NGUYỄN HỮU ĐIỂN

# THUẬT TOÁN VÀ LẬP TRÌNH

## QUYỂN 6
## THUẬT TOÁN QUAY LUI

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

NGUYỄN HỮU ĐIỂN

# THUẬT TOÁN VÀ LẬP TRÌNH

# QUYỂN 6
# THUẬT TOÁN QUAY LUI

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA HÀ NỘI

# LỜI NÓI ĐẦU

Những năm trước khi lập trình VieTeX tôi toàn dùng C/C++ thu thập tài liệu nhiều nhưng không có thời gian để viết lại. Nay muốn viết lại thì sức khỏe không ổn định. Tôi đã cố gắng gom lại thành các tập lập trình theo chủ đề. Nội dung mỗi thuật toán bắt đầu từ lý thuyết đến lập trình bằng C/C++.

Cuốn sách viết ra không dành riêng cho các bạn học tin học, mà các bạ học toán, thầy cô giáo, các bạn thích tìm hiểu về thuật toán. Cũng như tôi bắt đầu có biết gì về lập trình đâu, tự học và chăm chỉ là thành công thôi. Tôi dùng trình biên dịch Dev-C++: https://www.bloodshed.net/
Hiện nay Dev-C++ cải tiến rất nhiều và chạy tốt với môi trường unicode. Những ví dụ trong tài liệu các bạn chép thẳng vào soạn thảo và biên dịch không cần cấu hình trình biên dịch.

Tôi đã làm các quyển sách:

1. Thuật toán và số học.
2. Thuật toán và dữ liệu.
3. Thuật toán xắp xếp
4. Thuật toán tìm kiếm
5. Thuật toán đồ thị,
6. Thuật toán quay lui
7. Thuật toán chia để trị
8. Thuật toán động
9. Thuật toán tham
10. Thuật toán nén
11. Một số đề thi Olympic Tin học.

Cuốn sách dành cho học sinh phổ thông yêu toán, học sinh khá giỏi môn toán, các thầy cô giáo, sinh viên đại học ngành toán, ngành tin học và những người yêu thích Toán - Tin. Trong biên soạn không thể tránh khỏi sai sót và nhầm lẫn mong bạn đọc cho ý kiến.

Hà Nội, ngày 25 tháng 2 năm 2022

**Nguyễn Hữu Điển**

# NHỮNG KÝ HIỆU

Trong cuốn sách này ta dùng những kí hiệu với các ý nghĩa xác định trong bảng dưới đây:

| | |
|---|---|
| $\mathbb{N}$ | tập hợp số tự nhiên |
| $\mathbb{N}^*$ | tập hợp số tự nhiên khác 0 |
| $\mathbb{Z}$ | tập hợp số nguyên |
| $\mathbb{Q}$ | tập hợp số hữu tỉ |
| $\mathbb{R}$ | tập hợp số thực |
| $\mathbb{C}$ | tập hợp số phức |
| $\equiv$ | dấu đồng dư |
| $\infty$ | dương vô cùng (tương đương với $+\infty$) |
| $-\infty$ | âm vô cùng |
| $\emptyset$ | tập hợp rỗng |
| $C_m^k$ | tổ hợp chập $k$ của $m$ phần tử |
| $\vdots$ | phép chia hết |
| $\not\vdots$ | không chia hết |
| $UCLN$ | ước số chung lớn nhất |
| $BCNN$ | bội số chung nhỏ nhất |
| deg | bậc của đa thức |
| IMO | International Mathematics Olympiad |
| APMO | Asian Pacific Mathematics Olympiad |

# NỘI DUNG

# DANH SÁCH CÁC HÌNH

# DANH SÁCH CÁC BẢNG

# Danh sách chương trình

# CHƯƠNG 6
# THUẬT TOÁN QUAY LUI



Người ta thường chấp nhận rằng một vấn đề được coi là "được giải quyết tốt" nếu độ phức tạp của thuật toán tương ứng là tuyến tính hoặc đa thức (mức độ thấp). Tuy nhiên, chúng ta thường gặp các tác vụ quan trọng mà không có thuật toán nào "nhanh". Các bài toán này sẽ là chủ đề của chương này, và các kỹ thuật khác nhau để giải quyết chúng sẽ không chỉ được thảo luận ở đây mà còn ở các chương sau.

# 6.1. Phân loại bài toán

Khi chúng ta nói về các lớp bài toán, chúng ta hiểu việc nhóm các bài toán theo mức độ phức tạp của chúng - các bài toán có độ phức tạp tương tự được xếp vào cùng một lớp. Có nhiều tiêu chí khác nhau để phân loại bài toán theo độ "khó" của chúng. Chúng ta sẽ xem xét việc phân loại bài toán trên hai tiêu chí đánh giá quan trọng nhất: thời gian và bộ nhớ.

## 6.1.1. Phức tạp về thời gian

Hiện tại, chúng ta sẽ giới hạn bản thân chỉ xem xét các bài toán có thể xác minh (tức là những bài toán yêu cầu câu trả lời có hoặc không).

***Bài toán NP***

*NP* (từ thời gian đa thức không xác định) là lớp của các *bài toán có thể tính toán thời gian đa thức*. Một bài toán thuộc về lớp *NP*, nếu với độ phức tạp tính toán đa thức có thể kiểm tra xem một ứng viên có thực sự là một giải pháp hay không, mà không cần quan tâm sẽ mất bao lâu để tìm thấy một ứng viên như vậy. Nói một cách đơn, bài toán *NP* là những bài toán mà khi chúng ta cố gắng "tìm" ra ứng viên phù hợp cho một giải pháp, chúng ta có thể dễ dàng kiểm tra nó (xem ví dụ 2 của 6.1.4).

***Bài toán P***

Đây là những bài toán về khả năng tính toán mà có một giải pháp với *độ phức tạp của đa thức* (chữ viết tắt *P* bắt nguồn từ từ *polynom* trong tiếng Anh). Ngược lại, lớp này (là một lớp con của lớp *NP*) khá rộng: khi xem xét các bài toán thực tế, bậc của hàm đa thức là vô cùng quan trọng. Lớp này cũng chứa các thuật toán mà hàm của nó có tiệm cận tăng thấp hơn hàm tuyến tính. Đó là hàm logarit (thường các tác vụ này được xem xét trong một lớp con riêng biệt - Bài toán log), hàm Ackermann nghịch đảo và các hàm khác. (xem ví dụ 1 của 6.1.4).

***Bài toán lũy thừa***

Đây là những bài toán có thể được giải quyết với độ phức tạp theo lũy thừa. Lớp này chứa hai phần trước, nhưng không toàn diện

- có những bài toán mà các thuật toán tốt nhất có độ phức tạp cao hơn lũy thừa.

### 6.1.2. Độ phức tạp tính toán theo bộ nhớ

Trong cách phân loại này, bộ nhớ được sử dụng như một chức năng có kích thước của dữ liệu đầu vào là rất quan trọng. Chúng ta không quan tâm đến độ phức tạp của thuật toán để giải quyết chúng. Lớp này có thể được chia thành nhiều lớp con: *P-space* (các tác vụ yêu cầu bộ nhớ đa thức), *exp-space* (bộ nhớ hàm mũ) và các lớp khác. (xem ví dụ 3 của 6.1.4).

Rõ ràng là độ phức tạp tính toán của thuật toán luôn ít nhất bằng độ phức tạp của bộ nhớ. Thật vậy, không thể chiếm bộ nhớ có kích thước $\Theta(n)$ trong thời gian nhỏ hơn $\Theta(n)$ [Manev-1996].

### 6.1.3. Bài toán không thể giải được

Có những bài toán thuật toán mà [Manev-1996] có thể chứng minh rằng chúng không thể giải được, bất kể chúng ta có bao nhiêu thời gian và bộ nhớ. Tiếp theo chúng ta sẽ đưa ra các ví dụ cụ thể (xem ví dụ 4 của 6.1.4).

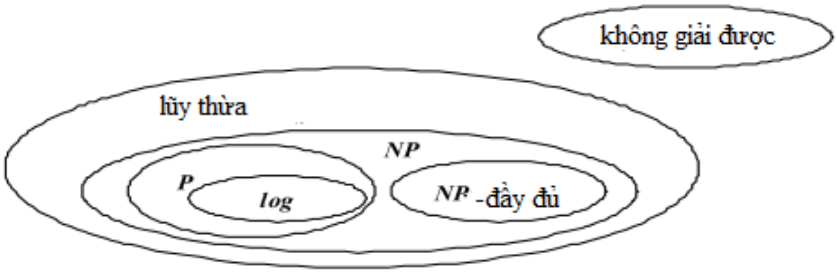
### 6.1.4. Các ví dụ

Một số lớp bài toán được liệt kê (và các kết nối giữa chúng) được thể hiện dưới dạng giản đồ trong Hình 6.1. Hình này cũng cho thấy loại bài toán $NP$-đầy đủ, chúng ta sẽ thảo luận chi tiết hơn trong 6.2. Hình 6.1 cho thấy một ví dụ về vị trí của các bài toán, vì vị trí chính xác của chúng không được xác định. Một trong những câu hỏi lớn trong tin học lý thuyết là liệu $P \subseteq NP$ hay $P \equiv NP$. Giả thuyết đầu tiên và Hình 6.1 được coi là có nhiều khả năng hơn, phản ánh điều này.

Chúng ta sẽ xem xét một số ví dụ về các bài toán thuộc các lớp khác nhau.

***Ví dụ 1.*** *Các bài toán có thể giải bằng đa thức*

Một số ví dụ về các bài toán đa thức có thể giải được có thể được liệt kê, nhiều trong số đó đã được thảo luận trong bài báo cho đến nay: các thuật toán từ lý thuyết đồ thị (để duyệt đồ thị và các sửa




Hình 6.1. Phân loại bài toán (theo thời gian).

đổi của chúng, thuật toán Dijkstra, Floyd, Prim, Kruskal và nhiều thuật toán khác - tất nhiên, chúng ta phải xác định chúng là các bài toán có thể xác minh), để tìm kiếm (ví dụ, tìm kiếm nhị phân - với độ phức tạp $\Theta(\log_2 n)$), hầu hết tất cả các thuật toán sắp xếp, để tính toán các hàm toán học khác nhau và nhiều hàm khác.

***Ví dụ 2.*** *Bài toán NP*

Ví dụ đơn giản sau đây minh họa rõ ràng ý nghĩa của lớp các bài toán $NP$: Một số tự nhiên $n$ là cho trước. Kiểm tra xem có ước của số tự nhiên $n$ nhỏ hơn số tự nhiên $q(1 < q < n)$ hay không.

Nếu chúng ta có một ứng cử viên cho nghiệm $q'$, chúng ta có thể dễ dàng kiểm tra xem có $p'$ sao cho $n = p'.q'$ hay không, nhưng đối với lời giải của bài toán trên thì tài nguyên tính toán cần thiết còn nhiều hơn nữa.

Hãy xem xét một ví dụ khác: một đồ thị $G(V, E)$, hai đỉnh $i, j \in V$ và một số tự nhiên $k$ được cho trước.

**Bài toán:** Kiểm tra xem có một đường đi đơn (không có đỉnh lặp lại) giữa $i$ và $j$ với độ dài ít nhất $k$.

*Bài toán này thuộc về lớp NP:* nếu chúng ta đã có được một đường dẫn, chúng ta có thể dễ dàng (với độ phức tạp tuyến tính) kiểm tra điều kiện xem đường đi có đơn không và nó có dài ít nhất $k$. Tuy nhiên, một thuật toán có độ phức tạp đa thức để tìm đường như vậy vẫn chưa được biết đến, vì vậy chúng ta không thể chắc chắn liệu bài toán có thuộc về lớp $P$ hay không.

Chúng ta sẽ lưu ý sau rằng bài toán này thuộc về một lớp bài toán mới được đề cập trong phần giới thiệu và sẽ là chủ đề chính

của chương này - lớp bài toán đầy đủ *NP*. Đối với họ, người ta cho rằng không có nghiệm đa thức.

**Ví dụ 3.** *Phân loại theo bộ nhớ*

Trong hầu hết các loại hình thể thao "trí tuệ" (ví dụ như cờ vua, các trò chơi bài khác nhau, v.v.), việc tham gia vào máy tính đã được tổ chức từ lâu. Ban đầu, trong những lần va chạm đầu tiên như vậy, các "tay đua" máy tính hầu như không có cơ hội, và chiến thắng của con người trước cỗ máy đã hỗ trợ cho luận điểm rằng trí tuệ tự nhiên khó có thể bị các quá trình tính toán nhân tạo cạnh tranh. Tuy nhiên, nhiều người ủng hộ quan điểm này đã "uống một cốc nước lạnh" khi vào năm 1997, Deep Blue, một máy tính IBM, lần đầu tiên đánh bại nhà vô địch cờ vua thế giới Gary Kasparov trong một trận đấu chính thức. Năm 1996, Deep Blue thắng một ván trước Kasparov, nhưng thua trận (tổng cộng 6 ván). Tuy nhiên, sau đó, vào năm 1997, anh đã đánh bại anh ta với 3,5: 2,5 điểm (2 thắng, 1 thua và 3 hòa), nhờ đó anh nhận được giải thưởng 100.000 đô la mà anh đã thiết lập 17 năm trước đó cho chương trình máy tính đầu tiên của mình. nhà vô địch cờ vua thế giới. Tuy nhiên, chúng ta sẽ lưu ý rằng khả năng của máy tính hiện đại vẫn không cho phép một chương trình máy tính trở thành nhà vô địch thế giới trong cờ vua, và nếu có thể, điều này vẫn chưa được chứng minh trong thực tế: Deep Blue chỉ thắng một phần trong các ván cờ. , và sau khi anh ấy được lập trình đặc biệt để thi đấu tốt với Kasparov. [Russell, Norvig-1995] [Loviglio-1997]

Tuy nhiên, ở những trò chơi đơn giản hơn như cờ caro, cỗ máy đã chứng tỏ được ưu thế của mình so với con người. Năm 1992, chương trình máy tính Chinook, được tạo ra bởi Jonathan Schaefer và các đồng nghiệp của ông, đã giành chức vô địch Mỹ mở rộng, nhưng không thể trở thành nhà vô địch thế giới, do Marion Tinsley giành chiến thắng với tỷ số 21,5: 18,5. Tinsley đã 40 năm vô địch thế giới (!) Và chỉ để thua 3 trận chính thức trong thời gian đó. Tuy nhiên, trong cuộc đụng độ với Chinook, anh đã ghi được trận thua thứ tư và thứ năm. Sau đó, vào năm 1994, Chinook trở thành nhà vô địch thế giới sau khi trận đấu chính thức gặp Tinsley bị đình chỉ vì lý do sức khỏe.

Nhưng hãy quay lại với cờ vua. Lập trình các chương trình cờ vua là gì? Chuỗi nước đi có thể được hiểu là một cái cây: trước mặt

người chơi bắt đầu trò chơi, 20 bước di chuyển khác nhau có thể được thực hiện, với mỗi bước di chuyển có nhiều bước di chuyển ứng phó khác nhau, v.v. Các chương trình cờ vua đi xung quanh cây này và kiểm tra hậu quả có thể xảy ra khi thực hiện mỗi nước đi có thể xảy ra.

Bài toán đối với một chương trình chơi cờ vua hoàn hảo là của lớp $exp - space$, nếu chúng ta coi kích thước của bàn cờ là kích thước của dữ liệu đầu vào. Ý nghĩa của việc phân loại trí nhớ? Trong cờ vua, kích thước của bàn cờ là không đổi - $8 \times 8$. Do đó, việc duy trì tất cả các cấu hình có thể có và nước đi tốt nhất cho mỗi người trong số họ yêu cầu bộ nhớ liên tục, mặc dù hằng số này rất lớn (giả sử rằng bất kỳ cấu hình cờ vua nào cũng có thể thực hiện được thì số cấu hình cờ vua khác nhau là $65^{32}$ - tại sao?). Mặt khác, chiều cao của cây cờ không lớn: một ván cờ dài hơn 150 nước đi rất hiếm khi được chơi. Do đó, nếu chúng ta tìm kiếm trên cây cho mỗi bước đi, vấn đề thiếu bộ nhớ sẽ biến mất, nhưng một vấn đề khác lại nảy sinh: thời gian dài (theo cấp số nhân) không thể chấp nhận được để tính toán mỗi nước đi.

Có những phương pháp thu thập thông tin được gọi là cắt ngắn alpha-beta (xem 6.5.3) hạn chế việc nghiên cứu cây quyết định, nhưng thậm chí chúng không thể xử lý số lượng lớn các đỉnh thu thập thông tin. Vì lý do này, các chương trình cờ vua thực sự tìm kiếm ít chuyên sâu hơn (ví dụ: 8 quân tiến lên) và từ thông tin thu thập được, cũng như dựa trên một số tiêu chí "chiến thuật" (ví dụ: số lượng và quân cờ còn lại trên bảng, v.v.). xem 6.5.4) xác định nước đi tốt nhất có thể.

**Ví dụ 4.** *Các bài toán có thể giải được*

Bài toán sau là không thể giải được: Hai chuỗi $x$ và $y$ và một tập hợp các quy tắc thay thế $v$ được đưa ra. Kiểm tra xem chuỗi $x$ có thể được chuyển đổi thành chuỗi y hay không bằng một loạt các ứng dụng quy tắc thay thế. Các quy tắc thay thế được áp dụng như sau: mỗi khi chuỗi con $v$ tham gia vào chuỗi tạm thời (phép biến đổi cuối cùng thu được), nó có thể được thay thế bằng $u$ (đối với một quy tắc $v \to u$ cho trước).

Hầu hết các bài toán liên quan đến hành vi của một chương trình cũng không thể giải quyết được. Cuối cùng, chúng ta sẽ xem xét

ví dụ sau: Khi một chương trình chạy trong một thời gian dài mà không cho kết quả, rất khó để đánh giá liệu nó, trong ngôn ngữ "lập trình", có bị "lặp" hay đơn giản là thuật toán cần thêm thời gian để đầy đủ. . Thật vậy, sẽ vô cùng tiện lợi nếu có một phương pháp tự động để kiểm tra như vậy (ví dụ, trình biên dịch có thể tự đánh giá). Tuy nhiên, bài toán này là không thể giải quyết được: không có thuật toán nào có thể luôn và không có lỗi xác định chương trình có nằm trong vòng lặp vô hạn hay không. Sau đây là một ví dụ về những gì đã được nói cho đến nay:

Phần chính của chương trình

```
int main() {
  unsigned a, b, c, i, x;
  for (x = 3;;) {
    for (a = 1; a <= x; a++)
      for (b = 1; b <= x; b++)
        for (c = 1; c <= x; c++)
          for (i = 3; i<= x; i++)
            if (pow(a, i) + pow(b, i) == pow(c, i))
              exit(0);
    x++;
  }
  return 0;
}
```

Đoạn được hiển thị kiểm tra xem có nghiệm của phương trình $a^n + b^n = c^n$, cho $a, b, c, n$ số tự nhiên và $n > 2$. Theo Định lý lớn được chứng minh gần đây của Fermat, bộ số tự nhiên không tồn tại. . Đối với một người (và một người biên dịch) không biết rằng định lý đã được chứng minh, sẽ rất khó để đánh giá liệu chương trình trên có tìm ra lời giải và kết thúc hay không. Hơn nữa, có nhiều bài toán tương tự mà kết quả lý thuyết tương ứng không được biết và hoàn toàn không biết liệu các chương trình tương ứng có lặp lại hay không.

Bài toán xác định xem một vòng lặp chương trình có được gọi là vấn đề dừng hay không. Việc xem xét lý thuyết của nó đối với một số hình thức tính toán hiện có (ví dụ, máy Turing - xem [Manev-1996]) cho phép chứng minh thực tế này một cách chặt chẽ.

**Ví dụ 5.** *Bài toán không xác định*

Đối với một số bài toán, người ta vẫn chưa xác định được liệu chúng có thuộc loại bài toán khó giải quyết hay không: một ví dụ tương tự là bài toán Scolem [MathWorld]:

Một ma trận $M_{3\times3}$ đã cho. Kiểm tra xem có tồn tại số tự nhiên $n$ sao cho nếu $M$ được nâng lên lũy thừa $n$ thì thu được ma trận có góc trên bên phải chứa 0.

