Phần 3 – CÁC ỨNG DỤNG CỦA CÁC LỚP CTDL

Chương 14 – ỨNG DỤNG CỦA NGĂN XẾP

Dựa trên tính chất của các giải thuật, các ứng dụng của ngăn xếp có thể được chia làm bốn nhóm như sau: đảo ngược dữ liệu, phân tích biên dịch dữ liệu, trì hoãn công việc và các giải thuật quay lui. Một điều đáng chú ý ở đây là khi xem xét các ứng dụng, chúng ta không bao giờ quan tâm đến cấu trúc chi tiết của ngăn xếp. Chúng ta luôn sử dụng ngăn xếp như một cấu trúc dữ liệu trừu tượng với các chức năng mà chúng ta đã định nghĩa cho nó.

14.1. Đảo ngược dữ liệu

Trong phần trình bày về ngăn xếp chúng ta đã được làm quen với một ví dụ xuất các phần tử theo thứ tự ngược với thứ tự nhập vào. Ở đây chúng ta tiếp tục tham khảo thêm ứng dụng đổi một số thập phân sang một số nhị phân.

Ứng dụng đổi số thập phân sang số nhị phân

Giải thuật dưới đây chuyển đổi số thập phân `decNum` sang một số nhị phân.

```
1  loop (decNum > 0)
   1  digit = decNum % 2
   2  xuất (digit)
   3  decNum = decNum / 2
2  endloop
```

Tuy nhiên các ký số được xuất ra sẽ là thứ tự ngược của kết quả mà chúng ta mong muốn. Chẳng hạn số 19 lẽ ra phải được đổi thành 10011 chứ không phải là 11001. Thực là dễ dàng nếu chúng ta sử dụng ngăn xếp để khắc phục điều này.

```
void DecimalToBinary (val int decNum)
post: số nhị phân tương đương với số thập phân decNum sẽ được xuất ra.
uses: sử dụng lớp Stack để đảo ngược thứ tự các số 1 và số 0.
{
   1. Stack<int> reverse; // Khởi tạo ngăn xếp để chứa các ký số 0 và 1.
   2. loop (decNum > 0)
      1. digit = decNum % 2
      2. reverse.push(digit)
      3. decNum = decNum / 2
   3. endloop
   4. loop (!reverse.empty())
      1. reverse.top(digit)
      2. reverse.pop()
      3. xuất(digit)
   5  endloop
}
```

Một điều dễ nhận thấy là nếu chúng ta dùng một mảng liên tục (`array` trong C++) để chứa các số `digit` rồi tìm cách in theo thứ tự đảo lại, chúng ta sẽ phải tiêu tốn sức lực vào việc quản lý các biến chỉ số chạy trên mảng. Đó là điều nên tránh. Việc tuân thủ lời khuyên này giúp chúng ta có thói quen tốt khi đụng phải những bài toán lớn hơn: chúng ta có thể tập trung vào giải quyết những vấn đề chính của bài toán.

14.2. Phân tích biên dịch (parsing) dữ liệu

Việc phân tích dữ liệu thường bao gồm phân tích từ vựng và phân tích cú pháp. Chẳng hạn, để chuyển đổi một chương trình nguồn được viết bởi một ngôn ngữ nào đó thành ngôn ngữ máy, trình biên dịch cần tách chương trình ấy ra thành các từ khóa, các danh hiệu, các ký hiệu, sau đó tiến hành kiểm tra tính hợp lệ về từ vựng, về cú pháp. Trong việc kiểm tra cú pháp thì việc kiểm tra cấu trúc khối lồng nhau một cách hợp lệ là một trong những điều có thể được thực hiện dễ dàng nhờ ngăn xếp.

Ứng dụng kiểm tra tính hợp lệ của các cấu trúc khối lồng nhau

Để kiểm tra tính hợp lệ của các cấu trúc khối lồng nhau, chúng ta cần kiểm tra các cặp dấu ngoặc như [], {}, () phải tuân theo một thứ tự đóng mở hợp lệ, có nghĩa là mỗi khối cần phải nằm gọn trong một khối khác, nếu có.

