

ĐỒ THỊ

1. Đường đi ngắn nhất

1.1. Đồ thị có trọng số

Trong các ứng dụng thực tế, chẳng hạn trong mạng lưới giao thông, người ta không chỉ quan tâm đến việc tìm đường đi giữa hai địa điểm mà còn phải lựa chọn một hành trình tiết kiệm nhất (theo tiêu chuẩn không gian, thời gian hay một đại lượng mà chúng ta cần giảm thiểu theo hành trình). Khi đó người ta gán cho mỗi cạnh của đồ thị một giá trị phản ánh chi phí đi qua cạnh đó và cố gắng tìm một con đường mà tổng chi phí các cạnh đi qua là nhỏ nhất.

Đồ thị có trọng số là một bộ ba $G = (V, E, w)$ trong đó $G = (V, E)$ là một đồ thị, w là hàm trọng số:

$$\begin{aligned} w: E &\rightarrow \mathbb{R} \\ e &\mapsto w(e) \end{aligned}$$

Hàm trọng số gán cho mỗi cạnh e của đồ thị một số thực $w(e)$ gọi là trọng số (weight) của cạnh. Nếu cạnh $e = (u, v)$ thì ta cũng ký hiệu $w(u, v) = w(e)$.

Tương tự như đồ thị không trọng số, có nhiều cách biểu diễn đồ thị có trọng số trong máy tính. Nếu ta sử dụng danh sách cạnh, danh sách kè hay danh sách liên thuộc, mỗi phần tử của danh sách sẽ chứa thêm một thông tin về trọng số của cạnh tương ứng. Trường hợp biểu diễn đơn đồ thị gồm n đỉnh, ta còn có thể sử dụng ma trận trọng số $W = \{w_{uv}\}_{n \times n}$ trong đó w_{uv} là trọng số của cạnh (u, v) . Trong trường hợp $(u, v) \notin E$ thì tùy bài toán cụ thể, w_{uv} sẽ được gán một giá trị đặc biệt để nhận biết (u, v) không phải là cạnh (chẳng hạn có thể gán bằng $+\infty$, 0 hay $-\infty$).

Đường đi, chu trình trong đồ thị có trọng số cũng được định nghĩa giống như trong trường hợp không trọng số, chỉ có khác là độ dài đường đi không tính

bằng số cạnh đi qua, mà được tính bằng tổng trọng số của các cạnh đi qua. Độ dài của một đường đi P được ký hiệu là $w(P)$.

1.2. Đường đi ngắn nhất xuất phát từ một đỉnh

Bài toán tìm đường đi ngắn nhất xuất phát từ một đỉnh (*single-source shortest path*) được phát biểu như sau: Cho đồ thị có trọng số $G = (V, E, w)$, hãy tìm các đường đi ngắn nhất từ đỉnh xuất phát $s \in V$ đến tất cả các đỉnh còn lại của đồ thị. Độ dài của đường đi từ đỉnh s tới đỉnh t , ký hiệu $\delta(s, t)$, gọi là *khoảng cách* (distance) từ s đến t . Nếu như không tồn tại đường đi từ s tới t thì ta sẽ đặt khoảng cách đó bằng $+\infty$. Có một vài biến đổi khác của bài toán tìm đường đi ngắn nhất xuất phát từ một đỉnh:

- Tìm các con đường ngắn nhất từ mọi đỉnh tới một đỉnh t cho trước. Bằng cách đảo chiều các cung của đồ thị, chúng ta có thể quy về bài toán tìm đường đi ngắn nhất xuất phát từ t .
- Tìm đường đi ngắn nhất từ đỉnh s tới đỉnh t cho trước. Dĩ nhiên nếu ta tìm được đường đi ngắn nhất từ s tới mọi đỉnh khác thì bài toán tìm đường đi ngắn nhất từ s tới t cũng sẽ được giải quyết. Hơn nữa, vẫn chưa có một thuật toán nào tìm đường đi ngắn nhất từ s tới t mà không cần quy về bài toán tìm đường đi ngắn nhất từ s tới mọi đỉnh khác.
- Tìm đường đi ngắn nhất giữa mọi cặp đỉnh của đồ thị: Mặc dù có những thuật toán đơn giản và hiệu quả để tìm đường đi ngắn nhất giữa mọi cặp đỉnh, chúng ta vẫn có thể giải quyết bằng cách thực hiện thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với mọi cách chọn đỉnh xuất phát.

a) Cấu trúc bài toán con tối ưu

Các thuật toán tìm đường đi ngắn nhất mà chúng ta sẽ khảo sát đều dựa vào một đặc tính chung: Mỗi đoạn đường trên đường đi ngắn nhất phải là một đường đi ngắn nhất.

Định lý 1-1

Cho đồ thị có trọng số $G = (V, E, w)$, gọi $P = \langle v_1, v_2, \dots, v_k \rangle$ là một đường đi ngắn nhất từ v_1 tới v_k , khi đó với mọi $i, j: 1 \leq i \leq j \leq k$, đoạn đường $P_{ij} = \langle v_i, v_{i+1}, \dots, v_j \rangle$ là một đường đi ngắn nhất từ v_i tới v_j .

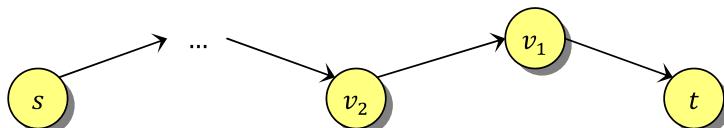
Chúng ta sẽ thấy rằng hầu hết các thuật toán tìm đường đi ngắn nhất đều là thuật toán quy hoạch động (ví dụ thuật toán Floyd) hoặc tham lam (ví dụ thuật toán Dijkstra) bởi tính chất bài toán con tối ưu nêu ra trong Định lý 1-1.

Nếu như đồ thị có chu trình âm (chu trình với độ dài âm) thì khoảng cách giữa một số cặp đỉnh nào đó có thể không xác định, bởi vì bằng cách đi vòng theo chu trình này một số lần đủ lớn, ta có thể chỉ ra đường đi giữa hai đỉnh nào đó trong chu trình này nhỏ hơn bất kỳ một số cho trước nào. Trong trường hợp như vậy, có thể đặt vấn đề tìm đường đi đơn ngắn nhất. Vấn đề đó lại là một bài toán NP-đầy đủ, hiện chưa ai chứng minh được sự tồn tại hay không một thuật toán đa thức tìm đường đi đơn ngắn nhất trên đồ thị có chu trình âm.

b) Quy về bài toán đo khoảng cách

Nếu như đồ thị không có chu trình âm thì có thể chứng minh được rằng một trong những đường đi ngắn nhất là đường đi đơn. Khi đó chỉ cần biết được khoảng cách từ s tới tất cả những đỉnh khác thì đường đi ngắn nhất từ s tới t có thể tìm được một cách dễ dàng qua thuật toán sau:

Trước tiên ta tìm đỉnh $v_1 \neq t$ để $\delta(s, t) = \delta(s, v_1) + c(v_1, t)$. Để thấy rằng luôn tồn tại đỉnh v_1 như vậy và đỉnh đó sẽ là đỉnh đứng liền trước t trên đường đi ngắn nhất từ s tới t . Nếu $v_1 = s$ thì đường đi ngắn nhất là đường đi trực tiếp theo cung (s, t) . Nếu không thì vấn đề trở thành tìm đường đi ngắn nhất từ s tới v_1 . Và ta lại tìm được một đỉnh $v_2 \notin \{t, v_1\}$ để $\delta(s, v_1) = \delta(s, v_2) + c(v_2, v_1)$...Cứ tiếp tục như vậy sau một số hữu hạn bước cho tới khi xét tới đỉnh $v_k = s$, Ta có dãy $t = v_0, v_1, v_2, \dots, v_k = s$ không chứa đỉnh lặp lại. Lật ngược thứ tự dãy cho ta đường đi ngắn nhất từ s tới t .



c) Nhãn khoảng cách và phép co

Tất cả những thuật toán chúng ta sẽ khảo sát để tìm đường đi ngắn nhất xuất phát từ một đỉnh đều sử dụng kỹ thuật gán nhãn khoảng cách: Với mỗi đỉnh $v \in V$, nhãn khoảng cách $d[v]$ là độ dài một đường đi nào đó từ s tới v . Trong

trường hợp chúng ta chưa xác định được đường đi nào từ s tới v , nhãn $d[v]$ được gán giá trị $+\infty$.

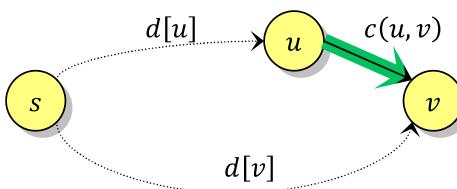
Ban đầu chúng ta chưa xác định được bất kỳ đường đi nào từ s tới các đỉnh khác nên các $d[v]$ được gán giá trị khởi tạo là:

$$d[v] = \begin{cases} 0, & \text{nếu } v = s \\ +\infty, & \text{nếu } v \neq s \end{cases} \quad (v = 1, 2, \dots, n)$$

```
procedure Init;
begin
  for  $\forall v \in V$  do  $d[v] := +\infty$ ;
   $d[s] := 0$ ;
end;
```

Do tính chất của nhãn khoảng cách, ta có $d[v] \geq \delta(s, v), \forall v \in V$. Các thuật toán tìm đường đi ngắn nhất sẽ cực tiểu hóa dần các nhãn $d[.]$ cho tới khi $d[v] = \delta(s, v), \forall v \in V$. Trong các thuật toán mà chúng ta sẽ khảo sát, việc cực tiểu hóa các nhãn khoảng cách được thực hiện bởi các *phép co*.

Phép co theo cạnh $(u, v) \in E$, gọi tắt là phép co (u, v) được thực hiện như sau: Giả sử chúng ta đã xác định được $d[u]$ là độ dài một đường đi từ s tới u , ta nối thêm cạnh (u, v) để được một đường đi từ s tới v với độ dài $d[u] + w(u, v)$. Nếu đường đi này có độ dài ngắn hơn $d[v]$, ta ghi nhận lại $d[v]$ bằng $d[u] + w(u, v)$. Điều này có nghĩa là nếu $s \sim u$ nối thêm cạnh (u, v) lại ngắn hơn đường đi $s \sim v$ đang có, thì ta hủy bỏ đường đi $s \sim v$ hiện tại và ghi nhận lại đường đi $s \sim v$ mới là đường đi $s \sim u \rightarrow v$.



