Chương 13 – ĐỒ THỊ

Chương này trình bày về các cấu trúc toán học quan trọng được gọi là đồ thị. Đồ thị thường được ứng dụng trong rất nhiều lĩnh vực: điều tra xã hội, hóa học, địa lý, kỹ thuật điện,.... Chúng ta sẽ tìm hiểu các phương pháp biểu điễn đồ thị bằng các cấu trúc dữ liệu và xây dựng một số giải thuật tiêu biểu liên quan đến đồ thị.

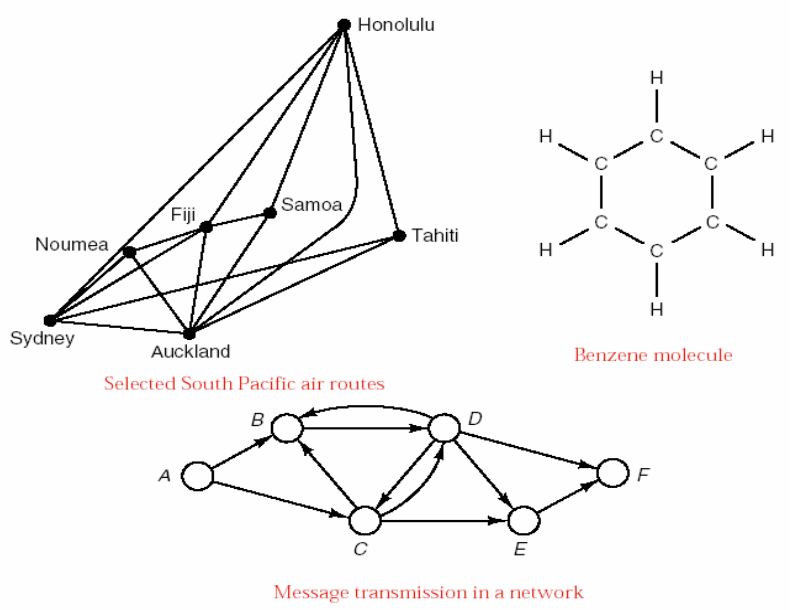
13.1. Nền tảng toán học

13.1.1. Các định nghĩa và ví dụ

Một đồ thị (*graph*) G gồm một tập V chứa các đỉnh của đồ thị, và tập E chứa các cặp đỉnh khác nhau từ V. Các cặp đỉnh này được gọi là các cạnh của G. Nếu $e = (\nu, \mu)$ là một cạnh có hai đỉnh ν và μ, thì chúng ta gọi ν và μ nằm trên e, và e nối với ν và μ. Nếu các cặp đỉnh không có thứ tự, G được gọi là đồ thị vô hướng (*undirected graph*), ngược lại, G được gọi là đồ thị có hướng (*directed graph*). Thông thường đồ thị có hướng được gọi tắt là *digraph*, còn từ *graph* thường mang nghĩa là đồ thị vô hướng. Cách tự nhiên để vẽ đồ thị là biểu diễn các đỉnh bằng các điểm hoặc vòng tròn, và các cạnh bằng các đường thẳng hoặc các cung nối các đỉnh. Đối với đồ thị có hướng thì các đường thẳng hay các cung cần có mũi tên chỉ hướng. Hình 13.1 minh họa một số ví dụ về đồ thị.

Đồ thị thứ nhất trong hình 13.1 có các thành phố là các đỉnh, và các tuyến bay là các cạnh. Trong đồ thị thứ hai, các nguyên tử *hydro* và *carbon* là các đỉnh, các liên kết hóa học là các cạnh. Hình thứ ba là một đồ thị có hướng cho biết khả năng truyền nhận dữ liệu trên mạng, các nút của mạng (A, B, ..., F) là các đỉnh và các đường nối các nút là có hướng. Đôi khi cách chọn tập đỉnh và tập cạnh cho đồ thị phụ thuộc vào giải thuật mà chúng ta dùng để giải bài toán, chẳng hạn bài toán liên quan đến quy trình công việc, bài toán xếp thời khóa biểu,...

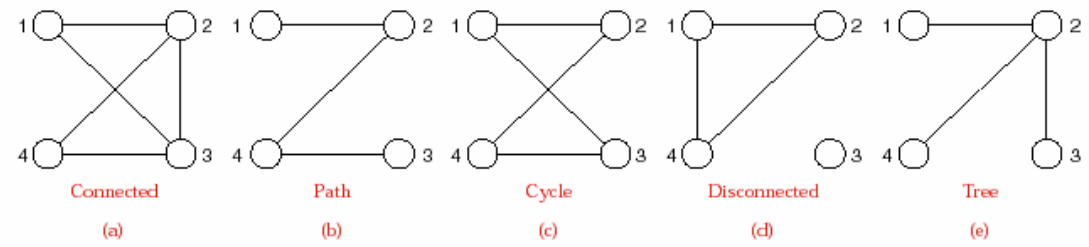
Đồ thị được sử dụng để mô hình hóa rất nhiều dạng quá trình cũng như cấu trúc khác nhau. Đồ thị có thể biểu diễn mạng giao thông giữa các thành phố, hoặc các thành phần của một mạch in điện tử và các đường nối giữa chúng, hoặc cấu trúc của một phân tử gồm các nguyên tử và các liên kết hóa học. Những người dân trong một thành phố cũng có thể được biểu diễn bởi các đỉnh của đồ thị mà các cạnh là các mối quan hệ giữa họ. Nhân viên trong một công ty có thể được biểu diễn trong một đồ thị có hướng mà các cạnh có hướng cho biết mối quan hệ của họ với những người quản lý. Những người này cũng có thể có những mối quan hệ "cùng làm việc" biểu diễn bởi các cạnh không hướng trong một đồ thị vô hướng.



Hình 13.1 – Các ví dụ về đồ thị

13.1.2. Đồ thị vô hướng

Một vài dạng của đồ thị vô hướng được minh họa trong hình 13.2. Hai đỉnh trong một đồ thị vô hướng được gọi là kề nhau (*adjacent*) nếu tồn tại một cạnh nối từ đỉnh này đến đỉnh kia. Trong đồ thị vô hướng trong hình 13.2 a, đỉnh 1 và 2 là kề nhau, đỉnh 3 và 4 là kề nhau, nhưng đỉnh 1 và đỉnh 4 không kề nhau. Một đường đi (*path*) là một dãy các đỉnh khác nhau, trong đó mỗi đỉnh kề với đỉnh kế tiếp. Hình (b) cho thấy một đường đi. Một chu trình (*cycle*) là một đường đi chứa ít nhất ba đỉnh sao cho đỉnh cuối cùng kề với đỉnh đầu tiên. Hình (c) là một chu trình. Một đồ thị được gọi là liên thông (*connected*) nếu luôn có một đường đi từ một đỉnh bất kỳ đến một đỉnh bất kỳ nào khác. Hình (a), (b), và (c) là các đồ thị liên thông. Hình (d) không phải là đồ thị liên thông. Nếu một đồ thị là không liên thông, chúng ta xem mỗi tập con lớn nhất các đỉnh liên thông nhau như một thành phần liên thông. Ví dụ, đồ thị không liên thông ở hình (d) có hai thành phần liên thông: một thành phần chứa các đỉnh 1,2 và 4; một thành phần chỉ có đỉnh 3.

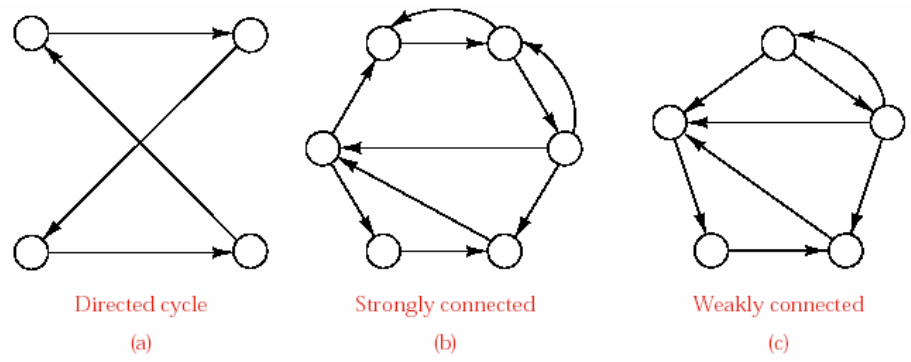


Hình 13.2 – Các dạng của đồ thị vô hướng

Phần (e) là một đồ thị liên thông không có chu trình. Chúng ta có thể nhận thấy đồ thị cuối cùng này thực sự là một cây, và chúng ta dùng đặc tính này để định nghĩa: Một cây tự do (*free tree*) được định nghĩa là một đồ thị vô hướng liên thông không có chu trình.

13.1.3. Đồ thị có hướng

Đối với các đồ thị có hướng, chúng ta có thể có những định nghĩa tương tự. Chúng ta yêu cầu mọi cạnh trong một đường đi hoặc một chu trình đều có cùng hướng, như vậy việc lần theo một đường đi hoặc một chu trình có nghĩa là phải di chuyển theo hướng chỉ bởi các mũi tên. Những đường đi (hay chu trình) như vậy được gọi là đường đi có hướng (hay chu trình có hướng). Một đồ thị có hướng được gọi là liên thông mạnh (*strongly connected*) nếu nó luôn có một đường đi có hướng từ một đỉnh bất kỳ đến một đỉnh bất kỳ nào khác. Trong một đồ thị có hướng không liên thông mạnh, nếu bỏ qua chiều của các cạnh mà chúng ta có được một đồ thị vô hướng liên thông thì đồ thị có hướng ban đầu được gọi là đồ thị liên thông yếu (*weakly connected*). Hình 13.3 minh họa một chu trình có hướng, một đồ thị có hướng liên thông mạnh và một đồ thị có hướng liên thông yếu.



Hình 13.3 – Các ví dụ về đồ thị có hướng

Các đồ thị có hướng trong phần (b) và (c) hình 13.3 có các cặp đỉnh có các cạnh có hướng theo cả hai chiều giữa chúng. Các cạnh có hướng là các cặp có thứ tự và các cặp có thứ tự (ν, μ) và (μ,ν) là khác nhau nếu $\nu \neq \mu$. Trong đồ thị vô hướng, chỉ có thể có nhiều nhất một cạnh nối hai đỉnh khác nhau. Tương tự, do các đỉnh trên một cạnh theo định nghĩa là phải khác nhau, không thể có một cạnh nối một đỉnh với chính nó. Tuy nhiên, cũng có những trường hợp mở rộng định nghĩa, người ta cho phép nhiều cạnh nối một cặp đỉnh, và một cạnh nối một đỉnh với chính nó.