### Bài tập

▷ **6.1.** Xác định số lượng các cấu hình cờ vua khác nhau.

▷ **6.2.** Đưa ra các ví dụ về các bài toán từ lý thuyết đồ thị có độ phức tạp lôgarit, đa thức hoặc hàm mũ.

▷ **6.3.** Thực hiện hai bài toán sau thuộc cùng một lớp:

- Kiểm tra số tự nhiên n có phải là số nguyên tố hay không.
- Kiểm tra xem có ước của số tự nhiên n nhỏ hơn số tự nhiên $q(q > 1, q < n)$ hay không.

## 6.2. Bài toán NP-đầy đủ

Từ quan điểm lý thuyết, một trong những câu hỏi thú vị nhất là liệu lớp $P$ có trùng với $NP$ hay không. Nói cách khác, nếu luôn có thể kiểm tra một ứng cử viên cho một nghiệm có độ phức tạp đa thức, thì liệu có thể tìm được toàn bộ lời giải với độ phức tạp như vậy không?

Người ta mong rằng câu trả lời cho câu hỏi trên là không. Tuy nhiên, vẫn chưa có bằng chứng nào chứng minh cho luận điểm này. Nghĩa là, chúng ta không thể chắc chắn rằng một vấn đề $NP$ nhất định không nằm trong P. Loại bài toán mà chúng ta gọi là $NP$-đầy đủ mang lại sự rõ ràng hơn khi tìm kiếm giải pháp cho vấn đề này. $NP$-đầy đủ là một loại bài toán $NP$ có thể được giảm bớt cho nhau với độ phức tạp đa thức (chúng ta sẽ xác định điều này một cách chặt chẽ trong giây lát) và mỗi bài toán từ $NP$ có thể được giảm đa thức thành một số bài toán $NP$-đầy đủ. Nếu chỉ một $NP$-đầy đủ được chứng minh là thuộc hoặc không thuộc cấp $P$, thì điều này sẽ áp dụng cho tất cả các bài toán khác của cấp $NP$ (Lưu ý từ $NP$-hoàn

chỉnh của cả lớp, và không chỉ từ $NP$-đầy đủ.). Việc khám phá ra một thuật toán đa thức cho một bài toán $NP$-đầy đủ là một bản tóm tắt "ý tưởng cố định" của mọi nhà toán học mới vào nghề - tìm ra một công thức đơn giản để tạo ra các số nguyên tố liên tiếp (điểm).

**Định nghĩa 6.1.** Chúng ta nói rằng bài toán $A$ có thể rút gọn thành bài toán $B$ và viết $A\langle B$ nếu có thể tìm được thuật toán giải bài $A$ bằng cách sử dụng một số đa thức các lệnh gọi đến chương trình giải $B$ và tất cả các phép tính bên ngoài các lệnh gọi này là độ phức tạp đa thức.

Sau đó, nếu một bài toán $B$ thuộc loại $P$ và $A\langle B$, thì nó theo sau rằng $A$ cũng thuộc loại $P$. Để minh họa những gì đã được nói cho đến nay bằng một ví dụ: tìm một chu trình Hamilton trong một đồ thị (xem **??**) - một chu kỳ trong đó mỗi đỉnh tham gia đúng một lần. Vấn đề có thể được giải quyết với sự trợ giúp của ví dụ 2:

```
for (<mỗi cạnh (i,j) từ đồ thị>) {
  if (<tồn tại đường đơn với độ dài n-1 đỉnh từ i đến j>) {
    return 1; /* Có chu trình Hamilton: đường đã tìm + cạnh (i,j)*/
  }
}
return 0;
```

Hãy biểu thị số cạnh trong đồ thị bằng $m$. Thuật toán sẽ tạo ra m tham chiếu đến bài toán tìm đường đi dài nhất trong đồ thị và độ phức tạp bên ngoài bài toán này sẽ là $\Theta(m)$. Nếu chúng ta áp dụng định nghĩa trên, nó có nghĩa là "Chu trình Hamilton "$\langle$". Đường đi đơn dài nhất". Tuy nhiên, vì không có thuật toán đa thức nào được biết đến để giải "con đường đơn giản dài nhất", chúng ta không thể nói rằng chu trình Hamilton là một bài toán đa thức có thể giải được.

**Định nghĩa 6.2.** Một bài toán $NP$-A được gọi là $NP$-đầy đủ khi và chỉ khi đối với mọi bài toán $B$ khác của lớp $NP$, nó tuân theo $B\langle A$.

**Định lý 6.1** (Cook)**.** *Có ít nhất một bài toán thuộc lớp NP-đầy đủ.*

Việc chứng minh định lý cuối cùng được thực hiện bằng cách tìm một bài toán $NP$-đầy đủ như vậy [Manev-1996] - ví dụ bài toán thỏa mãn một hàm Boolean (xem 6.3.1).

Việc áp dụng định nghĩa trên và định lý Cook cho nhiều bài toán đã dẫn đến việc chứng minh rằng chúng là $NP$-đầy đủ. Đoạn cuối cùng của chương (xem 6.6) Đưa ra các điều kiện cho hơn 70 bài toán đầy đủ $NP$. Có hàng trăm bài toán khác thuộc về lớp này. Nếu chỉ một trong số chúng chứng minh được rằng nó có thể hoặc không thể giải được bằng thuật toán đa thức, thì điều này sẽ xảy ra với tất cả những bài khác: không chỉ đối với những bài từ $NP$-đầy đủ, mà còn với $NP$ nói chung. Và từ đây nó sẽ trực tiếp theo $P \equiv NP$ đó.

Trong lý thuyết về độ phức tạp và khả năng tính toán của thuật toán, các lớp mà chúng ta đã xem xét cho đến nay ($P$, $NP$, $NP$-đầy đủ) là các bài toán về khả năng xác minh, tức là. trả lời một câu hỏi có hoặc không. Sau đây là một số ví dụ (bao gồm cả những ví dụ đã được thảo luận):

1. Để kiểm tra xem một số đếm có phải là Hamilton hay không.

2. Kiểm tra xem có chu trình Hamilton trong đồ thị có độ dài nhỏ hơn $k$ hay không.

Tuy nhiên, có một số tác vụ không phải là tác vụ có thể xác minh được, ví dụ:

3. Tìm chu trình Hamilton có độ dài nhỏ nhất.

Trong những trường hợp này, lý thuyết cần được mở rộng và một lớp bài toán khác được đưa vào: $NP$-khó.

**Định nghĩa 6.3.** Bài toán $A$ (không nhất thiết phải là bài toán kiểm chứng) thuộc cấp $NP$-khó, nếu và chỉ khi đối với mọi bài toán $B$ thuộc cấp $NP$, nó tuân theo $B \langle A$.

Theo hệ quả của định nghĩa, dễ dàng chứng minh rằng lớp của các bài toán $NP$-đầy đủ là một phần cắt ngang của các lớp $NP$ và $NP$-khó. Do đó, mối quan hệ giữa bài toán $NP$-đầy đủ và $NP$-khó có thể được biểu thị như: $NP$-đầy đủ là những bài toán $NP$-khó là những bài toán có thể xác minh được.

Đối với mỗi bài toán đầy đủ $NP$, có thể xác định một $NP$-khó tương ứng "không dễ hơn". Ví dụ, không có giải pháp nào khác cho lời giải của Bài toán 2 ở trên (trong trường hợp tổng quát của một đồ thị tùy ý) ngoài việc tìm chu trình Hamilton nhỏ nhất (chính xác là Bài toán 3).

### Bài tập

▷ **6.4.** Chỉ sử dụng định nghĩa 6.2. để chứng minh rằng nếu một bài toán $A$ $NP$-đầy đủ có thể rút gọn thành một bài toán $B$ khác, thì $B$ cũng là một bài toán $NP$-đầy đủ.

▷ **6.5.** Đưa ra các ví dụ về các bài toán từ lý thuyết đồ thị thuộc về các bài toán $NP$-đầy đủ bằng cách rút gọn thành tìm kiếm chu trình Hamilton trong đồ thị (chúng ta giả định rằng tính đầy đủ $NP$ của bài toán tìm chu trình Hamilton đã được chứng minh).

▷ **6.6.** bài toán "Kiểm tra xem có ước nào của số tự nhiên $n$ nhỏ hơn số tự nhiên $q(q > 1, q < n)$" thuộc lớp $NP$ không. Và nó có thuộc lớp $NP$-đầy đủ không?

## 6.3. Tìm kiếm với quay lui

Nếu chúng ta chấp nhận giả thuyết $P \neq NP$ là đúng (ký hiệu chính xác hơn sẽ là $P \subset NP$, vì rõ ràng là $P \subseteq NP$, nhưng $P \neq NP$ được sử dụng thường xuyên hơn trong tài liệu), thì cách tiếp cận duy nhất đảm bảo rằng chúng ta sẽ luôn tìm giải pháp chính xác cho một bài toán $NP$-đầy đủ (hoặc $NP$-khó) là vét cạn hoàn toàn. Một cách có thể để đạt được mức cạn kiệt hoàn toàn là phương pháp tìm kiếm quay lui.

Phương pháp quay lui là một kỹ thuật trong đó giải pháp của một vấn đề được xây dựng một cách tuần tự. Ở mỗi bước, một nỗ lực được thực hiện để mở rộng giải pháp từng phần (không đầy đủ) hiện tại với tất cả các phần mở rộng có thể có. Nếu không có phần mở rộng nào trong số này dẫn đến một giải pháp hoàn chỉnh sau đó, trường hợp được tuyên bố là vô vọng và thuật toán quay lại một bước. Các trường hợp giới hạn cho thuật toán xảy ra khi một giải pháp hoàn chỉnh cho vấn đề được tìm thấy hoặc khi không thể mở rộng giải pháp từng phần theo bất kỳ cách nào. Lược đồ đệ quy sau đây thường được sử dụng khi giải quyết các tác vụ tìm kiếm quay lui:

Lược đồ quay lui

```
void thu(<bước i>)
  { if (i > n) <kiểm tra xem có phải là nghiệm không>;
```

```
  else
  /* Mở rộng nghiệm riêng theo tất cả khả năng có được*/
  for (k = 1; k <= n; k++)
    if (<thành phần thứ k có thể chấp nhận>) {
      <Ghi nhận là thành phần chấp nhận>;
      thu(i+1);
      <Bỏ thành phần đã ghi nhận>;
    }
}
```

Trong các chương trước, phương pháp này đã được áp dụng nhiều lần. Ví dụ, trong bài toán cổ điển tìm chu trình Hamilton trong đồ thị, chúng ta sử dụng thuật toán sau:

1) Chúng ta bắt đầu xây dựng một chu trình Hamilton từ một đỉnh tùy ý $s \in V$ và đánh dấu nó như đã xét.

2) Tại mỗi bước, chúng ta cố gắng chuyển đến đỉnh v mới chưa được khám phá, sự cố của đỉnh hiện tại. Bằng cách thực hiện một quá trình chuyển đổi như vậy, chúng ta tiến thêm một bước - chúng ta đánh dấu đỉnh cao mới như đã xem xét và tiếp tục theo cách tương tự. Nếu không có đỉnh nào để vượt qua, có thể xảy ra hai trường hợp:

2.1) Nếu tất cả các đỉnh của đồ thị được đánh dấu và $v$ là ngẫu nhiên với $s$, thì sau đó chúng ta đã duyệt qua đồ thị và tìm ra chu trình Hamilton.

2.2) Nếu vẫn còn các đỉnh cần thiết (và vì chúng ta không thể di chuyển đến bất kỳ đỉnh nào trong số chúng), chúng ta tuyên bố giải pháp từng phần hiện tại là vô vọng: nó không thể được mở rộng theo bất kỳ cách nào bằng đỉnh hiện tại. Chúng ta đánh dấu đỉnh cao mà chúng ta đang có và lùi lại một bước để trở lại đỉnh cao mà chúng ta đã đạt được. Chúng ta tiếp tục thử các đỉnh ngẫu nhiên, không được đánh dấu khác, mà chúng ta đã không cố gắng vượt qua cho đến nay.

Nếu trong 2.1) khi tìm một chu trình Hamilton, chúng ta không làm gián đoạn thuật toán, nhưng quay lại, như trong 2.2), thì sự cạn kiệt hoàn toàn sẽ tiếp tục và chúng ta sẽ tìm thấy tất cả các chu trình Hamilton có thể có trong đồ thị.

Phương pháp tìm kiếm quay lui và các tính năng của nó sẽ vẫn là chủ đề chúng ta chú ý trong một vài điểm tiếp theo, nơi chúng ta sẽ xem xét một vài bài toán $NP$-đầy đủ và toàn diện hơn.

## Bài tập

▷ **6.7.** Vị trí nào trong sơ đồ trên là thích hợp để kiểm tra tính tối ưu (ví dụ, nếu bạn đang tìm một chu trình Hamilton với độ dài tối thiểu)?

### 6.3.1. Sự thỏa mãn của một hàm Boolean

Bài toán thỏa mãn hàm Boolean, theo một nghĩa nào đó, là bài toán đầy đủ $NP$ "đầu tiên". Nó thường được sử dụng để chứng minh định lý Cook từ 6.2, Bằng cách rút gọn nó để chứng minh tính đầy đủ $NP$ của nhiều bài toán khác. Chúng ta sẽ xem xét bài toán này và giải quyết nó bằng cách tìm kiếm quay lui.

Cho các biến Boolean $X_1, X_2, ..., X_n$ được cho trước. Bằng $\bar{X}_1, \bar{X}_2, ..., \bar{X}_n$ chúng ta sẽ biểu thị các phủ định của chúng (nghĩa là $X_i$ là "đúng" khi và chỉ khi $\bar{X}_i$ là "sai"). Chúng ta sẽ sử dụng các phép toán tiêu chuẩn để xây dựng các biểu thức Boolean $\wedge$ và $\vee$, có nghĩa là logic "và" (kết hợp) và logic "hoặc" (disjunction), cũng như các giá trị 0 và 1 để biểu thị "false" và " true ", tương ứng. Chúng ta sẽ cho phép sử dụng tính năng phủ định trên toàn bộ biểu thức (xem định luật De Morgan bên dưới). Đôi khi chúng ta biểu thị sự phủ định bằng dấu $\neg$ đặt trước biểu thức, ví dụ: $\neg(A \vee B)$. Liên kết $A \wedge B$ thường được viết là $A.B$ hoặc thậm chí $AB$.

Trong Bảng 6.1 kết quả có thể có của các phép toán logic $\wedge$ và $\vee$ trên hai biểu thức Boolean được hiển thị.

| $x$ | $y$ | $x \wedge y$ | $x \vee y$ |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

Bảng 6.1. Bảng chân lý

**Định nghĩa 6.4.** Biểu thức Boolean của các biến $X_1, X_2, ..., X_n$ được gọi là thỏa mãn nếu có một bộ giá trị "đúng" hoặc "sai" đối với $X_1, X_2, ..., X_n$ mà giá trị của biểu thức là "true".

**Bài toán:** Kiểm tra xem một biểu thức có đạt yêu cầu hay không.

Ví dụ, biểu thức $(\bar{X}_2 \vee X_2 \wedge X_1) \wedge \bar{X}_3$ thỏa mãn với $X_1 = 1, X_2 = 0, X_3 = 0$ hoặc $X_1 = 1, X_2 = 1, X_3 = 0$ Kiểm tra xem một biểu thức có thỏa mãn hay không là một bài toán $NP$: nếu một phép gán giá trị cho các biến được đưa ra, chúng ta có thể, như sẽ thấy ngay sau đây, với độ phức tạp đa thức kiểm tra xem nó có phải là một giải pháp hay không, nhưng để tìm ra phép gán này không có thuật toán đa thức nào được biết.

Trước khi chuyển sang lời giải của bài toán, chúng ta sẽ nhắc lại một số yếu tố của logic toán học.

**Định nghĩa 6.5.** Các hằng số 0 và 1, cũng như các biến và sự phủ định của chúng được gọi là *các chữ*.

**Định nghĩa 6.6.** *Một dạng biểu thức hoặc* là một biểu thức Boolean trong đó chỉ phép toán "hoặc" có liên quan.

**Định nghĩa 6.7.** Một biểu thức Boolean trong đó chỉ giải trừ được kết hợp bởi phép toán "và" tham gia được gọi là một biểu thức ở *dạng chuẩn liên hợp*.

Với việc áp dụng danh tính nhiều lần

$$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$$

bất kỳ biểu thức Boolean nào cũng có thể được biểu diễn ở dạng chuẩn liên hợp.

Các quy tắc hữu ích khác để biến đổi biểu thức được đưa ra bởi các định luật De Morgan:

$$\overline{A \vee B} = \bar{A} \wedge \bar{B}$$

$$\overline{A \wedge B} = \bar{A} \vee \bar{B}$$

Hãy xem một biểu thức ví dụ được viết ở dạng thông thường liên hợp:

$$(X_1 \vee X_4) \wedge (\bar{X}_1 \vee X_2) \wedge (X_1 \vee \bar{X}_3) \wedge (\bar{X}_2 \vee X_3 \vee \bar{X}_4) \wedge (\bar{X}_1 \wedge \bar{X}_\wedge \bar{X}_3)$$

Để kiểm tra xem biểu thức có thỏa mãn hay không, chúng ta có thể áp dụng phương pháp tổ hợp sau: chúng ta kiểm tra 2 n số phép gán giá trị có thể có cho các biến và với mỗi phép gán, chúng ta kiểm tra xem biểu thức có thỏa mãn hay không, với độ phức tạp $\Theta(2^n)$. Do đó, thuật toán tìm kiếm quay lui có dạng sau:

Chúng ta khởi tạo $i = 1$.

1) Gán cho biến thứ $i$ giá trị "true" và chuyển sang biến thứ $(i+1)$ (tiến thêm một bước). Nếu việc gán này không dẫn đến quyết định sau đó, chúng ta quay lại (lùi một bước), gán biến "lie" và thử tiếp tục một lần nữa.

2) Trường hợp giới hạn là khi tất cả các biến được gán một giá trị - sau đó giá trị của toàn bộ biểu thức được tính toán và kiểm tra xem nó có phải là "true" hay không.

Trong quá trình triển khai thuật toán được mô tả, biểu thức Boolean, được rút gọn thành dạng chuẩn liên hợp, sẽ được biểu diễn bằng một mảng hai chiều - một hàng của mảng được so sánh với mỗi hàng không tồn tại. Chúng ta sẽ biểu diễn các biến Boolean của một hàng (không tồn tại) với các chỉ số của chúng - số nguyên từ 1 đến $n$ và để phủ định biến $X_i$, chúng ta sẽ sử dụng số $-i$. Các giá trị `values[]` chứa các giá trị (0 hoặc 1) mà chúng ta đã gán cho mỗi biến. Hàm `true()` tính toán giá trị của biểu thức Boolean cho các phép gán được thực hiện trong các `values[]`. Việc kiểm tra này rất dễ thực hiện: Để toàn bộ biểu thức có giá trị là "true", thì mỗi phép toán bỏ quân phải có giá trị là "đúng" (vì phép toán giữa các lần bỏ quân là "và"). Để một từ không tồn tại có giá trị "đúng", ít nhất một trong các biến trong đó phải có giá trị "đúng" hoặc phủ định của một biến có giá trị "sai" (vì tất cả các ký tự trong một từ không tồn tại đều là kết hợp với phép toán "hoặc").