Hình 2.1. Phép co

Có thể hình dung hoạt động của phép co như sau: Căng một đoạn dây đàn hồi đọc theo đường đi s tới v , đoạn dây sẽ dãn ra tới độ dài $d[v]$. Tiếp theo ta thử lấy đoạn dây đó căng đọc theo đường đi từ s tới u rồi nối tiếp đến v . Nếu đoạn dây bị chùng xuống (co lại) hơn so với cách căng cũ, ta ghi nhận đường đi tương

ứng với cách căng mới, nếu đoạn dây không chùng xuống (hoặc căng thêm) thì ta vẫn giữ đoạn dây đó căng theo đường cũ. Chính vì phép co không làm “dài” thêm $d[v]$, ta nói rằng $d[v]$ bị cực tiểu hóa qua phép co (u, v) .

Phép co (u, v) được thực hiện bởi hàm *Relax*, hàm nhận vào cạnh (u, v) và trả về True nếu nhãn $d[v]$ bị giảm đi qua phép co (u, v) :

```
function Relax( $e = (u, v) \in E$ ) : Boolean;
begin
    Result :=  $d[v] > d[u] + w(e)$ ;
    if Result then
        begin
             $d[v] := d[u] + w(e)$ ; //cực tiểu hóa nhãn  $d[v]$ 
            trace[v] := u; //Lưu vết đường đi
        end
    end;
```

Mỗi khi $d[v]$ bị giảm xuống sau phép co (u, v) , ta lưu lại vết $trace[v] := u$ với ý nghĩa đường đi ngắn nhất từ s tới v cho tới thời điểm được ghi nhận sẽ là đường đi qua u trước rồi đi tiếp theo cung (u, v) , vết này được sử dụng để truy vết tìm đường đi khi thuật toán kết thúc.

d) Một số tính chất và quy ước

Các tính chất sau đây tuy đơn giản nhưng quan trọng để chứng minh tính đúng đắn của các thuật toán trong bài:

- Bất đẳng thức tam giác (triangle inequality): Với một cạnh $(u, v) \in E$, ta có $\delta(s, v) \leq \delta(s, u) + w(u, v)$.
- Cận dưới (lower bound) và sự hội tụ (convergence): Các $d[v]$ sau một loạt phép co sẽ giảm dần nhưng không bao giờ nhỏ hơn khoảng cách $\delta(s, v)$. Tức là khi $d[v] = \delta(s, v)$ ($d[v]$ đạt cận dưới) thì không một phép co nào làm giảm $d[v]$ đi được nữa.
- Cây đường đi ngắn nhất (shortest-path tree): Nếu ta khởi tạo các $d[v]$ và thực hiện các phép co cho tới khi $d[v]$ bằng khoảng cách từ s tới v ($\forall v \in V$) thì chúng ta cũng xây dựng được một cây gốc s trong đó nút v là con của nút $Trace[v]$. Đường đi trên cây từ nút gốc s tới một nút v chính là đường đi ngắn nhất.

- Sự không tồn tại đường đi (no path): Nếu không tồn tại đường đi từ s tới t thì cho dù chúng ta co như thế nào chăng nữa, $d[t]$ luôn bằng $+\infty$.

Để tiện trong trình bày thuật toán, ta đưa vào một quy ước khi cộng giá trị ∞ với hằng số C . Bởi trong máy tính không có khái niệm ∞ , các chương trình cài đặt thường sử dụng một hằng số đặc biệt để thay thế, nhưng có thể phải đi kèm với vài sửa đổi để hợp lý hóa các phép toán:

$$\begin{aligned} C + (+\infty) &= (+\infty) + C = +\infty \\ C + (-\infty) &= (-\infty) + C = -\infty \end{aligned}$$

Dưới đây ta sẽ xét một số thuật toán tìm đường đi ngắn nhất từ đỉnh s tới đỉnh t trên đồ thị có hướng $G = (V, E)$ có n đỉnh và m cung, các đỉnh được đánh số từ 1 tới n . Trong trường hợp đồ thị vô hướng với trọng số không âm, bài toán tìm đường đi ngắn nhất có thể quy đổi về bài toán tìm đường đi ngắn nhất trên phiên bản có hướng của đồ thị.

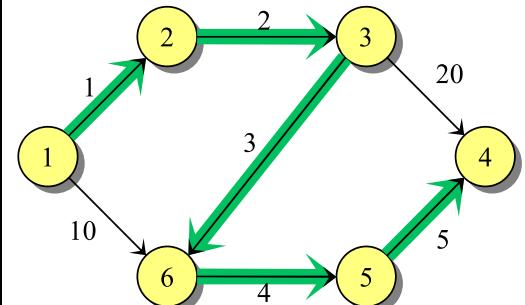
Input

- Dòng 1 chứa số đỉnh $n \leq 10^4$, số cung $m \leq 10^5$ của đồ thị, đỉnh xuất phát s , đỉnh đích t .
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, w , cho biết (u, v) là một cung $\in E$ và trọng số của cung đó là w (w là số nguyên có giá trị tuyệt đối $\leq 10^5$)

Output

Đường đi ngắn nhất từ s tới t và độ dài đường đi đó.

Sample Input	Sample Output
6 7 1 4 1 2 1 1 6 10 2 3 2 3 4 20 3 6 3 5 4 5 6 5 4	Distance from 1 to 4: 15 4<-5<-6<-3<-2<-1



e) Thuật toán Bellman-Ford

□ Thuật toán

Thuật toán Bellman-Ford[5][14] có thể sử dụng để tìm đường đi ngắn nhất xuất phát từ một đỉnh $s \in V$ trong trường hợp đồ thị $G = (V, E, w)$ không có chu trình âm. Thuật toán này khá đơn giản: Khởi tạo các nhãn khoảng cách $d[s] := 0$ và $d[v] := +\infty, \forall v \neq s$, sau đó thực hiện phép co theo mọi cạnh của đồ thị. Cứ lặp lại như vậy đến khi không thể cực tiểu hóa thêm bất kỳ một nhãn $d[v]$ nào nữa.

```

Init;
repeat
    Stop := True;
    for  $\forall e \in E$  do
        if Relax( $e$ ) then Stop := False;
    until Stop;
  
```

□ Tính đúng và tính dừng

Gọi $\delta_k(s, v)$ là độ dài ngắn nhất của một đường đi từ s tới v qua đúng k cạnh, nếu không tồn tại đường đi từ s tới v qua k cạnh thì $\delta_k(s, v) = +\infty$.

Ta chứng minh rằng sau mỗi lần lặp thứ k của vòng lặp repeat...until thì

$$d[v] \leq \delta_k(s, v), \forall v \in V \quad (1.2)$$

Tại bước khởi tạo, rõ ràng $d[v] = \delta_0(s, v)$. Giả sử bất đẳng thức (1.2) đúng trước lần lặp thứ k , ta chứng minh rằng bất đẳng thức vẫn đúng sau lần lặp thứ k . Thật vậy, đường đi ngắn nhất từ s tới v qua k cạnh sẽ phải thành lập bằng

cách lấy một đường đi ngắn nhất từ s tới một đỉnh u nào đó qua $k - 1$ cạnh rồi đi tiếp tới v bằng cung (u, v) : $s \rightsquigarrow u \rightarrow v$. Vì thế $\delta^k(s, v)$ có thể được tính bằng công thức truy hồi:

$$\begin{aligned}\delta_k(s, v) &= \min_{\substack{u \in V \\ (u, v) \in E}} \{\delta_{k-1}(s, u) + w(u, v)\} \\ &\geq \min_{\substack{u \in V \\ (u, v) \in E}} \{d[u] + w(u, v)\} \\ &\geq d[v]\end{aligned}\tag{1.3}$$

Bất đẳng thức thứ nhất đúng do giả thiết quy nạp và các phép co không bao giờ làm tăng $d[u]$. Bất đẳng thức thứ hai đúng vì sau khi căng theo tất cả các cạnh (\dots, v) thì không thể tồn tại u để $d[v] > d[u] + w(u, v)$ được nữa.

Trong các đường đi ngắn nhất từ s tới v , sẽ có một đường đi đơn (qua không quá $n - 1$ cạnh). Tức là $\delta(s, v) = \min_{k:1 \leq k \leq n-1} \delta_k(s, v)$. Sau $n - 1$ bước lặp của vòng lặp repeat...until, ta thu được các $d[v]$ thỏa mãn $d[v] \leq \delta_k(s, v)$ với mọi $v \in V$ và mọi số $k = 1, 2, \dots, n - 1$. Điều này chỉ ra rằng: $d[v] \leq \delta(s, v), \forall v \in V$. Mặt khác $d[v] \geq \delta(s, v)$ (tính bị chặn dưới), vậy $\forall v \in V: d[v] = \delta(s, v)$ sau $n - 1$ bước lặp repeat...until, điều này cũng cho thấy thuật toán Bellman-Ford sẽ kết thúc sau không quá $n - 1$ bước lặp repeat...until.

□ Cài đặt

Cách biểu diễn đồ thị tốt nhất để cài đặt thuật toán Bellman-Ford là sử dụng danh sách cạnh. Danh sách cạnh của đồ thị được lưu trữ trong mảng $e[1 \dots m]$, mỗi phần tử của mảng là một bản ghi chứa chỉ số hai đỉnh đầu mút u, v và trọng số w tương ứng với một cạnh

BELLMANFORD.PAS ✓ Thuật toán Bellman-Ford

```
{ $MODE OBJFPC }
program BellmanFordShortestPath;
const
  maxN = 10000;
  maxM = 100000;
  maxW = 100000;
  maxD = maxN * maxW;
type
```

```

TEdge = record //Cáu trúc biểu diễn cung
    x, y: Integer; //Đỉnh đầu và đỉnh cuối
    w: Integer; //Trọng số
end;
var
    e: array[1..maxM] of TEdge; //Danh sách cung
    d: array[1..maxN] of Integer; //Nhân trọng số
    trace: array[1..maxN] of Integer; //Vết
    n, m, s, t: Integer;
procedure Enter; //Nhập dữ liệu
var i: Integer;
begin
    ReadLn(n, m, s, t);
    for i := 1 to m do
        with e[i] do ReadLn(x, y, w);
    end;
procedure Init; //Khởi tạo
var v: Integer;
begin
    for v := 1 to n do d[v] := MaxD; //Các nhân d[v] := +∞
    d[s] := 0; //Ngoại trừ d[s] = 0
    end;
function Relax(const e: TEdge): Boolean; //Phép co theo cạnh e
begin
    with e do
        begin
            Result := (d[x] < maxD) and (d[y] > d[x] + w);
            if Result then //Co được
                begin
                    d[y] := d[x] + w; //Cực tiểu hóa nhân d[v]
                    trace[y] := x; //Lưu vết
                end;
        end;
    end;
procedure BellmanFord;
var
    Stop: Boolean;
    i, CountLoop: Integer;

```

```

begin
  for CountLoop := 1 to n - 1 do //Lặp tối đa n - 1 lần
    begin
      Stop := True; //Chưa có sự thay đổi nhẫn nào
      for i := 1 to m do
        if Relax(e[i]) then Stop := False;
        if Stop then Break; //Không nhẫn nào thay đổi, dừng
      end;
    end;
  procedure PrintResult; //In kết quả
  begin
    if d[t] = maxD then //d[t] = +∞, không có đường
      WriteLn('There is no path from ', s, ' to ', t)
    else
      begin
        WriteLn('Distance from ', s, ' to ', t, ':',
               ', ', d[t]);
        while t <> s do //Truy vết từ t
          begin
            Write(t, '<-');
            t := trace[t];
          end;
        WriteLn(s);
      end;
    end;
  begin
    Enter;
    Init;
    BellmanFord;
    PrintResult;
  end.