13.2. Biểu diễn bằng máy tính

Nếu chúng ta chuẩn bị viết chương trình để giải quyết một bài toán có liên quan đến đồ thị, trước hết chúng ta phải tìm cách để biểu diễn cấu trúc toán học của đồ thị như là một dạng nào đó của cấu trúc dữ liệu. Có nhiều phương pháp

được dùng phổ biến, về cơ bản chúng khác nhau trong việc lựa chọn kiểu dữ liệu trừu tượng để biểu diễn đồ thị, cũng như nhiều cách hiện thực khác nhau cho mỗi kiểu dữ liệu trừu tượng. Nói cách khác, chúng ta bắt đầu từ một định nghĩa toán học, đó là đồ thị, sau đó chúng ta tìm hiểu cách mô tả nó như một kiểu dữ liệu trừu tượng (tập hợp, bảng, hay danh sách đều có thể dùng được), và cuối cùng chúng ta lựa chọn cách hiện thực cho kiểu dữ liệu trừu tượng mà chúng ta chọn.

13.2.1. Biểu diễn của tập hợp

Đồ thị được định nghĩa bằng một tập hợp, như vậy một cách hết sức tự nhiên là dùng tập hợp để xác định cách biểu diễn nó như là dữ liệu. Trước tiên, chúng ta có một tập các đỉnh, và thứ hai, chúng ta có các cạnh như là tập các cặp đỉnh. Thay vì thử biểu diễn tập các cặp đỉnh này một cách trực tiếp, chúng ta chia nó ra thành nhiều phần nhỏ bằng cách xem xét tập các cạnh liên quan đến từng đỉnh riêng rẽ. Nói một cách khác, chúng ta có thể biết được tất cả các cạnh trong đồ thị bằng cách nắm giữ tập E_v các cạnh có chứa v đối với mỗi đỉnh v trong đồ thị, hoặc, một cách tương đương, tập A_v gồm tất cả các đỉnh kề với v. Thật vậy, chúng ta có thể dùng ý tưởng này để đưa ra một định nghĩa mới tương đương cho đồ thị:

Định nghĩa: Một đồ thị có hướng G bao gồm tập V, gọi là các đỉnh của G, và, đối với mọi $v \in V$, có một tập con A_v, gọi là tập các đỉnh kề của v.

Từ các tập con A_v chúng ta có thể tái tạo lại các cạnh như là các cặp có thứ tự theo quy tắc sau: cặp (v, w) là một cạnh nếu và chỉ nếu $w \in A_v$. Xử lý cho tập các đỉnh dễ hơn là tập các cạnh. Ngoài ra, định nghĩa mới này thích hợp với cả đồ thị có hướng và đồ thị vô hướng. Một đồ thị là vô hướng khi nó thỏa tính chất đối xứng sau: $w \in A_v$ kéo theo $v \in A_w$ với mọi v, $w \in V$. Tính chất này có thể được phát biểu lại như sau: Một cạnh không có hướng giữa v và w có thể được xem như hai cạnh có hướng, một từ v đến w và một từ w đến v.

13.2.1.1. Hiện thực các tập hợp

Có nhiều cách để hiện thực tập các đỉnh trong cấu trúc dữ liệu và giải thuật. Cách thứ nhất là biểu diễn tập các đỉnh như là một danh sách các phần tử của nó, chúng ta sẽ tìm hiểu phương pháp này sau. Cách thứ hai, thường gọi là chuỗi các bit (*bit string*), lưu một trị *Boolean* cho mỗi phần tử của tập hợp để chỉ ra rằng nó có hay không có trong tập hợp. Để đơn giản, chúng ta sẽ xem các phần tử có thể có của tập hợp được đánh chỉ số từ 0 đến `max_set-1`, với `max_set` là số phần tử tối đa cho phép. Điều này có thể được hiện thực một cách dễ dàng bằng cách sử dụng thư viện chuẩn (*Standard Template Library- STL*)

`std::bitset<max_set>`, hoặc lớp có sử dụng `template` cho kích thước tập hợp của chúng ta như sau:

```
template <int max_set>
struct Set {
  bool is_element[max_set];
};
```

Đây chỉ là một cách hiện thực đơn giản nhất của khái niệm tập hợp. Sinh viên có thể thấy rằng không có gì ngăn cản chúng ta đặc tả và hiện thực một CTDL tập hợp với các phương thức hội, giao, hiệu, xét thành viên của nó,..., một cách hoàn chỉnh nếu như cần sử dụng tập hợp trong những bài toán lớn nào đó.

Giờ chúng ta đã có thể đặc tả cách biểu diễn thứ nhất cho đồ thị của chúng ta:

```
// Tương ứng hình 13.4-b
template <int max_size>
class Digraph {
   int count;   //  Số đỉnh của đồ thị, nhiều nhất là max_size
   Set<max_size> neighbors[max_size];
};
```

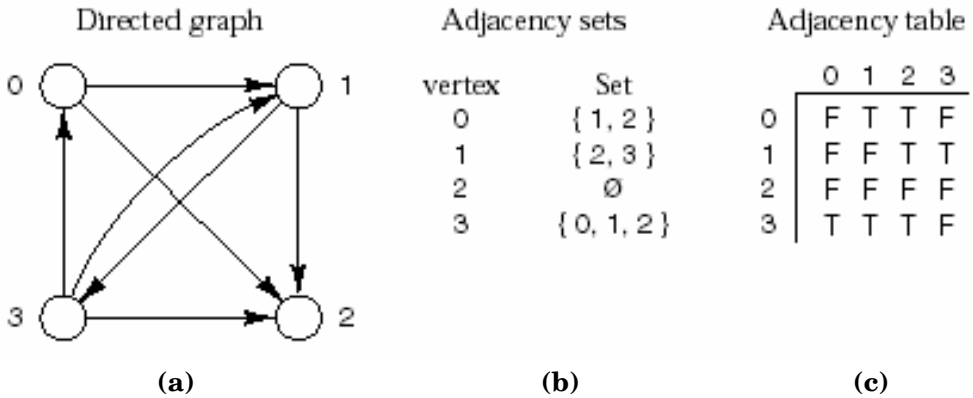
Trong cách hiện thực này, các đỉnh được đặt tên bằng các số nguyên từ 0 đến `count-1`. Nếu ν là một số nguyên thì phần tử `neighbors[ν]` của mảng là một tập các đỉnh kề với đỉnh ν.

13.2.1.2. Bảng kề

Trong cách hiện thực trên đây, cấu trúc `Set` được hiện thực như một mảng các phần tử kiểu `bool`. Mỗi phần tử chỉ ra rằng đỉnh tương ứng có là thành phần của tập hợp hay không. Nếu chúng ta thay thế tập các đỉnh kề này bằng một mảng, chúng ta sẽ thấy rằng mảng `neighbors` trong định nghĩa của lớp `Graph` có thể được biến đổi thành mảng các mảng (mảng hai chiều) như sau đây, và chúng ta gọi là bảng kề (*adjacency table*):

```
// Tương ứng hình 13.4-c
template <int max_size>
class Digraph {
   int count;   //  Số đỉnh của đồ thị, nhiều nhất là max_size.
   bool adjacency[max_size][max_size];
};
```

Bảng kề chứa các thông tin một cách tự nhiên như sau: `adjacency[v][w]` là `true` nếu và chỉ nếu đỉnh v là đỉnh kề của w. Nếu là đồ thị có hướng, `adjacency[v][w]` cho biết cạnh từ v đến w có trong đồ thị hay không. Nếu đồ thị vô hướng, bảng kề phải đối xứng, nghĩa là `adjacency[v][w]= adjacency[v][w]` với mọi v và w. Biểu diễn đồ thị bởi tập các đỉnh kề và bởi bảng kề được minh họa trong hình 13.4.



Hình 13.4 – Tập các đỉnh kề và bảng kề.

13.2.2. Danh sách kề

Một cách khác để biểu diễn một tập hợp là dùng danh sách các phần tử. Chúng ta có một danh sách các đỉnh, và, đối với mỗi đỉnh, có một danh sách các đỉnh kề. Chúng ta có thể xem xét cách hiện thực cho đồ thị bằng danh sách liên tục hoặc danh sách liên kết đơn. Tuy nhiên, đối với nhiều ứng dụng, người ta thường sử dụng các hiện thực khác của danh sách phức tạp hơn như cây nhị phân tìm kiếm, cây nhiều nhánh tìm kiếm, hoặc là `heap`. Lưu ý rằng, bằng cách đặt tên các đỉnh theo các chỉ số trong các cách hiện thực trước đây, chúng ta cũng có được cách hiện thực cho tập các đỉnh như là một danh sách liên tục.

13.2.2.1. Hiện thực dựa trên cơ sở là danh sách

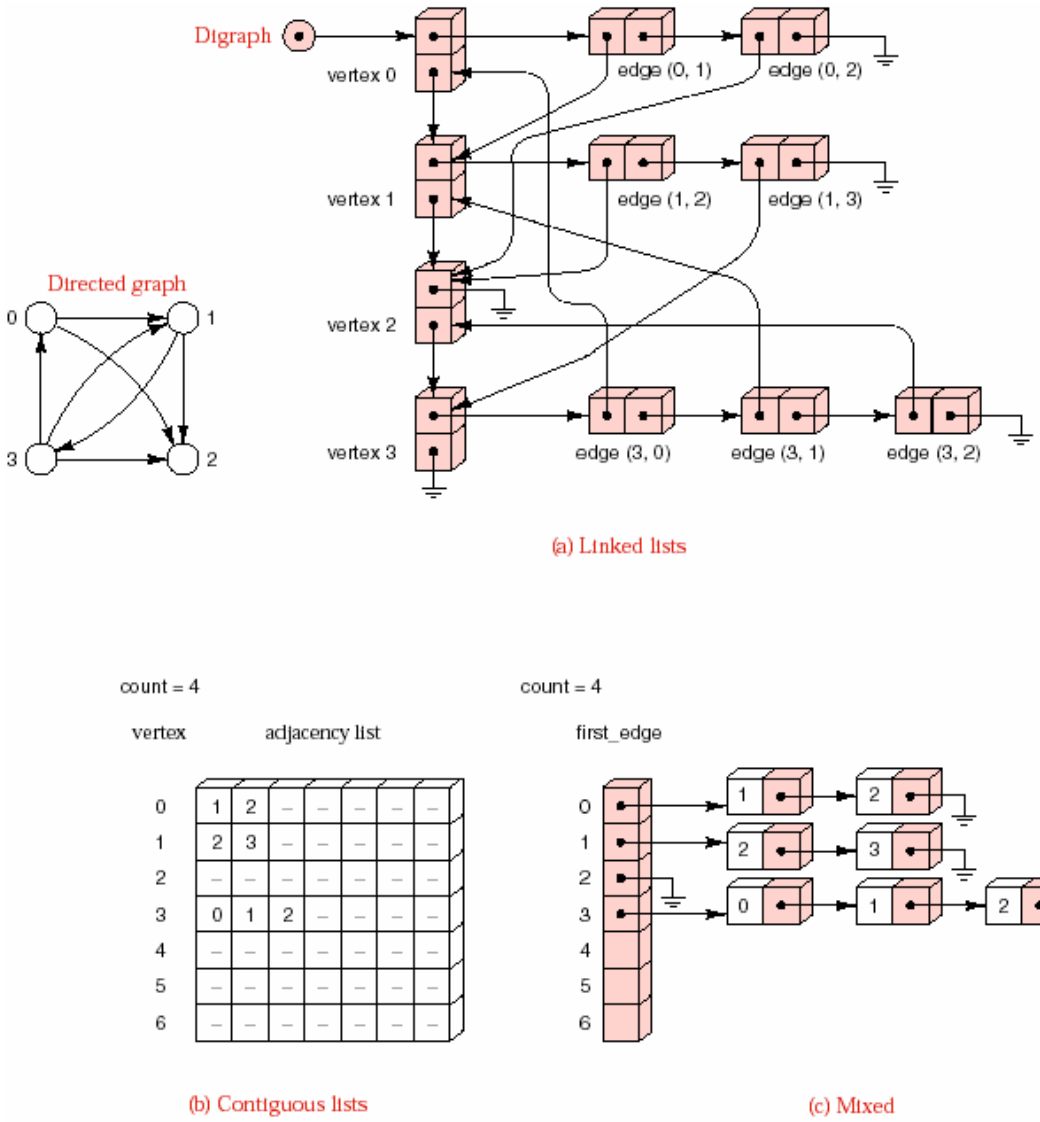
Chúng ta có được hiện thực của đồ thị dựa trên cơ sở là danh sách bằng cách thay thế các tập hợp đỉnh kề trước kia bằng các danh sách. Hiện thực này có thể sử dụng hoặc danh sách liên tục hoặc danh sách liên kết. Phần (b) và (c) của hình 13.5 minh họa hai cách hiện thực này.

```
// Tổng quát cho cả danh sách liên tục lẫn liên kết (hình 13.5-b và c).

typedef int Vertex;
template <int max_size>
class Digraph {
   int count;   //  Số đỉnh của đồ thị, nhiều nhất là max_size.
   List<Vertex> neighbors[max_size];
};
```

