Square(l1, l2);  T3.Add(sG1);  tmp = l2;  h++;  isAddT4 = true;  continue;  }  if (h + 1 < SquareRow \* (h / SquareRow + 1))  {  List<IPoint> l = AddListP(T, SquareRow, h + 1);  Square s = new Square(tmp, l);  //Thay phiên thêm cho T3 và T4  if (isAddT4)  {  isAddT4 = false;  if (s.lstPoints.Count != 0)  T4.Add(s);  }  else  {  isAddT4 = true;  if (s.lstPoints.Count != 0)  T3.Add(s);  }  tmp = l;  }  }  h++;  }  } |

Giao diện và cách sử dụng chương trình



Hình 5.4.1 Giao diện chính của giải thuật tìm cặp điểm gần nhất

Giao diện được chia thành 2 tab: *Algorithm* (1)-hiển thị chương trình tìm cặp điểm gần nhất sử dụng giải thuật trực tiếp, giải thuật chia để trị và giải thuật ngẫu nhiên và *Running time chart* (2)-hiển thị đồ thị thời gian chạy.

Nút *Randomly Generate* (3) có chức năng phát sinh mảng các điểm có tọa độ ngẫu nhiên và được hiển thị tại textbox (4) ,sau khi click chuột vào nút này sẽ có một hộp thoại yêu cầu chúng ta nhập kích thước của mảng.

Nút *Brute Force* (5), nút *Divide and Conquer* (6) và nút *Randomized Algorithm* (7) lần lượt có chức năng thực thi giải thuật trực tiếp, giải thuật chia để trị và giải thuật ngẫu nhiên tìm cặp điểm gần nhất trong mảng điểm ở textbox (4). Vùng (8) hiển thị thông tin kết quả trả về sau khi thực hiện giải thuật gồm: cặp điểm gần nhất, khoảng cách giữa chúng và thời gian thực hiện giải thuật.

Trình tự thao tác trên chương trình: đầu tiên, người dùng phải nhập một dãy điểm vào textbox (4) hoặc sử dụng nút *Randomly Generate* để phát sinh dãy các số ngẫu nhiên. Kế đó, người dùng chọn lựa giải thuật để tiến hành tìm kiếm bằng ba nút *Brute Force,* *Divide and Conquer, Randomized Algorithm*. Kết quả sẽ được trả về tại vùng (8). 

Hình 5.4.2 Thực hiện giải thuật ngẫu nhiên tìm cặp điểm gần nhất trong 50 điểm

So sánh giữa giải thuật trực tiếp, giải thuật chia để trị và giải thuật ngẫu nhiên

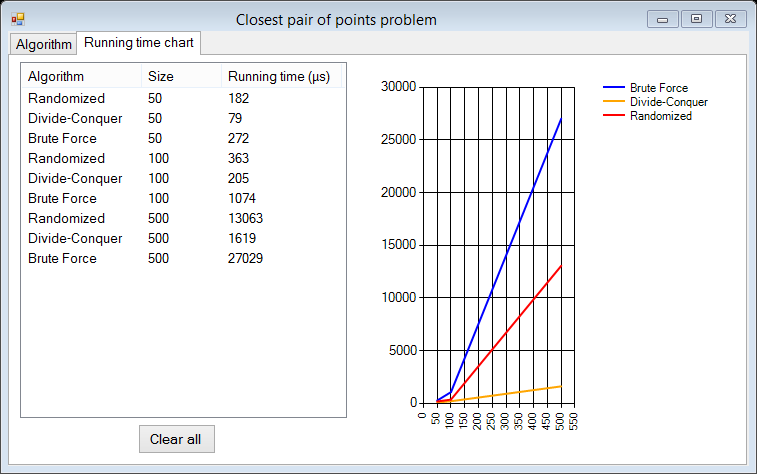
Nhắc lại độ phức tạp thuật toán theo thời gian đã được phân tích ở Phần 4.4, ta có độ phức tạp của giải thuật ngẫu nhiên là *O(n)*. Bên cạnh đó, ta có độ phức tạp của giải thuật trực tiếp là *O(n2)* và độ phức tạp của giải thuật chia để trị là *O(n log n)* ([8]).