```

Dễ thấy rằng thời gian thực hiện giải thuật Bellman-Ford trên đồ thị $G(V, E, w)$ là $O(|V||E|)$.

f) Thuật toán Dijkstra

□ Thuật toán

Trong **trường hợp đồ thị $G = (V, E, w)$ có trọng số trên các cung không âm**, thuật toán do Dijkstra [8] để xuất dưới đây hoạt động hiệu quả hơn nhiều so với thuật toán Bellman-Ford bởi quá trình sửa nhãn sẽ *chỉ xét mỗi cạnh tối đa một lần*. Tại mỗi bước, thuật toán đi tìm đỉnh u mà nhãn $d[u]$ đã đạt cận dưới bằng $\delta(s, u)$. Nhãn $d[u]$ chắc chắn không thể cực tiểu hóa được nữa, khi đó thuật toán mới tiến hành cực tiểu hóa các nhãn $d[v]$ khác bằng các phép sửa nhãn theo cạnh (u, v) . Các bước cụ thể được tiến hành như sau:

Bước 1: Khởi tạo

Gọi thủ tục Init để khởi tạo các nhãn khoảng cách $d[s] := 0$ và $d[v] := +\infty, \forall v \neq s$. Một nhãn $d[v]$ gọi là **cố định** nếu ta biết chắc $d[v] = \delta(s, v)$ và không thể cực tiểu hóa $d[v]$ thêm nữa bằng phép co, ngược lại nhãn $d[v]$ gọi là **tự do**. Ta sẽ đánh dấu trạng thái nhãn bằng mảng $avail[1..n]$ trong đó $avail[v] = \text{True}$ nếu nhãn $d[v]$ còn tự do. Ban đầu tất cả các nhãn đều tự do.

Bước 2: Lặp, bước lặp gồm hai thao tác:

- **Cố định nhãn:** Chọn trong các đỉnh có nhãn tự do, lấy ra đỉnh u là đỉnh có $d[u]$ nhỏ nhất, đánh dấu cố định nhãn đỉnh u ($avail[u] := \text{False}$).
- **Sửa nhãn:** Dùng đỉnh u , xét tất cả những đỉnh v nối từ u và thực hiện phép co theo cung (u, v) để cực tiểu hóa nhãn $d[v]$.

Bước lặp sẽ kết thúc khi mà đỉnh đích t được cố định nhãn (tìm được đường đi ngắn nhất từ s tới t); hoặc tại thao tác cố định nhãn, tất cả các đỉnh tự do đều có nhãn là $+\infty$ (không tồn tại đường đi).

Tại lần lặp đầu tiên, đỉnh s có $d[s]$ nhỏ nhất (bằng 0) sẽ được cố định nhãn. Có thể đặt câu hỏi tại sao đỉnh u có nhãn tự do nhỏ nhất được cố định nhãn tại từng bước, giả sử $d[u]$ còn có thể làm nhỏ hơn nữa thì tất phải có một đỉnh x mang nhãn tự do sao cho $d[u] > d[x] + w(x, u)$. Do trọng số $w(x, u)$ không âm nên $d[u] > d[x]$, trái với cách chọn $d[u]$ nhỏ nhất.

Bước 3: Truy vết

Kết hợp với việc lưu vết đường đi trên từng bước sửa nhãn, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi.

□ Cài đặt

Thuật toán Dijkstra hoạt động tốt nhất nếu đồ thị được biểu diễn bằng danh sách kè dạng forward star. Cấu trúc danh sách kè được khai báo như sau:

```
type
  TAdjNode = record //Cấu trúc nút của danh sách kè
    v: Integer; //Đỉnh kè
    w: Integer; //Trọng số cạnh tương ứng
  end;
var
  adj: array[1..maxM] of TAdjNode; //Mảng các nút
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số nút đầu tiên của danh sách kè u
  link: array[1..maxM] of Integer;
  //link[i]: Chỉ số nút kè tiếp nút adj[i] trong cùng một danh sách kè
```

Mỗi đỉnh u sẽ tương ứng với một danh sách các nút, mỗi nút chứa một đỉnh v và trọng số w của một cung (u, v) . Tất cả các nút được lưu trữ trong mảng $adj[1 \dots m]$ và mỗi nút sẽ thuộc đúng một danh sách kè. Các nút thuộc danh sách kè của đỉnh u là $adj[i_1], adj[i_2], adj[i_3]$, trong đó $i_1 = head[u]; i_2 = link[i_1]; i_3 = link[i_2] \dots$ Đây chính là cấu trúc dữ liệu biểu diễn danh sách mốc nối đơn, nhưng khác với những phép cài đặt truyền thống, ta sử dụng mảng các nút thay cho cơ chế cấp phát biến động và sử dụng chỉ số với vai trò như con trỏ.

DIJKSTRA.PAS ✓

```
{$MODE OBJFPC}
program DijkstraShortestPath;
const
  maxN = 10000;
  maxM = 100000;
  maxW = 100000;
  maxD = maxN * maxW;
type
  TAdjNode = record //Cấu trúc nút của danh sách kè
    v: Integer; //Đỉnh kè
    w: Integer; //Trọng số cung tương ứng
```

```

link: Integer; //Chỉ số nút kế tiếp trong cùng danh sách kè
end;

var
adj: array[1..maxM] of TAdjNode; //Mảng chứa tất cả các nút
head: array[1..maxN] of Integer;
//head[u]: Chỉ số nút đứng đầu danh sách kè của u
d: array[0..maxN] of Integer; //Nhận khoảng cách
avail: array[1..maxN] of Boolean; //Đánh dấu tự do/có định
trace: array[1..maxN] of Integer; //Vết đường đi
n, m, s, t: Integer;

procedure Enter; //Nhập dữ liệu
var i, u: Integer;
begin
ReadLn(n, m, s, t);
FillChar(head[1], n * SizeOf(head[1]), 0);
//Khởi tạo các danh sách kè rỗng
for i := 1 to m do
begin
ReadLn(u, adj[i].v, adj[i].w);
//Đọc một cung (u, v) trọng số w, đưa v và w vào trong nút adj[i]
adj[i].link := head[u]; //Chèn nút adj[i] vào đầu danh sách kè của u
head[u] := i; //Cập nhật chỉ số nút đứng đầu danh sách kè của u
end;
end;

procedure Init; //Khởi tạo
var v: Integer;
begin
for v := 0 to n do d[v] := MaxD;
//Các nhãn d[v] := +∞, d[0]: phần tử cầm cành
d[s] := 0; //Ngoại trừ d[s] = 0
FillChar(avail[1], n * SizeOf(avail[1]), True);
//Các nhãn đều tự do
end;

procedure Relax(u, v: Integer; w: Integer);
//Phép co theo cung (u, v) trọng số w
begin
if d[v] > d[u] + w then
begin
d[v] := d[u] + w;

```

```

    trace[v] := u;
end;
end;
procedure Dijkstra; //Thuật toán Dijkstra
var u, v, i: Integer;
begin
    repeat
        //Tìm đỉnh u có nhán tự do nhỏ nhất
        u := 0;
        for v := 1 to n do
            if avail[v] and (d[v] < d[u]) then
                u := v;
            if (u = 0) or (u = t) then Break;
            //u = 0: không tồn tại đường đi, u = t, xong
            avail[u] := False; //Cô định nhán đỉnh u
            //Co theo các cung nối từ u
            i := head[u]; //Duyệt từ đầu danh sách kè
            while i <> 0 do
                begin
                    Relax(u, adj[i].v, adj[i].w); //Thực hiện phép co
                    i := adj[i].link; //Chuyển sang nút kè tiếp trong danh sách kè
                end;
            until False;
        end;
        procedure PrintResult; //In kết quả
        begin
            if d[t] = maxD then
                WriteLn('There is no path from ', s, ' to ', t)
            else
                begin
                    WriteLn('Distance from ', s, ' to ', t, ':',
                           ', d[t]);'
                    while t <> s do
                        begin
                            Write(t, '<-');
                            t := trace[t];
                        end;
                    WriteLn(s);
                
```

```

end;
end;
begin
    Enter;
    Init;
    Dijkstra;
    PrintResult;
end.

```

Mỗi lượt của vòng lặp repeat...until sẽ có một đỉnh mang nhãn tự do bị cố định nhãn, suy ra số lượt lặp của vòng lặp repeat...until là $O(|V|)$. Việc tìm đỉnh u có nhãn tự do nhỏ nhất được thực hiện trong thời gian $O(|V|)$, vậy nên xét trên toàn thuật toán, tổng thời gian thực hiện của các pha cố định nhãn là $O(|V|^2)$.

Mỗi lượt của vòng lặp repeat...until khi cố định nhãn đỉnh u sẽ phải duyệt danh sách các đỉnh nối từ u để thực hiện pha sửa nhãn. Vì vậy xét trên toàn thuật toán, tổng thời gian thực hiện của các pha sửa nhãn là $O(\sum_{u \in V} \deg^+(u)) = O(|E|)$.

Vậy thủ tục Dijkstra thực hiện trong thời gian $O(|V|^2 + |E|)$.

□ Kết hợp với hàng đợi ưu tiên

Để thuật toán Dijkstra làm việc hiệu quả hơn, người ta thường kết hợp với một cấu trúc dữ liệu hàng đợi ưu tiên chứa các đỉnh tự do có nhãn $\neq +\infty$ để thuận tiện trong việc lấy ra đỉnh có nhãn nhỏ nhất cũng như cập nhật lại nhãn của các đỉnh. Hàng đợi ưu tiên cần hỗ trợ các thao tác sau:

- *Extract*: Lấy ra một đỉnh ưu tiên nhất (đỉnh u có $d[u]$ nhỏ nhất) khỏi hàng đợi ưu tiên.
- *Update(v)*: Thao tác này báo cho hàng đợi ưu tiên biết rằng nhãn $d[v]$ đã bị giảm đi, cần tổ chức lại (thêm v vào hàng đợi ưu tiên nếu v đang nằm ngoài).