13.2.2.2. Hiện thực liên kết

Bằng cách sử dụng các đối tượng liên kết cho cả các đỉnh và cho cả các danh sách kề, đồ thị sẽ có được tính linh hoạt cao nhất. Hiện thực này được minh họa trong hình 13.5-a và có các định nghĩa như sau:



Hình 13.5 – Hiện thực đồ thị bằng các danh sách

```
class Edge;
class Vertex {
   Edge *first_edge;     //  Chỉ đến phần tử đầu của DSLK các đỉnh kề.
   Vertex *next_vertex; //  Chỉ đến phần tử kế trong DSLK các đỉnh có trong đồ thị.
};

class Edge {
   Vertex *end_point;   //  Chỉ đến một đỉnh kề với đỉnh mà danh sách này thuộc về.
   Edge *next_edge;     //  Chỉ đến phần tử biểu diễn đỉnh kề kế tiếp trong danh sách các
                                  đỉnh kề với một đỉnh mà danh sách này thuộc về.
};
```

```
class Digraph {
   Vertex *first_vertex;//  Chỉ đến phần tử đầu tiên trong danh sách các đỉnh của đồ thị.
};
```

13.2.3. Các thông tin khác trong đồ thị

Nhiều ứng dụng về đồ thị không những cần những thông tin về các đỉnh kề của một đỉnh mà còn cần thêm một số thông tin khác liên quan đến các đỉnh cũng như các cạnh. Trong hiện thực liên kết, các thông tin này có thể được lưu như các thuộc tính bổ sung bên trong các bản ghi tương ứng, và trong hiện thực liên tục, chúng có thể được lưu trong các mảng các phần tử bên trong các bản ghi. Lấy ví dụ trường hợp mạng các máy tính, nó được định nghĩa như một đồ thị trong đó mỗi cạnh có thêm thông tin là tải trọng của đường truyền từ máy này qua máy khác. Đối với nhiều giải thuật trên mạng, cách biểu diễn tốt nhất là dùng bảng kề, trong đó các phần tử sẽ chứa tải trọng thay vì một trị kiểu `bool`. Chúng ta sẽ quay lại vấn đề này sau trong chương này.

13.3. Duyệt đồ thị

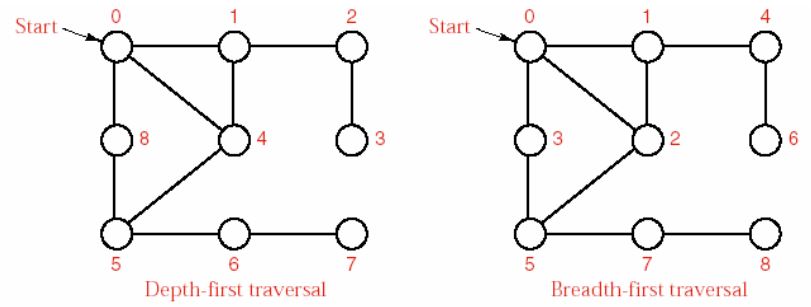
13.3.1. Các phương pháp

Trong nhiều bài toán, chúng ta mong muốn được khảo sát các đỉnh trong đồ thị theo một thứ tự nào đó. Tựa như đối với cây nhị phân chúng ta đã phát triển một vài phương pháp duyệt qua các phần tử một cách có hệ thống. Khi duyệt cây, chúng ta thường bắt đầu từ nút gốc. Trong đồ thị, thường không có đỉnh nào là đỉnh đặc biệt, nên việc duyệt qua đồ thị có thể bắt đầu từ một đỉnh bất kỳ nào đó. Tuy có nhiều thứ tự khác nhau để duyệt qua các đỉnh của đồ thị, có hai phương pháp được xem là đặc biệt quan trọng.

Phương pháp duyệt theo chiều sâu (*depth-first traversal*) trên một đồ thị gần giống với phép duyệt *preorder* cho một cây có thứ tự. Giả sử như phép duyệt vừa duyệt xong đỉnh v, và gọi w_1, w_2,...,w_k là các đỉnh kề với v, thì w_1 là đỉnh được duyệt kế tiếp, trong khi các đỉnh w_2,...,w_k sẽ nằm đợi. Sau khi duyệt qua đỉnh w_1 chúng ta sẽ duyệt qua tất cả các đỉnh kề với w_1, trước khi quay lại với w_2,...,w_k.

Phương pháp duyệt theo chiều rộng (*breadth-first traversal*) trên một đồ thị gần giống với phép duyệt theo mức (*level by level*) cho một cây có thứ tự. Nếu phép duyệt vừa duyệt xong đỉnh v, thì tất cả các đỉnh kề với v sẽ được duyệt tiếp sau đó, trong khi các đỉnh kề với các đỉnh này sẽ được đặt vào một danh sách chờ, chúng sẽ được duyệt tới chỉ sau khi tất cả các đỉnh kề với v đã được duyệt xong.

Hình 13.6 minh họa hai phương pháp duyệt trên, các con số tại các đỉnh biểu diễn thứ tự mà chúng được duyệt đến.



Hình 13.6 - Duyệt đồ thị

13.3.2. Giải thuật duyệt theo chiều sâu

Phương pháp duyệt theo chiều sâu thường được xây dựng như một giải thuật đệ quy. Các công việc cần làm khi gặp một đỉnh v là:

```
visit(v);
for (mỗi đỉnh w kề với đỉnh v)
    traverse(w);
```

Tuy nhiên, trong phép duyệt đồ thị, có hai điểm khó khăn mà trong phép duyệt cây không có. Thứ nhất, đồ thị có thể chứa chu trình, và giải thuật của chúng ta có thể gặp lại một đỉnh lần thứ hai. Để ngăn chặn đệ quy vô tận, chúng ta dùng một mảng các phần tử kiểu `bool` **`visited`**, `visited[v]` sẽ là `true` khi v vừa được duyệt xong, và chúng ta luôn xét trị của `visited[w]` trước khi xử lý cho w, nếu trị này đã là `true` thì w không cần xử lý nữa. Điều khó khăn thứ hai là, đồ thị có thể không liên thông, và giải thuật duyệt có thể không đạt được đến tất cả các đỉnh của đồ thị nếu chỉ bắt đầu đi từ một đỉnh. Do đó chúng ta cần thực hiện một vòng lặp để có thể bắt đầu từ mọi đỉnh trong đồ thị, nhờ vậy chúng ta sẽ không bỏ sót một đỉnh nào. Với những phân tích trên, chúng ta có phác thảo của giải thuật duyệt đồ thị theo chiều sâu dưới đây. Chi tiết hơn cho giải thuật còn phụ thuộc vào cách chọn lựa hiện thực của đồ thị và các đỉnh, và chúng ta để lại cho các chương trình ứng dụng.

```
template <int max_size>
void Digraph<max_size>::depth_first(void (*visit)(Vertex &)) const
/*
post: Hàm *visit được thực hiện tại mỗi đỉnh của đồ thị một lần, theo thứ tự duyệt theo chiều
      sâu.
uses: Hàm traverse thực hiện duyệt theo chiều sâu.
*/
```

```
{
   bool visited[max_size];
   Vertex v;
   for (all v in G) visited[v] = false;
   for (all v in G) if (!visited[v])
      traverse(v, visited, visit);
}
```

Việc đệ quy được thực hiện trong hàm phụ trợ **traverse**. Do hàm này cần truy nhập vào cấu trúc bên trong của đồ thị, nó phải là hàm thành viên của lớp Digraph. Ngoài ra, do traverse là một hàm phụ trợ và chỉ được sử dụng trong phương thức depth_first, nó nên được khai báo private bên trong lớp.

```
template <int max_size>
void Digraph<max_size>::traverse(Vertex &v, bool visited[],
                                 void (*visit)(Vertex &)) const
/*
pre:  v là một đỉnh của đồ thị Digraph.
post: Duyệt theo chiều sâu, hàm *visit sẽ được thực hiện tại v và tại tất cả các đỉnh có thể
      đến được từ v.
uses: Hàm traverse một cách đệ quy.
*/
{  Vertex w;
   visited[v] = true;
   (*visit)(v);
   for (all w adjacent to v)
     if (!visited[w])
        traverse(w, visited, visit);
}
```

13.3.3. Giải thuật duyệt theo chiều rộng

Do sử dụng đệ quy và lập trình với ngăn xếp về bản chất là tương đương, chúng ta có thể xây dựng giải thuật duyệt theo chiều sâu bằng cách sử dụng ngăn xếp. Khi một đỉnh đang được duyệt thì các đỉnh kề của nó được đẩy vào ngăn xếp, khi một đỉnh vừa được duyệt xong thì đỉnh kế tiếp cần duyệt là đỉnh được lấy ra từ ngăn xếp. Giải thuật duyệt theo chiều rộng cũng tương tự như giải thuật vừa được đề cập đến trong việc duyệt theo chiều sâu, tuy nhiên hàng đợi cần được sử dụng thay cho ngăn xếp.

```
template <int max_size>
void Digraph<max_size>::breadth_first(void (*visit)(Vertex &)) const
/*
post: Hàm *visit được thực hiện tại mỗi đỉnh của đồ thị một lần, theo thứ tự duyệt theo chiều
      rộng.
uses: Các phương thức của lớp Queue.
*/
{  Queue q;
   bool visited[max_size];
   Vertex v, w, x;
   for (all v in G) visited[v] = false;
```

```
   for (all v in G)
     if (!visited[v]) {
        q.append(v);
        while (!q.empty()){
           q.retrieve(w);
           if (!visited[w]) {
              visited[w] = true;
              (*visit)(w);
              for (all x adjacent to w)
                 q.append(x);
           }
           q.serve();
        }
     }
}
```

13.4. Sắp thứ tự topo

13.4.1. Đặt vấn đề

Nếu G là một đồ thị có hướng không có chu trình, thì thứ tự topo (*topological order*) của G là một cách liệt kê tuần tự mọi đỉnh trong G sao cho, với mọi ν, $\mu \in G$, nếu có một cạnh từ ν đến μ, thì ν nằm trước μ.