Xét về thời gian chạy thực của ba giải thuật trong chương trình ta có bảng số liệu sau:

Bảng 5.4‑1 So sánh thời gian chạy thực của ba giải thuật trực tiếp, chia để trị và ngẫu nhiên

|  |  |  |  |
| --- | --- | --- | --- |
| N | Giải thuật trực tiếp | Giải thuật chia để trị | Giải thuật ngẫu nhiên |
| 50 | 272 | 79 | 182 |
| 100 | 1074 | 205 | 363 |
| 500 | 27029 | 1619 | 13063 |

Sử dụng số liệu từ Bảng 5.4-1 để vẽ đồ thị minh họa, ta được đồ thị như trong Hình 5.4.3.



Hình 5.4.3 Đồ thị thời gian chạy thực của ba giải thuật tìm cặp điểm gần nhất

## Bài toán kiểm tra phép nhân ma trận

Cấu trúc dữ liệu

Trong chương trình kiểm tra phép nhân ma trận, các ma trận được lưu trữ trong các mảng hai chiều. Chúng tôi cũng xin lưu ý rằng các ma trận được sử dụng ở đầu vàolà ma trận vuông.

Hàm chính hiện thực giải thuật

Hàm MatrixEqualityTest() là hàm chính của giải thuật Freivalds với các tham số truyền vào là ma trận *A,B,C* kích thước và số lần thử *k*.

|  |
| --- |
| private bool MatrixEqualityTest(int[,] MTA, int[,] MTB, int[,] MTC,ref int k)  {  strVector = "";  int MTsize=MTA.GetLength(0);  for (int j = 0; j < k; j++)  {  //Tạo vector ngẫu nhiên r  int[,] vectorR = new int[MTsize, 1];  for (int i = 0; i < MTsize; i++)  {  vectorR[i, 0] = random.Next(0, 2);  }  //mỗi lần chạy in vector ngẫu nhiên  if(!startBF)  strVector +="random vector in trial "+(j+1).ToString()+":\r\n" +PrintMatrix(vectorR) + "\r\n";  int[,] MTResultL, MTResultR;  //A x (B x r)  MTResultL = MultiplyMT(MTB, vectorR, MTsize, MTsize, 1);  MTResultL = MultiplyMT(MTA, MTResultL, MTsize, MTsize, 1);  //C x r  MTResultR = MultiplyMT(MTC, vectorR, MTsize, MTsize, 1);  //A x (B x r)!C x r  if(!CompareMT(MTResultR, MTResultL))  return false;  }  return true;  } |

Giao diện và cách sử dụng chương trình



Hình 5.5.1 Giao diện chính của giải thuật kiểm tra phép nhân ma trận

Giao diện được chia thành 2 tab: *Algorithm* (1)-hiển thị chương trình kiểm tra phép nhân ma trận bằng giải thuật trực tiếp và giải thuật Freivalds và *Running time chart* (2)-hiển thị đồ thị thời gian chạy.

Nút *Random input Generator* (4) có chức năng phát sinh ba ma trận ngẫu nhiên *A, B* và *C* được hiển thị lần lượt tại textbox (5), (6) và (7). Kích thước của ba ma trận bắt buộc phải nhập vào textbox (3) trước khi bấm nút này.

Nút *Brute force* (5) và nút *Freivalds* (6) có chức năng thực thi giải thuật trực tiếp và giải thuật Freivalds kiểm tra phép nhân ma trận . Kết quả trả về sau khi thực hiện giải thuật trực tiếp gồm kết quả phép nhân ma trận *A*x*B* tại textbox (10) và xác nhận liệu *A*x*B* có bằng *C* cùng thời gian chạy hiển thị tại vùng (12). Kết quả trả về sau khi thực hiện giải thuật Freivalds gồm: các vectơ ngẫu nhiên được dùng trong các lần thử tại textbox(11) và hiển thị số lần thử, xác nhận liệu *A*x*B* có bằng *C* và thời gian chạy được hiển thị tại vùng (12).