Việc gán giá trị của các biến trong thuật toán trên được thực hiện bởi hàm gán (không dấu $i$), và trong trường hợp biên $(i = n)$, hàm true () được gọi để kiểm tra xem biểu thức có thỏa mãn hay không (nghĩa là kiểm tra xem việc chiếm đoạt có phải là một giải pháp cho vấn đề). Nhận thức đầy đủ như sau:

**Chương trình** 6.1. Biểu thức boolean (601bool.c)

```
#include <stdio.h>
```

```
#define MAXN 100 /* Số lượng tối đa biến boolean*/
#define MAXK 100 /* Số lượng biểu thức  hoặc*/
const unsigned n = 4; /*  Số  lượng các biến boolean*/
const unsigned k = 5; /*  Số  lượng biểu thức  hoặc */
const int expr[][MAXK] = {
  { 1, 4 },
  { -1, 2 },
  { 1, -3 },
  { -2, 3, -4 },
  { -1, -2, -3 }
};

int values[MAXN];

void printAssignment(void)
{ unsigned i;
  printf("Biểu thức thỏa mãn cho: ");
  for (i  = 0;  i  < n;  i++) printf("X%u=%u ", i+1, values[i]);
  printf("\n");
}

/*  ít  nhất một chữ phải có giá  trị  "đúng" trong mỗi biểu thức hoặc *
    /
int true(void)
{ unsigned i;
  for (i  = 0;  i  < k;  i++) {
    unsigned j = 0;
    char ok = 0;
    while (expr[i][j]  != 0)  {
      int p = expr[i][j];
      if ((p > 0)  && (1 == values[p-1])) {  ok = 1;  break; }
      if ((p < 0)  && (0 == values[-p-1])) {  ok = 1;  break; }
      j++;
    }
    if (!ok) return 0;
  }
  return 1;
}

/*  Gán giá trị  cho các  biến*/
```

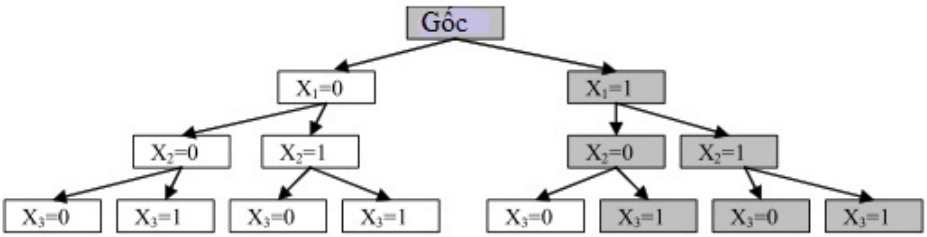
```
void assign(unsigned i)
{ if (i == n) {
    if (true()) printAssignment();
    return;
  }
  values[i] = 1; assign(i + 1);
  values[i] = 0; assign(i + 1);
}

int main() {
  assign(0);
  return 0;
}
```

*Kết quả thực hiện chương trình:*

Biểu thức thỏa mãn cho: X1 = 1 X2 = 1 X3 = 0 X4 = 0
Biểu thức thỏa mãn: X1 = 0 X2 = 0 X3 = 0 X4 = 1

Chúng ta sẽ minh họa cách thực hiện phép gán trong một ví dụ với 3 biến Boolean: Hình 6.2 - các đỉnh dẫn đến thỏa mãn biểu thức $X_1 \wedge (X_2 \vee X_3)$ được đánh dấu đậm hơn.




Hình 6.2. Thỏa mãn của biểu thức boolean

Trong các bài toán *NP*-đầy đủ (và đặc biệt trong bài toán thỏa mãn), các ứng cử viên cho giải pháp mặc nhiên tạo thành một cây (hoặc đồ thị) mà thuật toán tìm kiếm quay lui đi qua. Tuy nhiên, thông thường, nghiên cứu về phần lớn loài cây này là không cần thiết. Ví dụ, đối với biểu thức $X_1 \wedge (X_2 \vee X_3)$ khi gán $X_1 = 0$, ngay lập tức nhận thấy rằng chúng ta sẽ không đạt được một nghiệm, bất kể các giá trị sẽ nhận hai biến khác $X_2$ và $X_3$. Khi các biến nhiều hơn

và biểu thức Boolean phức tạp hơn, việc cắt cây được nghiên cứu như vậy có thể làm tăng đáng kể tốc độ của chương trình.

Chúng ta sẽ thực hiện việc cắt với sự trợ giúp của các sửa đổi sau trong chương trình trên:

1) Một đối số *t* mới được thêm vào hàm `true()`, nghĩa là chỉ các biến *t* đầu tiên mới được gán giá trị. Do đó, nếu một từ không tồn tại chỉ bao gồm các biến có số nhỏ hơn hoặc bằng*t* và không có biến nào trong số chúng có giá trị là "true", thì toàn bộ biểu thức Boolean sẽ là "false" và không có ý nghĩa gì khi gán giá trị Cho những biến nt khác. Nếu có các biến trong liên kết chưa được gán giá trị, thì giá trị của nó vẫn chưa được xác định và quá trình gán vẫn tiếp tục.

2) Trong quá trình tạo, hàm `true()` được gọi ở đầu mỗi bước. Do đó, việc tạo thêm sẽ bị gián đoạn ngay khi nó trở nên vô nghĩa - giá trị của biểu thức Boolean sẽ là một "lời nói dối", bất kể các phép gán khác sẽ được thực hiện như thế nào.

**Chương trình** 6.2. Biểu thức boolean (602boolcut.c)

```
#include <stdio.h>
/* Số lượng lớn nhất biến boolean */
#define MAXN 100
/* Số lượng lớn nhất biểu thức hoặc*/
#define MAXK 100
const unsigned n = 4; /* Số lượng các biến boolean*/
const unsigned k = 5; /* Số lượng biểu thức hoặc*/
const int expr[][MAXK] = {
  {  1,  4 },
  { -1,  2 },
  {  1, -3 },
  { -2,  3, -4 },
  { -1, -2, -3 }
};

int values[MAXN];

void printAssign(void)
{ unsigned i;
  printf("Biểu thức thỏa mãn với: ");
  for (i = 0; i < n; i++) printf("X%d=%d ", i + 1, values[i]);
  printf("\n");
```

```
}

/* ít nhất ký tự phải có giá trị "đúng" trong mọi biểu thức hoặc */
int true(int t)
{ unsigned i;
  for (i = 0; i < k; i++) {
    unsigned j = 0;
    char ok = 0;
    while (expr[i][j] != 0) {
      int p = expr[i][j];
      if ((p > t) || (-p > t)) { ok = 1; break; }
      if ((p > 0) && (1 == values[p-1])) { ok = 1; break; }
      if ((p < 0) && (0 == values[-p-1])) { ok = 1; break; }
      j++;
    }
    if (!ok) return 0;
  }
  return 1;
}

/* Gán giá trị cho các biến*/
void assign(unsigned i)
{ if (!true(i)) return;
  if (i == n) {
    printAssign();
    return;
  }
  values[i] = 1; assign(i + 1);
  values[i] = 0; assign(i + 1);
}

int main() {
  assign(0);
  return 0;
}
```

## Bài tập

▷ **6.8.** Hãy xuất các tiêu chí khác đối với việc loại cây của người đề cử cho lời giải.

▷ **6.9.** Biểu thức $(\bar{X}_2 \vee X_3 \wedge X_1) \wedge X_2 \wedge (\bar{X}_1 \vee \bar{X}_3)$ có thỏa mãn không?

▷ **6.10.** 3. Chứng minh đồng dạng $A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$ bằng bảng chân trị.

▷ **6.11.** Hãy chứng minh các định luật De Morgan.

▷ **6.12.** Sử dụng định luật De Morgan để chứng minh đẳng thức $A \vee B = \overline{\bar{A} \wedge \bar{B}}$.

▷ **6.13.** Phép bình đẳng từ bài toán trước thể hiện sự tách rời thông qua phép liên kết và phép phủ định. Đề xuất và chứng minh một công thức thể hiện sự liên kết thông qua phép loại bỏ và phủ định.

▷ **6.14.** Sử dụng hai bài toán trước, chứng minh rằng các tập $\{\neg, \wedge\}$ và $\{\neg, \vee\}$ có cùng biểu thức là $\{\neg, \wedge, \vee\}$.

▷ **6.15.** Có đúng là tập $\{\wedge, \vee\}$ có cùng tính biểu cảm với $\{\neg, \wedge, \vee\}$ không?

## 6.3.2. Tô màu đồ thị

Trong **??** chúng ta đã xác định vấn đề tô màu tối thiểu của một đồ thị. Nó thuộc về lớp *NP* - khó và trong đoạn này chúng ta sẽ giải quyết nó bằng cách sử dụng phương pháp tìm kiếm quay lui. Dễ dàng nhận thấy mối liên hệ giữa bài toán khó *NP* đối với việc tô màu tối thiểu và bài toán *NP*-đầy đủ để kiểm tra xem biểu đồ có thể tô màu r hay không:

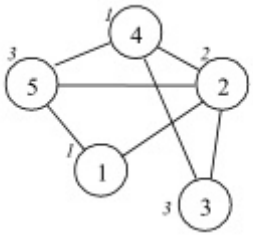
Hãy biểu thị số đỉnh trong đồ thị bằng $n$. Với mỗi $r$ ($1 \leq r \leq n$) chúng ta sẽ kiểm tra xem đồ thị có thể tô được $r$ màu hay không. Do đó, lời giải của bài toán tô màu tối thiểu là rút gọn đa thức thành tô màu $r$ của một đồ thị.

Hãy xem xét thuật toán sau để kiểm tra khả năng nhuộm $r$:

- Ta bắt đầu tô màu từ đỉnh đầu tiên của đồ thị: $i = 1$.
- *Bước i*: Tô màu đỉnh thứ i của biểu đồ với màu đầu tiên và chuyển sang tô màu đỉnh $(i + 1)$, sau đó tô màu đỉnh thứ i của biểu đồ bằng màu thứ hai và chuyển sang tô màu $(i + 1)$, v.v.
- Nếu tất cả các đỉnh đều được tô màu (tức là $i = n$) thì ta rơi vào trường hợp biên. Chúng ta kiểm tra xem có các đỉnh lân cận cùng màu hay không và nếu không, chúng ta đã tìm ra giải pháp.

Dễ dàng nhận thấy rằng thuật toán trên cực kỳ kém hiệu quả: độ phức tạp của nó là $\Theta\left(\sum_{r=1}^{n} r^n\right) = \Theta(n^n)$. Bài toán tương tự như trong bài toán thỏa mãn và như sau: nếu chúng ta tô màu hai đỉnh liền kề có cùng màu ngay từ đầu, chúng ta sẽ tiếp tục tô màu, mặc dù chắc chắn sẽ không dẫn đến giải pháp. Thiếu sót này của thuật toán có thể được khắc phục nếu, khi tô màu cho đỉnh thứ $i$, chúng ta kiểm tra xem một trong những hàng xóm của nó không còn được tô cùng màu hay không. Nếu trường hợp này xảy ra, thì thay vì đi đến đỉnh $(i+1)$, chúng ta sẽ cố gắng tô màu thứ $i$ bằng một màu khác. Nếu chúng ta không thể tìm thấy màu cho đỉnh thứ $i$ trong số các màu $r$, chúng ta lùi lại một bước. Trong kết quả này, nếu $i = 1$ thì việc tô màu của biểu đồ là không thể thực hiện được (chúng ta cần tăng số lượng màu).

Sau đây là mã nguồn của chương trình. Đồ thị được biểu diễn bằng ma trận lân cận $A[][]$, màu của các đỉnh được ghi vào mảng $col[]$ và hàm đệ quy cố gắng tô màu cho đỉnh thứ i của đồ thị là $nextCol(unsignedi)$. Dữ liệu đầu vào mẫu được thể hiện trong Hình 6.3.2.




Hình 6.3. Màu $r$ tối thiểu của đồ thị.

**Chương trình** 6.3. Tô màu ít nhất của đồ thị (603colorm2.c)

```
#include <stdio.h>
/* Số lượng lớn nhất trong đồ thị */
#define MAXN 200
/* Số đỉnh trong đồ thị */
const unsigned n = 5;
/* Ma trân kề của đồ thị */
const char A[MAXN][MAXN] = {
```

```
    { 0, 1, 0, 0, 1 },
    { 1, 0, 1, 1, 1 },
    { 0, 1, 0, 1, 0 },
    { 0, 1, 1, 0, 1 },
    { 1, 1, 0, 1, 0 }};

unsigned col[MAXN], maxCol, count = 0;
char foundSol = 0;

void showSol(void)
{ unsigned i;
  count++;
  printf("Tô màu ít nhất của đồ thị: \n");
  for (i = 0; i < n; i++)
    printf("Đỉnh %u - với màu %u \n", i + 1, col[i]);
}

void nextCol(unsigned i)
{ unsigned k, j, success;
  if (i == n) { showSol(); return; }
  for (k = 1; k <= maxCol; k++) {
    col[i] = k;
    success = 1;
    for (j = 0; j < n; j++)
      if (1 == A[i][j] && col[j] == col[i]) {
        success = 0;
        break;
      }
    if (success) nextCol(i + 1);
    col[i] = 0;
  }
}

int main() {
  unsigned i;
  for (maxCol = 1; maxCol <= n; maxCol++) {
    for (i = 0; i < n; i++) col[i] = 0;
    nextCol(0);
    if (count)
      break;
```

```
    }
    printf("Tổng số màu được tìm thấy với %u màu: %u \n",maxCol,
     count);
    return 0;
}
```

*Kết quả thực hiện chương trình:*

```
Màu tối thiểu của đồ thị:
Đỉnh 1 - với màu 1
Đỉnh 2 - với màu 2
Đỉnh 3 - với màu 3
Đỉnh 4 - với màu 1
Đỉnh 5 - với màu 3
Màu tối thiểu của đồ thị:
Đỉnh 1 - với màu 1
Đỉnh 2 - với màu 3
Đỉnh 3 - với màu 2
Đỉnh 4 - với màu 1
Đỉnh 5 - với màu 2
Màu tối thiểu của đồ thị:
Đỉnh 1 - với màu 2
Đỉnh 2 - với màu 1
Đỉnh 3 - với màu 3
Đỉnh 4 - với màu 2
Đỉnh 5 - với màu 3
Màu tối thiểu của đồ thị:
Đỉnh 1 - với màu 2
Đỉnh 2 - với màu 3
Đỉnh 3 - với màu 1
Đỉnh 4 - với màu 2
Đỉnh 5 - với màu 1
Màu tối thiểu của đồ thị:
Đỉnh 1 - với màu 3
Đỉnh 2 - với màu 1
Đỉnh 3 - với màu 2
Đỉnh 4 - với màu 3
Đỉnh 5 - với màu 2
Màu đồ thị tối thiểu:
Đỉnh 1 - với màu 3
Đỉnh 2 - với màu 2
```

Đỉnh 3 - với màu 1
Đỉnh 4 - với màu 3
Đỉnh 5 - với màu 1
Tổng số màu được tìm thấy với 3 màu: 6

## Bài tập

▷ **6.16.** Hãy đề xuất một thuật toán để tô màu tối thiểu các đỉnh của một đồ thị, tức là tìm số lượng màu tối thiểu một cách trực tiếp, mà không cần kiểm tra tuần tự từng $r$ ($1 \leq r \leq n$) xem đồ thị có thể tô được $r$ màu hay không.

▷ **6.17.** Đề xuất thêm tiêu chuẩn loại cây của đăng ký ra quyết định.

▷ **6.18.** Đề xuất và thực hiện một thuật toán để tô màu tối thiểu các cạnh của đồ thị.

### 6.3.3. Đường đi đơn dài nhất trong đồ thị chu trình

Cho đến nay, chúng ta đã xem xét một số cách giải thích vấn đề tìm đường đi đơn giản dài nhất trong đồ thị tuần hoàn có trọng số định hướng (độ dài đường đi được tính bằng tổng các cạnh mà nó chứa). Dưới đây chúng ta sẽ đưa ra một thuật toán của nguyên tắc tìm kiếm quay lui.

Ở mỗi bước, chúng ta cố gắng mở rộng con đường đã xây dựng một phần với một đỉnh khác. Gọi $j$ là đỉnh cuối cùng của con đường được xây dựng cho đến nay. Đối với đường, chúng ta thêm liên tiếp mỗi đỉnh kề với $j$ và không tham gia vào đường. Nếu tại một thời điểm nào đó, chúng ta không thể tìm thấy một đỉnh như vậy, thì điều đó có nghĩa là chúng ta đang ở trong một trường hợp ranh giới. Sau đó, chúng ta tính toán chiều dài của con đường và nếu nó lớn hơn mức tối đa tìm được cho đến nay, chúng ta lưu nó:

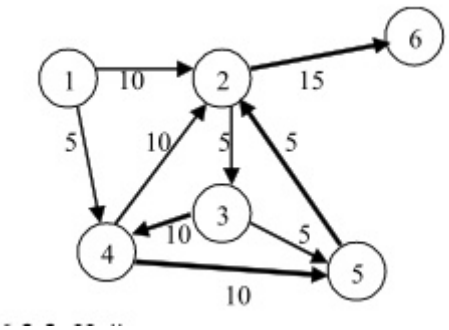
Sơ đồ thêm đỉnh

```
void themmotdinh(i) {
  if (<ta tìm theo đường dài>) {<ta lưu nó và chấm dứt nó>; }
  for (<mọi đỉnh k thừa kế của i>)
    if ((<đường không chứa đỉnh k>) &&
      (k là đỉnh kề của đỉnh tiếp theo trên đường>))
```

```
    {
      <Thêm đỉnh k vào đường>;
      <Thêm độ dài của cạnh mới vào độ dài của đường>;
      themmotdinh(i+1);
      /* Quay lui hồi quy */
      <Bỏ đánh dấu k là tham gia trong đường>;
      <Xóa đỉnh k từ đường>;
      <Trừ độ dài đường với độ dài cạnh mới thêm vào>;
    }
  }
}
```

Việc thực hiện theo sơ đồ trên. Tuy nhiên, nó không chỉ định đâu là các đỉnh bắt đầu. Rõ ràng là chỉ có đỉnh không có các cạnh đi trước thì không thể chọn làm điểm xuất phát (khi có các cạnh âm thì có thể bỏ sót một giải pháp tối ưu). Hay, việc tìm kiếm phải bắt đầu từ mọi đỉnh cao nhất có thể. Trong phần triển khai bên dưới, chúng ta sẽ gọi `addVertex()` tuần tự cho mỗi đỉnh của đồ thị.

Đồ thị được biểu diễn bằng ma trận lân cận `A[][]`, và các đầu vào mẫu được thể hiện trong Hình 6.4.




Hình 6.4. Đường dẫn đơn dài nhất trong một đồ thị có định hướng.

**Chương trình** 6.4. Tìm đường dài nhất (604longpath.c)

```
#include <stdio.h>
/* Số lượng lớn nhất của đỉnh */
#define MAXN 200
/* Số lượng đỉnh trong đồ thị*/
const unsigned n = 6;
```

```
/* Ma trận kề của đồ thị */
const char A[MAXN][MAXN] = {
  { 0, 10,  0,  5,  0,  0 },
  { 0,  0,  5,  0,  0, 15 },
  { 0,  0,  0, 10,  5,  0 },
  { 0, 10,  0,  0, 10,  0 },
  { 0,  5,  0,  0,  0,  0 },
  { 0,  0,  0,  0,  0,  0 }};

unsigned vertex[MAXN], savePath[MAXN];
char used[MAXN];
int maxLen, tempLen, si, ti;

void addVertex(unsigned i)
{ unsigned j, k;
  if (tempLen > maxLen) { /*Đa tìm đường đường dài nhất và lưu lại
    */
    maxLen = tempLen;
    for (j = 0; j <= ti; j++) savePath[j] = vertex[j];
    si = ti;
  }
  for (k = 0; k < n; k++) {
    if (!used[k]) { /*Nếu đỉnh k không tham gia vào đường đến hiệ
    n tại*/
       /* Nếu đỉnh như vậy thì thêm vào và là bên cạnh của đường
    cuối */
      if (A[i][k] > 0) {
        tempLen += A[i][k];
        used[k] = 1; /* đánh dấu k đã tham gia vào đường */
        vertex[++ti] = k; /* thêm đỉnh k vào đường*/
        addVertex(k);
        used[k] = 0; /* Quay lui hồi quy */
        tempLen -= A[i][k]; ti--;
      }
    }
  }
}

int main() {
  unsigned i;
```

```
    maxLen = 0; tempLen = 0; si = 0;  ti  = 1;
    for (i  = 0;  i  < n;  i++) used[i]  = 0;
    for (i  = 0;  i  < n;  i++) {
      used[i]  = 1;  vertex[0]  = i;
      addVertex(i);
      used[i]  = 0;
    }
    printf("Đường dài nhất là: \n");
    for (i  = 0;  i  < si;  i++) printf("%u ", savePath[i]  + 1);
    printf("\nvới độ dài chung %d\n", maxLen);
    return 0;
}
```

*Kết quả thực hiện chương trình:*

```
Cách dài nhất là:
3 4 5 2 6
với tổng chiều dài là 40.
```

## Bài tập

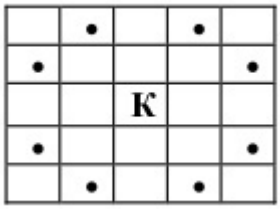
▷ **6.19.** Bài toán tìm đường đi đơn dài nhất trong đồ thị tuần hoàn có trọng số có định hướng có thay đổi về nguyên tắc không nếu chiều dài đường đi được tính bằng tích các trọng lượng của các sườn mà nó chứa. Và nếu chiều dài được xác định bởi trọng lượng của xương sườn tối thiểu (tối đa) liên quan đến đường?