Lý do sử dụng ngăn xếp được giải thích như sau: theo thứ tự xuất hiện, một dấu ngoặc mở xuất hiện sau cần phải có dấu ngoặc đóng tương ứng xuất hiện trước. Ví dụ [...(...)...] là hợp lệ, [...(...]...) là không hợp lệ. Điều này rõ ràng liên quan đến nguyên tắc FILO của ngăn xếp. Mỗi cấu trúc khối sẽ được chúng ta biết đến

khi bắt đầu gặp dấu ngoặc mở của nó, và chúng ta sẽ chờ cho đến khi nào gặp dấu ngoặc đóng tương ứng của nó thì xem như chúng ta đã duyệt qua cấu trúc đó. Các dấu ngoặc mở mà chúng ta gặp, chúng ta sẽ lần lượt lưu vào ngăn xếp, nếu đoạn chương trình hợp lệ, thì chúng ta cứ yên tâm rằng các dấu ngoặc đóng tương ứng của chúng sẽ xuất hiện theo đúng thứ tự ngược lại. Như vậy, mỗi khi gặp một dấu ngoặc đóng, việc cần làm là lấy từ ngăn xếp ra một dấu ngoặc mở và so trùng.

Văn bản cần kiểm tra thường là một biểu thức tính toán hay một đoạn chương trình.

Giải thuật: Đọc đoạn văn bản từng ký tự một. Mỗi dấu ngoặc mở (, [, { được xem như một dấu ngoặc chưa so trùng và được lưu vào ngăn xếp cho đến khi gặp một dấu ngoặc đóng),], } so trùng tương ứng. Mỗi dấu ngoặc đóng cần phải so trùng được với dấu ngoặc mở vừa được lưu cuối cùng, và như vậy dấu ngoặc mở này sẽ được lấy ra khỏi ngăn xếp và bỏ đi. Như vậy việc kiểm tra sẽ được lặp cho đến khi gặp một dấu ngoặc đóng mà không so trùng được với dấu ngoặc mở vừa lưu trữ (lỗi các khối không lồng nhau) hoặc đến khi hết văn bản cần kiểm tra. Trường hợp dấu ngoặc đóng xuất hiện mà ngăn xếp rỗng là trường hợp văn bản bị lỗi thừa dấu ngoặc đóng (tính đến vị trí đang xét); ngược lại, khi đọc hết đoạn văn bản, nếu ngăn xếp không rỗng thì do lỗi thừa dấu ngoặc mở.

Chương trình có thể mở rộng hơn đối với nhiều cặp dấu ngoặc khác nhau, hoặc cho cả trường hợp đặc biệt về đoạn chú thích trong một chương trình C (/* ...phần trong này dĩ nhiên không cần kiểm tra tính hợp lệ của các cặp dấu ngoặc ...*/)

```
int main()
/*
post: Chương trình sẽ báo cho người sử dụng khi đoạn văn bản cần phân tích gặp lỗi.
uses: lớp Stack.
*/
{
   Stack<char> openings;
   char symbol;
   bool is_matched = true;
   while (is_matched && (symbol = cin.get()) != '\n') {
      if (symbol == '{' || symbol == '(' || symbol == '[')
         openings.push(symbol);
      if (symbol == '}' || symbol == ')' || symbol == ']') {
         if (openings.empty()) {
            cout << "Unmatched closing bracket " << symbol
                 << " detected." << endl;
            is_matched = false;
         }
         else {
            char match;
            openings.top(match);
            openings.pop();
            is_matched = (symbol == '}' && match == '{')
                         || (symbol == ')' && match == '(')
                         || (symbol == ']' && match == '[');
```

```
            if (!is_matched)
               cout << "Bad match " << match << symbol << endl;
         }
      }
   }
   if (!openings.empty())
      cout << "Unmatched opening bracket(s) detected." << endl;
}
```

14.3. Trì hoãn công việc

Khi sử dụng ngăn xếp để đảo ngược dữ liệu, toàn bộ dữ liệu cần được duyệt xong, chúng ta mới bắt đầu lấy dữ liệu từ ngăn xếp. Nhóm ứng dụng liên quan đến việc trì hoãn công việc thường chỉ cần trì hoãn việc xử lý dữ liệu trong một thời gian nhất định nào đó mà thôi.