Trình tự thao tác trên chương trình: đầu tiên, người dùng phải nhập kích thước của ba ma trận vào textbox (3). Sau đó có thể dùng nút *Random input Generator* để sinh ma trận ngẫu nhiên hoặc nhập vào tại textbox (5),(6) và (7), lưu ý rằng kích thước của ba ma trận phải bằng với số được nhập tại textbox (3). Cuối cùng chọn giải thuật để thực hiện việc kiểm tra phép nhân ma trận bằng nút (8) và (9). 

Hình 5.5.2 Kiểm tra phép nhân ma trận với kích thước ma trận bằng 10

So sánh giữa giải thuật truyền thống và giải thuật Freivalds

Xét về thời gian chạy thực của giải thuật trực tiếp và giải thuật Freivalds trong chương trình ta có bảng số liệu sau:

Bảng 5.5‑1 So sánh thời gian chạy thực của giải thuật trực tiếp và giải thuật Freivalds

|  |  |  |
| --- | --- | --- |
| N | Giải thuật trực tiếp | Giải thuật Freivalds |
| 10 | 18 | 22 |
| 20 | 215 | 37 |
| 40 | 1502 | 490 |

Sử dụng số liệu từ Bảng 5.5-1 để vẽ đồ thị minh họa, ta được đồ thị như trong Hình 5.5.3

Hình 5.5.3 Đồ thị thời gian chạy thực của hai giải thuật kiểm tra ma trận

## Giải thuật ngẫu nhiên tìm cây bao trùm nhỏ nhất của đồ thị

Cấu trúc dữ liệu

Chúng tôi mô tả một đồ thị là danh sách các cạnh, mỗi cạnh bao gồm 2 *đỉnh* (vertex), *trọng số của cạnh* (cost), vị trí hiển thị giá trị cost trên form (point).

Mỗi vertex bao gồm các thuộc tính *tên đỉnh* (name), *vùng* (rank), *nút gốc* (root), *vị trí hiển thị đỉnh trên form* (point). (Rank và root dùng để xác định các cạnh đang xét có cùng một cây hay không?).

Hàm chính hiện thực giải thuật

Ở giải thuật này chúng tôi trình bày hàm thực hiện Boruvka step.