Trong suốt phần này, chúng sẽ chỉ xem xét các đồ thị có hướng không có chu trình. Thuật ngữ *acyclic* có nghĩa là một đồ thị không có chu trình. Các đồ thị như vậy xuất hiện trong rất nhiều bài toán. Như một ví dụ đầu tiên về thứ tự topo, chúng ta hãy xem xét các môn học trong một trường đại học như là các đỉnh của đồ thị, trong đó một cạnh nối từ môn này đến môn kia có nghĩa là môn thứ nhất là môn tiên quyết của môn thứ hai. Như vậy thứ tự topo sẽ liệt kê tất cả các môn sao cho mọi môn tiên quyết của một môn sẽ nằm trước môn đó. Ví dụ thứ hai là từ điển các thuật ngữ kỹ thuật. Các từ trong từ điển được sắp thứ tự sao cho không có từ nào được sử dụng trong một định nghĩa của từ khác trước khi chính nó được định nghĩa. Tương tự, các tác giả của các sách sử dụng thứ tự topo cho các đề mục trong sách. Hai thứ tự topo khác nhau của một đồ thị có hướng được minh họa trong hình 13.7.

Chúng ta sẽ xây dựng hàm để sinh ra thứ tự topo cho các đỉnh của một đồ thị không có chu trình theo hai cách: sử dụng phép duyệt theo chiều sâu và phép duyệt theo chiều rộng. Cả hai phương pháp được dùng cho một đối tượng của lớp Digraph sử dụng hiện thực dựa trên cơ sở là danh sách. Chúng ta có đặc tả lớp như sau:

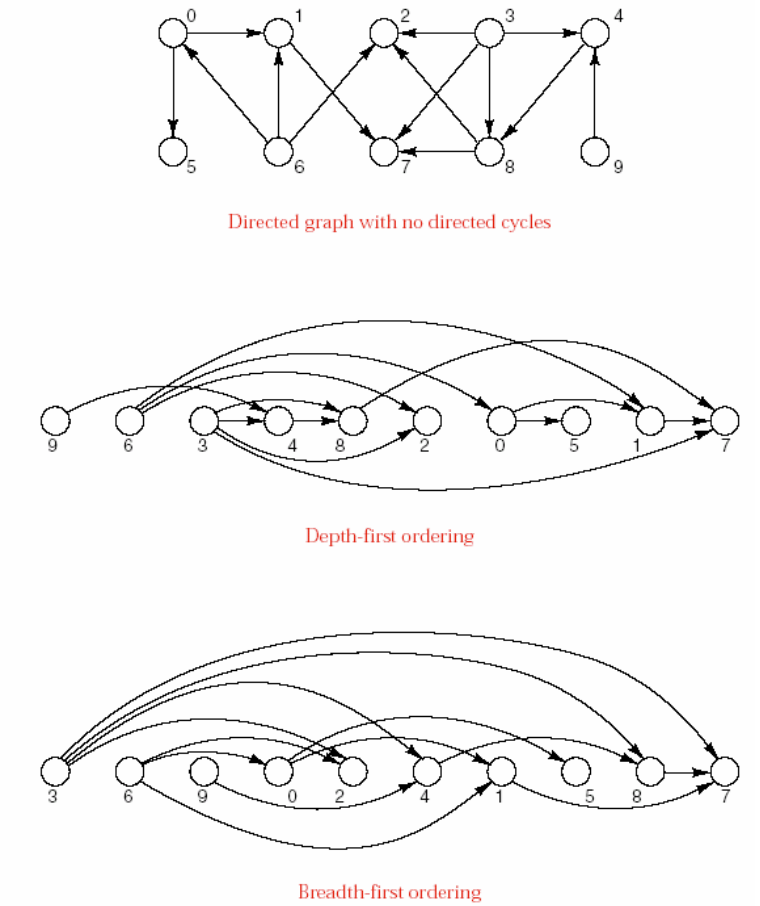
```
typedef int Vertex;

template <int graph_size>
class Digraph {
public:
   Digraph();
   void read();
   void write();
```

```
//  Các phương thức sắp thứ tự topo.
   void depth_sort(List<Vertex> &topological_order);
   void breadth_sort(List<Vertex> &topological_order);

private:
   int count;
   List <Vertex> neighbors[graph_size];
   void recursive_depth_sort(Vertex v, bool visited[],
                             List<Vertex> &topological_order);
};
```

Hàm phụ trợ **recursive_depth_sort** được sử dụng bởi phương thức **depth_sort**. Cả hai phương pháp sắp thứ tự đều sẽ tạo ra một danh sách các đỉnh của đồ thị theo thứ tự topo tương ứng.



Hình 13.7 – Các thứ tự topo của một đồ thị có hướng

13.4.2. Giải thuật duyệt theo chiều sâu

Trong thứ tự topo, mỗi đỉnh phải xuất hiện trước mọi đỉnh kề của nó trong đồ thị. Giải thuật duyệt theo chiều sâu này đặt dần các đỉnh vào mảng thứ tự topo từ phải sang trái. Bắt đầu từ một đỉnh chưa từng được duyệt đến, chúng ta cần gọi đệ quy để đến được các đỉnh mà không còn đỉnh kề, các đỉnh này sẽ được đặt vào mảng thứ tự topo ở các vị trí cuối mảng. Tính từ phải sang trái trong mảng thứ tự topo này, khi các đỉnh kề của một đỉnh đã được duyệt xong, thì chính đỉnh đó

có thể có mặt trong mảng. Đó chính là lúc các lần gọi đệ quy bên trong lùi về lần gọi đệ quy bên ngoài. Phương pháp này là một cách hiện thực trực tiếp của thủ tục duyệt theo chiều sâu một cách tổng quát đã được trình bày ở trên. Điểm khác biệt là việc xử lý tại mỗi đỉnh (ghi vào mảng thứ tự topo) chỉ được thực hiện sau khi các đỉnh kề của nó đã được xử lý.

```
template <int graph_size>
void Digraph<graph_size>::depth_sort(List<Vertex> &topological_order)
/*
post: Các đỉnh của một đồ thị có hướng không có chu trình được xếp theo thứ tự topo tương ứng
      cách duyệt đồ thị theo chiều sâu.
uses: Các phương thức của lớp List, hàm đệ quy recursive_depth_sort.
*/
{
   bool visited[graph_size];
   Vertex v;
   for (v = 0; v < count; v++) visited[v] = false;
   topological_order.clear();
   for (v = 0; v < count; v++)
      if (!visited[v]) // Thêm các đỉnh kề của v và sau đó là v vào mảng tứ tự topo từ
                                                                    phải sang
                                                                    trái.
         recursive_depth_sort(v, visited, topological_order);
}
```

Hàm phụ trợ **recursive_depth_sort** thực hiện việc đệ quy, dựa trên phác thảo của hàm **traverse** tổng quát, trước hết đặt tất cả các đỉnh sau của một đỉnh v vào các vị trí của chúng trong thứ tự topo, sau đó mới đặt v vào.

```
template <int graph_size>
void Digraph<graph_size>::recursive_depth_sort(Vertex v, bool *visited,
                          List<Vertex> &topological_order)
/*
pre:  Đỉnh v chưa có trong mảng thứ tự topo.
post: Thêm các đỉnh kề của v và sau đó là v vào mảng tứ tự topo từ phải sang trái.
uses: Các phương thức của lớp List và hàm đệ quy recursive_depth_sort.
*/
{  visited[v] = true;
   int degree = neighbors[v].size();
   for (int i = 0; i < degree; i++) {
      Vertex w;
      neighbors[v].retrieve(i, w);  // Một đỉnh kề của v.
      if (!visited[w])              // Duyệt tiếp xuống đỉnh w.
         recursive_depth_sort(w, visited, topological_order);
   }
   topological_order.insert(0, v);    // Đặt v vào mảng thứ tự topo.
}
```

Do giải thuật này duyệt qua mỗi đỉnh của đồ thị chính xác một lần và xem xét mỗi cạnh cũng một lần, đồng thời nó không hề thực hiện việc tìm kiếm nào, nên thời gian chạy là O(n+e), với n là số đỉnh và e là số cạnh của đồ thị.

13.4.3. Giải thuật duyệt theo chiều rộng

Trong thứ tự topo theo chiều rộng của một đồ thị có hướng không có chu trình, chúng ta bắt đầu bằng cách tìm các đỉnh có thể là các đỉnh đầu tiên trong thứ tự topo và sau đó chúng ta áp dụng nguyên tắc rằng, mọi đỉnh cần phải xuất hiện trước tất cả các đỉnh sau của nó trong thứ tự topo. Giải thuật này sẽ lần lượt đặt các đỉnh của đồ thị vào mảng thứ tự topo từ trái sang phải.