Có nhiều giải thuật mà dữ liệu cần xử lý có thể xuất hiện bất cứ lúc nào, chúng sẽ được lưu giữ lại để chương trình lần lượt giải quyết. Trong trường hợp dữ liệu cần được xử lý theo đúng thứ tự mà chúng xuất hiện, chúng ta sẽ dùng hàng đợi làm nơi lưu trữ dữ liệu. Ngược lại, nếu thứ tự xử lý dữ liệu ngược với thứ tự mà chúng xuất hiện, chúng ta sẽ dùng ngăn xếp do nguyên tắc FILO của nó.

14.3.1. Ứng dụng tính trị của biểu thức *postfix*

Chúng ta sẽ xem xét ví dụ về cách tính trị của một biểu thức ở dạng Balan ngược (*reverse Polish calculator*- còn gọi là *postfix*). Trong biểu thức này toán tử luôn đứng sau toán hạng của nó. Trong quá trình duyệt biểu thức, khi gặp các toán hạng chúng ta phải hoãn việc tính toán cho đến khi gặp toán tử tương ứng của chúng, do đó chúng sẽ được đẩy vào ngăn xếp. Khi gặp toán tử, các toán hạng được lấy ra khỏi ngăn xếp, phép tính được thực hiện và kết quả lại được đẩy vào ngăn xếp (do kết quả này có thể lại là toán hạng của một phép tính khác mà toán tử của nó chưa xuất hiện). Thứ tự FILO được nhìn thấy ở chỗ: toán tử của những toán hạng xuất hiện trước luôn đứng sau toán tử của những toán hạng xuất hiện sau. Chẳng hạn, với 8 5 2 - + (tương đương 8 + (5-2)), số 8 xuất hiện trước số 2, nhưng phép trừ của (5 - 2) lại có trước phép cộng.

Việc phân tích một biểu thức liên quan đến việc xử lý chuỗi để tách ra các toán hạng cũng như các toán tử. Do phần tiếp theo đây chỉ chú trọng đến ý tưởng sử dụng ngăn xếp trong giải thuật, nên chương trình sẽ nhận biết các thành phần của biểu thức một cách dễ dàng thông qua việc cho phép người sử dụng lần lượt nhập chúng. Việc phân tích biểu thức có thể được xem như bài tập khi sinh viên kết hợp với các kiến thức khác có liên quan đến việc xử lý chuỗi ký tự.

Trong chương trình, người sử dụng nhập dấu ? để báo trước sẽ nhập một toán hạng, toán hạng này sẽ được chương trình lưu vào ngăn xếp. Khi các dấu +, -, *, / được nhập, chương trình sẽ lấy các toán hạng từ ngăn xếp, tính và đưa kết quả

vào ngăn xếp; dấu = yêu cầu hiển thị phần tử tại đỉnh ngăn xếp (nhưng không lấy ra khỏi ngăn xếp), đó là kết quả của một phép tính mới nhất vừa được thực hiện.

Giả sử a, b, c, d biểu diễn các giá trị số. Dòng nhập **? a ? b ? c - = * ? d + =** được thực hiện như sau:

? a: đẩy a vào ngăn xếp;
? b: đẩy b vào ngăn xếp
? c: đẩy c vào ngăn xếp
- : lấy c và b ra khỏi ngăn xếp, đẩy b-c vào ngăn xếp
= : in giá trị b-c
* : lấy 2 toán hạng từ ngăn xếp là trị (b-c) và a, tính a * (b-c), đưa kết quả vào ngăn xếp.
? d: đẩy d vào ngăn xếp.
+ : lấy 2 toán hạng từ ngăn xếp là d và trị (a * (b-c)), tính (a * (b-c)) + d, đưa kết quả vào ngăn xếp.
= : in kết quả (a * (b-c)) + d

Ưu điểm của cách tính Balan ngược là mọi biểu thức phức tạp đều có thể được biểu diễn không cần cặp dấu ngoặc ().