|  |
| --- |
| private IList<Edge> Boruvka(IList<Edge> graph, IList<Edge> solvedGraph\_dauVao, out IList<Edge> solvedGraph)  {  solvedGraph = new List<Edge>();// solvedGraph là những cạnh gạch liền trong hình 4.6.2  IList<Edge> solvedGraph2 = new List<Edge>();// đồ thị hình 4.6.3  IList<Edge> dsCanhCL = new List<Edge>();// danh sách các cạnh còn lại sau khi thực hiện Boruvka Step, đồ thị hình 4.6.4  foreach (Edge ed in solvedGraph\_dauVao) // với mỗi cạnh từ solvedGraph\_dauVao add vào solvedGraph\_daura  solvedGraph.Add(ed);  #region lấy tất cả các đỉnh của đồ thị  IList<int> tenDinh=new List<int>();  foreach (Edge ed in graph)  {  bool kt1 = false;    foreach (int i in tenDinh)  {  if ( ed.V1.Name2 == i)  {  kt1 = true;  break;  }    }  if (kt1 == false)  tenDinh.Add(ed.V1.Name2);  bool kt2 = false;  if (ed.V1.Name2 != ed.V2.Name2)  {  foreach (int i in tenDinh)  {  if (ed.V2.Name2 == i)  {  kt2 = true;  break;  }  }  if (kt2 == false)  tenDinh.Add(ed.V2.Name2);  }  }  #endregion  // Nếu số cạnh bằng số đỉnh -1  if (tenDinh.Count - 1 == graph.Count)  return graph;  #region tính với mỗi đỉnh tìm cạnh có trọng số nhỏ nhất trả về solvedGraph  for (int i = 0; i < tenDinh.Count; i++)// với mỗi đỉnh tìm cạnh có trọng số nhỏ nhất  {  int minCost=int.MaxValue;  Edge minCostEdge = graph[0]; // cạnh có trọng số nhỏ nhất liên thuộc với đỉnh i  foreach (Edge ed in graph) // Tìm cạnh có trọng số nhỏ nhất  {    if (ed.V1.Name2 == tenDinh[i] || ed.V2.Name2 == tenDinh[i]) // nếu cạnh này là liên thuộc với đỉnh i  {  if (minCost > ed.Cost)  {  minCost = ed.Cost;  minCostEdge = ed;  }  }  }    // join lại, để biết rằng các cạnh được chọn nằm liên thông với nhau  Vertex root1 = minCostEdge.V1.GetRoot();  Vertex root2 = minCostEdge.V2.GetRoot();  if (root1.Name != root2.Name)  {  Vertex.Join(root1, root2);  solvedGraph.Add(minCostEdge);  }  }  #endregion  #region tim danh sach cac cạnh còn lại, trả về dsCanhCL  foreach (Edge ed in graph)  {  bool kiemtra = false;  foreach (Edge ed2 in solvedGraph)  {  if (ed == ed2)  {  kiemtra = true;  break;  }  }  if (kiemtra == false) // nếu cạnh này không có trong SolvedGraph  {  dsCanhCL.Add(ed);  }  }  #endregion  #region tính đồ thị hình thành sau khi bỏ cạnh trùng và khuyên, trả về solvedgraph2  // trong số các cạnh còn lại nếu một đỉnh thuộc root này, một thuộc root kia. tìm cạnh có trọng số nhỏ nhất  foreach (Edge ed in dsCanhCL)  {  // them lenh get root  Vertex root1 = ed.V1.GetRoot();  Vertex root2 = ed.V2.GetRoot();  if (root1.Name != root2.Name)  {  Edge minCostDScanh = ed;  foreach (Edge ed2 in dsCanhCL)  {  Vertex root3 = ed2.V1.GetRoot();  Vertex root4 = ed2.V2.GetRoot();  if (((root3.Name == root1.Name) && (root4.Name == root2.Name)) || ((root3.Name == root2.Name) && (root4.Name == root1.Name)))  {  if (minCostDScanh.Cost > ed2.Cost)  minCostDScanh = ed2;  }  }  // kiểm tra solvedgraph2 không có cạnh này mới thêm cạnh này vào  bool ketqua = false;  foreach (Edge ed3 in solvedGraph2)  {    if (ed3 == minCostDScanh)  {  ketqua = true;  break;  }    }  if(ketqua==false)  solvedGraph2.Add(minCostDScanh);  }  }  #endregion  return solvedGraph2; // đồ thị hình 4.6.5  } |

Giao diện và cách sử dụng chương trình

Giao diện chính của phần hiện thực cho giải thuật tìm cây bao trùm nhỏ nhất:



Hình 5.6.1 Giao diện chính của giải thuật tìm cây bao trùm nhỏ nhất

Giao diện được chia thành 2 tab: *Algorithm* (1) - hiển thị giải thuật và *Running time chart* (2) - hiển thị đồ thị thời gian chạy.

Vùng (3) là khu vực vẽ đồ thị mới, hiển thị đồ thị đã có lên đồng thời cũng là nơi hiển thị kết quả (cây bao trùm nhỏ nhất cần tìm).