Các đỉnh có thể là đỉnh đầu tiên chính là các đỉnh không là đỉnh sau của bất kỳ đỉnh nào trong đồ thị. Để tìm được chúng, chúng ta tạo một mảng **predecessor_count**, mỗi phần tử tại chỉ số v chứa số đỉnh đứng ngay trước đỉnh v. Các đỉnh cần tìm chính là các đỉnh mà không có đỉnh nào đứng ngay trước nó, trị trong phần tử tương ứng của mảng bằng 0. Các đỉnh như vậy đã sẵn sàng được đặt vào mảng thứ tư topo. Như vậy chúng ta sẽ khởi động quá trình duyệt theo chiều rộng bằng cách đặt các đỉnh này vào một hàng các đỉnh sẽ được xử lý. Khi một đỉnh cần được xử lý, nó sẽ được lấy ra từ hàng và đặt vào vị trí kế tiếp trong mảng topo, lúc này, việc xem xét nó xem như kết thúc và được đánh dấu bằng cách cho các phần tử trong mảng **predecessor_count** tương ứng với các đỉnh là đỉnh sau của nó giảm đi 1. Khi một phần tử nào trong mảng này đạt được trị 0, thì cũng có nghĩa là đỉnh tương ứng với nó có các đỉnh trước đã được duyệt xong, và đỉnh này đã sẵn sàng được duyệt nên được đưa vào hàng đợi.

```
template <int graph_size>
void Digraph<graph_size>::breadth_sort(List<Vertex> &topological_order)
/*
post: Các đỉnh của một đồ thị có hướng không có chu trình được xếp theo thứ tự topo tương ứng
      cách duyệt đồ thị theo chiều rộng.
uses: Các phương thức của các lớp List, Queue.
*/
{  topological_order.clear();
   Vertex v, w;
   int predecessor_count[graph_size];
   for (v = 0; v < count; v++) predecessor_count[v] = 0;
   for (v = 0; v < count; v++)
      for (int i = 0; i < neighbors[v].size(); i++) { // Cập nhật số đỉnh đứng
                                                      trước cho mỗi đỉnh.
         neighbors[v].retrieve(i, w);
         predecessor_count[w]++;
      }
   Queue<Vertex> ready_to_process;
   for (v = 0; v < count; v++)
      if (predecessor_count[v] == 0)
         ready_to_process.append(v);

   while (!ready_to_process.empty()) {
      ready_to_process.retrieve(v);
      topological_order.insert(topological_order.size(), v);
      for (int j = 0; j < neighbors[v].size(); j++) { // Giảm số đỉnh đứng trước
         neighbors[v].retrieve(j, w);             // của mỗi đỉnh kề của v đi 1
         predecessor_count[w]--;
```

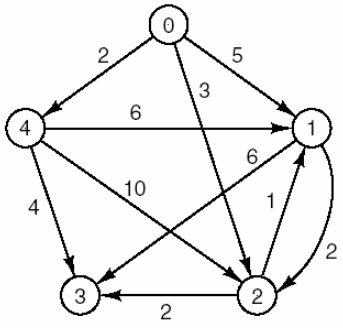
```
            if (predecessor_count[w] == 0)
               ready_to_process.append(w);
         }
         ready_to_process.serve();
      }
   }
```

Giải thuật này cần đến một trong các hiện thực của lớp `Queue`. `Queue` có thể có hiện thực theo bất kỳ cách nào đã được mô tả trong chương 3. Do các phần tử trong `Queue` là các đỉnh. Cũng như duyệt theo chiều sâu, thời gian cần cho hàm `breadth_first` là O(n+e), với n là số đỉnh và e là số cạnh của đồ thị.

13.5. Giải thuật Greedy: Tìm đường đi ngắn nhất

13.5.1. Đặt vấn đề

Như một ứng dụng khác của đồ thị, chúng ta xem xét một bài toán hơi phức tạp sau đây. Chúng ta có một đồ thị có hướng G, trong đó mỗi cạnh được gán một con số không âm gọi là tải trọng (*weight*). Bài toán của chúng ta là tìm một đường đi từ một đỉnh v đến một đỉnh w sao cho tổng tải trọng trên đường đi là nhỏ nhất. Chúng ta gọi đường đi như vậy là đường đi ngắn nhất (*shortest path*), mặc dù tải trọng có thể biểu diễn cho giá cả, thời gian, hoặc một vài đại lượng nào khác thay vì khoảng cách. Chúng ta có thể xem G như một bản đồ các tuyến bay, chẳng hạn, mỗi đỉnh của đồ thị biểu diễn một thành phố và tải trọng trên mỗi cạnh biểu diễn chi phí bay từ thành phố này sang thành phố kia. Bài toán của chúng ta là tìm lộ trình bay từ thành phố v đến thành phố w sao cho tổng chi phí là nhỏ nhất. Chúng ta hãy xem xét đồ thị ở hình 13.8. Đường ngắn nhất từ đỉnh 0 đến đỉnh 1 đi ngang qua đỉnh 2 có tổng tải trọng là 4, so với tải trọng là 5 đối với cạnh nối trực tiếp từ 0 sang 1, và tổng tải trọng là 8 nếu đi ngang qua đỉnh 4.



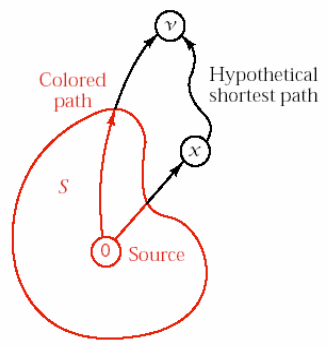
Hình 13.8 – Đồ thị có hướng với các tải trọng

Có thể dễ dàng giải bài toán một cách tổng quát như sau: bắt đầu từ một đỉnh, gọi là đỉnh nguồn, tìm đường đi ngắn nhất đến mọi đỉnh còn lại, thay vì chỉ tìm đường đến một đỉnh đích. Chúng ta cần tải trọng phải là những số không âm.

13.5.2. Phương pháp

Giải thuật sẽ được thực hiện bằng cách nắm giữ một tập S các đỉnh mà đường đi ngắn nhất từ đỉnh nguồn đến chúng đã được biết. Mới đầu, đỉnh nguồn là đỉnh duy nhất trong S. Tại mỗi bước, chúng ta thêm vào S các đỉnh còn lại mà đường đi ngắn nhất từ nguồn đến chúng vừa được tìm thấy. Bài toán bây giờ trở thành bài toán xác định đỉnh nào để thêm vào S tại mỗi bước. Chúng ta hãy xem những đỉnh đã có trong S như đã được tô một màu nào đó, và các cạnh nằm trong các đường đi ngắn nhất từ đỉnh nguồn đến các đỉnh có màu cũng được tô màu.

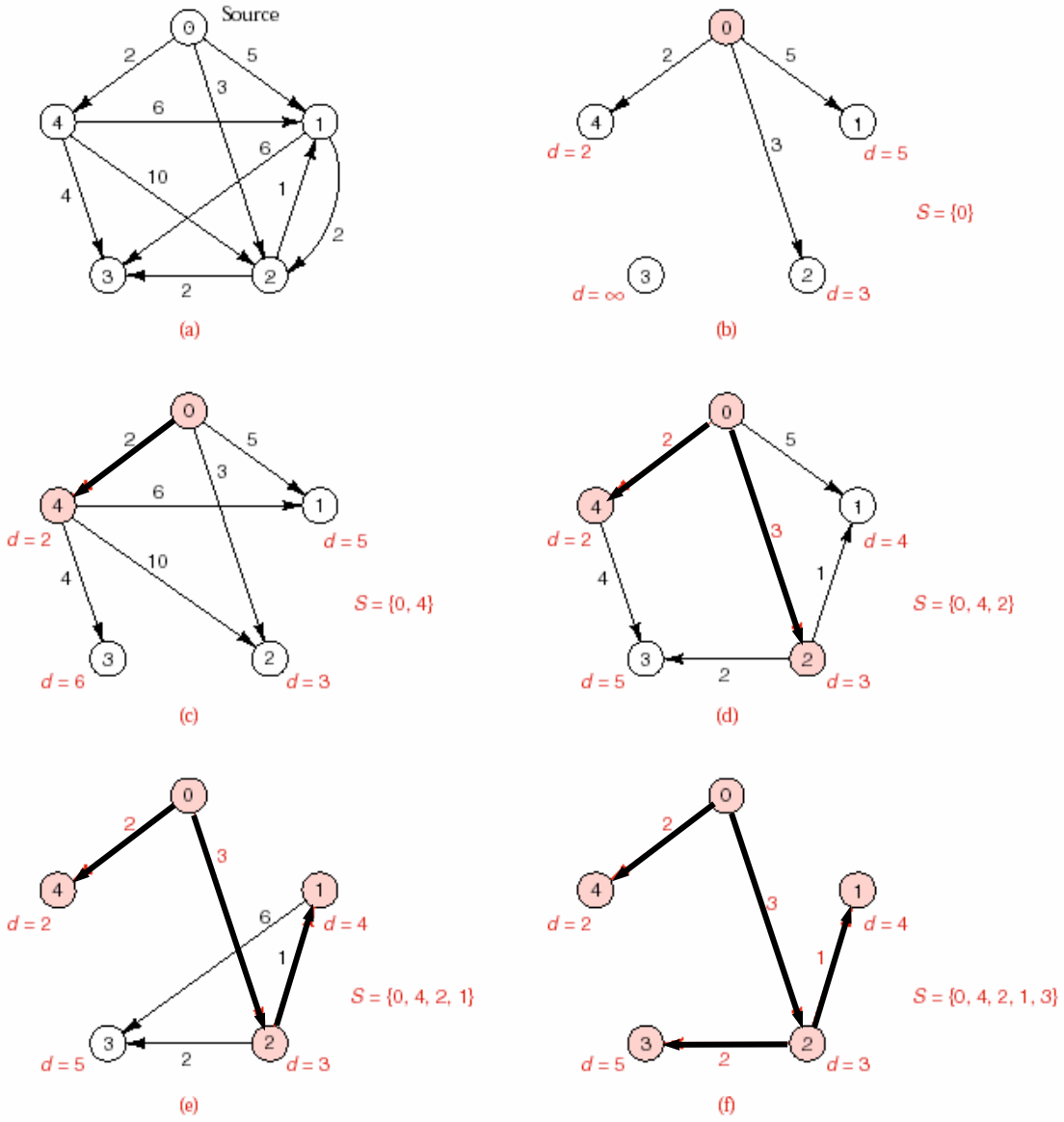
Chúng ta sẽ nắm giữ một mảng `distance` cho biết rằng, đối với mỗi đỉnh v, khoảng cách từ đỉnh nguồn dọc theo đường đi có các cạnh đã có màu, có thể trừ cạnh cuối cùng. Nghĩa là, nếu v thuộc *S*, thì `distance[v]` chứa khoảng cách ngắn nhất đến v, và mọi cạnh nằm trên đường đi tương ứng đều có màu. Nếu v không thuộc S, thì `distance[v]` chứa chiều dài của đường đi từ đỉnh nguồn đến một đỉnh w nào đó cộng với tải trọng của cạnh nối từ w đến v, và mọi cạnh nằm trên đường đi này, trừ cạnh cuối, đều có màu. Mảng `distance` được khởi tạo bằng cách gán từng `distance[v]` với trị của tải trọng của cạnh nối từ đỉnh nguồn đến v nếu tồn tại cạnh này, ngược lại nó được gán bằng vô cực.