Khi đó thuật toán Dijkstra có thể viết theo mô hình mới sử dụng hàng đợi ưu tiên:

```

Init;
PQ := (s); //Hàng đợi ưu tiên được khởi tạo chỉ gồm đỉnh xuất phát
repeat
    u := Extract; //Lấy ra đỉnh u có d[u] nhỏ nhất

```

```

if u = t then Break;
for  $\forall (u, v) \in E$  do
    if Relax(u, v) then Update(v);
until PQ =  $\emptyset$ ;
if d[t] =  $+\infty$  then
    Output  $\leftarrow$  Không có đường
else
    «Truy vết tìm đường đi từ s tới t»

```

Vòng lặp repeat...until mỗi lần sẽ lấy một đỉnh khỏi hàng đợi ưu tiên và đỉnh lấy ra sẽ không bao giờ bị đẩy vào hàng đợi ưu tiên lại nữa. Vòng lặp for bên trong xét trong tổng thể cả chương trình sẽ duyệt qua tất cả các cung (u, v) của đồ thị. Vậy thuật toán Dijkstra cần thực hiện không quá n phép Extract và m phép Update với n là số đỉnh và m là số cung của đồ thị.

Trong chương trình cài đặt thuật toán Dijkstra dưới đây tôi sử dụng Binary Heap để biểu diễn hàng đợi ưu tiên, khi đó thời gian thực hiện giải thuật sẽ là $O(n \lg n + m \lg n)$. Ngoài cấu trúc Binary Heap, người ta cũng đã nghiên cứu nhiều cấu trúc dữ liệu hiệu quả hơn để biểu diễn hàng đợi ưu tiên, chẳng hạn Fibonacci Heap[17], Relaxed Heap[11], 2-3 Heap[39], v.v... Những cấu trúc dữ liệu này cho phép cài đặt thuật toán Dijkstra chạy trong thời gian $O(n \lg n + m)$. Tuy nhiên việc cài đặt các cấu trúc dữ liệu này khá phức tạp, các bạn có thể tham khảo trong những tài liệu khác.

DIJKSTRAHEAP.PAS ✓ Thuật toán Dijkstra và cấu trúc Heap

```

{ $MODE OBJFPC }
program DijkstraShortestPathUsingHeap;
const
    maxN = 10000;
    maxM = 100000;
    maxW = 100000;
    maxD = maxN * maxW;
type
    TAdjNode = record //Cấu trúc nút của danh sách kề
        v: Integer; //Đỉnh kề
        w: Integer; //Trọng số cung tương ứng
        link: Integer; //Chỉ số nút kề tiếp trong cùng danh sách kề
    end;

```

```

end;
THeap = record //Cáu trúc Heap
  Items: array[1..maxN] of Integer; //Các phần tử chứa trong
  nItems: Integer; //Số phần tử chứa trong
  Pos: array[1..maxN] of Integer;
  //Pos[v] = vị trí của đỉnh v trong Heap
end;

var
  adj: array[1..maxM] of TAdjNode; //Mảng chứa tất cả các nút
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số nút đứng đầu danh sách kè của u
  d: array[1..maxN] of Integer;
  trace: array[1..maxN] of Integer;
  n, m, s, t: Integer;
  Heap: THeap;
procedure Enter; //Nhập dữ liệu
var i, u: Integer;
begin
  ReadLn(n, m, s, t);
  FillChar(head[1], n * SizeOf(head[1]), 0);
  //Khởi tạo các danh sách kè rỗng
  for i := 1 to m do
    begin
      ReadLn(u, adj[i].v, adj[i].w);
      //Đọc một cung (u, v) trọng số w, đưa v và w vào trong nút adj[i]
      adj[i].link := head[u]; //Chèn nút adj[i] vào đầu danh sách kè của u
      head[u] := i; //Cập nhật chỉ số nút đứng đầu danh sách kè của u
    end;
  end;
procedure Init;
var v: Integer;
begin
  for v := 1 to n do d[v] := MaxD;
  d[s] := 0;
  with Heap do //Khởi tạo Heap chỉ chứa mỗi phần tử s
    begin
      FillChar(Pos[1], n * SizeOf(Pos[1]), 0);
      Items[1] := s;
      Pos[s] := 1;
    end;

```

```

nItems := 1;
end;
end;
function Extract: Integer; //Lấy đỉnh u có nhãn d[u] nhỏ nhất ra khỏi Heap
var p, c, v: Integer;
begin
  with Heap do
    begin
      Result := Items[1]; //Trả về đỉnh ở gốc Heap
      v := Items[nItems]; //Vun lại Heap bằng phép Down-Heap
      Dec(nItems);
      p := 1; //Bắt đầu từ gốc
      repeat
        //Tìm c là nút con chứa đỉnh mang nhãn khoảng cách nhỏ hơn trong hai nút con
        c := p * 2;
        if (c < nItems)
          and (d[Items[c + 1]] < d[Items[c]]) then
            Inc(c);
        if (c > nItems)
          or (d[v] <= d[Items[c]]) then Break;
        Items[p] := Items[c]; //Chuyển đỉnh từ c lên p
        Pos[Items[p]] := p; //Cập nhật vị trí
        p := c; //Di xuống nút con
      until False;
      Items[p] := v; //Đặt đỉnh v vào nút p của Heap
      Pos[v] := p; //Cập nhật vị trí
    end;
  end;
procedure Update(v: Integer); //d[v] vừa bị cực tiểu hóa, tổ chức lại Heap
var p, c: Integer;
begin
  with Heap do
    begin
      c := Pos[v]; //c là vị trí của đỉnh v trong Heap
      if c = 0 then //Nếu v chưa có trong Heap
        begin
          Inc(nItems);
          c := nItems; //Cho v vào Heap ở vị trí một nút lá
        end;
    end;
  
```

```

    end;
repeat //Thực hiện Up-Heap
    p := c div 2; //Xét nút cha của c
    if (p = 0) or (d[Items[p]] <= d[v]) then Break;
    //Dừng nếu đã xét lên gốc hoặc gặp vị trí đúng
    Items[c] := Items[p]; //Kéo đỉnh từ nút cha xuống nút con
    Pos[Items[c]] := c; //Cập nhật vị trí
    c := p; //Di lên nút cha
until False;
Items[c] := v; //Đặt v vào nút c
Pos[v] := c; //Cập nhật vị trí
end;
end;

function Relax(u, v: Integer; w: Integer): Boolean;
//Phép co theo cạnh (u, v) trọng số w
begin
    Result := d[v] > d[u] + w;
    if Result then
        begin
            d[v] := d[u] + w;
            trace[v] := u;
        end;
    end;
end;

procedure Dijkstra; //Thuật toán Dijkstra
var u, i: Integer;
begin
repeat
    u := Extract; //Lấy ra đỉnh u có d[u] nhỏ nhất
    if (u = 0) or (u = t) then Break;
    i := head[u]; //Duyệt từ đầu danh sách kề của u
    while i <> 0 do
        begin
            if Relax(u, adj[i].v, adj[i].w) then
                //Nếu thực hiện được phép co
                Update(adj[i].v); //Tổ chức lại Heap
            i := adj[i].link; //Chuyển sang nút kế tiếp trong danh sách kề
        end;
until Heap.nItems = 0;
end;

```

```

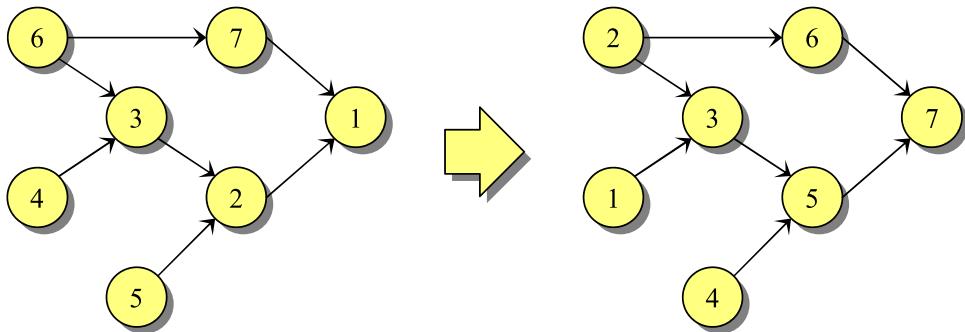
procedure PrintResult; //In kết quả
begin
  if d[t] = maxD then
    WriteLn('There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('Distance from ', s, ' to ', t, ':',
              ', d[t]);
      while t <> s do
        begin
          Write(t, '<-');
          t := trace[t];
        end;
      WriteLn(s);
    end;
  end;
begin
  Enter;
  Init;
  Dijkstra;
  PrintResult;
end.

```

g) Đường đi ngắn nhất trên đồ thị không có chu trình

Thuật toán

Xét **trường hợp đồ thị có hướng, không có chu trình** (Directed Acyclic Graph - DAG), có một thuật toán hiệu quả để tìm đường đi ngắn nhất dựa trên kỹ thuật sắp xếp Tô pô (Topological Sorting), cơ sở của thuật toán dựa vào định lý: Nếu $G = (V, E)$ là một DAG thì các đỉnh của nó có thể đánh số sao cho mỗi cung của G chỉ nối từ đỉnh có chỉ số nhỏ hơn đến đỉnh có chỉ số lớn hơn.



Hình 2.2. Phép đánh chỉ số lại theo thứ tự tô pô

Phép đánh số theo thứ tự tô pô đã được trình bày trong thuật toán Kosaraju-Sharir (**Error! Reference source not found.**, **Mục Error! Reference source not found.**): Dùng thuật toán tìm kiếm theo chiều sâu trên đồ thị đảo chiều và đánh số các đỉnh theo thứ tự duyệt xong (Finish).

```

procedure TopoSort;
    procedure DFSVisit (v $\in$ V);
        //Thuật toán tìm kiếm theo chiều sâu từ đỉnh v trên đồ thị đảo chiều
        begin
            avail[v] := False; //avail[v] = False  $\Leftrightarrow$  v đã thăm
            for  $\forall u \in V: (u, v) \in E$  do //Duyệt mọi đỉnh v chưa thăm nối đến u
                if avail[u] then
                    DFSVisit (u); //Gọi đệ quy để tìm kiếm theo chiều sâu từ đỉnh u
                «Đánh số v»; //Đánh số v theo thứ tự duyệt xong
            end;
        begin
            for  $\forall v \in V$  do avail[v] := True; //Đánh dấu mọi đỉnh đều chưa thăm
            for  $\forall v \in V$  do
                if avail[v] then DFSVisit (v);
            end.