▷ **6.20.** Bài toán tìm đường đi nguyên tố dài nhất trong đồ thị tuần hoàn có định hướng có thay đổi về nguyên tắc không, nếu độ dài đường đi được tính bằng tổng trọng số của các đỉnh mà nó chứa (chúng ta giả sử rằng đối với mỗi đỉnh thì một trọng số - a số tự nhiên).

▷ **6.21.** Sửa đổi chương trình trên để in tất cả các đường có độ dài tối đa.

### 6.3.4. Đường đi quân ngựa

Một tác vụ phổ biến minh họa tốt cho phương pháp tìm kiếm trả về được gọi là "đi bộ ngựa":



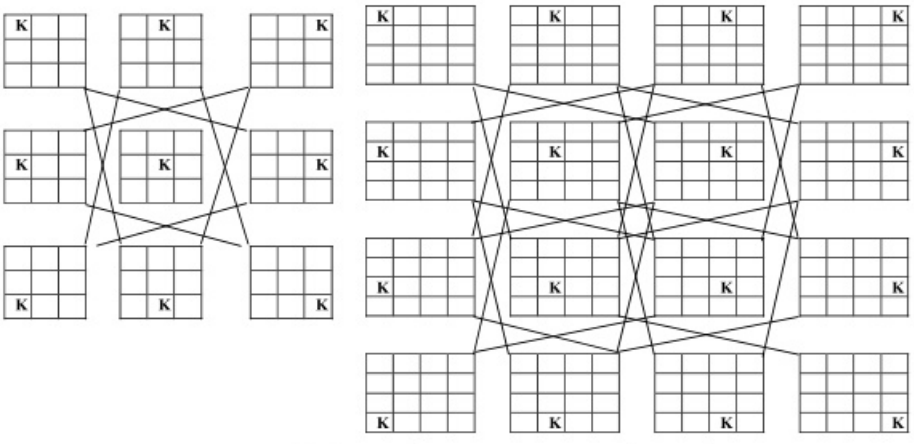
Hình 6.5. Vị trí quân ngựa được di chuyển.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | 22 | 7 | 44 | 39 | 24 | 9 | 28 | 63 |
| 7 | 43 | 40 | 23 | 8 | 45 | 62 | 25 | 10 |
| 6 | 6 | 21 | 42 | 59 | 38 | 27 | 64 | 29 |
| 5 | 41 | 58 | 37 | 46 | 61 | 54 | 11 | 26 |
| 4 | 20 | 5 | 60 | 53 | 36 | 47 | 30 | 51 |
| 3 | 57 | 2 | 35 | 48 | 55 | 52 | 15 | 12 |
| 2 | 4 | 19 | 56 | 33 | 14 | 17 | 50 | 31 |
| 1 | 1 | 34 | 3 | 18 | 49 | 32 | 13 | 16 |
| | *a* | *b* | *c* | *d* | *e* | *f* | *g* | *h* |

Hình 6.6. Lời giải bài toán với $n = 8$.

**Bài toán:** Cho một số tự nhiên $n$ ($n > 4$) và một bàn cờ tổng quát có kích thước $n \times n$ ô. Tìm một con bò trên bảng với đường đi của con ngựa. Mỗi ô trên bảng phải được truy cập chính xác một lần, với "chuyến tham quan" bắt đầu từ ô ở góc dưới bên trái. Nước đi cho phép của ngựa là nước cờ tiêu chuẩn, như trong Hình 6.5.

*Lời giải* của bài toán cho $n = 8$ được thể hiện trong Hình 6.6. Tại mỗi vị trí của con ngựa trên bàn cờ (tổng cộng là $n^2$) một đỉnh của đồ thị vô hướng có thể được so sánh. Một đường sườn trong đồ thị này sẽ nối hai đỉnh, nếu có thể chuyển đổi giữa các vị trí tương ứng bằng hành trình của con ngựa. Đây là những đồ thị tương tự cho $n = 3$ và $n = 4$ trông như thế nào (Hình 6.7):



Hình 6.7. Đồ thị bài toán với $n = 3$ và $n = 4$

Để tìm đường đi ngang cần thiết của bàn cờ, chúng ta phải tìm một đường đi trong đồ thị mà mỗi đỉnh đi đúng một lần, tức là đường đi của Hamilton. Vì vậy, chúng ta nhận được điều đó

"Bài toán nước đi quân ngựa" ⟨ "Đường Hamilton"

Hơn nữa, bài toán ngựa đi bộ có thể được rút gọn thành bài toán tìm đường đi Hamilton trong đồ thị trong đó hoành độ của mỗi đỉnh di bị giới hạn, trong trường hợp này là $d_i \leq 8$ (xem các bài toán 5, 7 và 8 ở cuối phần này).

Tất nhiên, chúng ta sẽ không giải quyết vấn đề với một đồ thị, thứ nhất, vì có một thuật toán rõ ràng hơn nhiều, và thứ hai, bởi vì một đồ thị với $n^2$ đỉnh như vậy sẽ yêu cầu khoảng $8.n^2$ bộ nhớ (để biểu diễn thông qua một danh sách các . thậm chí sẽ cần nhiều bộ nhớ hơn - xem Bảng **??**). Một lần nữa, chúng ta sẽ sử dụng tìm kiếm trả về, được mô tả bằng sơ đồ bởi hàm đệ quy sau:

Bước đi của quân ngựa

```
/* board[][] là bàn cờ cỡ n x n */
void BuocTiepTheo(x, y, i)
{ if (i == n*n) { /*in ra nghiệm bài toán*/ }
  board[x][y] = i;
  for (<mỗi bước có thể đi (u,v) của quân ngựa trên ô (x,y))
    if (0 == board[u][v]) /* Nếu ô đặt là trống */
      BuocTiepTheo(u, v, i+1);
  board[x][y] = 0; /*quay lui */
}
```

Trong quá trình thực hiện sau đó, một bản tóm tắt của bài toán được xem xét. Đầu tiên, chuyến tham quan có thể bắt đầu từ bất kỳ ô nào (đặt trước với tọa độ `startX` và `startY`). Ngoài ra, các nét cho phép đối với con ngựa có thể được xác định lại, tức là. chúng có thể không phải là những cái tiêu chuẩn được thể hiện trong Hình 6.5, mà là những cái khác ngẫu nhiên.

Các bước di chuyển hợp lệ cho ngựa được đặt ở đầu chương trình trong các mảng không đổi `diffX[]` và `diffY[]` - các giá trị trong đó có nghĩa là ngựa có thể di chuyển từ ô $(x, y)$ sang bất kỳ ô nào khác `(x + diffX [i], y + diffY [i])`, cho $i = 1, 2, ..., maxDiff$:

**Chương trình** 6.5. Bài toán các bước đi quan ngựa (605knight.c)

```
#include <stdio.h>
#include <stdlib.h>
/* Cỡ lớn nhất của bàn cờ */
#define MAXN 10
/* Số lớn nhất quy tắc di chuyển của quân ngựa*/
#define MAXD 10
/* Cỡ của bàn cờ thật */
const unsigned n = 6;
/* Vị trí bắt đầu */
const unsigned startX = 1;
const unsigned startY = 1;
/* Quy tắc di chuyển của quân ngựa*/
const unsigned maxDiff = 8;
const int diffX[MAXD] = { 1, 1, -1, -1, 2, -2, 2, -2 };
const int diffY[MAXD] = { 2, -2, 2, -2, 1, 1, -1, -1 };
unsigned board[MAXN][MAXN];
unsigned newX, newY;

void printBoard(void)
{ unsigned i, j;
  for (i = n; i > 0; i--) {
    for (j = 0; j < n; j++) printf("%3u", board[i-1][j]);
    printf("\n");
  }
  exit(0); /* thoát khỏi chương trình */
}

void nextMove(unsigned X, unsigned Y, unsigned i)
{ unsigned k;
  board[X][Y] = i;
  if (i == n * n)
    { printBoard(); return; }
  for (k = 0; k < maxDiff; k++) {
    newX = X + diffX[k]; newY = Y + diffY[k];
    if ((newX >= 0 && newX < n && newY >= 0 && newY < n) &&
        (0 == board[newX][newY]))
      nextMove(newX, newY, i + 1);
  }
  board[X][Y] = 0;
```

```
}

int main() {
  unsigned i, j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      board[i][j] = 0;
  nextMove(startX-1, startY-1, 1);
  printf("Bài toán không có nghiệm. \n");
  return 0;
}
```

*Kết quả thực hiện chương trình:*

```
10 27  6 19 16 25
 7 20  9 26  5 18
28 11  4 17 24 15
21  8 23 32  3 34
12 29  2 35 14 31
 1 22 13 30 33 36
```

*Lưu ý:* Bài toán đi ngang bàn cờ với một cú đánh của con ngựa thường được coi là một ví dụ kinh điển về việc áp dụng phương pháp tìm kiếm trả về, mặc dù nó không phải là *NP*-đầy đủ. vì nó có một nghiệm đa thức (theo sau đó là một nghiệm đa thức để tìm đường đi Hamilton trong dạng đặc biệt của đồ thị trong Hình 6.7). Thuật toán này (sẽ được thảo luận chi tiết trong Chương 9 - Thuật toán tham lam) thường dựa trên sơ đồ sau: đối với mỗi nước đi liên tiếp của ngựa, ô này được chọn, từ đó sẽ có ít cơ hội nhất cho nước đi tiếp theo .

Tuy nhiên, với những sửa đổi nhỏ cho bài toán, chúng ta có thể dễ dàng thấy mình rơi vào tình huống mà chúng ta không thể chắc chắn rằng bài toán có phải là đa thức hay không. Ví dụ như vậy là các bài toán sau:

## Bài tập

▷ **6.22.** Hãy sửa đổi chương trình trên để nó tìm thấy tất cả các lời giải.

▷ **6.23.** Hãy sửa đổi chương trình trên để nó tìm thấy tất cả các giải

pháp bất đối xứng khác nhau. Chúng ta sẽ xem xét các nghiệm đối xứng có thể nhận được từ nhau bằng cách quay hoặc bằng phép đối xứng về các đường ngang, đường thẳng đứng hoặc hai đường chéo chính của bàn cờ.

▷ **6.24.** Tìm số đường đi khác nhau của bàn cờ:

a) tất cả các đường duyệt qua

(b) tất cả các đường dfuyệt không đối xứng khác nhau

▷ **6.25.** Cho một số tự nhiên $n(n > 4)$ và một bàn cờ tổng quát có kích thước $n \times n$ ô. Tìm một con đường trên bảng với đường đi của con ngựa. Mỗi ô phải được truy cập chính xác một lần, với "chuyến tham quan" bắt đầu từ bất kỳ ô nào được đặt trước trên bàn cờ.

▷ **6.26.** Cho một số tự nhiên $n(n > 4)$ và một bàn cờ tổng quát có kích thước $n \times n$ ô được đưa ra, một số bị cấm. Tìm một con đường trên bảng với bước đi của con ngựa. Mỗi ô không bị cấm trên bàn cờ phải được truy cập chính xác một lần, với "chuyến tham quan" bắt đầu từ ô ở góc dưới bên trái.

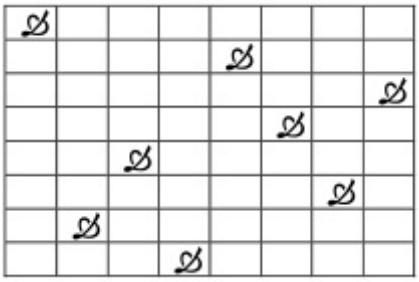
▷ **6.27.** Một số tự nhiên $n(n > 4)$ và một bàn cờ tổng quát có kích thước $n \times n$ ô được đưa ra. Tìm một con đường đi trên bảng với bước đi của quân ngựa với giao điểm tối thiểu của quỹ đạo quân ngựa.

▷ **6.28.** Xác định hạng của bài toán tìm đường đi Hamilton trong đồ thị có tung độ đỉnh giới hạn (tức là tung độ của mỗi đỉnh nhỏ hơn một số tự nhiên $k$ cố định).

▷ **6.29.** Tính toán đa thức bài toán 6.28 thành bài toán 6.26 có được không?

### 6.3.5. Bài toán tám quân Hậu

Bài toán đặt quân hậu trên bàn cờ là một bài toán kinh điển. Nó một lần nữa cho thấy tầm quan trọng của việc chặt cây đệ quy đúng cách đối với việc áp dụng hiệu quả phương pháp triệt tiêu hoàn toàn. Gauss đã xem xét bài toán này ngay từ năm 1850 và đã thành công chính xác do thực tế là với nghiên cứu phân tích, các ứng cử



Hình 6.8. Lời giải bài toán quân Hậu với $n = 8$

viên cho một giải pháp có thể giảm đáng kể. Do đó, mục tiêu của chúng ta sẽ là thu hẹp phạm vi nghiên cứu càng nhiều càng tốt và chỉ sau đó mới thực hiện vét cạn hoàn toàn.

**Bài toán:** Đặt $n$ quân Hậu trên một bàn cờ tổng quát có kích thước $n \times n$ ($n \geq 2$) sao cho không có hai quân nào bị tác động ăn quân (không nằm trong cùng một hàng ngang, dọc hoặc chéo).

Một lời giải khả thi của bài toán cho $n = 8$ được thể hiện trong Hình 6.8. Chúng ta sẽ giải quyết vấn đề bằng cách tìm kiếm với quay lui theo sơ đồ sau:

Chúng ta khởi tạo $i = 1$.

- Nếu chúng ta đã triển khai thành công các quân Hậu cho đến nay, chúng ta đang cố gắng tìm một vị trí "phù hợp" của quân Hậu (i + 1) (bước tiến).
- Nếu chúng ta đã đặt $n$ quân Hậu (tức là $i = n$) và chúng đáp ứng điều kiện của bài toán, thì sau đó chúng ta đã tìm ra lời giải, in ra, và cuối cùng dừng phép tính.
- Nếu không có vị trí trống cho quân hậu thứ $i$, chúng ta lùi lại một bước và tìm kiếm vị trí khác cho quân hậu (thứ $i - 1$).

Chúng ta sẽ làm rõ ý nghĩa của vị trí "thích hợp" ở trên. Một giải pháp không hiệu quả cho bài toán là thử tất cả các cách kết hợp có thể có $C_{n^2}^{n}$ để đặt các quân hậu trên bàn cờ và loại bỏ những kết hợp không phải là giải pháp cho bài toán. Vì chỉ có thể có nhiều nhất một quân Hậu trong mỗi hàng, nên chỉ có thể thử $n^n$ vị trí quân hậu khác nhau, một trong mỗi hàng và có thể tìm ra giải pháp cho chúng. Nếu chúng ta tính đến việc có thể có nhiều nhất một

quân Hậu trong một cột, thì số khả năng sinh ra giảm xuống còn $n!$. Chúng ta sẽ giới thiệu một mảng $n$ phần tử chứa số cột - ở vị trí thứ $i$ được viết số cột mà ở đó vị trí quân Hậu của hàng thứ $i$. Mỗi hoán vị như vậy sẽ xác định vị trí rõ ràng của các quân Hậu trên bàn cờ. Cấu hình xác minh có thể được giảm hơn nữa - chỉ các vị trí an toàn sẽ được chọn khi đặt từng quân Hậu kế tiếp, tức là những vị trí không ở cùng vị trí ngang, dọc hoặc chéo với quân Hậu đã được đặt. Đây là thuật toán được mô tả bằng sơ đồ trông như thế nào:

Đặt Hậu trên hàng

```
void DatHauHang(i) { /* Đặt Hậu hàng thứ i */
  if (i > n) { <In ra Lời giải>; }
  for (<mọi cột k quân Hậu đã chiếm>)
    if (<không có hậu trên đường chéo qua (i,k)>) {
    /* vị trí (i,k) không bị đe dọa */
    <Đặt Hậu i ở vị trí (i,k)>;
    DatHauHang(i+1);
    <loại bỏ hâu thứ i từ vị trí (i,k)>;
  }
}
```

Trong quá trình triển khai để đánh dấu và kiểm tra xem một vị trí có an toàn hay không, chúng ta sẽ sử dụng bốn mảng một chiều:

1) `queens[N]`: `queens[i]` chứa số lượng của trụ cột mà quân hậu của hàng thứ $i$ được đặt.

2) `col[N]`: `col[i]` có giá trị là 1 nếu không có quân hậu nào được đặt trong cột $i$ và 0 - ngược lại.

3) `RD[2*N-1]` và `LD[2*N-1)]` cho biết liệu có nữ hoàng trên đường chéo chính thứ k và đường chéo phụ tương ứng ($k = 1, 2, ..., 2n - 1$). Trong đó:

- `RD [i + k]` cho biết có quân hậu trong đường chéo chính đi qua ô (i, k) hay không.
- `LD [N + i-k]` cho biết có quân hậu trong đường chéo phụ đi qua ô (i, k) hay không.

Chương trình dưới đây kết thúc với một giải pháp. Chúng ta cho phép người đọc cố gắng sửa đổi nó để nó tìm và suy ra tất cả các giải pháp có thể cho một $n$ nhất định.

**Chương trình** 6.6. Bài toán tám quân Hậu (606queens.c)

```
#include <stdio.h>
#include <stdlib.h>
/* Kích thước lớn nhất bàn cờ */
#define MAXN 100
/* Kích thước bàn cờ */
const unsigned n = 13;

unsigned col[MAXN], RD[2*MAXN - 1],
LD[2*MAXN], queens [MAXN];

/* In ra vị trí các quân Hậu */
void printBoard()
{ unsigned i, j ;
  for (i = 0; i < n; i++) {
    printf("\n");
    for (j = 0; j < n; j++)
      if (queens[i] == j)
        printf("x ");
      else
        printf("o ");
  }
  printf("\n");
  exit(0);
}

/* Tìm vị trí tiếp theo để đặt Hậu */
void generate(unsigned i)
{ unsigned j;
  if (i == n) printBoard();
  for (j = 0; j < n; j++)
    if (col[j] && RD[i + j] && LD[n + i - j]) {
      col[j] = 0; RD[i + j] = 0; LD[n + i - j] = 0; queens[i] = j;
      generate(i + 1);
      col[j] = 1; RD[i + j] = 1; LD[n + i - j] = 1;
    }
}

int main() {
  unsigned i;
```

```
    for (i = 0; i < n; i++) col[i] = 1;
    for (i = 0; i < (2*n - 1); i++) RD[i] = 1;
    for (i = 0; i < 2*n; i++) LD[i] = 1;
    generate(0);
    printf("Bài toán không có nghiệm! \n");
    return 0;
}
```

*Kết quả thực hiện chương trình:*

```
x o o o o o o o o o o o o
o o x o o o o o o o o o o
o o o o x o o o o o o o o
o x o o o o o o o o o o o
o o o o o o o o x o o o o
o o o o o o o o o o o x o
o o o o o o o o o x o o o
o o o o o o o o o o o o x
o o o x o o o o o o o o o
o o o o o x o o o o o o o
o o o o o o o x o o o o o
o o o o o o o o o x o o o
o o o o o o x o o o o o o
```

Các tối ưu hóa bổ sung của thuật toán cũng có thể được thực hiện, tuy nhiên, điều này sẽ không làm giảm độ phức tạp $\Theta(n!)$ Của nó [Nakov-1998] [Reingold, Nivergelt, Deo-1980].

## Bài tập

▷ **6.30.** Đề xuất và thực hiện một thuật toán để giải một dạng bài toán sau: In tất cả các vị trí không đối xứng có thể có của các quân Hậu trên bảng. Hai giải pháp được coi là đối xứng nếu có thể nhận được một giải pháp từ phương pháp kia bằng cách xoay bàn cờ hoặc đối xứng về các hàng, cột, đường chéo chính hoặc phụ.