Cách biểu diễn Balan ngược rất tiện lợi trong các trình biên dịch cũng như các phép tính toán.

Hàm phụ trợ **get_command** nhận lệnh từ người sử dụng, kiểm tra hợp lệ và chuyển thành chữ thường bởi **tolower()** trong thư viện cctype.

```
int main()
/*
post: chương trình thực hiện tính toán trị của biểu thức số học dạng postfix do người sử dụng
      nhập vào.
uses: lớp Stack và các hàm introduction, instructions, do_command,  get_command.
*/

{
   Stack<double> stored_numbers;
   introduction(); // Giới thiệu về chương trình.
   instructions(); // Xuất các hướng dẫn sử dụng chương trình.
   while (do_command(get_command(), stored_numbers));
}
```

```
char get_command()
/*
post: trả về một  trong những ký tự hợp lệ do người sử dụng gõ vào (?, =, +, -, *, /, q).
*/
{
```

```
   char command;
   bool waiting = true;
   cout << "Select command and press <Enter>:";

   while (waiting) {
      cin >> command;
      command = tolower(command);
      if (command == '?' || command == '=' || command == '+' ||
          command == '-' || command == '*' || command == '/' ||
          command == 'q' ) waiting = false;
      else {
         cout << "Please enter a valid command:"   << endl
              << "[?]push to stack   [=]print top" << endl
              << "[+] [-] [*] [/]   are arithmetic operations" << endl
              << "[Q]uit." << endl;
      }
   }
   return command;
}
```

Ngăn xếp stored_numbers làm thông số cho hàm do_command được khai báo là tham chiếu do nó cần phải thay đổi khi hàm được gọi.

```
bool do_command(char command, Stack<double> &numbers)
/*
pre:  command chứa ký hiệu của phép tính số học (+, - *, /) hoặc các ký tự đã quy định (q, =, ?).
post: Việc xử lý tùy thuộc thông số  command.  Hàm trả về true, ngoại trừ trường hợp kết thúc
      viêc tính toán khi command là 'q'.
uses: lớp Stack.
*/
{
   double p, q;
   switch (command) {
   case '?':
      cout << "Enter a real number: " << flush;
      cin >> p;
      if (numbers.push(p) == overflow)
         cout << "Warning: Stack full, lost number" << endl;
      break;

   case '=':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else
         cout << p << endl;
      break;

   case '+':
      if (numbers.top(p) == underflow)
         cout << "Stack empty" << endl;
      else {
         numbers.pop();
         if (numbers.top(q) == underflow) {
            cout << "Stack has just one entry" << endl;
            numbers.push(p);
         }

         else {
```

```
            numbers.pop();
            if (numbers.push(q + p) == overflow)
               cout << "Warning: Stack full, lost result" << endl;
         }
      }
      break;

   //   Add options for further user commands.

    case 'q':
      cout << "Calculation finished.\n";
      return false;
   }
   return true;
}
```

14.3.2. Ứng dụng chuyển đổi biểu thức dạng *infix* thành dạng *postfix*

Ngược với biểu thức dạng *postfix, b*iểu thức dạng *infix* cho phép có các dấu ngoặc đóng mở quy ước về độ ưu tiên của các phép tính. Chúng ta có độ ưu tiên từ cao xuống thấp theo thứ tự sau đây:

Độ ưu tiên 2 (cao nhất): các phép tính * và /
Độ ưu tiên 1 : các phép tính + và -
Độ ưu tiên 0 : dấu (,)

Khi duyệt biểu thức *infix* từ trái sang phải, các toán hạng trong biểu thức *infix* đều được đưa ngay vào biểu thức *postfix*, các toán tử cần được hoãn lại nên được đưa vào ngăn xếp. Trong biểu thức *postfix*, toán tử nào gặp trước sẽ được tính toán trước. Do đó, từ biểu thức *infix,* trước khi một toán tử nào đó cần được đưa vào ngăn xếp thì phải lấy từ đỉnh ngăn xếp tất cả các toán tử có độ ưu tiên cao hơn nó để đặt vào biểu thức *postfix* trước. Riêng trường hợp độ ưu tiên của toán tử đang cần đưa vào ngăn xếp bằng với độ ưu tiên của toán tử đang ở đỉnh ngăn xếp, thì toán tử ở đỉnh ngăn xếp cũng được lấy ra trước nếu các toán tử này là các toán tử kết hợp trái (Việc tính toán xử lý từ trái sang phải).