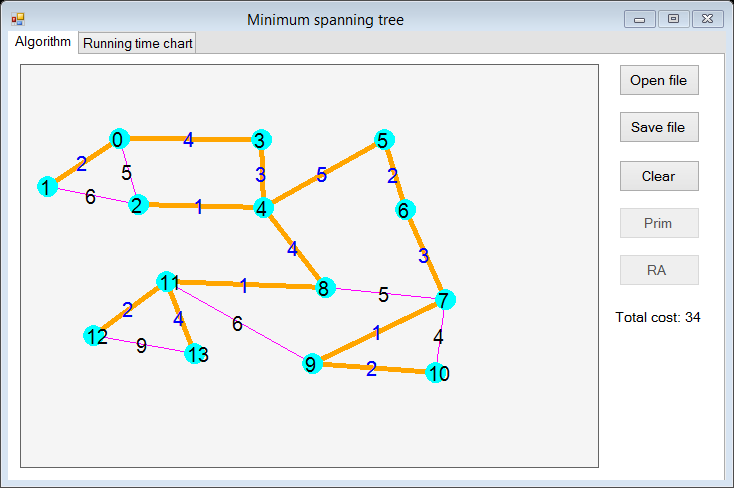
Khi muốn vẽ một đồ thị mới,chúng ta click chuột trên vùng (3), mỗi lần click ta sẽ nhận được một đỉnh của đồ thị tại vị trí click. Sau khi đã vẽ tất cả các đỉnh, chúng ta vẽ các cạnh bằng cách giữ phím Ctrl click chọn lần lượt 2 đỉnh thuộc cạnh cần vẽ. Một hộp thoại hiện lên ta nhập trọng số của cạnh này vào.

Sau khi vẽ xong đồ thị chúng ta có thể sử dụng nút *Save file* (5) để lưu lại đồ thị đang có, để tiện sử dụng cho lần sau. Khi click *Save file,* một cửa số hiện ra ta chọn vị trí lưu, và đặt tên file, click *Save.*

Chúng ta có thể mở đồ thị đã vẽ sẵn (đã được lưu trước đó) bằng cách click nút *Open file* (4)*.* Một của sổ hiện ra, ta chọn file cần mở, rồi click *Open.* Đồ thị đã lưu sẽ hiện lại lên vùng (3), chúng ta có thể sử dụng để chạy giải thuật hay vẽ thêm cạnh vào đồ thị này.

Nút *Clear* (6) cho phép ta xóa đồ thị đang có trên vùng (3).

Nút *Prim* (7) và *RA* (8), đây là 2 nút quan trọng nhất , nút *Prim* cho phép ta tìm cây bao trùm nhỏ nhất theo giải thuật Prim, nút *RA* cho phép ta tìm cây bao trùm nhỏ nhất theo giải thuật ngẫu nhiên đã trình bày ở chương 4. Sau khi thực hiện giải thuât chương trình sẽ hiển thị cây bao trùm nhỏ nhất tìm được, *tổng trọng số* (total cost) của cây bao trùm nhỏ nhất ở tab *Algorithm* (như Hình 5.6.2) và vẽ đồ thị thời gian chạy ở tab *Running time chart*.