▷ **6.31.** Bài toán đối với trường hợp tổng quát của một bảng hình chữ nhật có hợp lý không (và nếu có, với những giải thích bổ sung nào)?

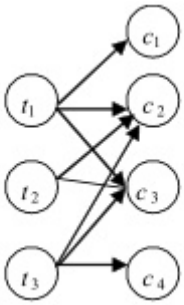
### 6.3.6. Thời khóa biểu của trường học

Lập lịch là một ví dụ khác về một bài toán được giải quyết bằng phương pháp tìm kiếm quay lui và vét cạn hoàn toàn. Nếu không xem xét chi tiết câu hỏi rộng này (người đọc có thể tìm thêm thông tin trong [Christofides-1975]), chúng ta sẽ cố gắng làm rõ nó nói chung bằng cách xem xét một phiên bản đơn giản của bài toán biên soạn chương trình giảng dạy ở trường học:

**Bài toán**: Lớp $C$ và $t$ giáo viên được đưa ra ($c, t$ - số tự nhiên). Số giờ `cl[i][j]` (số tự nhiên, $i = 1, 2, ..., c, j = 1, 2, ...t$) mà giáo viên $j$ phải thuyết trình cho lớp $i$ được biết. Để tìm một lịch trình cho chương trình học, trong đó các điều kiện sẽ được đáp ứng:

1) Trong mỗi bài học, một giáo viên có thể dạy nhiều nhất một lớp.

2) Tổng thời lượng của chương trình phải ở mức tối thiểu.




Hình 6.9. Lịch dạy học

Xét ví dụ sau: Có 4 lớp $c_1, c_2, c_3$ và $c_4$ và ba giáo viên $t_1, t_2$ và $t_3$. Các giờ sẽ được tổ chức là:
Giáo viên dạy $t_1$ mỗi lớp $c_1$-5 giờ, $t_1$ mỗi $c_2$ - 5 giờ, $t_1$ mỗi $c_3$ - 5 giờ,
Giáo viên dạy $t_2$ trong lớp $c_2$ - 5 giờ, $t_2$ trong $c_3$ - 5 giờ,
Giáo viên dạy $t_3$ mỗi lớp $c_2$ - 5 giờ, $t_3$ mỗi $c_3$ - 5 giờ, $t_3$ mỗi $c_4$ - 5 giờ.

Chúng ta đã trình bày cấu hình này theo sơ đồ với một đồ thị hai phần có định hướng trong Hình 6.9: Chúng ta chưa xác định trọng số của các sườn: đối với ví dụ đã chọn, tất cả các trọng số sẽ bằng 5. Với $t_i \rightarrow c_j(x)$, chúng ta nghĩa là giáo viên dạy bạn $x$ giờ của lớp $c_j$. Do đó, hai lịch trình sau đây có thể thực hiện được cho ví dụ đã

chọn:

*Lựa chọn 1.*

1) $t_1 \rightarrow c_2$ (5), $t_3 \rightarrow c_3$ (5), $t_2$ nghỉ vì cả hai lớp $c_2$ và $c_3$ đều bị chiếm.
2) $t_1 \rightarrow c_3$ (5), $t_3 \rightarrow c_2$ (5), $t_2$ nghỉ.
3) $t_2 \rightarrow c_2$ (5), $t_1 \rightarrow c_1$ (5), $t_3 \rightarrow c_4$ (5)
4) $t_2 \rightarrow c_3$ (5)

Chương trình đã đầy đủ và tổng thời lượng của nó là 20 giờ.

*Lựa chọn 2.*

1) $t_1 \rightarrow c_1$ (5), $t_2 \rightarrow c_2$ (5), $t_3 \rightarrow c_3$ (5)
2) $t_1 \rightarrow c_2$ (5), $t_2 \rightarrow c_3$ (5), $t_3 \rightarrow c_4$ (5)
3) $t_1 \rightarrow c_3$ (5), $t_3 \rightarrow c_2$ (5), $t_2$ nghỉ.

Đến đây chương trình hoàn thành trong 15 giờ. Có thể thấy rằng tổng thời lượng phụ thuộc vào trình tự dạy học.

Cả hai lựa chọn được thể hiện trong Bảng **??**.

Phương án 1.

| | 1-5 | 6-10 | 11-15 | 16-20 |
|---|---|---|---|---|
| $t_1$ | $c_2$ | $c_3$ | $c_1$ | × |
| $t_2$ | × | × | $c_2$ | $c_1$ |
| $t_3$ | $c_3$ | $c_2$ | $c_4$ | × |

Phương án 2.

| | 1-5 | 6-10 | 11-15 |
|---|---|---|---|
| $t_1$ | $c_2$ | $c_3$ | $c_1$ |
| $t_2$ | $c_2$ | $c_3$ | × |
| $t_3$ | $c_3$ | $c_4$ | $c_2$ |

Bảng 6.2. Các phương án lịch

Như một điểm khởi đầu để giải quyết vấn đề, chúng ta sẽ xem xét thuật toán tổ hợp sau: Để tìm một chương trình có thời lượng tối thiểu, chúng ta sẽ xây dựng và đánh giá tất cả các lịch trình có thể có. Hãy xem bảng sau (Bảng 6.3):

Bảng cho biết việc làm của mỗi giáo viên trong mỗi giờ, tức là trong ô của hàng thứ i, cột thứ j sẽ ghi lớp học mà giáo viên thứ i sẽ dạy trong giờ thứ j. Chúng ta sẽ điền vào các cột của bảng một cách

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | ⋯ |
|---|---|---|---|---|---|---|---|---|---|
| $t_1$ | | | | | | | | | |
| $t_2$ | | | | | | | | | |
| ⋯ | | | | | | | | | |

Bảng 6.3. Bảng phân công

nhất quán với tất cả các tổ hợp lớp có thể có để đáp ứng các điều kiện của bài toán:

- Tại một thời điểm, mỗi giáo viên có thể dạy nhiều nhất một lớp (nghĩa là không thể có sự lặp lại các lớp trong một cột).
- Mỗi giáo viên nên dành chính xác bao nhiêu giờ cho lớp học tương ứng theo điều kiện quy định.

Ví dụ, đối với *Lựa chọn 2* ở trên, bảng sẽ được điền như trong Bảng 6.4.

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $t_1$ | $c_1$ | $c_1$ | $c_1$ | $c_1$ | $c_1$ | $c_2$ | $c_2$ | $c_2$ | $c_2$ | $c_2$ | $c_3$ | $c_3$ | $c_3$ | $c_3$ | $c_3$ |
| $t_2$ | $c_2$ | $c_2$ | $c_2$ | $c_2$ | $c_2$ | $c_3$ | $c_3$ | $c_3$ | $c_3$ | $c_3$ | × | × | × | × | × |
| $t_3$ | $c_3$ | $c_3$ | $c_3$ | $c_3$ | $c_3$ | $c_4$ | $c_4$ | $c_4$ | $c_4$ | $c_4$ | $c_2$ | $c_2$ | $c_2$ | $c_2$ | $c_2$ |

Bảng 6.4. Bảng phân công

Quyết định điền vào một bảng như vậy sẽ không hiệu quả, vì chúng ta tạo ra một lịch trình, xem xét từng giờ riêng biệt. Trong ví dụ này, tất cả giáo viên dành 5 giờ - vì vậy chúng ta có thể tạo lịch trình cho mỗi 5 giờ liên tục (Bảng **??**):

| | 5 | 10 | 15 | 20 | ⋯ |
|---|---|---|---|---|---|
| $t_1$ | | | | | |
| $t_2$ | | | | | |
| ⋯ | | | | | |

Bảng 6.5. Bảng phân công

Trong trường hợp chung, vì giờ của mỗi giáo viên là số tự nhiên ngẫu nhiên nên chúng ta sẽ thực hiện như sau ở mỗi bước:

- Ta tính $mX = \min(cl[i][j])$, với mỗi $i = 1, 2, ..., c, j = 1, 2, ..., t, cl[i][j] \neq 0$.
- Chúng ta tạo ra tất cả các kết hợp có thể có của các bài tập trên lớp cho mỗi giáo viên trong khoảng thời gian $mX$ giờ: mỗi giáo viên $t_i \rightarrow$ lớp $c_j$, với mỗi $i = 1, 2, ..., t$, và đối với mỗi bài tập, chúng ta sẽ giảm đi $mX$ giờ $cl[i][j]$.

Quá trình tạo tiếp tục cho đến khi tất cả các giá trị của $cl[i][j]$ bằng 0 - khi đó sẽ có một trường hợp giới hạn trong đó nó được tính toán thời lượng chương trình kéo dài và nếu số kết quả nhỏ hơn thời lượng nhỏ nhất được tìm thấy cho đến nay , lịch trình được duy trì.

Chúng ta sẽ triển khai thuật toán dưới dạng một hàm `generate(teacher, level, mX,totalHours)` với 4 tham số: biến `teacher` chứa số giáo viên mà một lớp sẽ được chỉ định, $mX$ là số giờ và `totalHours` - tổng thời lượng của chương trình cho đến nay. Đối với mỗi mức (dòng điện được giữ ở `level`), đầu tiên $mX$ được tính theo thuật toán mô tả ở trên, sau đó `generate(0, level+1, mX, totalHours+mX)` . Điều này có nghĩa là trong tương lai, một lớp học phải được chỉ định cho mỗi giáo viên để dạy số giờ $mX$. Việc giao lớp $k$ cho giáo viên được thực hiện theo thuật toán cho các tổ hợp không lặp lại (xem **??**). Sau khi phân công, số giờ dạy `cl[k][teacher]` giảm đi $mX$, và khi một lớp được chỉ định cho mỗi giáo viên ( `teacher == N`), lớp đó sẽ được chuyển sang cấp độ tiếp theo và toàn bộ quá trình được lặp lại từ đầu - $mX$ được tính toán lại, v.v. n. Phần cuối của đệ quy đạt được khi không còn giờ dạy - sau đó nó được kiểm tra xem `totalHours < minimal` (thời lượng của chương trình mới được tạo có nhỏ hơn thời lượng tối thiểu cho đến thời điểm này hay không) và nếu hoàn thành, lịch trình sẽ được lưu. . Đệ quy cũng bị cắt:

```
if (totalHours > = Minimum) return;
```

nghĩa là, nếu thời lượng của chương trình được tạo một phần trở nên dài hơn thời lượng tối thiểu được tìm thấy cho đến nay, thì việc tiếp tục tạo sẽ không có ý nghĩa gì.

**Chương trình** 6.7. Lập thời khóa biểu (607program.c)

```
#include <stdio.h>
/* Số giáo viên lớn nhất */
#define MAXT 100
/* Số lới lớn nhất */
#define MAXC 100
/* Số giáo viên */
const unsigned t = 3;

/*Số lớp */
const unsigned c = 4;
unsigned cl[MAXC][MAXT] = {
  { 5, 5, 5 }, /* thông tin lớp 1 */
  { 5, 5, 5 },
  { 5, 0, 0 },
  { 0, 0, 5 } /* thông tin lớp C */
};

const unsigned MAX_VALUE = 20000;

char usedC[100][MAXC];
unsigned teach[MAXT], minimal;

void generate(unsigned teacher, unsigned level,
              unsigned mX, unsigned totalHours)
  { unsigned i, j;
    if (totalHours >= minimal) return;
    if (teacher == t) {
      unsigned min = MAX_VALUE;
      for (i = 0; i < c; i++)
        for (j = 0; j < t; j++)
          if (cl[i][j] < min && 0 != cl[i][j]) min = cl[i][j];
      if (min == MAX_VALUE) {
        if (totalHours < minimal) minimal = totalHours;
      }
      else {
        generate(0, level + 1, min, totalHours + min);
      }
      return;
  }
```

```
  /* Xác định lớp cho giáo viên dạy, mà sử dụng giờ nhỏ nhất*/
  for (i = 0; i < c; i++) {
    if (cl[i][teacher] > 0 && !usedC[level][i]) {
      cl[i][teacher] -= mX;
      usedC[level][i] = 1;
      generate(teacher + 1, level, mX, totalHours);
      usedC[level][i] = 0; /* quay lui */
      cl[i][teacher] += mX;
    }
  }

int main() {
  unsigned i, j;
  for (i = 0; i < 100; i++)
    for (j = 0; j < c; j++) usedC[i][j] = 0;
  minimal = MAX_VALUE;
  generate(t, 0, 0, 0);
  printf("Lịch trình tối thiểu là %u giờ.\n", minimal);
  return 0;
}
```

*Kết quả thực hiện chương trình:*

```
Lịch trình tối thiểu là 15 giờ.
```

## Bài tập

▷ **6.32.** Sửa đổi chương trình trên sao cho in bản phân phối để dạy học cùng với thời lượng tối thiểu của chương trình.

▷ **6.33.** Độ phức tạp của thuật toán được đề xuất trên là gì?

### 6.3.7. Dịch mật mã

Mã Morse thường được sử dụng, mặc dù cũ hơn một chút, như một phương tiện liên lạc - trong điện báo, tàu thủy, để liên lạc giữa các địa điểm quân sự và những nơi khác. Trong đó, mỗi ký hiệu (chữ cái, số, v.v.) được biểu diễn dưới dạng một chuỗi các dấu chấm và dấu gạch ngang. Do đó, biểu tượng có thể được truyền bằng tín hiệu radio (hoặc ánh sáng) ngắn hơn và dài hơn tương ứng với dấu chấm và dấu gạch ngang. Điều quan trọng là quá trình truyền giữa

các ký tự phải được tạm dừng, nếu không, bản dịch ngược lại rất mơ hồ, ví dụ:

Chữ "A" tương ứng với "01" (0 sẽ chỉ dấu chấm, và 1 - dấu gạch ngang), chữ "M" tương ứng với "11" và chữ "J" - "0111". Do đó, nếu nhận được dãy mã "0111", chúng ta sẽ không biết làm thế nào để giải mã nó - là "J" hoặc là "AM".

Bài toán chúng ta sẽ xem xét như sau: Hai bảng chữ cái được đưa ra và đối với mỗi chữ cái của một bảng chữ cái, chúng ta biết chữ cái nào của bảng kia tương ứng với nó ("các chữ cái" của cả hai bảng chữ cái có thể đại diện cho nhiều hơn một ký hiệu). Ví dụ: nếu các ký hiệu tạo thành các chữ cái của bảng chữ cái đầu tiên là 0 và 1, và của thứ hai - $a, b, c, d$, thì chúng ta có thể xem xét hai bảng chữ cái sau:

1) "0", "1", "01", "10"

2) "$a$", "$b$", "$c$", "$d$"

Tất cả các chuỗi này được coi là các chữ cái và mỗi chữ cái của một bảng chữ cái được so sánh duy nhất với một chữ cái khác, ví dụ:

"0" $\Leftrightarrow$ "$a$"
"1" $\Leftrightarrow$ "$b$"
"01" $\Leftrightarrow$ "$c$"
"10" $\Leftrightarrow$ "$d$"

Theo một từ (chuỗi các chữ cái) từ một bảng chữ cái, chúng ta đang tìm kiếm bản dịch sang ngôn ngữ của bảng chữ cái kia. bài toán phức tạp là chúng ta không có dấu phân cách giữa các chữ cái riêng lẻ. Do đó, chuỗi "0110" có thể được dịch theo một số cách: là "$abba$", "$cd$" và các cách khác. Tất cả các bản dịch có thể có của một từ đều được tìm kiếm.

Bài toán được đặt ra đòi hỏi vét canh hoàn toàn các khả năng, vì vậy chúng ta sẽ áp dụng tìm kiếm quay lại. Hãy để từ mà chúng ta đang dịch được viết trong chuỗi `str1`. Chúng ta sẽ sử dụng tham số `count`, cho biết ký hiệu mà chúng ta đã đạt được cho đến nay (số đếm ban đầu có giá trị là 0). Chúng ta tìm thấy một chuỗi con bắt đầu bằng số đếm, phải là một chữ cái của bảng chữ cái đầu tiên.

Có thể tìm thấy một số chữ cái như vậy (ví dụ, ở trên trong chuỗi "0110", chúng ta sẽ tìm thấy các chữ cái "0" và "01"). Đối với mỗi ký tự như vậy, chúng ta kiểm tra đệ quy (dịch) phần còn lại của chuỗi sau khi loại trừ nó. Trên thực tế, chúng ta sẽ không sửa đổi chuỗi trong thực tế, nhưng sẽ tăng số lượng theo độ dài của ký tự tìm được. Nếu tại một thời điểm nào đó giá trị của số đếm trở nên bằng độ dài của chuỗi, thì có một trường hợp giới hạn - chúng ta đã nhận được một bản dịch khả dĩ.

Trong quá trình thực hiện, dữ liệu đầu vào được khởi tạo trong hàm `initLanguage()`. Số chữ cái là $N$, trường của mảng `transf[i].st1` chứa chữ cái thứ $i$ $(1 \leq i \leq n)$ của bảng chữ cái đầu tiên và nó tương ứng với chữ cái thứ $i$ của bảng chữ cái thứ hai, được viết trong trường `transf [i].st2`. Khi thực hiện phép dịch, trong mảng `int translate[]`, chúng ta chỉ viết số của các chữ cái đã dịch và nếu chúng ta đi đến quyết định, chúng ta `in` các chữ cái của bảng chữ cái thứ hai tương ứng với các số này. Trong chương trình, chúng ta đã sử dụng bảng chữ cái tiếng Latinh và các số từ 0 đến 9 làm dữ liệu đầu vào mẫu. Bảng chữ cái thứ hai bao gồm mã Morse tương ứng cho mỗi ký hiệu.

**Chương trình** 6.8. Các bản dịch mật mã (608translat.c)

```
#include <stdio.h>
#include <string.h>
#define MAXN 40 /* Số lớn nhất tương ứng giữa các chữ cái */
#define MAXTL 200 /* Độ dài lớn nhất của từ để dịch*/
/* Số lượng tương ứng*/
const unsigned n = 38;
/* Từ phải dịch */
char *str1 = "101001010";

struct transType {
  char *st1;
  char *st2;
};
struct transType transf[MAXN];

unsigned translation[MAXTL], pN, total = 0;

* Ví dụ dùng bảng morse: 0 là tạch, còn 1 là tè */
```

```
void initLanguage(void)
{  transf[0].st1 = "A"; transf[0].st2 = "01";
transf[1].st1 = "B"; transf[1].st2 = "1000";
transf[2].st1 = "C"; transf[2].st2 = "011";
transf[3].st1 = "D"; transf[3].st2 = "110";
transf[4].st1 = "E"; transf[4].st2 = "100";
transf[5].st1 = "F"; transf[5].st2 = "0";
transf[6].st1 = "G"; transf[6].st2 = "0001";
transf[7].st1 = "H"; transf[7].st2 = "1100";
transf[8].st1 = "I"; transf[8].st2 = "00";
transf[9].st1 = "K"; transf[9].st2 = "0111";
transf[10].st1 = "L"; transf[10].st2 = "101";
transf[11].st1 = "M"; transf[11].st2 = "0100";
transf[12].st1 = "N"; transf[12].st2 = "11";
transf[13].st1 = "O"; transf[13].st2 = "10";
transf[14].st1 = "P"; transf[14].st2 = "111";
transf[15].st1 = "Q"; transf[15].st2 = "0110";
transf[16].st1 = "R"; transf[16].st2 = "010";
transf[17].st1 = "S"; transf[17].st2 = "000";
transf[18].st1 = "T"; transf[18].st2 = "1";
transf[19].st1 = "U"; transf[19].st2 = "001";
transf[20].st1 = "V"; transf[20].st2 = "0010";
transf[21].st1 = "X"; transf[21].st2 = "0000";
transf[22].st1 = "Y"; transf[22].st2 = "1010";
transf[23].st1 = "Z"; transf[23].st2 = "1110";
transf[24].st1 = "W"; transf[24].st2 = "1111";
transf[25].st1 = "Â"; transf[25].st2 = "1101";
transf[26].st1 = "Ă"; transf[26].st2 = "0011";
transf[27].st1 = "Ê"; transf[27].st2 = "0101";
transf[28].st1 = "1"; transf[28].st2 = "01111";
transf[29].st1 = "2"; transf[29].st2 = "00111";
transf[30].st1 = "3"; transf[30].st2 = "00011";
transf[31].st1 = "4"; transf[31].st2 = "00001";
transf[32].st1 = "5"; transf[32].st2 = "00000";
transf[33].st1 = "6"; transf[33].st2 = "10000";
transf[34].st1 = "7"; transf[34].st2 = "11000";
transf[35].st1 = "8"; transf[35].st2 = "11100";
transf[36].st1 = "9"; transf[36].st2 = "11110";
transf[37].st1 = "0"; transf[37].st2 = "11111";
}
```

```
/* In bản dịch */
void printTranslation(void)
{ unsigned i;
  total++;
  for (i = 0; i < pN; i++)
    printf("%s", transf[translation[i]].st1);
  printf("\n");
}

/* tìm chữ cái tiếp sau */
void nextLetter(unsigned count)
{ unsigned i, k;
  if (count == strlen(str1)) { printTranslation(); return; }
  for (k = 0; k < n; k++) {
    unsigned len = strlen(transf[k].st2);
    for (i = 0; i < len; i++)
      if (str1[i + count] != transf[k].st2[i]) break;
    if (i == len) {
      translation[pN++] = k;
      nextLetter(count + strlen(transf[k].st2));
      pN--;
    }
  }
}

int main() {
  printf("Danh sách tất cả khả năng dịch: \n");
  initLanguage();
  pN = 0;
  nextLetter(0);
  printf("Số lượng chung cách dịch khác nhau: %u \n", total);
  return 0;
}
```

## Bài tập

▷ **6.34.** Sửa đổi chương trình trên để tìm bản dịch có độ dài tối thiểu. Những điều kiện nào để bỏ cây đối với các ứng cử viên cho quyết định có thể được áp dụng?