Chúng ta quan sát ba ví dụ sau đây:

a + b * c được chuyển thành a b c * + (1)
a * b **+** c được chuyển thành a b * c **+** (2)
a + b **-** c được chuyển thành a b + c **-** (3)

Ví dụ 1, dấu + vẫn ở trong ngăn xếp, và dấu * được đẩy vào ngăn xếp.
Ví dụ 2, dấu * được lấy ra khỏi ngăn xếp trước khi đưa dấu + vào.
Ví dụ 3, dấu + được lấy ra khỏi ngăn xếp trước khi đưa dấu - vào.

Trong trường hợp có dấu ngoặc, dấu mở '(' cần được lưu trong ngăn xếp cho đến khi gặp dấu đóng ')' tương ứng. Chúng ta quy ước độ ưu tiên của dấu ngoặc mở thấp nhất là hợp lý. Mọi toán tử xuất hiện sau dấu ngoặc mở đều không thể làm cho dấu này được lấy ra khỏi ngăn xếp, trừ khi dấu ngoặc đóng tương ứng được duyệt đến. Khi gặp dấu đóng ')', xem như kết thúc mọi việc tính toán trongcặp dấu (), mọi toán tử còn nằm trên dấu mở '(' trong ngăn xếp đều được lấy ra để đưa vào biểu thức dạng *postfix*. Do các dấu ngoặc không bao giờ xuất hiện trong biểu thức *postfix,* dấu '(' được đặt vào ngăn xếp để chờ dấu ')' tương ứng, khi nó được lấy ra thì sẽ bị vất bỏ. Dấu ')' để giải quyết cho dấu '(', và cũng sẽ bị vất đi.

Các toán tử +, -, *, / đều là các toán tử kết hợp từ trái sang phải. Biểu thức a - b - c (được hiểu là `(a-b)-c`) được chuyển đổi thành a b - c - (không phải a b c - -). Với một số toán tử kết hợp từ phải sang trái, chẳng hạn phép tính lũy thừa thì 2^2^3 = 2^(2^3)= 2^8=256, không phải (2^2)^3= 4^3=64, thì các xử lý trên cần được sửa đổi cho hợp lý. Chương trình hoàn chỉnh chuyển đổi biểu thức dang *infix* sang biểu thức dạng *postfix*, cũng như việc xử lý đặc biệt cho trường hợp toán tử kết hợp phải được xem như bài tập.

14.4. Giải thuật quay lui (*backtracking*)

Ngăn xếp còn được sử dụng trong các giải thuật quay lui nhằm lưu lại các thông tin đã từng duyệt qua để có thể quay ngược trở lại. Chúng ta sẽ xem xét các ví dụ sau đây.

14.4.1. Ứng dụng trong bài toán tìm đích (*goal seeking*).