```

Một cách khác để đánh số theo thứ tự tô pô là sử dụng thuật toán tìm kiếm theo chiều rộng: Với mỗi đỉnh v ta tính và lưu trữ $\text{deg}^-[v]$ là bán bậc vào của nó. Sử dụng một hàng đợi chứa các đỉnh có bán bậc vào bằng 0 (đỉnh không có cung đi vào). Sau đó cứ lấy một đỉnh u khỏi hàng đợi, đánh số cho đỉnh u đó và xóa đỉnh u khỏi đồ thị. Việc xóa đỉnh u khỏi đồ thị tương đương với việc giảm tất cả các $\text{deg}^-[v]$ của những đỉnh v nối từ u đi 1, nếu $\text{deg}^-[v]$ bị giảm về 0 thì đẩy v vào hàng đợi để chờ...Quá trình đánh số sẽ kết thúc khi hàng đợi rỗng (tất cả

các đỉnh đều đã được đánh số thứ tự mới). Phương pháp này tuy cài đặt có dài dòng hơn và chậm hơn một chút nhưng không cần sử dụng đệ quy.

Nếu các đỉnh được đánh số sao cho mỗi cung phải nối từ một đỉnh tới một đỉnh khác mang chỉ số lớn hơn thì thuật toán tìm đường đi ngắn nhất có thể thực hiện khá đơn giản:

```
Init;  
for v := s + 1 to t do  
    for ∀u: (u, v) ∈ E do Relax(u, v); //Co theo các cung nối tới v
```

Có thể thấy rằng sau mỗi bước lặp với đỉnh v , nhãn $d[v]$ không thể co thêm được nữa ($d[u] = \delta(s, u)$).

□ Cài đặt

Vì mục tiêu của chúng ta chỉ cần tìm đường đi từ s tới t , vì thế chúng ta chỉ cần đánh số thứ tự tô pô cho những đỉnh đến được t bằng lời gọi $DFSVisit(t)$, những đỉnh đến được từ t (có thứ tự tô pô lớn hơn t) sẽ bị bỏ qua.

Cách cài đặt thông thường nhất để tìm đường đi ngắn nhất trên đồ thị không có chu trình là tách biệt hai pha: Sắp xếp tô pô và tối ưu nhãn. Pha sắp xếp tô pô trước hết thực hiện tìm kiếm theo chiều sâu trên đồ thị đảo chiều bằng lời gọi $DFSVisit(t)$. Mỗi khi một đỉnh được duyệt xong (đánh số thứ tự tô pô), nó sẽ được đẩy vào một hàng đợi. Pha tối ưu nhãn lần lượt lấy các đỉnh ra khỏi hàng đợi (theo đúng thứ tự tô pô) và thực hiện phép co theo tất cả các cung nối tới đỉnh vừa lấy ra. Cách cài đặt này có ưu điểm là khá sáng sủa, trong trường hợp mà ta bỏ qua được khâu sắp xếp tô pô hoặc có thể xác định thứ tự tô pô bằng một cách đơn giản hơn, chương trình trở nên rất gọn.

Tuy nhiên nếu ta phải giải quyết bài toán tổng quát bao gồm cả hai pha sắp xếp tô pô và tối ưu nhãn thì có thể khéo léo lồng pha tối ưu nhãn vào pha sắp xếp tô pô: Trước khi đỉnh v được đánh số (duyệt xong), ta thực hiện tất cả các phép co theo các cung (u, v) với mọi đỉnh u nối đến v .

💻 SHORTESTPATHDAG.PAS ✓ Đường đi ngắn nhất trên DAG

```
{ $MODE OBJFPC }  
program DAGShortestPath;  
const  
    maxN = 10000;
```

```

maxM = 100000;
maxW = 100000;
maxD = maxN * maxW;

type
  TAdjNode = record //Cấu trúc nút của danh sách kề dạng reverse star
    u: Integer; //Định kề
    w: Integer; //Trọng số cung tương ứng
    link: Integer; //Chỉ số nút kề tiếp trong cùng danh sách kề
  end;
var
  adj: array[1..maxM] of TAdjNode; //Mảng chứa tất cả các nút
  head: array[1..maxN] of Integer;
  //head[u]: Chỉ số nút đứng đầu danh sách kề của u
  d: array[1..maxN] of Integer; //Nhận khoảng cách
  trace: array[1..maxN] of Integer; //Vết
  avail: array[1..maxN] of Boolean;
  n, m, s, t: Integer;
procedure Enter; //Nhập dữ liệu
var i, v: Integer;
begin
  ReadLn(n, m, s, t);
  FillChar(head[1], n * SizeOf(head[1]), 0);
  for i := 1 to m do
    begin
      ReadLn(adj[i].u, v, adj[i].w);
      //Đọc một cung (u, v) trọng số w, đưa u và w vào trong nút adj[i]
      adj[i].link := head[v]; //Chèn nút adj[i] vào đầu danh sách kề của v
      head[v] := i; //Cập nhật chỉ số nút đứng đầu danh sách kề của v
    end;
  end;
procedure Init;
var v: Integer;
begin
  FillChar(avail[1], n * SizeOf(avail[1]), True);
  for v := 1 to n do d[v] := maxD;
  d[s] := 0;
end;
procedure Relax(u, v: Integer; w: Integer);

```

```

begin
  if (d[u] < maxD) and (d[v] > d[u] + w) then
    begin
      d[v] := d[u] + w;
      trace[v] := u;
    end;
  end;
procedure DFSVisit(v: Integer); //DFS trên đồ thị đảo chiều
var i: Integer;
begin
  avail[v] := False;
  i := head[v];
  while i <> 0 do //Duyệt danh sách các đỉnh nối đến v
    begin
      if avail[adj[i].u] then
        //adj[i].u là một đỉnh nối đến v, nếu adj[i].u chưa thăm thì đi thăm,
        DFSVisit(adj[i].u); //sau lời gọi này d[adj[i].u] sẽ bằng δ(s, adj[i].u)
        Relax(adj[i].u, v, adj[i].w);
        //Thực hiện luôn phép co (adj[i].u, v)
        i := adj[i].link; //Nhảy sang nút kế tiếp trong danh sách kế
    end;
  end; //Khi thu tục kết thúc, d[v] sẽ bằng δ(s,v)
procedure PrintResult; //In kết quả
begin
  if d[t] = maxD then
    WriteLn('There is no path from ', s, ' to ', t)
  else
    begin
      WriteLn('Distance from ', s, ' to ', t, ':',
              ', ', d[t]);
      while t <> s do
        begin
          Write(t, '<-');
          t := trace[t];
        end;
      WriteLn(s);
    end;
  end;

```

```

begin
    Enter;
    Init;
    DFSVisit(t);
    PrintResult;
end.

```

Thời gian thực hiện giải có thể đánh giá qua thời gian thực hiện giải thuật DFS, tức là bằng $O(|E|)$ khi đồ thị được biểu diễn bởi danh sách kề.

1.3. Đường đi ngắn nhất giữa mọi cặp đỉnh

Trong một số ứng dụng thực tế, đôi khi người ta có nhu cầu tính sẵn *đường đi ngắn nhất giữa mọi cặp đỉnh* của đồ thị (*all-pairs shortest paths*) để trả lời nhanh những truy vấn tìm đường đi ngắn nhất mà không cần thực hiện lại thuật toán. Rõ ràng ta có thể áp dụng thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với n lượt chọn đỉnh xuất phát, nhưng những thuật toán trong mục này có thể thực hiện nhanh hơn và đơn giản hơn nhiều.

a) Thuật toán Floyd

Cho đơn đồ thị có hướng, có trọng số $G = (V, E)$ với n đỉnh và m cung. Thuật toán Floyd tính tất cả các phần tử của ma trận khoảng cách $D = \{d_{uv}\}_{n \times n}$, trong đó $d[u, v]$ là khoảng cách từ u tới v . Cách làm tương tự như thuật toán Warshall để tìm bao đóng đồ thị: từ ma trận trọng số $W = \{w[u, v]\}_{n \times n}$, trong đó $w[v, v] = 0, \forall v \in V$, thuật toán Floyd tính lại các $w[u, v]$ thành độ dài đường đi ngắn nhất từ u tới v theo cách sau: Với $\forall k \in V$ được xét theo thứ tự từ 1 tới n , thuật toán xét mọi cặp đỉnh u, v và cực tiểu hóa $w[u, v]$ theo công thức:

$$w[u, v]_{\text{mi}} := \min\{w[u, v]_{\text{cũ}}, w[u, k] + w[k, v]\} \quad (1.4)$$

Tức là nếu như đường đi từ u tới v đang có lại dài hơn đường đi từ u tới k cộng với đường đi từ k tới v thì ta huỷ bỏ đường đi từ u tới v hiện thời và coi đường đi từ u tới v sẽ là nối của hai đường đi từ u tới k rồi từ k tới v :

```

for k := 1 to n do
    for u := 1 to n do
        for v := 1 to n do
            w[u, v] := min(w[u, v], w[u, k] + w[k, v]);

```

□ Tính đúng của thuật toán

Gọi $\delta_k(u, v)$ là độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k\}$. Rõ ràng khi $k = 0$ thì $\delta_0(u, v) = w[u, v]$ (đường đi ngắn nhất là đường đi trực tiếp không qua đỉnh trung gian nào).

Nếu đường đi ngắn nhất từ u tới v mà chỉ qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k\}$ lại:

- Không đi qua đỉnh k , tức là chỉ qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k - 1\}$ thì $\delta_k(u, v) = \delta_{k-1}(u, v)$.
- Có đi qua đỉnh k , thì đường đi đó sẽ là nối của một đường đi ngắn nhất từ u tới k và một đường đi ngắn nhất từ k tới v , hai đường đi này chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k - 1\}$, vậy $\delta_k(u, v) = \delta_{k-1}(u, k) + \delta_{k-1}(k, v)$.

Vì ta muốn $\delta_k(u, v)$ nhỏ nhất nên suy ra:

$$\delta_k(u, v) = \min\{\delta_{k-1}(u, v), \delta_{k-1}(u, k) + \delta_{k-1}(k, v)\} \quad (1.5)$$

Cuối cùng ta quan tâm tới các $\delta_n(u, v)$: Độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, n\}$, tức là khoảng cách giữa u và v : $\delta(u, v)$.