## 6.4. Phương pháp nhánh và ranh giới

Một trường hợp đặc biệt của phương pháp tìm kiếm quay lui là phương pháp các nhánh và các đường biên. Nó áp dụng cho các bài toán áp dụng hai điều kiện sau:

1) Mỗi ứng cử viên cho một quyết định được so sánh giá trị - *giá cả* và mục tiêu là tìm ra *giải pháp tối ưu* (ví dụ: giải pháp có giá tối thiểu).

2) Mỗi giải pháp của bài toán phải có thể được trình bày dưới dạng thu được từ các phần mở rộng liên tiếp của các giải pháp từng phần với giá tăng đơn điệu.

Trong Chương 5 (xem 5.4.4.) Chúng ta đã giải quyết vấn đề của khách du lịch thương mại bằng phương pháp phân nhánh và ranh giới. Trong bài toán này, cả hai điều kiện đã được đáp ứng:

1) Mỗi lần từ đồ thị (và cụ thể là chu trình Hamilton), một giá được so sánh - tổng trọng lượng của các sườn mà nó chứa. Chúng ta đang tìm kiếm một chu kỳ Hamilton với giá tối thiểu.

2) Vì chúng ta đã đưa ra hạn chế bổ sung là trọng số của các cạnh của đồ thị phải là số dương nên điều kiện thứ hai cũng được đáp ứng: Giá của mỗi nghiệm riêng trong đó các đỉnh tham gia $(v_1, v_2, ..., v_k)$ nhỏ hơn giá của phần mở rộng tùy ý của nó $(v_1, v_2, ..., v_k, v_{k+1})$, thu được bằng cách thêm một đỉnh khác.

Điều quan trọng đối với phương pháp nhánh và đường biên là cây đệ quy mà người ứng viên xin quyết định thu được có thể bị "cắt" trên cơ sở một nguyên tắc khác: Nếu một giải pháp từng phần có giá lớn hơn hoặc bằng giá trị nhỏ nhất được tìm thấy vào thời điểm hiện tại , không có ý nghĩa gì khi mở rộng thêm, vì điều này chắc chắn sẽ không dẫn đến một giải pháp tốt hơn. Chúng ta đã thực hiện một phần tương tự về những quyết định vô vọng trong bài toán dành cho khách du lịch thương mại. Chúng ta sẽ xem xét một bài toán *NP*-đầy đủ khác và biến thể của nó, trong đó có thể tìm lời giải bằng phương pháp nhánh và đường biên.

### 6.4.1. Bài toán ba lô (lựa chọn tối ưu)

**Bài toán 1.** Cho một ba lô có khối lượng (độ chứa) $M$ kg. Cho $N$ vật, mỗi vật nặng $m_i (1 \leq i \leq N)$. Kiểm tra xem có một tập hợp con các đồ vật lấp đầy ba lô chính xác không, tức là tổng trọng lượng của chúng chính xác là $M$.

**Bài toán 2.** (phiên bản tối ưu hóa tổng quát của bài trước) Một ba lô có thể chứa $M$ kg được đưa ra. Cho trước $N$ vật, mỗi vật có khối lượng $m_i$ và giá trị (giá) $c_i$. Tìm một tập hợp con các món có tổng chi phí lớn nhất để đựng vừa ba lô, tức là tổng khối lượng của chúng nhỏ hơn hoặc bằng $M$.

Trong một số trường hợp đặc biệt, ví dụ khi trọng số là số nguyên và có đủ bộ nhớ $(\Theta(M.N))$, các thuật toán hiệu quả có thể được áp dụng để giải các bài toán trên dựa trên nguyên tắc tối ưu động (Chương 8). Tuy nhiên, trong trường hợp tổng quát, khi giá và trọng số là số thực, hai bài toán $NP$-đầy đủ này được giải quyết triệt để. Chúng ta sẽ xem xét thứ hai trong số họ và giải quyết nó bằng phương pháp các nhánh và ranh giới.

Chúng ta sẽ tạo ra tất cả các giải pháp có thể bằng cách tìm kiếm có trả lại và trong quá trình tạo, chúng ta sẽ giữ lại giải pháp tối ưu. Trong bài toán thứ hai, một tập hợp các đối tượng được đưa ra. Từ tất cả $2^N$ tập con của nó, ta phải chọn một tập thỏa mãn điều kiện của bài toán: tổng trọng lượng của các vật trong đó không được vượt quá $M$ và tổng giá trị của chúng phải lớn nhất.

Chúng ta hãy xem xét những điểm chính trong việc lập thuật toán. (Để đơn giản hóa ký hiệu, chúng ta sẽ xem $c_i$ và $m_i$ là các mảng $c[i]$ và $m[i]$, tương ứng.)

Ta có một nghiệm riêng (tập hợp con của các đối tượng) trong đó các đối tượng $A = a_1, a_2, ..., a_k$ được lấy. Theo $S(a_1, a_2, ..., a_k)$ chúng ta có nghĩa là tổng $S(a_1, a_2, ..., a_k) = m[a_1] + m[a_2] + ... + m[a_k]$ bằng trọng lượng của chúng. Đối với tập hợp con $A$, chúng ta cố gắng thêm một đối tượng mới $a_{k+1}$ trong số những đối tượng chưa được lấy cho đến nay:

- Nếu $S(a_1, a_2, ..., a_k) + m[a_{k+1}] \leq M$ thì sau đó phép cộng đối tượng là có thể và ta tiếp tục khai triển đệ quy tập con A theo mọi cách có thể. .

- Nếu không tồn tại đối tượng $a_{k+1}$ sao cho $S(a_1, a_2, ..., a_k) + m[a_{k+1}] \leq M$ thì theo đó tập $a_1, a_2, ..., a_k$ là một giới hạn (tức là không thể mở rộng thêm nữa).

Khi đó giá trị của nó $c[A] = c[a_1] + c[a_2] + \cdots + c[a_k]$ được tính và nếu nó lớn hơn giá trị lớn nhất tìm được cho đến nay, nó được giữ nguyên:

Sơ đồ chọn đồ vật

```
void generate(i) {
  if (<nếu trong lượng chung của vật đã chọn > M>) return;
  if (i == N) {
    /* trường hợp cận biên, kiểm tra có giải pháp nào tốt hơn*/
    /* trong số tối đa được tìm thấy cho đến nay */
    return;
  }
  <lấy vật thứ i>;
  generate(i+1);
  <loại bỏ vật thứ i>;
  generate(i+1); /* nghĩa là chọn vật thứ i được */
}
```

Trong thực tế, chúng ta đã xác định khi nào một giải pháp từng phần có thể được mở rộng. Vì vậy, chúng ta đảm bảo rằng không phải tất cả 2 tập con $N$-có thể được xem xét, mà chỉ những tập mà tổng trọng lượng của các đối tượng nhỏ hơn hoặc bằng $M$. Tuy nhiên, điều này không kết thúc vấn đề cắt cây đệ quy của các ứng cử viên.

Trong trường hợp của bài toán người bán hàng, nơi tìm kiếm một chu trình Hamilton tối thiểu, một giải pháp từng phần được tuyên bố là vô vọng nếu chiều dài của con đường đang được xây dựng trở nên lớn hơn mức tối thiểu được tìm thấy cho đến nay. Khi tìm giải pháp tối đa, như trong bài toán chọn tối ưu, kiểu cắt này phải được thay đổi: Trong quá trình xây dựng một tập hợp con các đối tượng (như trong quy trình giản đồ trên) chúng ta đã lấy các đối tượng $A = \{a_1, a_2, ..., a_k\}$, và chúng ta đã bỏ qua các đối tượng $B = \{b_1, b_2, ..., b_p\}$. Với $VT$, chúng ta sẽ biểu thị tổng các giá trị của tất cả các đối tượng, và với $Vmax$, chúng ta sẽ biểu thị giá trị lớn nhất thu được từ các giải pháp đã thử nghiệm cho đến nay. Sau đó

nếu

$$VT - c[b_1] - c[b_2] - ... - c[b_p] \leq Vmax,$$

theo đó là không có ích lợi gì khi mở rộng giải pháp hiện tại thêm nữa. Điều này là do ngay cả khi có thể bao gồm tất cả các mục khác, giá trị tối đa mà chúng ta sẽ nhận được là:

(giá trị của các mục đã lấy cho đến nay) + (giá trị còn lại) =

$$= (c[a1] + ... + c[ak]) + (VT - (c[a1] + ... + c[ak] + c[b1] + ... + c[bp])) =$$

$$= VT - c[b1] - c[b2] - ... - c[bp],$$

trong khi đó, nếu biểu thức sau nhỏ hơn $Vmax$, thì giải pháp hiện tại là vô vọng và việc xem xét thêm sẽ không dẫn đến kết quả tốt hơn. Sau đây là một lược đồ chi tiết hơn của hàm đệ quy `generate()`:

Hàm đệ quy generate()

```
float Ttemp; /* Trọng số chung của các vật đã chọn*/
float Vtemp; /* Giá chung của các vật đã chọn*/
float VmaX; /* Giá cực đại của lời giải được chọn cho đến nay*/
float totalV; /* Giá chung của vật còn lại*/

void generate(unsigned i)
{ if (Ttemp > K) return;
  if (Vtemp + totalV < VmaX) return; /* giải pháp vô vọng */
  if (i == N) {
    if (Vtemp > VmaX) {
      <lưu lời giải>;
      VmaX = Vtemp;
  }
  return;
  }
  Vtemp += c[i]; Ttemp += m[i]; totalV -= c[i]; /*lấy vật i */
  generate(i+1);
  Vtemp -=c [i]; Ttemp -= m[i]; /* loại vật i */
  generate(i+1); /* tức là khi lấy đối tượng thứ i bị "bỏ qua" */
  totalV += c[i];
}

int main() {
  totalV = c[1] + c[2] + ... + c[N];
```

```
  generate(0);
  return 0;
}
```

Để có được một chương trình làm thật, chúng ta cũng phải chỉ rõ cách chúng ta sẽ thực hiện việc đưa vào và loại trừ chủ thể thứ $i$. Với mục đích này, chúng ta sẽ nhập `token[]` và một biến $tN$, hiển thị số phần tử của nó. Chúng ta sẽ lưu giải pháp tốt nhất được tìm thấy cho đến nay trong mảng `saveTaken[]`, với phần tử $sn$. Khi kết thúc hàm `main()`, chúng ta sẽ in ra giải pháp tối ưu tìm được. Dữ liệu đầu vào `N, M, c [N]` và `m[N]` được đặt là hằng số ở đầu chương trình.

**Chương trình** 6.9. Bài toán ba lô (609bagrec.c)

```
#include <stdio.h>
#define MAXN 100

const unsigned n = 10;
const double M = 10.5;
const double c[MAXN] = {10.3,9.0,12.0,8.0,4.0,8.4,9.1,17.0,6.0,9.7};
const double m[MAXN] = {4.0,2.6,3.0,5.3,6.4,2.0,4.0,5.1,3.0,4.0};

unsigned taken[MAXN], saveTaken[MAXN], tn, sn;
double VmaX, Vtemp, Ttemp, totalV;

void generate(unsigned i)
{ unsigned k;
  if (Ttemp > M) return;
  if (Vtemp + totalV < VmaX) return;
  if (i  == n) {
    if (Vtemp > VmaX) { /* Giữ nghiệm tối ưu */
      VmaX = Vtemp; sn = tn;
      for (k = 0;  k < tn;  k++)
        saveTaken[k] = taken[k];
    }
    return;
  }

  taken[tn++] = i;  Vtemp += c[i]; totalV  -= c[i];  Ttemp += m[i];
  generate(i  + 1);
```

```
    tn--; Vtemp -= c[i]; Ttemp -= m[i];
    generate(i + 1);
    totalV += c[i];
  }

  int main() {
    unsigned i;
    tn = 0; VmaX = 0; totalV = 0;
    for (i = 0; i < n; i++)
      totalV += c[i];
    generate(0);
    printf("Giá lớn nhất: %.2lf\nChọn vật: \n", VmaX);
    for (i = 0; i < sn; i++)
      printf("%u ", saveTaken[i] + 1);
    printf("\n");
    return 0;
  }
```

*Kết quả thực hiện chương trình:*

```
Giá tối đa: 37,40
Các mục đã chọn:
3 6 8
```

### Bài tập

▷ **6.35.** Hãy biên dịch và thực hiện một bài toán giải thuật toán 1.

▷ **6.36.** Sửa đổi chương trình trên để tìm tất cả các giải pháp với giá tối đa.

## 6.5. Các chiến lược tối ưu cho trò chơi

Ví dụ về cờ vua, mà chúng ta đã đưa ra ở đầu chương, chúng ta sẽ xem xét thêm hai lần nữa - ở đây và trong 6.5.2. Trong 6.3.4 chúng ta đã chỉ ra một cách khá sơ đồ về cách một cái cây có thể được xây dựng dựa trên tất cả các loại chuyển động. Bây giờ, khi nhìn, chúng ta sẽ không sử dụng một cái cây, mà là một đồ thị có định hướng. Mỗi cấu hình cờ vua có thể sẽ được biểu diễn bằng một đỉnh của đồ thị $G(V, E)$. Cạnh $(i, j) \in E$ khi và chỉ khi của cấu hình tương ứng

với $i$, cấu hình $j$ có thể nhận được bằng cách thực hiện một bước di chuyển. Chúng ta sẽ xác định một số khái niệm:

- Cấu hình cờ vua được gọi là thiết bị đầu cuối nếu trò chơi kết thúc (quân đen, quân trắng hoặc hòa. Trong cờ vua, một trận hòa xảy ra khi:
  - *nam* - không thể có nước đi nào trước người chơi đang đến lượt chơi.
  - *bị chiếu vĩnh viễn* - lặp lại cùng một nước đi ba lần (thường bị ép bằng cờ vua của một trong các đối thủ, người đang tìm kiếm một kết quả hòa).
  - *không có khả năng chiến thắng* của một trong hai đối thủ (có quá ít và / hoặc quân cờ yếu / không thể cơ động trên bàn cờ).
  - *thỏa thuận hòa* giữa các kỳ thủ (kể cả trường hợp "chủ quan" này cũng không nên bỏ qua và có giá trị trong máy tính nhận thức cờ vua).

  Khi có cấu hình đầu cuối, đỉnh tương ứng của đồ thị không có người thừa kế.
- *Cấu hình không đầu cuối* được gọi là *chiến thắng* nếu có ít nhất một người kế nhiệm là người thua cuộc (tức là chỉ với một nước đi ta có thể đạt đến một cấu hình mà đối thủ đang di chuyển và mỗi nước đi đều dẫn đến một trận thua trong tương lai).
- Cấu hình không đầu cuối được gọi là *thua* nếu tất cả các cấu hình kế tiếp của nó đều là cấu hình thắng (tức là chúng ta chơi nước đi nào thì đối thủ của chúng ta sẽ có nước đi thắng ngay sau đó).
- Bất kỳ cấu hình nào khác là *không chắc chắn* (nếu cả hai người chơi chơi tối ưu, trò chơi sẽ kết thúc với tỷ số hòa).

Vì vậy, mục tiêu của một chương trình chơi cờ hoàn hảo là luôn cố gắng đạt được cấu hình chiến thắng. Nhất cử nhất động, chiến thắng chỉ là vấn đề thời gian và hoàn toàn chắc chắn, bất kể đối thủ ra sao.

### 6.5.1. Trò chơi "X" và "O"

Xây dựng một biểu đồ đại diện cho một phân tích đầy đủ của một trò chơi cờ vua không phải là thế mạnh của kỹ thuật hiện tại (nó không cho phép bảo tồn hoặc nghiên cứu một biểu đồ có khoảng 10.100 đỉnh). Do đó, chúng ta sẽ giải quyết vấn đề về chiến lược chiến thắng với một trò chơi đơn giản hơn nhưng cũng nổi tiếng:

Hai người chơi trên bảng có kích thước $3 \times 3$. Ở mỗi nước đi, người chơi lần lượt điền vào các ô trên bảng - người chơi 1 sử dụng ký hiệu "X" và người chơi 2 sử dụng ký hiệu "O". Người chiến thắng là người đầu tiên điền vào toàn bộ hàng ngang, hàng dọc hoặc một trong hai đường chéo chính của bàn cờ.

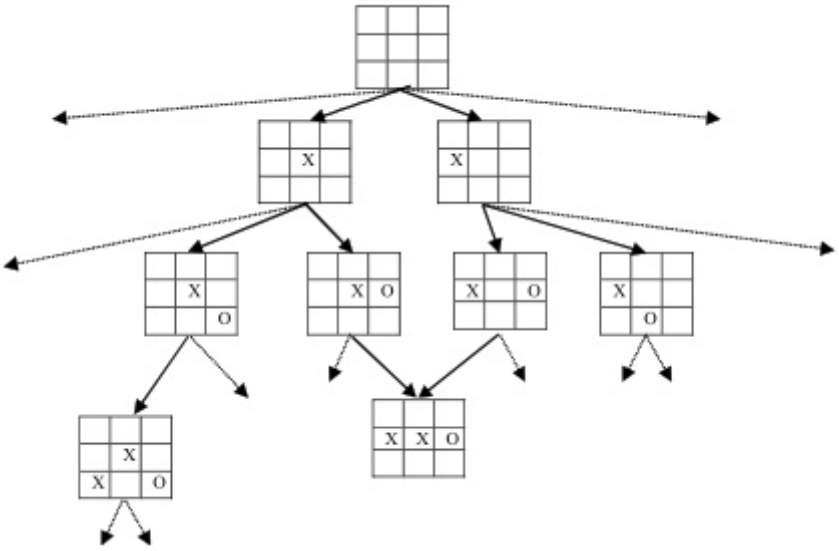
Chúng ta sẽ trình bày các cấu hình có thể có trong trò chơi dưới dạng các nút đồ thị (lưu ý rằng, không giống như cờ vua, trong trò chơi "X" và "O", đồ thị là dòng xoay). Một phần của đồ thị được thể hiện trong Hình 6.10.

Các điều khoản của đoạn trước cũng có giá trị ở đây - ví dụ trong Hình 6.11. cấu hình $T_3$ và $T_4$ là đầu cuối (người chơi 1 đã thắng trò chơi), cấu hình $T_2$ là thua (bất kỳ nước đi nào mà người chơi 2 chơi, anh ta sẽ thua), và $T_1$ là chiến thắng.