Hình 13.9 – Tìm một đường đi ngắn

Để xác định đỉnh được thêm vào S tại mỗi bước, chúng ta áp dụng một "tiêu chí tham lam" (*greedy criterion*) trong việc chọn ra một đỉnh v có khoảng cách nhỏ nhất từ trong mảng `distance` sao cho v chưa có trong *S*. Chúng ta cần chứng minh rằng, đối với đỉnh v này, khoảng cách chứa trong mảng `distance` thực sự là chiều dài của đường đi ngắn nhất từ đỉnh nguồn đến v. Giả sử có một đường đi ngắn hơn từ nguồn đến v, như hình 13.9. Đường đi này đi ngang một đỉnh *x* nào đó chưa thuộc S rồi mới đến v (có thể lại qua một số đỉnh khác có nằm trong *S* trước khi gặp v). Nhưng nếu đường đi này ngắn hơn đường đi đã được tô màu đến v, thì đoạn đường ban đầu trong nó từ nguồn đến *x* còn ngắn hơn nữa, nghĩa là `distance[x] < distance[v]`, và như vậy tiêu chí tham lam đã phải chọn *x* thay vì v là đỉnh kế tiếp được thêm vào S.

Khi thêm v và S, chúng ta sẽ tô màu v và tô màu luôn đường đi ngắn nhất từ đỉnh nguồn đến v (mọi cạnh trong nó trừ cạnh cuối thực sự đã được tô màu trước



Hình 12.10 - Ví dụ về các đường đi ngắn nhất

đó). Kế tiếp, chúng ta cần cập nhật lại các phần tử của mảng `distance` bằng cách xem xét đối với mỗi đỉnh w nằm ngoài *S*, đường đi từ nguồn qua v rồi đến w có ngắn hơn khoảng cách từ nguồn đến w đã được ghi nhận trước đó hay không. Nếu điều này xảy ra có nghĩa là chúng ta vừa phát hiện được một đường đi mới cho đỉnh w có đi ngang qua v ngắn hơn cách đi đã xác định trước đó, và như vậy chúng ta cần cập nhật lại `distance[w]` bằng `distance[v]` cộng với tải trọng của cạnh nối từ v đến *w*.

13.5.3. Ví dụ

Trước khi viết một hàm cho phương pháp này, chúng ta hãy xem qua ví dụ ở hình 13.10. Đối với đồ thị có hướng ở hình (a), trạng thái ban đầu được chỉ ra ở hình (b): tập S (các đỉnh đã được tô màu) chỉ gồm có nguồn là đỉnh 0, các phần tử của mảng `distance` chứa các con số được đặt cạnh mỗi đỉnh còn lại. Khoảng cách đến đỉnh 4 ngắn nhất, nên 4 được thêm vào S như ở hình (c), và `distance[3]` được cập nhật thành 6. Do khoảng cách đến 1 và 2 ngang qua 4 lớn hơn khoảng cách đã chứa trong mảng `distance`, nên các khoảng cách này trong `distance` được giữ không đổi. Đỉnh kế tiếp gần nhất đối với nguồn là đỉnh 2, nó được thêm vào S như hình (d), khoảng cách đến các đỉnh 1 và 3 được cập nhật lại ngang qua đỉnh này. Hai bước cuối cùng, trong hình (e) và (f), đỉnh 1 và 3 được thêm vào và các đường đi ngắn nhất từ đỉnh nguồn đến các đỉnh còn lại được chỉ ra trong sơ đồ cuối.

13.5.4. Hiện thực

Để hiện thực giải thuật tìm đường ngắn nhất trên, chúng ta cần chọn cách hiện thực cho đồ thị có hướng. Việc dùng bảng kề cho phép truy xuất ngẫu nhiên đến mọi đỉnh của đồ thị. Hơn nữa, chúng ta có thể sử dụng bảng vừa để chứa các tải trọng vừa để chứa thông tin về các đỉnh kề. Trong đặc tả dưới đây của đồ thị có hướng, chúng ta thêm thông số `template` cho phép người sử dụng chọn lựa kiểu của tải trọng theo ý muốn. Lấy ví dụ, người sử dụng khi dùng lớp `Digraph` để mô hình hóa mạng các tuyến bay, họ có thể chứa giá vé là một số nguyên hay một số thực.

```
template <class Weight, int graph_size>
class Digraph {
public:
    //  Thêm constructor và các phương thức nhập và xuất đồ thị.
   void set_distances(Vertex source, Weight distance[]) const;
protected:
   int count;
   Weight adjacency[graph_size][graph_size];
};
```

Thuộc tính **`count`** chứa số đỉnh của một đối tượng đồ thị. Trong các ứng dụng, chúng ta cần bổ sung các phương thức để nhập hay xuất các thông tin của một đối tượng đồ thị, nhưng do chúng không cần thiết trong việc hiện thực phương thức **`distance`** để tìm các đường đi ngắn nhất, chúng ta xem chúng như là bài tập.

Chúng ta sẽ giả sử rằng lớp `Weight` đã có các tác vụ so sánh. Ngoài ra, người sử dụng sẽ phải khai báo trị lớn nhất có thể có của `Weight`, gọi là **`infinity`**. Chẳng hạn, chương trình của người sử dụng với tải trọng là số nguyên có thể sử dụng thư viện chuẩn ANSI C++`<limits>` với định nghĩa toàn cục như sau:

```
const Weight infinity = numeric_limits<int>::max();
```

Chúng ta sẽ đặt trị của infinity vào các phần tử của mảng distance tương ứng với các cạnh từ không tồn tại nguồn đến mỗi đỉnh. Phương thức **set_distance** sẽ tìm các đường đi ngắn nhất và các khoảng cách ngắn nhất này sẽ được trả về qua tham biến distance[].

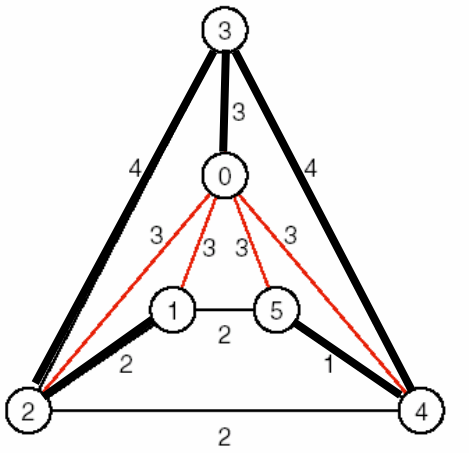
```
template <class Weight, int graph_size>
void Digraph<Weight, graph_size>::set_distances(Vertex source,
                                    Weight distance[]) const
/*
post: Mảng distance chứa đường đi có tải trọng ngắn nhất từ đỉnh nguồn đến mỗi đỉnh trong
      đồ thị.
*/
{  Vertex v, w;
   bool found[graph_size];   //  Biểu diễn các đỉnh trong S.
   for (v = 0; v < count; v++) {
      found[v] = false;
      distance[v] = adjacency[source][v];
   }
   found[source]=true;// Khởi tạo bằng cách bỏ đỉnh nguồn vào S.
   distance[source] = 0;
   for(int i = 0; i < count; i++){ // Mỗi lần lặp bỏ thêm một đỉnh vào S.
      Weight min = infinity;
      for (w = 0; w < count; w++) if (!found[w])
         if (distance[w] < min) {
            v = w;
            min = distance[w];
         }
      found[v] = true;
      for (w = 0; w < count; w++) if (!found[w])
         if (min + adjacency[v][w] < distance[w])
             distance[w] = min + adjacency[v][w];
   }
}
```

Để ước đoán thời gian cần để chạy hàm này, chúng ta thấy rằng vòng lặp chính thực hiện n-1 lần, trong đó n là số đỉnh, và bên trong vòng lặp chính có hai vòng lặp khác, mỗi vòng lặp này thực hiện n-1 lần. Vậy các vòng lặp thực hiện $(n-1)^2$ lần. Các lệnh bên ngoài vòng lặp chỉ hết O(n), nên thời gian chạy của hàm là $O(n^2)$.

13.6. Cây phủ tối tiểu

13.6.1. Đặt vấn đề

Giải thuật tìm đường đi ngắn nhất của phần trên có thể được áp dụng với mạng hay đồ thị có hướng cũng như mạng hay đồ thị không có hướng. Ví dụ, hình 13.11 minh họa ứng dụng tìm các đường đi ngắn nhất từ đỉnh nguồn 0 đến các đỉnh khác trong mạng.

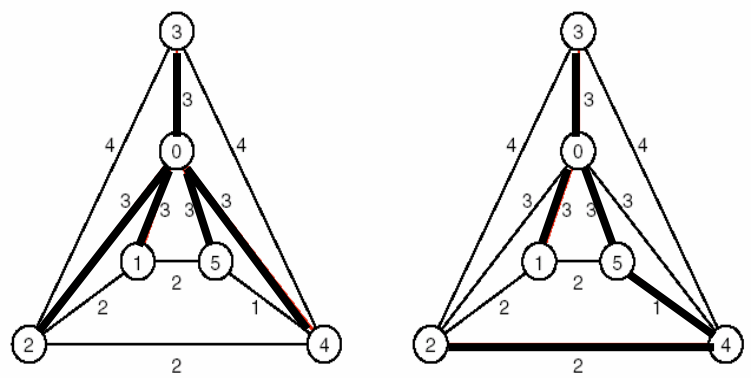


Hình 13.11 – Tìm đường đi ngắn nhất trong một mạng

Nếu mạng dựa trên cơ sở là một đồ thị liên thông G, thì các đường đi ngắn nhất từ một đỉnh nguồn nào đó sẽ nối nguồn này với tất cả các đỉnh khác trong G. Từ đó, như trong hình 13.11, nếu chúng ta kết hợp các đường đi ngắn nhất tính được lại với nhau, chúng ta có một cây nối tất cả các đỉnh của G. Nói cách khác, đó là cây được tạo bởi tất cả các đỉnh và một số cạnh của đồ thị G. Chúng ta gọi những cây như vậy là cây phủ (*spanning tree*) của G. Cũng như phần trước, chúng ta có thể xem một mạng trên một đồ thị G như là một bản đồ các tuyến bay, với mỗi đỉnh biểu diễn một thành phố và tải trọng trên một cạnh là giá vé bay từ thành phố này sang thành phố kia. Một cây phủ của G biểu diễn một tập các đường bay cho phép các hành khách hoàn tất một chuyến du lịch qua khắp các thành phố. Dĩ nhiên rằng, hành khách cần phải thực hiện một số tuyến bay nào đó một vài lần mới hoàn tất được chuyến du lịch. Điều bất tiện này được bù đắp bởi chi phí thấp. Nếu chúng ta hình dung mạng trong hình 13.11 như một hệ thống điều khiển tập trung, thì đỉnh nguồn tương ứng với sân bay trung tâm, và các đường đi từ đỉnh này là những hành trình bay. Một điều quan trọng đối với một sân bay theo hệ thống điều khiển tập trung là giảm tối đa chi phí bằng cách chọn lựa một hệ thống các đường bay mà tổng chi phí nhỏ nhất.