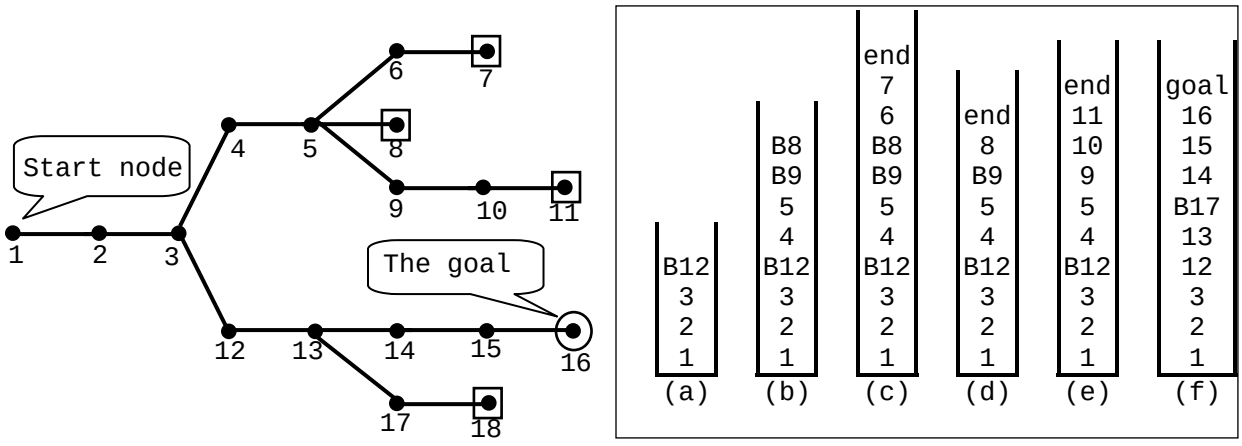
Hình 14.1 minh họa cho bài toán tìm đích. Chúng ta có một nút bắt đầu và một nút gọi là đích đến. Để đơn giản, chúng ta xét đồ thị không có chu trình và chỉ có duy nhất một đường đi từ nơi bắt đầu đến đích. Nhìn hình vẽ chúng ta có thể nhận ra ngay đường đi này. Tuy nhiên máy tính cần một giải thuật thích hợp để tìm ra được con đường này.

Chúng ta bắt đầu từ nút 1, sang nút 2 và nút 3. Tại nút 3 có 2 ngã rẽ, giả sử chúng ta đi theo đường trên, đến nút 4 và nút 5. Tại nút 5 chúng ta lại đi theo đường trên đến nút 6 và nút 7. Đến đây chúng ta không còn đường đi tiếp và cũng chưa tìm được đích cần đến, chúng ta phải quay trở lại nút 5 để chọn lối đi khác. Tại nút 8 chúng ta lại phải quay lại nút 5 để đi sang nút 9,.... Bằng cách này, từ nút 13, khi chúng ta tìm được nút 16 thì chúng ta không cần phải quay lui để thử với nút 17, 18 nữa. Giải thuật kết thúc khi tìm thấy đích đến.

Giải thuật của chúng ta cần lưu các nút để quay lại. So sánh nút 3 và nút 5, chúng ta thấy rằng trên đường đi chúng ta gặp nút 3 trước, nhưng diểm quay về để thử trước lại là nút 5. Do đó cấu trúc dữ liệu thích hợp chính là ngăn xếp với

nguyên tắc FILO của nó. Ngoài ra nếu chúng ta lưu nút 3 và nút 5 thì có sự bất tiện ở chỗ là khi quay về, thông tin lấy từ ngăn xếp không cho chúng ta biết các nhánh nào đã được duyệt qua và các nhánh nào cần tiếp tục duyệt. Do đó, tại nút 3, trước khi đi sang 4, chúng ta lưu nút 12, tại nút 5, trước khi đi sang 6 chúng ta lưu nút 8 và nút 9,...

Với giải thuật trên chúng ta có thể tìm đến đích một cách dễ dàng. Tuy nhiên, nếu bài toán yêu cầu in ra các nút trên đường đi từ nút bắt đầu đến đích thì chúng ta chưa làm được. Như vậy chúng ta cũng cần phải lưu cả các nút trên đường mà chúng ta đã đi qua. Những nút nằm trên những đoạn đường không dẫn đến đích sẽ được dỡ bỏ khỏi ngăn xếp khi chúng ta quay lui. Ở đây chúng ta gặp phải một vấn đề cũng tương đối phổ biến trong một số bài toán khác, đó là những gì chúng ta bỏ vào ngăn xếp không có cùng mục đích. Có hai nhóm thông tin khác nhau: một là các nút nằm trên đường đang đi qua, hai là các nút nằm trên các nhánh rẽ khác mà chúng ta sẽ lần lượt thử tiếp khi gặp thất bại trên con đường đang đi. Trong những trường hợp như vậy, việc giải quyết rất là dễ dàng: chúng ta dùng cách đánh dấu để phân biệt từng trường hợp, khi lấy ra khỏi ngăn xếp, căn cứ vào cách đánh dấu này chúng ta sẽ biết phải xử lý như thế nào cho thích hợp (Việc duyệt cây theo thứ tự LRN trong chương 10 nếu dùng ngăn xếp cũng là một ví dụ). Trong hình 14.1, ký tự B trong ngăn xếp cho biết đó là những nút dành cho việc quay lui (*Backtracking*) để thử với nhánh khác. Vậy khi gặp điểm cuối của một con đường không dẫn đến đích, chúng ta dỡ bỏ khỏi ngăn xếp các nút cho đến khi gặp một nút có ký tự 'B', bỏ lại nút này vào ngăn xếp (không còn ký tự 'B'), và đi tiếp các nút kế tiếp theo nút này. Cuối cùng khi gặp đích, con đường được tìm thấy chính là các nút đang lưu trong ngăn xếp mà không có ký tự 'B' ở đầu.