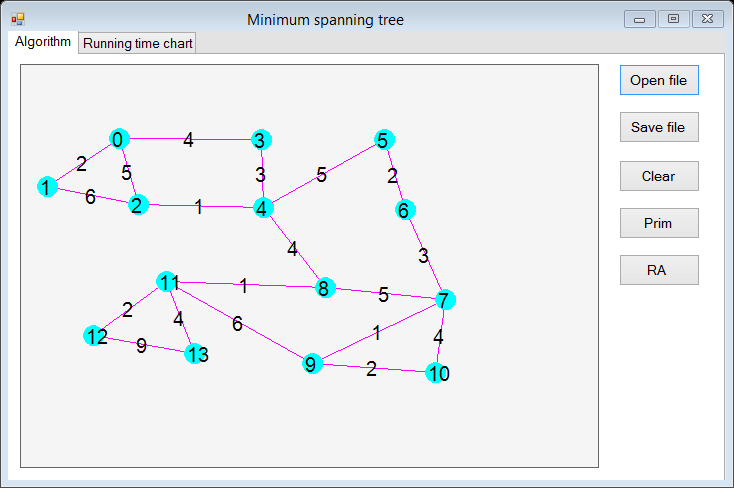


Hình 5.6.2 Kết quả thu được khi thực hiện giải thuật tìm cây bao trùm nhỏ nhất

Dữ liệu thử

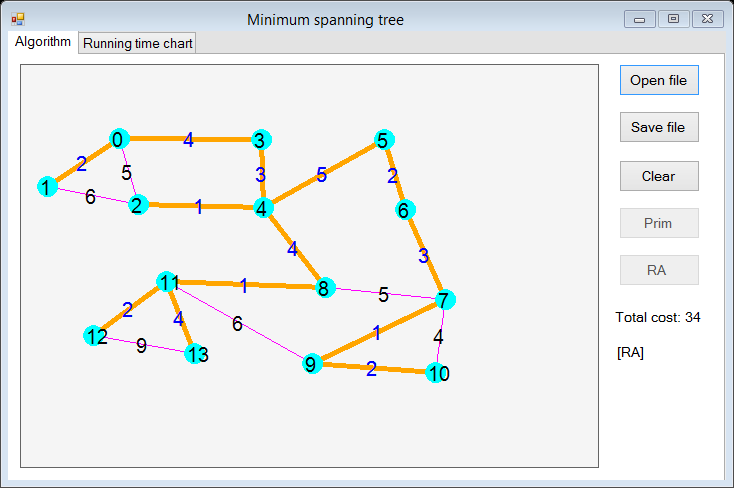
Đồ thị 1

Đồ thị đưa vào (Hình 5.6.3)



Hình 5.6.3 Đồ thị 1

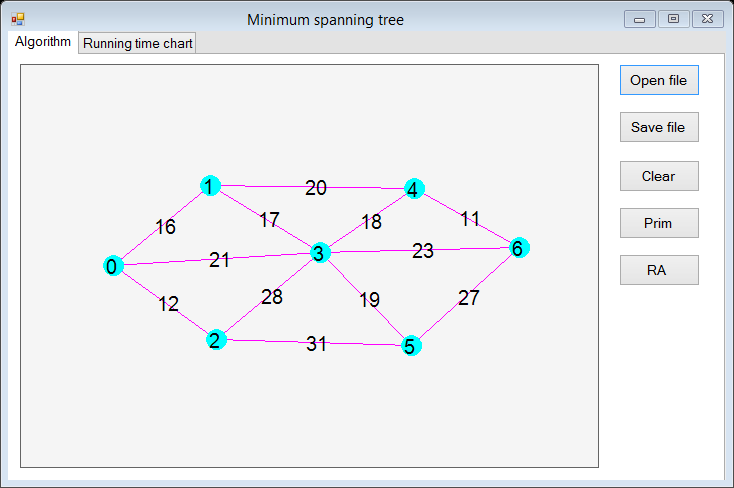
Cây bao trùm nhỏ nhất tìm được (bằng giải thuật ngẫu nhiên) (Hình 5.6.5)

******

Hình 5.6.4 Cây bao trùm nhỏ nhất của đồ thị 1

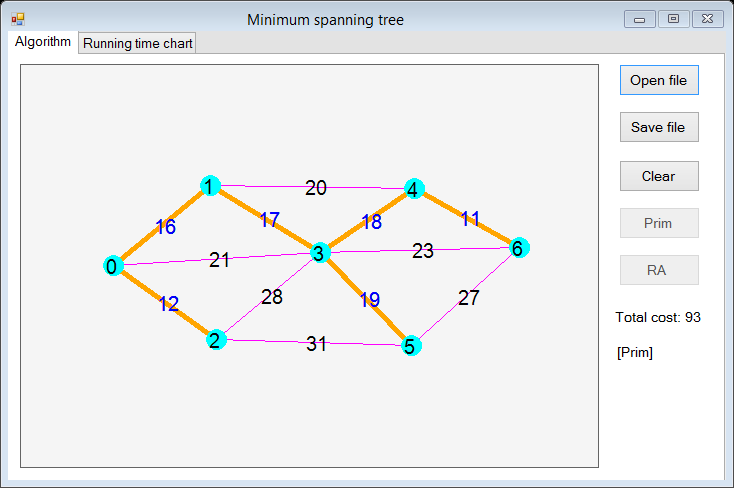
Đồ thị 2

Đồ thị đưa vào (Hình 5.6.6)