Chúng ta sẽ đưa ra một thuật toán có thể chơi trò chơi "X" và "O" một cách tối ưu, thực hiện hết hoàn toàn (toàn bộ biểu đồ trò chơi). Chúng ta sẽ sử dụng chức năng đệ quy `checkPosition (int player, board)`, kiểm tra xem cấu hình `board` là thắng, thua hay không xác định cho `player`. Với `board1, board2, ..., boardn`, chúng ta sẽ biểu thị tất cả các cấu hình có thể nhận được từ `board` trong một lần:

Sơ đồ kiểm tra vị trí chơi

```
/* Trả lại: 1 - nếu cấu hình mang lại lợi thế cho người chơi, */
/*          2 - nếu nó là một thế thua cuộc */
/*          3 - nếu nó không xác định */
int checkPosition(int player, board)
  if (<cấu hình là thiết bị đầu cuối>) {
  if (<trò chơi kết thúc>) return 3; /* cờ hòa */
  if (<player == người chơi thắng>) return 1;
  if (<player != người chơi thắng>) return 2;
 } else {
```



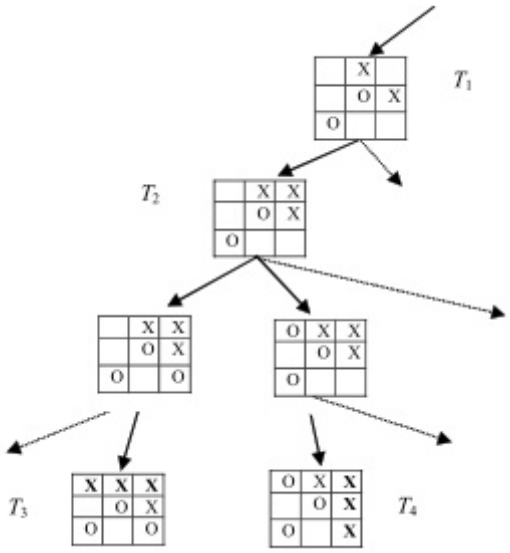
Hình 6.10. Đồ thị trò chơi "X" và "O"

```
  if (checkPosition(<người chơi khác>,boardi)==1 với mọi i=1,..,n)
    return 2;
  if (checkPosition(<người chơi khác>,boardi)==2 với mọi i=1,..,n)
    return 1;
  return 3;
 }
}
```

Sơ đồ trên sử dụng cùng một giá trị để chỉ ra một vị trí không xác định và một vị trí cuối cùng mà tại đó trò chơi kết thúc với tỷ số hòa. Như đã lưu ý trong đoạn 6.5, thực tế là một vị trí không chắc chắn (ngay cả khi cây trò chơi được xem xét toàn bộ) có nghĩa là với cách chơi tối ưu cho cả hai người chơi, trò chơi chắc chắn sẽ kết thúc với tỷ số hòa.

Lược đồ của hàm `checkPosition()` là phổ quát. Nó có thể được áp dụng trong tất cả các trò chơi liên quan đến hai người chơi luân phiên và mỗi cấu hình có thể nhận được trong một lần di chuyển từ người khác.

Nó cũng có thể được sử dụng trong cờ vua, nhưng do kích thước của cây, nó sẽ chỉ giúp ích trong một số trường hợp đơn giản hơn




Hình 6.11. Các vị trí kết thúc trong cây trò chơi

Hình 6.12. Chiếu tướng sau 3 nước đi (trắng đi trước)

(ví dụ, Hình 6.12 cho thấy một vị trí chiến thắng cho quân trắng). Với nó, chúng ta có thể giải quyết các vấn đề cờ vua cổ điển kiểu "cờ tướng trong 3 nước đi" và các bài toán khác.

Sau đây là cách triển khai có thể có của hàm `checkPosition()` cho các trò chơi "X" và "O". Chúng ta sử dụng hàm `terminal()` để kiểm tra xem cấu hình có phải là thiết bị đầu cuối hay không và nếu

có, trả về người chiến thắng là ai hoặc trò chơi đã kết thúc với tỷ số hòa:

**Chương trình** 6.10. Trò chơi tic và tắc (610tictac.c)

```
#include <stdio.h>
/* Người chơi bắt đầu */
const char startPlayer = 2;
/* Cấu hình bắt đầu */
char board[3][3] = {
  { '.', '.', '.' },
  { '.', 'X', '.' },
  { 'X', '.', 'O' }};

/* kết quả: 1, nếu cấu hình kết thúc và chiến thắng là người 1,
* 2, nếu cấu hình kết thúc và chiến thắng là người 2,
* 3, nếu cấu hình kết thúc và cờ hòa
* 0, nếu cấu hình không kết thúc
*/
char terminal(char a[3][3])
{ unsigned i, j;
  for (i = 0; i < 3; i++) {
  /* kiểm tra các đường ngang */
  for (j = 0; j < 3; j++)
    if (a[i][j] != a[i][0]) break;
  if (3 == j && a[i][0] != '.') {
    if (a[i][0] == 'X') return 1; else return 2;
  }

  /* kiểm tra các đường đứng */
  for (j = 0; j < 3; j++)
    if (a[j][i] != a[0][i]) break;
  if (3 == j && a[i][0] != '.') {
    if (a[0][i] == 'X') return 1; else return 2;
  }

  /* kiểm tra các đường chéo */
  if (a[0][0] == a[1][1] && a[1][1] == a[2][2] && a[1][1] != '.')
    if (a[0][0] == 'X') return 1; else return 2;
  if (a[2][0] == a[1][1] && a[1][1] == a[0][2] && a[1][1] != '.')
    if (a[2][0] == 'X') return 1; else return 2;
  }
```

```
  /* phải chăng là hòa (phải chăng các vị trí đều bận) */
  for (i = 0; i < 3; i++)
    for (j = 0; j < 3; j++)
      if (a[i][j] == '.') return 0;
  return 3;
}

/* trả về: 1, nếu cấu hình không thắng cho người chơi,
* 2, nếu là thua và
* 3, nếu không xác định */
char checkPosition(char player, char board[3][3])
{ unsigned i, j, result;
  int t = terminal(board);
  if (t) {
    if (player == t) return 1;
  if (3 == t) return 3;
  if (player != t) return 2;
  }
  else {
    char otherPlayer, playerSign;
    if (player == 1) { playerSign = 'X'; otherPlayer = 2; }
      else { playerSign = 'O'; otherPlayer = 1; }
  /* char board[3][3];
  * xác định vị trí
  */
  for (i = 0; i < 3; i++) {
    for (j = 0; j < 3; j++) {
      if (board[i][j] == '.') {
        board[i][j] = playerSign;
        result = checkPosition(otherPlayer, board);
        board[i][j] = '.';
        if (result == 2) return 1; /* cấu hình thua cho đối thủ, */
        /* suy ra thắng cho người chơi */
        if (result == 3) return 3; /* cấu hình không xác định */
      }
  }
}
       /* tất cả bước sau đều cấu hình thắng và
       * suy ra nó thua cho người chơi */
```

```
            return 2;
        }
    }

    int main() {
        printf("%u\n", checkPosition(startPlayer, board));
        return 0;
    }
```

Trong vài điểm tiếp theo chúng ta sẽ xem xét vấn đề này - để so sánh "chất lượng" của hai vị trí không xác định, cũng như việc chặt một phần cây sẽ được nghiên cứu.

## Bài tập

▷ **6.37.** Hãy viết một chương trình mô phỏng trò chơi "X" -chọn và "O" giữa người và máy tính, và máy tính phải chơi một cách tối ưu: để giành chiến thắng bất cứ khi nào anh ta có được cơ hội như vậy, tức là trong trường hợp lỗi của con người, và không mắc lỗi cho mình. Khi vị trí của máy tính không chắc chắn, nó có thể thực hiện bất kỳ hành động nào không gây tổn thất.

▷ **6.38.** Hãy viết chương trình chơi chữ "X" và "O" trong trường hợp chung, khi một bảng có kích thước $n \times n$ được đưa ra, và người chiến thắng là người chơi tạo được một hàng, cột hoặc đường chéo của $m$ ký tự. Trong biến thể phổ biến nhất, $m$ là 5.

▷ **6.39.** Hãy chứng minh rằng với bàn là $3 \times 3$, nếu cả hai người chơi đều chơi đúng, trò chơi phải kết thúc với tỷ số hòa.

▷ **6.40.** Tìm số của:

a) tất cả các loại cấu hình có thể nhận được trong trò chơi

b) tất cả các loại cấu trúc gỗ (so sánh với a))

c) số lượng lá trên cây (tức là cấu hình đầu cuối)

### 6.5.2. Nguyên tắc minimum và maximum

Chúng ta sẽ tiếp tục với trò chơi "X" và "O". Không giống như đoạn trước, bây giờ chúng ta sẽ trình bày các chuyển động có thể xảy ra không phải với đồ thị (như trong Hình 6.10), Mà là với một

cây, tại mỗi đỉnh có một số ước lượng được so sánh. Nó xác định vị trí tối ưu như thế nào đối với người chơi 1. Chúng ta sẽ bắt đầu với danh sách trên cây (cấu hình thiết bị đầu cuối): một trang tính có xếp hạng $+\infty$ nếu người chơi 1 thắng, xếp hạng $-\infty$ nếu người chơi 2 thắng và 0 xếp hạng nếu 0 trận đấu kết thúc với tỷ số hòa. Xếp hạng của tất cả các đỉnh khác được xác định bởi những người kế thừa của chúng như sau: Đối với đỉnh $i$ với những người thừa kế $i_1, i_2, ..., i_k$, xếp hạng $V[i]$ được xác định dựa trên *nguyên tắc minimum và maximum*:

$$V[i] = \begin{cases} \max(V[i_1], V[i_2], ..., V[i_k],) & \text{nếu bước đi là người chơi 1} \\ \min(V[i_1], V[i_2], ..., V[i_k],) & \text{nếu bước đi là người chơi 2} \end{cases}$$

Một đỉnh được gọi là *đỉnh tối đa hóa* nếu người chơi 1 đang tiến hành cấu hình phù hợp với nó, nếu không nó được gọi là *đỉnh tối thiểu* (các định nghĩa này dựa trên thực tế là xếp hạng đỉnh được xác định là mức tối đa / tối thiểu của xếp hạng của người kế nhiệm ). Sau đây là một hàm (dựa trên phương pháp tìm kiếm trả về) tính toán ước lượng của bất kỳ đỉnh nào từ cây.

Hàm đánh giá trên mỗi đỉnh của cây

```
value minimax(<đỉnh i>) {
  if (<i là lá>) return <đánh giá lá>;
  <Gọi (i1 , i2 ,..., in ) là những thừa kế của i>;
  if (<i là đỉnh với nhỏ nhất>)
  return min(minimax(i1 ), ... , minimax(in ));
  if (<i là đỉnh với cực đại hóa>)
  return max(minimax(i1 ), ... , minimax(in ));
}
```

Vì vậy, bài toán của việc thực hiện tốt nhất tiếp theo là tìm ra người kế vị hàng đầu với số điểm cao nhất. Trong trò chơi "X" -chets và "O" điểm của các đỉnh-lá có thể là $-\infty, +\infty$ và 0. Do đó điểm của mỗi đỉnh của cây nhất thiết phải là một trong ba giá trị này. Tuy nhiên, trong nhiều trò chơi khác, có thể các ước lượng khác nhau cho các lá nhiều hơn hoặc việc đánh giá phần ngọn của cây được xác định mà không cần phải kiểm tra toàn bộ cây con của nó (tức là, đánh giá được thực hiện bởi một số tiêu chí chủ quan - xem 6.5.4).

Đặc điểm của hàm `minimax()` là nó kiểm tra từng đỉnh của cây. Thường thì điều này là không cần thiết.

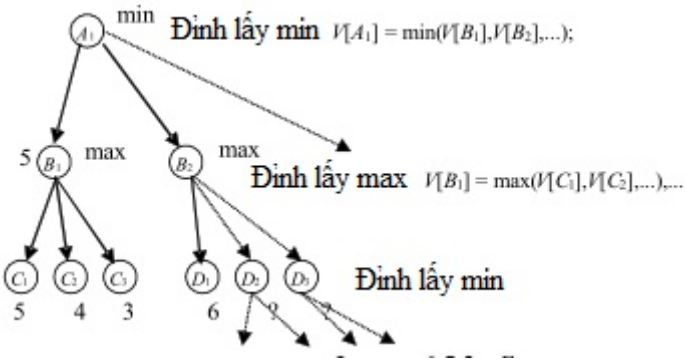
## Bài tập

▷ **6.41.** Viết chương trình mô phỏng trò chơi chữ "X" và chữ "O" giữa người và máy tính, thực hiện nguyên tắc tối thiểu và tối đa.

### 6.5.3. Nhát cắt alpha-beta

*Nhát cắt alpha-beta* là một phương pháp giảm các đỉnh được nghiên cứu trên nguyên tắc tối thiểu và tối đa. Hãy xem hai ví dụ.

Trong Hình 6.13 một cây được hiển thị trong đó thuật toán phải tính toán ước lượng đỉnh để tối thiểu hóa $A_1$. Điểm của $A_1$ là điểm tối thiểu trong số các điểm kế thừa của anh ta $V[B_1]$, $V[B_2]$, $V[B_3]$, v.v.

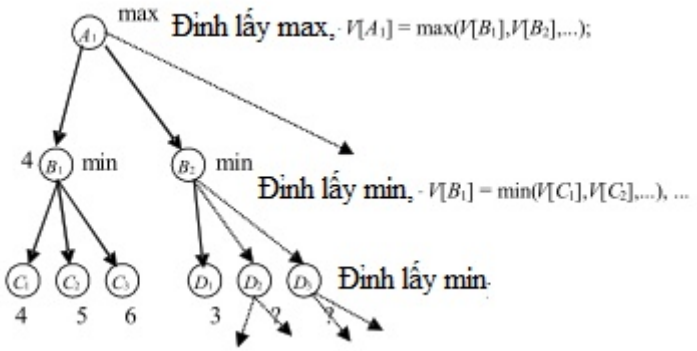



Hình 6.13. Nhát cắt beta

Thuật toán đầu tiên cố gắng tìm giá trị của ước lượng $V[B_1]$ - nó được xác định là giá trị lớn nhất trong số các ước lượng kế tiếp của nó. Giả sử rằng các ước lượng kế tiếp của $B_1$ đã được tính toán, vì vậy $V[B_1] = \max(V[C_1, C_2, C_3]) = \max(5, 4, 3) = 5$. Vì như $A_1$ là đỉnh cực tiểu, từ $V[B_1] = 5$ ước tính của $A_1$ sẽ nhỏ hơn hoặc bằng 5. Quá trình thu thập thông tin tiếp tục với đỉnh $B_2$. Giá trị $V[B_2]$ là giá trị lớn nhất trong các giá trị kế tiếp của nó là $D_1, D_2, D_3, \ldots$. Chúng ta đến với phần thực của việc nhát cắt alpha-beta - trong ví dụ này, chúng ta giả định rằng ước lượng thu được sau khi nghiên cứu cây

con có gốc $D_1$ là $V[D_1] = 6$. Theo đó ước lượng của $B_2$ sẽ ít nhất là 6 (chúng ta đang tìm kiếm giá trị tối đa). Mặt khác, chúng ta đã thấy rằng $V[A_1] \leq 5$. chúng ta không quan tâm và sẽ không được nghiên cứu, vì chúng ta đã có được rằng $V[B_2] \geq 6$ và $V[A_1] \leq 5$.

Nhát cắt alpha được áp dụng khi xem xét một đỉnh tối đa hóa. Xét ví dụ trong Hình 6.14: Giả sử chúng ta đã tính được $V[C_1] = 4, V[C_2] = 5, V[C_3] = 6$. Như vậy $V[B_1] = \min(V[C_1], V[C_2], V[C_3]) = \min(4, 5, 6) = 4$. Theo đó $V[A_1] \geq 4$. Tiếp theo, hóa ra $V[D_1] = 3$, theo đó là $V[B_2] \leq 3$. Từ $V[A_1] \geq 4$ và $V[B_2] \leq 3$, việc nghiên cứu thêm về những người thừa kế của $B_2$ là không cần thiết (nhát cắt alpha).




Hình 6.14. Nhát cắt alpha

Việc triển khai nhát cắt alpha và beta được thực hiện như sau: Đối với mỗi đỉnh được duyệt, ngoài đánh giá của nó, hai giá trị khác được tính toán - *giá trị alpha* và *giá trị beta* của đỉnh:

*Giá trị alpha của đỉnh*

Ta khởi tạo các giá trị alpha như sau: Nếu đỉnh là lá thì giá trị alpha là $-\infty$, ngược lại thì trùng với ước lượng của nó. Sau đó, khi thu thập thông tin, giá trị alpha của mỗi đỉnh *tối đa hóa* được xác định là giá trị lớn nhất của các ước tính của các giá trị kế thừa được nghiên cứu cho đến nay. Đối với các đỉnh cực tiểu, giá trị alpha là giá trị alpha của *giá trị đỉnh trước đó*.

*Giá trị beta các đỉnh*

Các giá trị beta của đỉnh lá tính được khởi tạo với các ước tính

của chúng. Tất cả các đỉnh khác được khởi tạo với giá trị beta là $+\infty$. Khi duyệt sau đó, giá trị beta của đỉnh tối thiểu hóa được xác định là giá trị nhỏ nhất của các ước tính của các giá trị kế thừa được nghiên cứu cho đến nay. Giá trị beta của đỉnh tối đa hóa là giá trị beta của giá trị tiền nhiệm.

Bài toán này đảm bảo rằng:

- Điểm đỉnh sẽ luôn không nhỏ hơn giá trị alpha của đỉnh và không lớn hơn giá trị beta của nó.
- Giá trị alpha và beta có thể thay đổi khi bạn duyệt qua, nhưng chắc chắn rằng giá trị alpha sẽ không bao giờ giảm và giá trị beta sẽ tăng lên.
- Giá trị alpha của mỗi đỉnh (trừ đỉnh gốc) luôn lớn hơn hoặc bằng giá trị alpha của đỉnh trước của nó.
- Giá trị beta của mỗi đỉnh (trừ đỉnh gốc) luôn nhỏ hơn hoặc bằng giá trị beta của đỉnh trước của nó.

Chúng ta sẽ hiển thị một sơ đồ mà cây có thể được duyệt. Hàm `minimaxCutoff()` được gọi với các tham số đỉnh gốc cây, $-\infty$ và $+\infty$ (giá trị alpha và beta, tương ứng):

Nhát cắt minimax

```
value minimaxCutoff (N, A, B) {
  Na = A; / * Na là giá trị alpha của đỉnh được tính * /
  Nb = B; / * Nb là giá trị beta của đỉnh sẽ được tính * /
  if (<N là đỉnh lá>)    return <giá trị giá của lá>;
  if (<N là đỉnh lấy min>)
    for (<với mọi thừ kế Ni của N>) {
      val = minimaxCutoff (Ni, Na, Nb);
      Nb = min (Nb, val);
      if (Nb <= Na) break; / * nhát cắt alpha-beta * /
    }
    return Nb;
}
else {/ * nếu N là đỉnh lấy max* /
    for (<với mỗi đỉnh thừa kế Ni của N>) {
      val = minimaxCutoff (Ni, Na, Nb);
      Na = max (Na, val);
      if (Na> = Nb) break; / * nhát cắt alpha-beta * /
    }
```

```
    return Na;
  }
}
```

Trong hai dòng (có chữ " nhát cắt alpha-beta "), nghiên cứu thêm về alpha-beta bị gián đoạn, vì nó là vô vọng.

## Bài tập

▷ **6.42.** Viết chương trình mô phỏng trò chơi "X-lần" và "O" giữa người và máy tính, thực hiện nguyên tắc tối thiểu và tối đa với alpha-beta clipping. So sánh với tùy chọn không bị cắt ngắn alpha-beta.