Hình 14.1- Ví dụ và ngăn xếp minh họa quá trình *backtracking*.

```
Algorithm seek_goal(val graph_type graph, val node_type start,
                                                  val node_type goal)
/*
pre: graph chứa đồ thị không có chu trình, trong đó có một nút start và một nút goal.
post: nếu đường đi từ start đến goal được tìm thấy sẽ được in ra theo thứ tự từ goal ngược
     về start
*/
{
   1. Stack<node_type> nodes. // node_type là kiểu dữ liệu thích hợp.
   2. node_type current_node = start
   3. boolean fail = FALSE
   4. loop ((current_node chưa là goal) AND (!fail))
      1. nodes.push(current_node)
      2. if (current_node là điểm rẽ nhánh)
         1. next_node = nút rẽ vào 1 nhánh
         2. loop (còn nhánh chưa đưa vào ngăn xếp) // đưa các nhánh
                                                   // còn lại vào ngăn xếp.
            1. nodes.push(nút rẽ vào 1 nhánh, có kèm 'B')
         3. endloop
      3. else
         1. if (tồn tại nút kế)
            1. next_node = nút kế
         2. else
            1. repeat
               1. if (nodes.empty())
                  1. fail = TRUE
               2. else
                  1. nodes.top(next_node)
                  2. nodes.pop()
               3. endif
            2. until ((fail) OR (next_node có 'B'))
         3. endif
      4. endif
      5. current_node = next_node
   5. endloop
   6. if (!fail)
      1. print("The path to your goal is:")
      2. print(current_node)  // chính là goal
      3. loop (!nodes.empty())
         1. nodes.top(current_node)
         2. nodes.pop()
         3. if (current_node không có ký tự 'B')
            1. print(current_node)
         4. endif
      4. endloop
   7. else
      1. print("Path not found!")
   8. endif
}
```

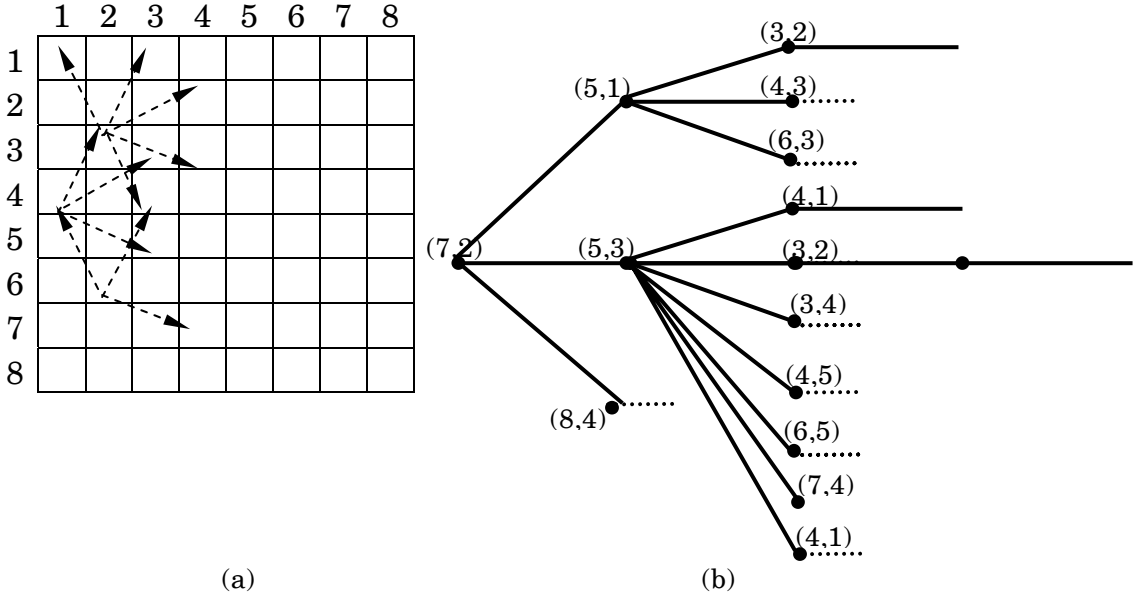
Trên đây là mã giả cho bài toán tìm đích, cấu trúc dữ liệu để chứa đồ thị chúng ta sẽ được tìm hiểu sau trong chương 13.