Ta sẽ chứng minh rằng sau mỗi bước lặp của vòng lặp “for $k\dots$ ”, thì:

$$w[u, v] \leq \delta_k(u, v) \quad (1.6)$$

Phép chứng minh được thực hiện quy nạp theo k . Ký hiệu $w_k[u, v]$ là giá trị $w[u, v]$ sau vòng lặp thứ k , khi $k = 0$ thì như đã chỉ ra ở trên, $w_0[u, v] = \delta_0(u, v)$. Giả sử bất đẳng thức đúng với $k - 1$, trước hết dễ thấy rằng các $w[u, v]$ sẽ được tối ưu hóa giảm dần theo từng bước. Từ công thức (1.5), ta có:

$$\begin{aligned} \delta_k(u, v) &= \min \left\{ \underbrace{\delta_{k-1}(u, v)}_{\geq w_{k-1}(u, v)}, \underbrace{\delta_{k-1}(u, k) + \delta_{k-1}(k, v)}_{\geq w_{k-1}(u, k) + w_{k-1}(k, v)} \right\} \\ &\geq w_k[u, v] \end{aligned}$$

Mặt khác có thể thấy thuật toán Floyd tìm được $w_n[u, v]$ là độ dài của một đường đi từ u tới v . Tức là $w_n[u, v] \geq \delta_n(u, v)$. Từ những kết quả trên suy ra

khi kết thúc thuật toán, $w_n(u, v) = \delta_n(u, v) = \delta(u, v)$ là độ dài đường đi ngắn nhất từ u tới v .

□ Cài đặt

Ta sẽ cài đặt thuật toán Floyd trên đồ thị có hướng gồm n đỉnh, m cung với khuôn dạng Input/Output như sau:

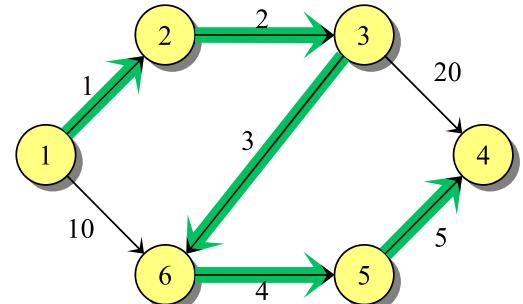
Input

- Dòng 1 chứa số đỉnh $n \leq 10^3$, số cung $m \leq 10^6$ của đồ thị, đỉnh xuất phát s , đỉnh cần đến t .
- m dòng tiếp theo, mỗi dòng có dạng ba số u, v, w , cho biết (u, v) là một cung $\in E$ và trọng số của cung đó là w (w là số nguyên có giá trị tuyệt đối $\leq 10^5$)

Output

Đường đi ngắn nhất từ s tới t và độ dài đường đi đó.

Sample Input	Sample Output
6 7 1 4 1 2 1 1 6 10 2 3 2 3 4 20 3 6 3 5 4 5 6 5 4	Distance from 1 to 4: 15 1->2->3->6->5->4



Thuật toán Floyd chỉ cần thực hiện một lần trên đồ thị và khi cần tìm đường đi ngắn nhất giữa một cặp đỉnh khác, ta chỉ cần dò đường dựa trên ma trận khoảng cách mà thôi. Trên thực tế người ta thường kết hợp với một cơ chế lưu vết để trả lời nhanh nhiều truy vấn về đường đi ngắn nhất: Gọi $trace[u, v]$ là đỉnh đứng liền sau u trên đường đi ngắn nhất từ u tới v . Sau mỗi phép cực tiểu hóa $c[u, v] := c[u, k] + c[k, v]$ (đường đi ngắn nhất từ u tới v phải đi vòng qua k), ta cập nhật lại vết $trace[u, v] := trace[u, k]$.



FLOYD.PAS ✓ Thuật toán Floyd

```
{$MODE OBJFPC}
program FloydAllPairsShortestPaths;
const
  maxN = 1000;
  maxW = 1000;
  maxD = maxN * maxW;
var
  w: array[1..maxN, 1..maxN] of Integer; //Ma trận trọng số
  trace: array[1..maxN, 1..maxN] of Integer; //Vết
  n, m, s, t: Integer;
procedure Enter; //Nhập dữ liệu, các cạnh không có được gán trọng số +∞
var
  i, u, v, weight: Integer;
begin
  ReadLn(n, m, s, t);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then w[u, v] := 0
      else w[u, v] := maxD;
  for i := 1 to m do
    begin
      ReadLn(u, v, weight);
      if w[u, v] > weight then
        //Phòng trường hợp nhiều cung nối từ u tới v (đa đồ thị)
        w[u, v] := weight; //Chỉ ghi nhận cung trọng số nhỏ nhất
    end;
  end;
procedure Floyd;
var k, u, v: Integer;
begin
  for u := 1 to n do
    for v := 1 to n do trace[u, v] := v;
    //Khởi tạo đường đi ngắn nhất là đường đi trực tiếp
  for k := 1 to n do
    for u := 1 to n do
      if w[u, k] < maxD then
        for v := 1 to n do
```

```

if (w[k, v] < maxD)
    and (w[u, v] > w[u, k] + w[k, v]) then
        begin //Cực tiểu hóa c[u, v]
            w[u, v] := w[u, k] + w[k, v];
            //Ghi nhận đường đi vòng qua k
            trace[u, v] := trace[u, k]; //Lưu vết
        end;
    end;
procedure PrintResult; //In kết quả
begin
    if w[s, t] = maxD then
        WriteLn('There is no path from ', s, ' to ', t)
    else
        begin
            WriteLn('Distance from ', s, ' to ', t, ':',
                    ', w[s, t]);
            while s <> t do
                begin
                    Write(s, '->');
                    s := trace[s, t];
                end;
                WriteLn(t);
            end;
        end;
    begin
        Enter;
        Floyd;
        PrintResult;
    end.

```

Dễ thấy rằng thời gian thực hiện giải thuật Floyd là $\Theta(n^3)$ và chi phí bộ nhớ là $\Theta(n^2)$.

b) Thuật toán Johnson

Trong trường hợp $G = (V, E)$ là đồ thị thưa gồm n đỉnh và m cạnh: $m \ll n^2$, thuật toán Johnson tìm đường đi ngắn nhất giữa mọi cặp đỉnh hoạt động hiệu quả hơn thuật toán Floyd. Bản chất của thuật toán Johnson là thực hiện thuật

toán Bellman-Ford để *gán lại trọng số* và thực hiện tiếp thuật toán Dijkstra để tìm đường đi ngắn nhất.

Nếu đồ thị không có cạnh trọng số âm, ta có thể thực hiện thuật toán Dijkstra n lần với cách chọn lần lượt n đỉnh làm đỉnh xuất phát. Bằng cách kết hợp với một hàng đợi ưu tiên được tổ chức dưới dạng Fibonacci Heap, thời gian thực hiện một lần thuật toán Dijkstra là $O(n \lg n + m)$, và như vậy ta có thể tìm đường đi ngắn nhất giữa mọi cặp đỉnh trong thời gian $O(n^2 \lg n + mn)$.

Nếu đồ thị có cạnh trọng số âm nhưng không có chu trình âm, thuật toán Johnson thực hiện một kỹ thuật gọi là *gán lại trọng số* (*re-weighting*). Tức là trọng số $w: E \rightarrow \mathbb{R}$ sẽ được biến đổi thành trọng số $\hat{w}: E \rightarrow \mathbb{R}$ thỏa mãn hai điều kiện sau đây:

- Với mọi cặp đỉnh $u, v \in V$, đường đi P là đường đi ngắn nhất từ u tới v ứng với trọng số w nếu và chỉ nếu P cũng là đường đi ngắn nhất từ u tới v ứng với trọng số \hat{w}
- Với mọi cạnh $e \in E$, trọng số $\hat{w}(e)$ là một số không âm

Định lý 1-2

Cho đồ thị $G = (V, E)$ với trọng số $w: E \rightarrow \mathbb{R}$. Gọi $h: V \rightarrow \mathbb{R}$ là một hàm gán cho mỗi đỉnh v một số thực $h(v)$. Xét trọng số $\hat{w}: E \rightarrow \mathbb{R}$ định nghĩa bởi:

$$\hat{w}(u, v) = w(u, v) + h(u) - h(v)$$

Xét $p = \langle v_0, v_1, \dots, v_k \rangle$ là một đường đi từ v_0 tới v_k , khi đó p là đường đi ngắn nhất từ v_0 tới v_k với trọng số w nếu và chỉ nếu p là đường đi ngắn nhất từ v_0 tới v_k với trọng số \hat{w} . Hơn nữa G có chu trình âm tương ứng với trọng số w nếu và chỉ nếu G có chu trình âm với trọng số \hat{w} .

Chứng minh

Ký hiệu $w(p)$ và $\hat{w}(p)$ lần lượt là độ dài đường đi p với trọng số w và \hat{w} . Ta có

$$\begin{aligned} \hat{w}(p) &= \sum_{i=1}^k \hat{w}(v_{i-1}, v_i) \\ &= \sum_{i=1}^k (w(v_{i-1}, v_i) + h(v_{i-1}) - h(v)) \end{aligned}$$

$$\begin{aligned}
&= \left(\sum_{i=1}^k w(v_{i-1}, v_i) \right) + h(v_0) - h(v_k) \\
&= w(p) + h(v_0) - h(v_k)
\end{aligned}$$

Vậy qua phép biến đổi trọng số, độ dài mọi đường đi từ v_0 tới v_k sẽ được cộng thêm một lượng $h(v_0) - h(v_k)$. Hay nói cách khác, đường đi nào ngắn nhất với trọng số w cũng là đường đi ngắn nhất với trọng số \widehat{w} .

Nếu p là một chu trình, $v_0 = v_k$, ta suy ra $\widehat{w}(p) = w(p)$. Độ dài của chu trình được bảo toàn qua phép biến đổi trọng số, tức là G có chu trình âm với trọng số w nếu và chỉ nếu G có chu trình âm với trọng số \widehat{w} .

Mục tiêu tiếp theo là chỉ ra một phép gán trọng số mới cho đồ thị G để đảm bảo các trọng số không âm. Nếu G không có chu trình âm ta thêm vào đồ thị một đỉnh giả s và các cung nối từ đỉnh s tới tất cả các đỉnh còn lại của đồ thị, trọng số của các cung này được đặt bằng 0. Do đỉnh s không có cung đi vào, đồ thị mới tạo thành cũng không có chu trình âm. Thực hiện thuật toán Bellman-Ford trên đồ thị mới với đỉnh xuất phát s để xác định các $h[v] = \delta(s, v)$ là độ dài đường đi ngắn nhất từ s tới v tương ứng với trọng số c .