Định nghĩa: Một cây phủ tối tiểu (*minimal spanning tree*) của một mạng liên thông là cây phủ mà tổng các tải trọng trên các cạnh của nó là nhỏ nhất.

Mặc dù việc so sánh hai cây phủ trong hình 13.12 là không khó khăn, nhưng cũng khó mà biết được còn có cây phủ nào có trị nhỏ hơn nữa hay không. Bài toán của chúng ta là xây dựng một phương pháp xác định một cây phủ tối tiểu của một mạng liên thông.



Hình 13.12 – Hai cây phủ trong một mạng

13.6.2. Phương pháp

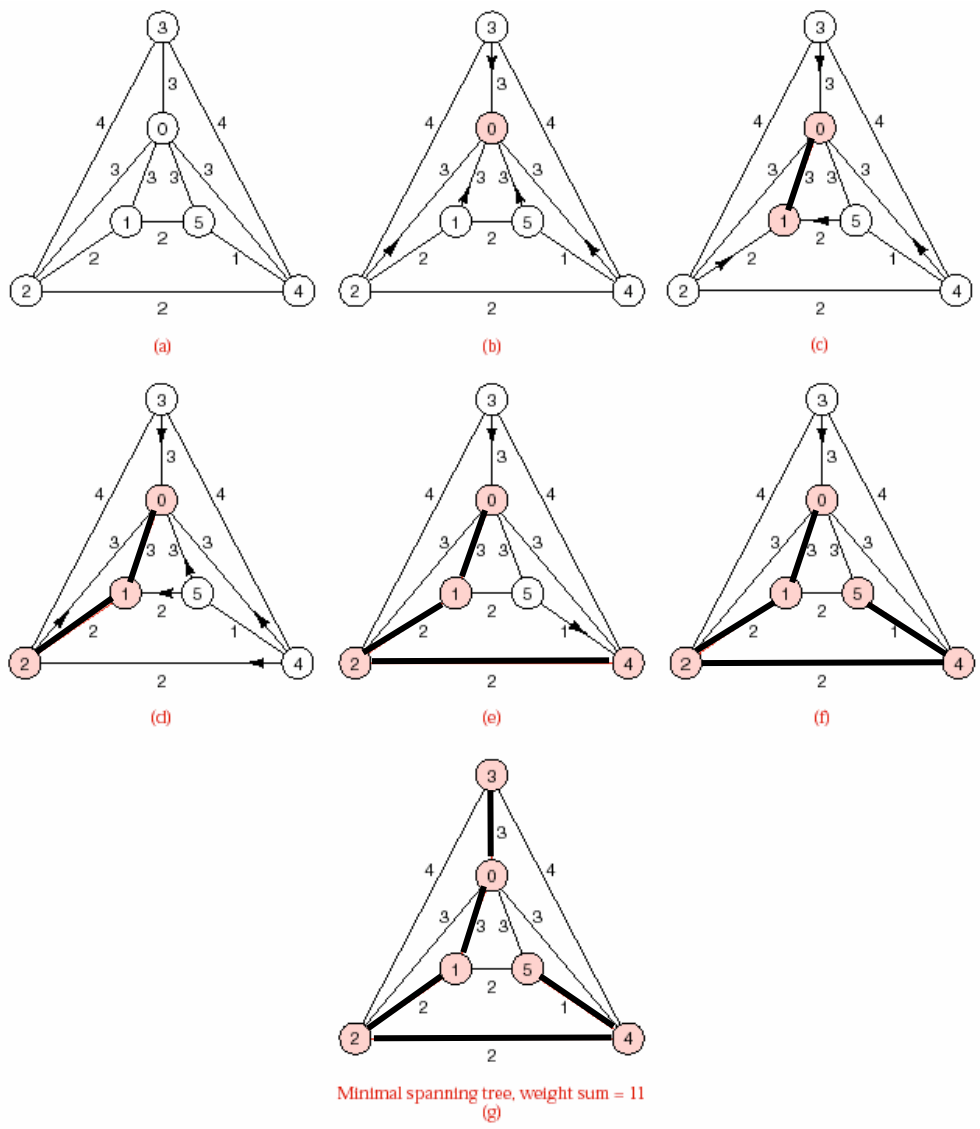
Chúng ta đã biết giải thuật tìm cây phủ trong một đồ thị liên thông, do giải thuật tìm đường ngắn nhất đã có. Chúng ta có thể thay đổi chút ít trong giải thuật tìm đường ngắn nhất để có được phương pháp tìm cây phủ tối tiểu mà R. C. Prim đã đưa ra lần đầu vào năm 1957.

Trước hết chúng ta chọn ra một đỉnh bắt đầu, gọi là nguồn, và trong khi tiến hành phương pháp, chúng ta nắm giữ một tập X các đỉnh mà đường đi từ chúng đến đỉnh nguồn thuộc về cây phủ tối tiểu mà chúng ta đã tìm thấy. Chúng ta cũng cần nắm một tập Y gồm các cạnh nối các đỉnh trong X mà thuộc cây đang được xây dựng. Như vậy, chúng ta có thể hình dung rằng các đỉnh trong *X* và các cạnh trong Y đã tạo ra một phần của cây mà chúng ta cần tìm, cây này sẽ lớn lên thành cây phủ tối tiểu cuối cùng. Lúc khởi đầu, đỉnh nguồn là đỉnh duy nhất trong X, và Y là tập rỗng. Tại mỗi bước, chúng ta thêm một đỉnh vào X: đỉnh này được chọn sao cho nó có một cạnh nối với một đỉnh nào đó đã có trong X có tải trọng nhỏ nhất, so với các tải trọng của tất cả các cạnh khác nối các đỉnh còn nằm ngoài X với các đỉnh đã có trong X. Cạnh tối tiểu này sẽ được đưa vào Y.

Việc chứng minh giải thuật Prim đem lại cây phủ tối tiểu không thực dễ dàng, chúng ta tạm hoãn việc này lại sau. Tuy nhiên, chúng ta có thể hiểu về việc chọn lựa một đỉnh mới để thêm vào X và một cạnh mới để thêm vào Y . Tiêu chí Prim cho chúng ta một cách dễ dàng nhất để thực hiện việc kết nối này, và như vậy, theo tiêu chí tham lam, chúng ta nên sử dụng nó.

Khi hiện thực giải thuật Prim, chúng ta cần nắm giữ một danh sách các đỉnh thuộc X như là các phần tử của một mảng kiểu `bool`. Cách nắm giữ các cạnh trong Y sẽ dễ dàng nếu như chúng ta bắt chước cách lưu trữ các cạnh trong một đồ thị.

Chúng ta sẽ cần đến một mảng phụ **`neighbor`** để biết thêm rằng, đối với mỗi đỉnh v, đỉnh nào trong X có cạnh đến v có tải trọng nhỏ nhất. Các tải trọng nhỏ nhất này cũng chứa trong mảng **`distance`** tương ứng. Nếu một đỉnh v không có một cạnh nào nối được với một đỉnh nào đó trong X, trị của phần tử tương ứng của nó trong `distance` sẽ là **`infinity`**.



Hình 13.13 – Ví dụ về giải thuật Prim

Mảng `neighbor` được khởi tạo bằng cách gán `neighbor[v]` đến đỉnh nguồn cho tất cả các đỉnh v, và `distance` được khởi tạo bằng cách gán `distance[v]` bởi tải trọng của cạnh nối từ đỉnh nguồn đến v hoặc `infinity` nếu cạnh này không tồn tại.

Để xác định đỉnh nào sẽ được thêm vào tập X tại mỗi bước, chúng ta chọn đỉnh v trong số các đỉnh chưa có trong X mà trị tương ứng trong mảng `distance` là nhỏ nhất. Sau đó chúng ta cần cập nhật lại các mảng để phản ánh đúng sự thay đổi mà chúng ta đã làm đối với tập X như sau: với mỗi w chưa có trong X, nếu có

một cạnh nối v và w, chúng ta xem thử tải trọng của cạnh này có nhỏ hơn `distance[w]` hay không, nếu quả thực như vậy thì `distance[w]` cần được cập nhật lại bằng trị của tải trọng này, và `neighbor[w]` sẽ là v.

Lấy ví dụ, chúng ta hãy xem xét mạng trong hình (a) của hình 13.13. Trạng thái ban đầu trong hình (b): Tập X (các đỉnh được tô màu) chỉ gồm đỉnh nguồn 0, và đối với mỗi đỉnh w, đỉnh được lưu trong `neighbor[w]` được chỉ bởi các mũi tên từ w. Trị của `distance[w]` là tải trọng của cạnh tương ứng. Khoảng cách từ nguồn đến đỉnh 1 là một trong những trị nhỏ nhất, nên 1 được thêm vào X như hình (c), và trong các phần tử của các mảng `neighbor` và `distance` chỉ có các phần tử tương ứng với đỉnh 2 và đỉnh 5 là được cập nhật lại. Đỉnh kế tiếp gần với các đỉnh trong X nhất là đỉnh 2, nó được thêm vào X như hình (d), trong này cũng đã chỉ ra các trị của các mảng đã được cập nhật lại. Các bước cuối cùng được minh họa trong hình (e), (f) và (g).