******

Hình 5.6.5 Đồ thị 2

Cây bao trùm nhỏ nhất tìm được (bằng giải thuật ngẫu nhiên) (Hình 5.6.7)



Hình 5.6.6 Cây bao trùm nhỏ nhất của đồ thị 2

Đồ thị 3

Đồ thị đưa vào (Hình 5.6.9)



Hình 5.6.7 Đồ thị 3

Cây bao trùm nhỏ nhất tìm được (bằng giải thuật ngẫu nhiên) (Hình 5.6.10)



Hình 5.6.8 Cây bao trùm nhỏ nhất của đồ thị 3

So sánh giữa giải thuật ngẫu nhiên và giải thuật Prim

Như chúng tôi đã trình bày ở Phần 4.6, giải thuật Prim,có độ phức tạp ), còn giải thuật ngẫu nhiên tìm cây bao trùm nhỏ nhất có độ phức tạp là Còn dưới đây là thời gian chạy thực tế của 2 giải thuật.

Bảng 5.6‑1 Bảng so sánh thời gian chạy (đơn vị tính: microsecond) của giải thuật ngẫu nhiên và giải thuật Prim

|  |  |  |
| --- | --- | --- |
| Đồ thị | Thời gian chạy của giải thuật Prim | Thời gian chạy của giải thuật ngẫu nhiên |
| Đồ thị 1 | 116 | 124 |
| Đồ thị 2 | 145 | 65 |
| Đồ thị 3 | 45240 | 1092 |

Trong bảng trên chúng ta thấy rõ ràng giải thuật ngẫu nhiên chạy nhanh hơn giải thuật cổ điển. Điều này càng thể hiện rõ hơn ở đồ thị thời gian chạy của hai giải thuật tìm cây bao trùm nhỏ nhất như Hình 5.6.12 bên dưới. Trong hình, Đường màu xanh là đường biểu diễn thời gian chạy của giải thuật Prim, đường màu đỏ là đường biểu diễn thời gian chạy của giải thuật ngẫu nhiên.



Hình 5.6.9 Đồ thị so sánh thời gian chạy của hai giải thuật tìm cây bao trùm nhỏ nhất

## Kết luận

Trong chương này, chúng tôi đã hiện thực thành công các giải thuật ngẫu nhiên đã đề cập ở chương 4. Một điều đáng phấn khởi là các giải thuật này đều chạy nhanh hơn giải thuật cổ điển. Điều này là cơ sở, là động lực để các giải thuật ngẫu nhiên được đưa vào thực tiễn. Tiếp theo chương này là phần tổng kết ở chương 6.

# Chương 6

# TỔNG KẾT VÀ ĐỀ NGHỊ



## Tổng kết

## Đề nghị

# TÀI LIỆU THAM KHẢO

[1] Karp Richard M. *An introduction to randomized algorithms*. Discrete Applied Mathematics, 34, 1991, 165-201.

[2] Lee R.C.T, Tseng S.S, Chang R.C, Tsai Y.T. *Introduction to The design and Analysis of Algorithms-A Strategic Approach*. McGraw-Hill Education, 2005.

[3] Levitin A. *Introduction to The design and Analysis of Algorithms*. Addison-Wesley, 2012.

[4] Mitzenmacher M., Upfal E. *Probability and Computing Randomized Algorithms and Probabilistic Analysis*. Cambridge university press, 2005

[5] Motwani R., Prabhakar Raghavan P. *Randomized algorithms*. Cambridge university press, 1995.

[6] Rosen K.H. *Discrete Mathematics*, Prentice Hall Inc., 2012

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald D. Rivest, clifford Stein. *Introduction to Algorithms*. McGraw-Hill Book Company, 2009.

[8] en.wikipedia.org/

[9] vi.wikipedia.org/

[10] http://www.codeproject.com/Articles/2728/C-BigInteger-Class