14.4.2. Bài toán mã đi tuần và bài toán tám con hậu

Ví dụ tiếp theo liên quan đến bài toán mã đi tuần. Thực ra bài toán tám con hậu được trình bày trong chương 6 cũng hoàn toàn tương tự. Sinh viên có thể tham khảo ý tưởng được trình bày dưới đây để giải bài toán tám con hậu sử dụng ngăn xếp thay vì đệ quy.

Với bàn cờ 64 ô, bài toán mã đi tuần yêu cầu chúng ta chỉ ra đường đi cho con mã bắt đầu từ một ô nào đó, lần lượt đi qua tất cả các ô, mà không có ô nào lập lại hơn một lần.

Từ một vị trí, con mã có tối đa là 8 vị trí chung quanh có thể đi được. Giải thuật quay lui rất gần với hướng suy nghĩ tự nhiên của chúng ta. Trong các khả năng có thể, chúng ta thường chọn ngẫu nhiên một khả năng để đi. Và với vị trí mới chúng ta cũng làm điều tương tự. Mỗi ô đi qua chúng ta đánh dấu lại. Trong quá trình thử này, nếu có lúc không còn khả năng lựa chọn nào khác vì các ô nằm trong khả năng đi tiếp đều đã được đánh dấu, chúng ta cần phải lùi lại để thử những khả năng khác. Thay vì biểu diễn đường đi cho con mã trên bàn cờ (hình 14.2a), chúng ta vẽ lại các khả năng đi và thấy chúng không khác gì so với bài toán tìm đích (hình 14.2b). Và như vậy, chúng ta thấy cách sử dụng ngăn xếp trong những bài toán dạng này hoàn toàn đơn giản và tương tự. Phương pháp quay lui trong trường hợp này đôi khi còn được gọi là phương pháp thử sai (*trial and error method*).



Hình 14.2- Bài toán mã đi tuần.

Hình trên đây minh họa bài toán mã đi tuần với điểm bắt đầu là (7,2). Đích đến không cụ thể như bài toán trên, mà đường đi cần tìm chính là đường đi nào trong đồ thị này có đủ 64 nút. Bài toán này phụ thuộc vào số ô của bàn cờ và vị

trí bắt đầu của con mã, khả năng có lời giải và có bao nhiêu lời giải chúng ta không xem xét ở đây. Chúng ta chỉ quan tâm đến cách sử dụng ngăn xếp trong những bài toán tương tự. Bản chất những bài này đều có cùng một đặc trưng của bài toán tìm đích trên một đồ thị không có chu trình. Đích đến có thể là nhiều hơn một (tương ứng với nhiều lời giải được tìm thấy) như trong bài toán tám con hậu mà cách giải đệ quy được trình bày trong chương 6. Sự khác nhau cơ bản giữa chúng chỉ là một vài kỹ thuật nho nhỏ dùng để chuyển từ dữ liệu ban đầu của bài toán sang dạng đồ thị biểu diễn các khả năng di chuyển trong quá trình tìm đến đích mà thôi.