13.6.3. Hiện thực

Để hiện thực giải thuật Prim, chúng ta cần chọn một lớp để biểu diễn cho mạng. Sự tương tự của giải thuật này so với giải thuật tìm đường ngắn nhất trong phần trước giúp chúng ta quyết định thiết kế lớp `Network` dẫn xuất từ lớp `Digraph`.

```
template <class Weight, int graph_size>
class Network: public Digraph<Weight, graph_size> {
public:
   Network();
   void read();   //  Định nghĩa lại để nhập thông tin về mạng.
   void make_empty(int size = 0);
   void add_edge(Vertex v, Vertex w, Weight x);
   void minimal_spanning(Vertex source,
                         Network<Weight, graph_size> &tree) const;
};
```

Chúng ta sẽ viết lại phương thức nhập **`read`** để đảm bảo rằng tải trọng của cạnh (v, w) luôn trùng với tải trọng của cạnh (w, v), với mọi v và w, vì đây là một mạng vô hướng. Phương thức mới **`make_empty(int size)`** tạo một mạng có **`size`** đỉnh và không có cạnh. Phương thức khác, **`add_edge`**, thêm một cạnh có một tải trọng cho trước vào mạng. Chúng ta cũng đã giả sử rằng lớp `Weight` đã có đầy đủ các toán tử so sánh. Ngoài ra, người sử dụng cần khai báo trị lớn nhất có thể của `Weight` gọi là **`infinity`**. Phương thức **`minimal_spanning`** mà chúng ta sẽ viết ở đây sẽ tìm cây phủ tối tiểu và trả về qua tham biến **`tree`**. Tuy phương thức chỉ có thể tìm cây phủ khi thực hiện trên một mạng liên thông, nó cũng có thể tìm một cây phủ cho một thành phần liên thông có chứa đỉnh **`source`** trong mạng.

```
template <class Weight, int graph_size>
void Network<Weight, graph_size>::minimal_spanning(Vertex source,
                     Network<Weight, graph_size> &tree) const
/*
post: Xác định cây phủ tối tiểu trong thành phần liên thông có chứa đỉnh source của mạng.
*/
{
   tree.make_empty(count);
   bool component[graph_size]; // Các đỉnh trong tập X.
   Vertex neighbor[graph_size];// Phần tử thứ i chứa đỉnh trước của nó sao cho khoảng
                               // cách giữa chúng nhỏ nhất so với các khoảng cách từ
                               // các đỉnh trước khác đã có trong tập X đến nó.
   Weight distance[graph_size];// Các khoảng cách nhỏ nhất tương ứng với từng phần tử
                               // trong mảng neighbor trên.
   Vertex w;

   for (w = 0; w < count; w++) {
      component[w] = false;
      distance[w] = adjacency[source][w];
      neighbor[w] = source;
   }
   component[source] = true;    //  Tập X chỉ có duy nhất đỉnh nguồn.
   for (int i = 1; i < count; i++) {
      Vertex v;                 //  Mỗi lần lặp bổ sung thêm một đỉnh vào tập X.
      Weight min = infinity;
      for (w = 0; w < count; w++) if (!component[w])
         if (distance[w] < min) {
            v = w;
            min = distance[w];
         }

      if (min < infinity) {
         component[v] = true;
         tree.add_edge(v, neighbor[v], distance[v]);
         for (w = 0; w < count; w++) if (!component[w])
            if (adjacency[v][w] < distance[w]) {
               distance[w] = adjacency[v][w];
               neighbor[w] = v;
            }
      }
      else break;      //  Xong một thành phần liên thông trong đồ thị không liên thông.
   }
}
```

Vòng lặp chính trong hàm trên thực hiện n-1 lần, với n là số đỉnh, và trong vòng lặp chính còn có hai vòng lặp khác, mỗi vòng lặp này thực hiện n-1 lần. Vậy các vòng lặp thực hiện $(n-1)^2$ lần. Các lệnh bên ngoài vòng lặp chỉ hết O(n), nên thời gian chạy của hàm là $O(n^2)$.

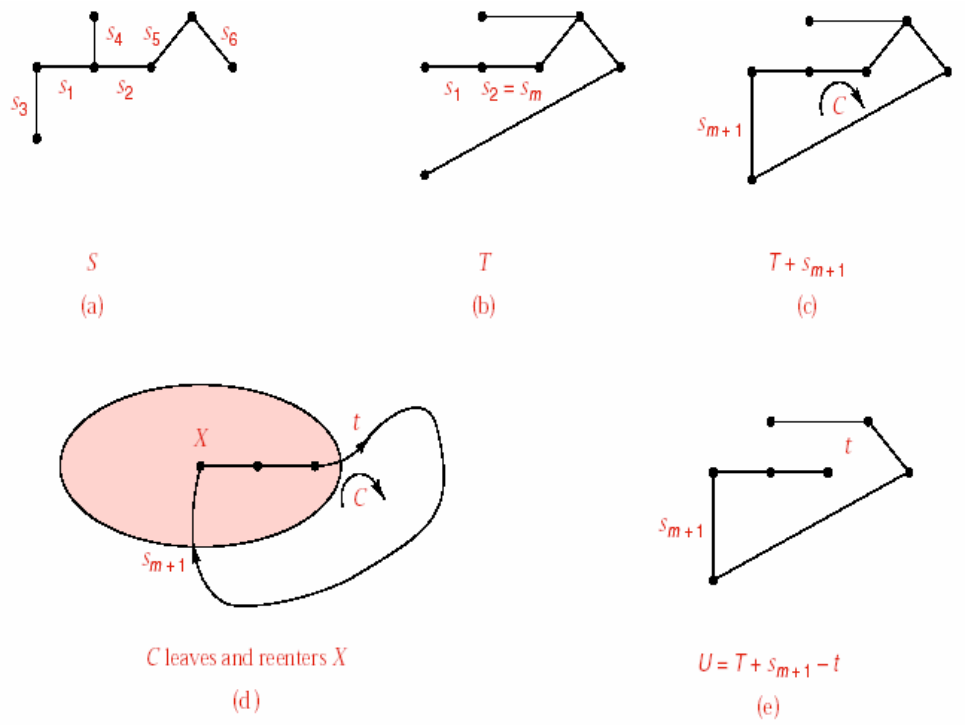
13.6.4. Kiểm tra giải thuật Prim

Chúng ta cần chứng minh rằng, đối với đồ thị liên thông G, cây phủ S sinh ra bởi giải thuật Prim phải có tổng tải trọng trên các cạnh nhỏ hơn so với bất kỳ

một cây phủ nào khác của G. Giải thuật Prim xác định một chuỗi các cạnh s_1, s_2, ..., s_n tạo ra cây phủ S. Như hình 13.14, s_1 là cạnh thứ nhất được thêm vào tập Y trong giải thuật Prim, s_2 là cạnh thứ hai được thêm vào, và cứ thế.

Để chứng minh S là một cây phủ tối tiểu, chúng ta chứng minh rằng nếu m là một số nguyên, $0 \le m \le n$, thì sẽ có một cây phủ tối tiểu chứa cạnh s_i với $i \le m$. Chúng ta sẽ chứng minh quy nạp trên m. Trường hợp cơ bản, khi m = 0, rõ ràng là đúng, vì bất kỳ cây phủ tối tiểu nào cũng đều phải chứa một tập rỗng các cạnh. Ngoài ra, khi chúng ta hoàn tất việc quy nạp, trường hợp cuối cùng với m = n chỉ ra rằng có một cây phủ tối tiểu chứa tất cả các cạnh của S, và chính là S. (Lưu ý rằng nếu thêm bất kỳ một cạnh nào vào một cây phủ thì cũng tạo ra một chu trình, nên bất kỳ cây phủ nào chứa mọi cạnh của S đều phải chính là S). Nói cách khác, khi chúng ta hoàn tất việc quy nạp, chúng ta đã chứng minh được rằng S chính là cây phủ tối tiểu.

Như vậy chúng ta cần trình bày các bước quy nạp bằng cách chứng minh rằng nếu $m < n$ và T là một cây phủ tối tiểu chứa các cạnh s_i với $i \le m$, thì phải có một cây phủ tối tiểu U cũng chứa các cạnh trên và thêm một cạnh s_{m+1}. Nếu s_{m+1} đã có trong T, chúng ta có thể đơn giản cho U = T, như vậy chúng ta có thể giả sử rằng s_{m+1} không là một cạnh trong T. Chúng ta hãy xem hình (b) của hình 13.14.



Hình 13.14 – Kiểm tra giải thuật Prim

Chúng ta hãy gọi X là tập các đỉnh trong S thuộc các cạnh s_1, s_2, ..., s_m và R là tập các đỉnh còn lại trong S. Trong giải thuật Prim, cạnh s_{m+1} nối một đỉnh trong

X với một đỉnh trong R, và s_{m+1} là một trong các cạnh có tải trọng nhỏ nhất nối giữa hai tập này. Chúng ta hãy xét ảnh hưởng của việc thêm cạnh s_{m+1} này vào T, như minh họa trong hình (c). Việc thêm vào này phải tạo ra một chu trình C, do mạng liên thông T rõ ràng là có chứa một đường đi có nhiều cạnh nối hai đầu của cạnh s_{m+1}. Chu trình C phải có chứa một cạnh $t \neq s_{m+1}$ nối tập X với tập R, do nếu chúng ta di dọc theo đường đi khép kín C chúng ta phải đi vào tập X một số lần bằng với số lần chúng ta đi ra khỏi nó. Chúng ta hãy xem hình (d). Giải thuật Prim bảo đảm rằng tải trọng của s_{m+1} nhỏ hơn hoặc bằng tải trọng của t. Do đó, cây phủ mới U trong hình (e), có được từ T bằng cách loại đi t và thêm vào s_{m+1}, sẽ có tổng tải trọng không lớn hơn tải trọng của T. Như vậy chúng ta đã có được U là một cây phủ tối tiểu của G, mà U chứa các cạnh s_1, s_2, ..., s_m, s_{m+1}. Điều này hoàn tất được quá trình quy nạp của chúng ta.

13.7. Sử dụng đồ thị như là cấu trúc dữ liệu

Trong chương này, chúng ta đã tìm hiểu chỉ một ít ứng dụng của đồ thị, nhưng chúng ta đã bắt đầu thâm nhập vào những vấn đề sâu sắc của các giải thuật về đồ thị. Nhiều giải thuật trong số đó, đồ thị xuất hiện như các cấu trúc toán học và đã nắm bắt được các đặc trưng thiết yếu của bài toán, thay vì chỉ là những công cụ tính toán cho ra được những lời giải của chúng. Lưu ý rằng trong chương này chúng ta đã nói về các đồ thị như là các cấu trúc toán học, chứ không như các cấu trúc dữ liệu, do chúng ta đã sử dụng chúng để đặc tả các vấn đề trong toán học, và để viết các giải thuật, chúng ta đã hiện thực các đồ thị trong các cấu trúc dữ liệu như danh sách hoặc bảng. Tuy vậy, rõ ràng là đồ thị tự bản thân nó có thể được xem như các cấu trúc dữ liệu - các cấu trúc dữ liệu mà có chứa các mối quan hệ giữa các dữ liệu phức tạp hơn những gì đã được mô tả trong một danh sách hoặc một cây. Do tính tổng quát và mềm dẻo, đồ thị là cấu trúc dữ liệu rất hiệu quả và đã tỏ rõ những giá trị của nó trong những ứng dụng cấp tiến như thiết kế hệ quản trị cơ sở dữ liệu chẳng hạn. Tất nhiên, một công cụ mạnh như vậy càng nên được sử dụng mọi khi cần thiết, nhưng việc sử dụng nó cần phải được kết hợp với việc xem xét một cách cẩn thận để sức mạnh của nó không làm cho cho chúng ta bị rối. Cách an toàn nhất trong việc sử dụng một công cụ mạnh là dựa trên sự chính quy; nghĩa là, chúng ta chỉ sử dụng công cụ mạnh trong những phương pháp đã được định nghĩa một cách cẩn thận và dễ hiểu. Do tính tổng quát của đồ thị, việc tuân thủ nguyên tắc vừa nêu ra trong việc sử dụng nó không phải luôn dễ dàng.