Định lý 1-3

Trọng số $\widehat{w}: E \rightarrow \mathbb{R}$ xác định bởi $\widehat{w}(u, v) = w(u, v) + h[u] - h[v]$ là một hàm trọng số không âm.

Chứng minh

Khi thuật toán Bellman-Ford kết thúc, sẽ không tồn tại một cạnh (u, v) nào mà

$$h(v) > h(u) + w(u, v)$$

hay nói cách khác, với mọi cạnh (u, v) của đồ thị, ta có

$$w(u, v) + h(u) - h(v) \geq 0$$

Các nhận xét kể trên, đặc biệt là Định lý 1-2 và Định lý 1-3, cho ta mô hình cài đặt thuật toán Johnson:

- Thêm vào đồ thị một đỉnh s và các cung trọng số 0 nối từ s tới tất cả các đỉnh khác, dùng thuật toán Bellman-Ford tính các $h[v]$ là độ dài đường đi ngắn nhất từ s tới v . Thời gian thực hiện giải thuật $O(nm)$
- Loại bỏ s và các cung mới thêm vào khỏi đồ thị, gán lại trọng số cạnh $\widehat{w}(u, v) := w(u, v) + h[u] - h[v]$ với mọi cạnh $(u, v) \in E$. Thời gian thực hiện giải thuật $\Theta(m)$

- Lần lượt lấy các đỉnh $v \in V$ làm đỉnh xuất phát, thực hiện thuật toán Dijkstra để tìm đường đi ngắn nhất từ v tới tất cả các đỉnh khác. Thời gian thực hiện giải thuật $O(n^2 \lg n + nm)$ nếu sử dụng Fibonacci Heap

Vậy thuật toán Johnson tìm đường đi ngắn nhất giữa mọi cặp đỉnh có thể thực hiện trong thời gian $O(n^2 \lg n + nm)$. Thuật toán dựa trên hai thuật toán đã biết để tìm đường đi ngắn nhất xuất phát từ một đỉnh, việc cài đặt xin dành cho bạn đọc.

1.4. Một số chú ý

Ở một số chương trình trong bài, đôi khi ta sử dụng ma trận trọng số và đem trọng số $+\infty$ gán cho những cạnh không có trong đồ thị ban đầu, hay khi khởi tạo các nhãn khoảng cách, chúng ta thường gán $d[v] := +\infty$ và cực tiểu hóa dần các nhãn đó. Trên máy tính thì không có khái niệm trừu tượng $+\infty$ nên ta sẽ phải chọn một số dương $maxD$ đủ lớn để thay. Như thế nào là đủ lớn?. Số đó phải đủ lớn hơn tất cả trọng số của các đường đi đơn để cho dù đường đi thật có tồi tệ đến đâu vẫn tốt hơn đường đi trực tiếp theo cạnh tưởng tượng ra đó.

Trong trường hợp đồ thị có cạnh trọng số âm, cần cẩn thận với phép cộng trọng số: Nếu một trong hai hạng tử là $maxD$, ta coi như tổng bằng $maxD$ ($C + \infty = \infty$) và không cần cộng nữa. Lý do thứ nhất là để hạn chế lỗi tràn số khi hằng số $maxD$ trong bài toán cụ thể quá lớn, lý do thứ hai là để không bị tính sai khi cộng $maxD$ với một số âm được kết quả $< maxD$, khi đó rất có thể $d[t] < maxD$ mặc dù không tồn tại đường đi từ s tới t .

Những thuật toán tìm đường đi ngắn nhất bộc lộ rất rõ ưu, nhược điểm trong từng trường hợp cụ thể (Ví dụ như số đỉnh của đồ thị quá lớn làm cho không thể biểu diễn bằng ma trận trọng số thì thuật toán Floyd sẽ gặp khó khăn, hay thuật toán Ford-Bellman làm việc khá chậm). Vì vậy cần phải hiểu bản chất và thành thạo trong việc cài đặt tất cả các thuật toán trên để có thể sử dụng chúng một cách uyển chuyển trong từng bài toán thực tế.

Bài tập

- 2.1. Cho đồ thị có hướng $G = (V, E)$ không có chu trình âm, hãy tìm thuật toán $O(|V||E|)$ để tính tất cả các $\delta^*(u) = \min_{v \in V} \{\delta(u, v)\}$
- 2.2. Cho đồ thị có hướng $G = (V, E)$ gồm n đỉnh và m cung, hãy tìm thuật toán $O(nm)$ để xác định đồ thị có chu trình âm hay không và chỉ ra một chu trình âm nếu có.

Gợi ý: Thực hiện thuật toán Bellman-Ford (tối đa $n - 1$ lần quét danh sách cạnh và thực hiện phép co), sau đó ta quét lại danh sách cạnh xem có thể thực hiện được phép co nào nữa hay không. Nếu còn có thể co theo cạnh (u, v) nào đó, ta kết luận đồ thị có chu trình âm và thực hiện phép co này, sau đó lần ngược vết đường đi từ đỉnh v , nếu quá trình lần vết đi lặp lại một đỉnh x nào đó, ta có một chu trình bắt đầu và kết thúc ở đỉnh x .

- 2.3. Hệ ràng buộc: Cho x_1, x_2, \dots, x_n là các biến số, cho m ràng buộc, mỗi ràng buộc có dạng:

$$x_j - x_i \leq w_{ij}, (w_{ij} \in \mathbb{R})$$

Vấn đề đặt ra là hãy tìm cách gán giá trị cho các biến x_1, x_2, \dots, x_n thỏa mãn tất cả các ràng buộc đã cho.

Gợi ý: Có nhiều ví dụ về hệ ràng buộc trên thực tế, chẳng hạn một công trình xây dựng có n công đoạn. Vì lý do kỹ thuật, một công đoạn x_j không được bắt đầu muộn hơn w_{ij} thời gian so với công đoạn x_i . Ràng buộc này có thể viết dưới dạng $x_j - x_i \leq w_{ij}$. Một dạng ràng buộc khác là công đoạn x_j phải bắt đầu sau khi công đoạn x_i bắt đầu được ít nhất w_{ij} thời gian, ràng buộc này cũng có thể viết dưới dạng $x_j - x_i \geq w_{ij}$ hay $x_i - x_j \leq -w_{ij}$.

Coi mỗi biến là một đỉnh của đồ thị, mỗi ràng buộc $x_j - x_i \leq w_{ij}$ cho tương ứng với một cạnh (x_i, x_j) có trọng số w_{ij} . Khi đó nếu đồ thị có chu trình âm thì không tồn tại giải pháp. Thật vậy, giả sử tồn tại chu trình âm, không giảm tính tổng quát, giả sử chu trình âm đó là $C = \langle x_1, x_2, \dots, x_k, x_1 \rangle$, ta có

$$x_2 - x_1 \leq w_{12}$$

$$x_3 - x_2 \leq w_{23}$$

...

$$x_1 - x_k \leq w_{k1}$$

Tổng về trái của các bất đẳng thức trên bằng 0 và tổng về phải chính là trọng số của chu trình (âm), bất đẳng thức $0 \leq w(c) < 0$ không thể được thỏa mãn.

Nếu đồ thị không có chu trình âm, ta thêm vào một đỉnh s và các cung nối trọng số 0 từ s tới mọi đỉnh khác. Dùng thuật toán Bellman-Ford để tìm đường đi ngắn nhất từ s , khi đó các nhãn $d[x_i] = \delta(s, x_i)$ chính là một cách gán giá trị thỏa mãn tất cả các ràng buộc. Hơn nữa cách này còn làm khoảng giá trị gán cho các biến là hẹp nhất:

$$\max_{1 \leq i \leq n} \{x_i\} - \min_{1 \leq j \leq n} \{x_j\} \rightarrow \min$$

- 2.4.** (Cải tiến của Yen cho thuật toán Bellman-Ford) Với đồ thị có hướng $G = (V, E)$ gồm n đỉnh, ta đánh số các đỉnh từ 1 tới n , chia tập cạnh E làm hai tập con: E_1 gồm các cung nối từ đỉnh có chỉ số nhỏ tới đỉnh có chỉ số lớn và E_2 gồm các cung nối từ đỉnh có chỉ số lớn tới đỉnh có chỉ số nhỏ. Đặt $G_1 = (V, E_1)$ và $G_2 = (V, E_2)$, hai đồ thị này là đồ thị có hướng không có chu trình được biểu diễn bằng danh sách kề.

Thuật toán Bellman-Ford sau đó được thực hiện như sau:

```

Init;
repeat
    Stop := True;
    for u := 1 to n - 1 do
        for v: (u, v) ∈ E1 do
            if Relax(u, v) then Stop := False;
    for u := n downto 2 do
        for v: (u, v) ∈ E2 do
            if Relax(u, v) then Stop := False;
until Stop;

```

Bên trong vòng lặp repeat...until là hai pha tối ưu nhãn: pha thứ nhất xét các đỉnh theo thứ tự tăng dần còn pha thứ hai xét các đỉnh theo thứ tự giảm dần của chỉ số. Mỗi khi một đỉnh được xét và thực hiện phép co theo tất cả

các cung đi ra khỏi u , mỗi pha thực hiện tương tự như thuật toán tìm đường đi ngắn nhất trên đồ thị không có chu trình.

Chỉ ra rằng vòng lặp repeat...until trong cài tiến của Yen lặp không quá $[n/2]$ lần. Cài đặt thuật toán và so sánh với cách cài đặt chuẩn của thuật toán Bellman-Ford.

- 2.5.** Arbitrage là một cách sử dụng sự bất hợp lý trong hối đoái tiền tệ để kiếm lời. Ví dụ nếu 1\$ mua được 0.7£, 1£ mua được 190¥, 1¥ mua được 0.009\$ thì từ 1\$, ta có thể đổi sang 0.7£, sau đó sang $0.7 \times 190 = 133$ ¥, rồi đổi lại sang $133 \times 0.009 = 1.197$ \$. Kiếm được 0.197\$ lãi.

Giả sử rằng có n loại tiền tệ đánh số từ 1 tới n . Bảng $R = \{r_{ij}\}_{n \times n}$ cho biết tỉ lệ hối đoái: một đơn vị tiền i đổi được r_{ij} đơn vị tiền j . Hãy tìm thuật toán để xác định xem có thể kiếm lời từ bảng tỉ giá hối đoái này bằng phương pháp arbitrage hay không? Nếu có thể sử dụng arbitrage, hãy chỉ ra một cách kiếm lời.