### 6.5.4. Duyệt alpha-beta đến một độ sâu nhất định

Chúng ta sẽ tập trung lại vào ví dụ về cờ vua. Để thực hiện duyệt thông tin alpha-beta nhằm xác định điểm cao nhất, chúng ta sẽ phải duyệt từ một cái cây khổng lồ, nơi ngay cả những tối ưu hóa như nhát cắt alpha-beta cũng không giúp ích gì đáng kể cho chúng ta. Thường trong những trường hợp như vậy, việc duyệt đến một độ sâu nhất định được áp dụng. Khi đạt đến điểm đó, quá trình duyệt bị dừng lại và việc đánh giá đỉnh điểm mà nó bị gián đoạn được tính theo một số tiêu chí khác. Đối với cờ vua, tiêu chí này có thể là, quân nào còn lại trên bàn cờ, quyền tự do di chuyển của họ, ai là người sở hữu tâm bàn cờ, v.v. Hàm tính toán ước tính cấu hình sẽ không yêu cầu nhiều thời gian tính toán, vì nó sẽ được thực hiện thường xuyên. Một cách có thể để ước tính là bằng cách so sánh các giá trị của các số liệu [Brassard, Bratley - 1996]:

Quân Hậu - 10; Quân xe 5; Tượng, Ngựa - 3,25; Quân Tốt - 1;

Đánh giá một cấu hình (tích cực hoặc tiêu cực) có thể được định nghĩa là tổng điểm của các quân trắng trừ đi tổng điểm của các quân đen. Đối với các vị trí đầu cuối, chúng ta giả định rằng khi quân đen ở vị trí mờ, điểm cấu hình là $+\infty$ và khi người da trắng mờ, điểm là $-\infty$. Khi trò chơi kết thúc với tỷ số hòa - giá trị là 0.

Hàm ước lượng đỉnh tương tự như trong 6.5.2 với sự khác biệt duy nhất mà nó gây ra khi tính toán giá trị cấu hình khi đạt đến độ sâu nhất định:

Hàm tính giá trị theo độ sâu

```
value eval(i) {
  return <Tính theo tiêu chuẩn cho cấu hình>;
}
value minimax(i, depth) {
  if (<i là đỉnh lá>) return <đánh giá của đỉnh lá ($+\infty,-\infty,
    0$);
  if (depth > maxDepth) return eval(i); / * tính toán ràng buộc * /
    <lấy (i1 , i2 , ..., in )là các thừa kế của i>;
  if (<i là đỉnh lấy min>)
    return min(minimax(i1 ,depth+1), .., minimax(in ,depth+1));
  if (<i là đỉnh lấy max>)
    return max(minimax(i1 ,depth+1), .., minimax(in ,depth+1));
}
```

Độ sâu của `maxDepth` thu thập thông tin càng lớn thì ước tính càng chính xác. Tuy nhiên, chúng ta sẽ không bao giờ chắc chắn rằng đánh giá là chính xác tuyệt đối. Ví dụ: luôn có thể có một quá trình phát triển trò chơi hy sinh một con số trong cấu hình độ sâu maxDepth (điều này sẽ làm giảm xếp hạng `eval (i)`) để có được vị trí tốt hơn: Trong một số bước tiếp theo, thậm chí `maxDepth + 1` ), mà sẽ không được xem xét, có thể giành chiến thắng một con số mạnh hơn hoặc thậm chí kiểm tra đối thủ.

Cùng với việc thu thập thông tin alpha-beta, bạn có thể áp dụng phương pháp nhát cắt alpha-beta. Vào năm 1997, tốc độ của máy tính DeepBlue, đã đánh bại nhà vô địch thế giới, cho phép 200.000.000 lượt kiểm tra cấu hình cờ vua mỗi giây. Với việc bổ sung thêm alpha-beta clipping, độ sâu của cây mà máy tính này có thể khám phá là khá cao. Ngoài ra, hàm `eval()` cực kỳ chính xác (một số người chơi cờ chuyên nghiệp đã thực hiện điều này). Có thể nói, phần cứng máy tính hiện đã qua thời điểm phát triển quan trọng, lúc này người ta không còn có thể cạnh tranh trong trò chơi cờ vua với một chương trình cờ vua được thiết kế tốt.

## Bài tập

▷ **6.43.** Hãy đề xuất phương án đánh giá chất lượng cấu hình cho các trò chơi "X" và "O".

▷ **6.44.** Để xuất ra một biến thể của trò chơi "X" và "O" bằng cách sử dụng nhát cắt alpha-beta đến một độ sâu nhất định.

## 6.6. Câu hỏi và bài tập

Trong phần cuối của chương, chúng ta sẽ đưa ra danh sách các bài toán NP-đầy đủ nổi tiếng nhất, kèm theo lời giải thích ngắn gọn về một số trong số đó. Các bài toán *NP*-đầy đủ thường xuất hiện trong thực tế và việc nhận ra chúng như vậy có thể giúp chúng ta tiết kiệm rất nhiều công sức. Thực tế là một bài toán *NP*-đầy đủ sẽ cho phép chúng ta không lãng phí thời gian để tìm kiếm một thuật toán nhanh, mà tập trung vào một trong những cách tiếp cận tiêu chuẩn sau để giải quyết:

- *Sử dụng thuật toán heuristic (xem Chương 9).* Nếu vấn đề không thể được giải quyết nhanh chóng trong mọi trường hợp, thì vẫn có thể có một phương pháp nhanh chóng chỉ có thể giải quyết trong một số trường hợp.
- *Giải quyết vấn đề gần đúng (xem Chương 9).* Đối với một số bài toán *NP*-đầy đủ, các thuật toán xấp xỉ tồn tại. Họ sẽ không giải quyết vấn đề một cách chính xác, nhưng họ sẽ luôn tìm ra một giải pháp có thể được chứng minh là đủ gần.
- Nhiều bài toán thực tế, vốn có tính chất *NP*-đầy đủ, có thể được chia thành một số trường hợp *đặc biệt chính*. Trong một số trường hợp, có thể có một thuật toán phức tạp đa thức sử dụng, ví dụ, bộ nhớ bổ sung (tối ưu hóa động - Chương 8).
- *Sử dụng vét kiệt hoàn toàn.* Ở đây, cách nhận thức cũng rất quan trọng. Đối với một số trường hợp trong bài toán có thể chắc chắn rằng chúng sẽ không dẫn đến một giải pháp cho phép loại trừ chúng khỏi quá trình nghiên cứu và do đó làm giảm độ phức tạp về thời gian. Những điều này đã được thảo luận trong chương này.

Nhiều bài toán chúng ta sẽ xem xét được xuất bản lần đầu tiên vào năm 1979 trong "bách khoa toàn thư" về các bài toán *NP*-đầy đủ - [Garey, Johnson-1979]. Để đầy đủ, danh sách cũng chứa các bài toán đã được xem xét (một số trong số chúng đã được giải quyết). Mỗi bài toán dưới đây được thảo luận trong sơ đồ sau:

- *Tên của bài toán* (và tên viết tắt, nếu biết). Đây là tên mà bài

toán có thể được tìm thấy thường xuyên nhất trong tài liệu.

- *Điều kiện của bài toán.* Nó bao gồm hai phần: dữ liệu đầu vào (thường chúng là các cấu trúc cụ thể - đồ thị, tập hợp, biểu thức logic, v.v.) và những gì bạn đang tìm kiếm. Câu hỏi thứ hai được xây dựng như một câu hỏi mà đối với các đầu vào cụ thể, thuật toán phải trả lời có hoặc không.
- *Nhận xét về bài toán.* Phần này (không phải lúc nào cũng có) bao gồm một bình luận ngắn về bài toán, có thể bao gồm các hướng dẫn (hoặc các cách tiếp cận ít được biết đến hơn) để giải quyết bài toán, các biến thể, ứng dụng của nó, v.v. Ở đây có thể có các tài liệu tham khảo khác liên quan đến bài toán. Như chúng ta đã nói, nhiều bài toán trong danh sách này có thể được tìm thấy được phân loại trong [Garey, Johnson-1979], vì vậy chúng ta sẽ không đề cập đến cuốn sách này trong từng bài toán cụ thể. Sau đây là một ví dụ:

▷ **6.45.** *Đồ thị 3 màu (3-COL)*

- Một đồ thị được đưa ra.
- Kiểm tra xem có thể "tô màu" mỗi đỉnh của đồ thị (so sánh nó) với một trong ba màu: lục, lam, đỏ, sao cho không có hai đỉnh liền kề nào được tô cùng màu.

Vì vậy, đặt ra, bài toán liên quan đến việc tìm ra lời giải cho một đồ thị tùy ý. Điều kiện chỉ muốn kiểm tra xem có thể tô màu cho cột 3 màu hay không. Đối với nhiều bài toán trong danh sách này, điều kiện có thể được tóm tắt để tìm ra giải pháp tối ưu - đối với ví dụ cụ thể, có thể tìm cách tô màu với số lượng màu tối thiểu, v.v.

Hơn nữa, các bài toán theo sau với độ khó tăng dần. Các bài toán từ đầu danh sách được đặt ở dạng đơn giản hơn, chúng liên quan đến một bộ máy toán học đơn giản hơn, được thảo luận rộng rãi trong các tài liệu. Tình huống hoàn toàn ngược lại là với các bài toán ở cuối danh sách - điều kiện phức tạp hơn và ít được biết đến hơn.

▷ **6.46.** *Đường Hamilton trong đồ thị*

- Một đồ thị không định hướng được đưa ra.
- Kiểm tra xem có một đường đi đơn giản nào chứa tất cả các đỉnh của đồ thị hay không.
- Bài toán được xem xét trong: **??**, 6.2, **??** và những người khác.

▷ **6.47.** *Chu trình tối đa*

- Một đồ thị không định hướng với $n$ đỉnh và số tự nhiên $k, 1 \leq k \leq n$ là cho trước.
- Kiểm tra xem có một vòng lặp đơn giản trong đồ thị chứa ít nhất $k$ đỉnh hay không.
- Đây là bản tóm tắt của bài toán tìm đường đi Hamilton. Trong thực tế, bài toán kiểm tra xem một đồ thị có chu trình Hamilton hay không là một trường hợp đặc biệt (nếu chúng ta đặt $k = n$).

▷ **6.48.** *Sự thỏa mãn của một hàm Boolean*

- Thứ m loại bỏ $C_1, C_2, ..., C_m$ của các biến Boolean $X_1, X_2, ..., X_n$ và các số phủ định của chúng được đưa ra.
- Để kiểm tra xem có sự gán giá trị "true" hoặc "false" cho các biến $X_1, X_2, ..., X_n$, mà giá trị của tất cả các liên từ $C_1, C_2, ..., C_m$ là "true".
- Chúng ta đã giải quyết vấn đề trong 6.3.1.

▷ **6.49.** *Con đường dài nhất*

- Một đồ thị không định hướng với n đỉnh và số tự nhiên $k, 1 \leq k \leq n$ là cho trước. Hai đỉnh $s, t$ của đồ thị là cố định.
- Kiểm tra xem có một đường đi đơn giản từ $s$ đến $t$ trong đó ít nhất $k$ cạnh tham gia hay không.
- Chúng ta đã giải quyết một biến thể của bài toán trong 6.3.3.

▷ **6.50.** *Một bài toán cho khách du lịch*

- Một đồ thị có trọng số không định hướng với n đỉnh và số tự nhiên $k, k \geq 1$ đã cho.
- Kiểm tra xem có một chu trình đơn giản bao gồm tất cả các đỉnh của đồ thị, với độ dài (là tổng trọng lượng của các sườn) nhiều nhất là $k$.
- Đây là một dạng biến thể của bài toán nổi tiếng về tìm chu trình Hamilton nhỏ nhất trong đồ thị, mà chúng ta đã xem xét nhiều lần.

▷ **6.51.** *Bài toán ba lô*

- Một tập hợp hữu hạn $A$ được cho trước và một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử $a \in A$. Số tự nhiên $k$ đã cho.

- Kiểm tra xem có tồn tại một tập con $B \subseteq A$ sao cho tổng các giá trị $s(x)$ của $x \in B$ là chính xác $k$.
- Đây là bài toán 1 của 6.4.1. Chúng ta quyết định biến thể tối ưu hóa của nó theo phương pháp phân nhánh và ranh giới. Chúng ta sẽ xem xét nó một lần nữa trong **??**, Nơi chúng ta sẽ giải quyết nó với tối ưu hóa động.

▷ **6.52.** *Bài toán của Alan và Bob*

- Tập hợp hữu hạn $A$. Cho trước, một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử $a$.
- Kiểm tra xem $A$ có thể chia thành hai tập $A_1$ và $A_2$ hay không mà $\sum_{x \in A_1} s(x) = \sum_{y \in A_2} s(y)$.
- Tên của bài toán "Alan và Bob" xuất phát từ bài toán thực tế là chia một số phần quà (mỗi phần quà có giá trị nhất định) giữa hai anh em (Alan và Bob) sao cho tổng giá trị của các phần quà của một là bằng tổng từ các giá trị của quà tặng của người kia. bài toán được xem xét trong **??**.

▷ **6.53.** *Tích từ một tập hợp con.*

- Một tập hợp hữu hạn $A$ được cho trước và một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử $a \in A$. Số tự nhiên $k$ đã cho.
- Kiểm tra xem có tồn tại tập con $B \subseteq A$ sao cho tích của các phần tử $s(x), x \in B$ chính xác bằng $k$.
- Đây là một biến thể của bài toán ba lô.

▷ **6.54.** *Đóng gói hộp*

- Tập hợp $A$ hữu hạn có $n$ phần tử đã cho. Một số tự nhiên $s(a)$ được ánh xạ tới mỗi phần tử $a \in A$. Các số tự nhiên $b$ và $k$ đã cho, $1 \leq k \leq n$.
- Kiểm tra xem $A$ có thể chia thành $k$ tập $A_1, A_2, ..., A_k$ sao cho với mỗi $A_i (1 \leq i \leq k)$, tổng các giá trị của các phần tử của nó không vượt quá $b$.
- Bài toán là một bản tóm tắt của bài toán cho ba lô - $k$ ba lô được đưa ra, mỗi ba lô có thể chứa b kg và phải phân phối các vật dụng trong đó. Một biến thể tối ưu hóa của vấn đề cũng có thể xảy ra - để tìm k tối thiểu mà các đối tượng có thể được phân phối theo cách được mô tả.

▷ **6.55.** *Phủ hình vuông*

- $N$ màu (ký hiệu là số nguyên $1, 2, ..., n$) và danh sách các ô vuông tô màu có một cạnh được đưa ra. Mỗi hình vuông được cho bởi bốn $(a, b, c, d)$ cho biết mặt trên tô màu $a$, mặt dưới tô màu $b$, mặt trái tô màu $c$ và mặt bên phải tô màu $d$. Số tự nhiên $k$ đã cho, $1 \leq k \leq n$.
- Kiểm tra xem có một ô vuông có kích thước $k \times k$ được tô bằng cách sử dụng các ô vuông đã cho để các mặt tiếp xúc của chúng có cùng màu hay không.
- [Kelevedzhiev-5/1998]

▷ **6.56.** *Trò chơi ô chữ*

- Tập hợp $n$ ký hiệu $S = \{s_1, s_2, ..., s_n\}$ và tập $W = \{w_1, w_2, ..., w_{2n}\}$ gồm $2n$ từ được đưa ra sao cho mỗi wi là một dãy gồm đúng n ký tự từ $S$.
- Để kiểm tra xem có thể thu được ô chữ có kích thước $n \times n$ từ $2n$ từ đã cho hay không, tức là nếu $C$ là ma trận có kích thước $n \times n$, thì ký hiệu của $S$ có thể được viết trong mỗi ô sao cho mỗi dòng và một cột của ma trận để biểu diễn một từ từ $W$.

▷ **6.57.** *Tập độc lập trong đồ thị*

- Một đồ thị không định hướng $G(V, E)$ với n đỉnh và số tự nhiên $k$, $1 \leq k \leq n$ là cho trước.
- Kiểm tra xem có tồn tại một tập con $U \subseteq V$ chứa ít nhất $k$ đỉnh trong đó không tồn tại hai đỉnh $i, j \in U$ sao cho $(i, j) \in E$.
- Một biến thể của bài toán được xem xét trong 5.7.2.

▷ **6.58.** *Đồ thị không có hình tam giác*

- Một đồ thị không định hướng $G(V, E)$ được cho.
- Để kiểm tra xem có thể chia tập $E$ thành hai tập con $E_1$ và $E_2$ sao cho không trong chúng có ba bộ lạc có dạng $(i, j), (j, k), (k, i)$ (nghĩa là trong hai đồ thị $G_1(V, E_1), G_2(V, E_2)$ không có chu trình nào có độ dài 3).

▷ **6.59.** *Bộ thống trị*

- Một đồ thị không định hướng $G(V, E)$ với n đỉnh và số tự nhiên $k$, $1 \leq k \leq n$ là cho trước.

- Kiểm tra rằng $G$ có chứa tập con $U$, $U \subseteq V$ với nhiều nhất $k$ đỉnh và sao cho mỗi đỉnh $i \in V$, $i \in U$ tồn tại một đỉnh $j \in U$ sao cho $(i, j) \in E$.

▷ **6.60.** *Số màu cạnh*

- Đồ thị không định hướng $G(V, E)$ với m cạnh và số tự nhiên $k$, $1 \leq k \leq m$ là cho trước.
- Kiểm tra xem có thể so sánh một số tự nhiên duy nhất từ 1 đến $k$ trên mỗi sườn hay không (tô màu cho sườn với màu từ 1 đến $k$) để không có hai sườn ngẫu nhiên nào có màu giống nhau.
- Định lý Vizing chỉ ra rằng trường hợp phức tạp duy nhất trong bài toán này là khi k bằng hoành độ của đỉnh của đồ thị. Tính hoàn chỉnh NP của bài toán này đã được chứng minh gần đây. Có rất nhiều thuật toán heuristic và song song để giải nó [Gibbons, Rytter-1987].

▷ **6.61.** *Chuỗi con chung ngắn nhất*

- Cho tập $S = \{s_1, s_2, ..., s_n\}$ gồm các xâu nhị phân (dãy 0 và 1) và một số tự nhiên $k$.
- Kiểm tra xem có tồn tại một chuỗi $s$ có độ dài tối đa $k$ sao cho mỗi chuỗi $p \in S$ là một chuỗi con của $s$ hay không.
- Tóm tắt của bài toán là khi nhiều nhân vật khác nhau có thể tham gia vào các chuỗi, nhưng ngay cả trong trường hợp đơn giản này, bài toán vẫn hoàn thành $NP$.

▷ **6.62.** *Bài toán của bưu tá*

- Cho đồ thị $G(V, E)$ với m cạnh, tập con $E_1$, $E_1 \subseteq E$ và số tự nhiên $k$, $1 \leq k \leq m$.
- Kiểm tra xem có chu trình nào trong cột (không nhất thiết phải đơn giản) chứa mỗi cạnh $e \in E_1$ ít nhất một lần và có tổng số cạnh nhỏ hơn hoặc bằng $k$.

▷ **6.63.** *Bộ phận "cấp trên"*

- Hai dãy số tăng nghiêm ngặt $A =< a_1, a_2, ..., a_n >$ và $B =< b_1, b_2, ..., b_m >$ của các số nguyên dương được cho.
- Với $Div(x, Y)$ ($Y$ là một dãy số tự nhiên), chúng ta sẽ ký hiệu số phần tử $y \in Y$, sao cho $x$ là ước chính xác của $y$. Kiểm tra xem có tồn tại số tự nhiên $c$ mà $Div(c, A) > Div(c, B)$ hay không.

- Đây và ba bài toán tiếp theo là một loại đặc biệt của bài toán $NP$-đầy đủ - từ lập trình số nguyên. Trong bài toán cụ thể, chúng ta có thể tìm kiếm c một cách tuần tự, bắt đầu từ 1. Tuy nhiên, thuật toán này không hiệu quả và bài toán là $NP$-đầy đủ. Điều này đáng chú ý nếu chúng ta so sánh kích thước của dữ liệu đầu vào (số bit để biểu diễn số trong hệ thống số nhị phân) với độ phức tạp của thuật toán trong trường hợp xấu nhất - chúng ta có một hàm số mũ.

▷ **6.64.** *Phương trình Diophantine bậc hai*

- Các số tự nhiên $a, b, c$ đã cho.
- Kiểm tra xem có tồn tại các số nguyên $x$ và $y$ sao cho $(a.x^2) + (b.y) = c$.
- Nhận xét tương tự áp dụng ở đây như trong Bài toán 19 - tìm kiếm tuần tự cho các số $x$ và $y$ có độ phức tạp theo cấp số nhân bằng số bit cần thiết để biểu diễn các số $a, b$ và $c$.