- 2.6.** (Thuật toán Karp tìm chu trình có trung bình trọng số nhỏ nhất) Cho đồ thị có hướng $G = (V, E)$ gồm n đỉnh, hàm trọng số $w: E \rightarrow \mathbb{R}$. Ta định nghĩa trung bình trọng số của một chu trình C gồm các cạnh $\langle e_1, e_2, \dots, e_k \rangle$ là:

$$\mu(C) = \frac{1}{k} \sum_{i=1}^k w(e_i)$$

Đặt

$$\mu^* = \min_C \{\mu(C)\}$$

Khi đó chu trình C có $\mu(C) = \mu^*$ gọi là chu trình có trung bình trọng số nhỏ nhất (*minimum mean-weight cycle*). Chu trình có trung bình trọng số nhỏ nhất có nhiều ý nghĩa trong các thuật toán tìm luồng với chi phí cực tiểu.

Không giảm tính tổng quát, giả sử mọi đỉnh $v \in V$ đều đến được từ một đỉnh $s \in V$ (Ta có thể thêm một đỉnh giả s và cung trọng số 0 nối từ s tới mọi đỉnh khác, s không nằm trên chu trình đơn nào nên không ảnh hưởng tới tính đúng đắn của thuật toán). Đặt $\delta(v)$ là độ dài đường đi ngắn nhất từ s tới v . Đặt $\delta_k(v)$ là độ dài đường đi ngắn nhất trong số các đường đi từ s

tới v qua đúng k cạnh (ta có thể thêm vào các cung trọng số đủ lớn để với mọi cặp đỉnh u, v luôn tồn tại cung (u, v) và (v, u) , việc tìm chu trình trung bình trọng số nhỏ nhất không bị ảnh hưởng bởi những cung thêm vào và $\delta_k(v)$ luôn là giá trị hữu hạn).

a) Chứng minh rằng nếu $\mu^* = 0$, đồ thị G không có chu trình âm và:

$$\delta(v) = \min_{0 \leq k \leq n-1} \{\delta_k(v)\}, \forall v \in V$$

b) Chứng minh rằng nếu $\mu^* = 0$ thì

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(v) - \delta_k(v)}{n - k} \geq 0, \forall v \in V$$

(Gợi ý: Sử dụng kết quả câu a)

c) Gọi C là một chu trình trọng số 0, u, v là hai đỉnh nằm trên C , giả sử $\mu^* = 0$ và x là độ dài đường đi từ u tới v dọc theo chu trình C . Chứng minh rằng

$$\delta(v) = \delta(u) + x$$

d) Chứng minh rằng nếu $\mu^* = 0$ thì trên mỗi chu trình trọng số 0 sẽ tồn tại một đỉnh v sao cho:

$$\max_{0 \leq k \leq n-1} \frac{\delta_n(v) - \delta_k(v)}{n - k} = 0$$

e) Chứng minh rằng nếu $\mu^* = 0$ thì

$$\min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(v) - \delta_k(v)}{n - k} = 0$$

f) Chỉ ra rằng nếu chúng ta cộng thêm một hằng số Δ vào tất cả các trọng số cạnh thì μ^* tăng lên Δ . Sử dụng tính chất này để chứng minh rằng

$$\mu^* = \min_{v \in V} \max_{0 \leq k \leq n-1} \frac{\delta_n(v) - \delta_k(v)}{n - k}$$

g) Tìm thuật toán $O(|V||E|)$ và lập chương trình để tính μ^* và chỉ ra một chu trình có trung bình trọng số nhỏ nhất.

- 2.7. Trên mặt phẳng cho n đường tròn, đường tròn thứ i được cho bởi bộ ba số thực (x_i, y_i, r_i) , (x_i, y_i) là toạ độ tâm và r_i là bán kính. Chi phí di chuyển trên mỗi đường tròn bằng 0. Chi phí di chuyển giữa hai đường tròn bằng

khoảng cách giữa chúng. Hãy tìm phương án di chuyển giữa hai đường tròn s, t cho trước với chi phí ít nhất.

- 2.8.** Thuật toán Dijkstra có thể sai nếu đồ thị có cạnh trọng số âm, hãy chỉ ra một ví dụ.
- 2.9.** Cho đồ thị vô hướng $G = (V, E)$ có n đỉnh và m cạnh, các cạnh có trọng số là số nguyên trong phạm vi từ 0 tới k . Hãy thay đổi thuật toán Dijkstra để được thuật toán $O(kn + m)$ tìm đường đi ngắn nhất xuất phát từ một đỉnh $s \in V$.

Gợi ý: Để ý rằng nếu một nhãn $d[v] < +\infty$ thì nhãn này phải là số nguyên nằm trong khoảng $[0 \dots (n - 1) \times k]$, đồng thời nếu xét nhãn khoảng cách của các đỉnh lấy ra khỏi hàng đợi ưu tiên thì các nhãn khoảng cách này được sắp xếp theo thứ tự không giảm. Ta tổ chức hàng đợi ưu tiên dưới dạng bảng băm: $q[0 \dots (n - 1) \times k]$ trong đó $q[x]$ là chốt của một danh sách mốc nối chứa các đỉnh v mà $d[v] = x$. Khi đó các phép chèn, cập nhật trên hàng đợi ưu tiên chỉ mất thời gian $O(1)$. Phép lấy ra một phần tử trong hàng đợi ưu tiên tính tổng thể mất thời gian $O(kn)$.

- 2.10.** Tương tự như Bài tập 2.9 nhưng hãy tìm một thuật toán $O((m + n) \log k)$

Gợi ý: Để ý rằng tại mỗi bước của thuật toán Dijkstra, có tối đa $k + 2$ giá trị khác nhau của các nhãn $d[v]$ trong hàng đợi ưu tiên. Mỗi giá trị x sẽ cho tương ứng với một danh sách mốc các nút v mà $d[v] = x$, các chốt của danh sách mốc nối được lưu trữ trong một Binary Heap, khi đó các phép đẩy vào, lấy ra, co nhãn khoảng cách được thực hiện trong thời gian $O(\log k)$.

- 2.11.** (Thuật toán Gabow) Xét đồ thị $G = (V, E)$ có các trọng số cạnh là số tự nhiên: $w: E \rightarrow \mathbb{N}$. Giả sử rằng từ đỉnh xuất phát s có đường đi tới mọi đỉnh khác. Gọi k là trọng số lớn nhất của các cạnh trong E . Gọi $z = \lceil \lg(k + 1) \rceil$, khi đó trọng số mỗi cạnh có thể được biểu diễn bằng một dãy z bit. Với mỗi cạnh $e \in E$ mang trọng số $w(e)$, ta ký hiệu $w_i(e)$ là số tạo thành bằng i bit đầu tiên của $w(e)$, tức là:

$$w_i(e) = w(e) \text{ div } 2^{z-i}, (\forall i = 1, 2, \dots, z).$$

Ví dụ $z = 5$ và $w(e) = 11 = 01011_{(2)}$. Ta có:

$$\begin{aligned}
w_1(e) &= 0_{(2)} = 0 \\
w_2(e) &= 01_{(2)} = 1 \\
w_3(e) &= 010_{(2)} = 2 \\
w_4(e) &= 0101_{(2)} = 5 \\
w_5(e) &= 01011_{(2)} = 11
\end{aligned}$$

Định nghĩa $\delta_i(s, v)$ là độ dài đường đi ngắn nhất từ s tới v trên đồ thị G với hàm trọng số w_i . Rõ ràng $w_z(e) = w(e)$ nên $\delta_z(s, v) = \delta(s, v)$ với $\forall v \in V$.

- a) Giả sử rằng $\delta(s, v) \leq |E|$ với mọi đỉnh v có đường đi từ s , tìm thuật toán $O(|E|)$ để xác định tất cả các $\delta(s, v)$. (Gợi ý: Sử dụng hàng đợi ưu tiên như trong **Error! Reference source not found.**).
- b) Chứng minh rằng các $\delta_1(s, v)$ có thể tính được trong thời gian $O(|E|)$. (Gợi ý: Chú ý rằng các trọng số $w_1(e) \in \{0, 1\}$).
- c) Chỉ ra rằng với mọi $i = 2, 3, \dots, z$: $w_i(e) = 2 \cdot w_{i-1}(e)$ hoặc $w_i(e) = 2 \cdot w_{i-1}(e) + 1$. Từ đó chứng minh rằng:

$$2 \cdot \delta_{i-1}(s, v) \leq \delta_i(s, v) < 2 \cdot \delta_{i-1}(s, v) + |V|$$

(Gợi ý: Độ dài đường đi ngắn nhất từ s tới v sẽ nhân đôi nếu ta nhân đôi các trọng số cạnh)

- d) Với mọi cạnh $e = (u, v) \in E$, định nghĩa:

$$\widehat{w}_i(e) = \widehat{w}_i(u, v) = w_i(u, v) + 2 \cdot \delta_{i-1}(s, u) - 2 \cdot \delta_{i-1}(s, v)$$

Chứng minh rằng với mọi đường đi $p: u \rightsquigarrow v$, ta có:

$$\widehat{w}_i(p) = w_i(p) + 2 \cdot \delta_{i-1}(s, u) - 2 \cdot \delta_{i-1}(s, v)$$

- e) Định nghĩa $\hat{\delta}_i(s, v)$ là độ dài đường đi ngắn nhất từ s tới v trên đồ thị G với hàm trọng số \widehat{w}_i . Chứng minh rằng với $i = 2, 3, \dots, z$ và $\forall v \in V$:

$$\hat{\delta}_i(s, v) = \delta_i(s, v) - 2 \cdot \delta_{i-1}(s, v) \leq |E|$$

- f) Tìm thuật toán tính các $\delta_i(s, v)$ từ các $\delta_{i-1}(s, v)$ trong thời gian $O(|E|)$. Từ đó chứng minh rằng có thể tìm đường đi ngắn nhất trên đồ thị G trong thời gian $O(|E| \cdot z) = O(|E| \lg k)$.

- 2.12.** Cho một bảng các số tự nhiên kích thước $m \times n$. Từ một ô có thể di chuyển sang một ô kề cạnh với nó. Hãy tìm một cách đi từ ô (x, y) ra một ô biên sao cho tổng các số ghi trên các ô đi qua là nhỏ nhất.
- 2.13.** Cho một dãy số nguyên $A = (a_1, a_2, \dots, a_n)$. Hãy tìm một dãy con gồm nhiều nhất các phần tử của dãy đã cho mà tổng của hai phần tử liên tiếp là số nguyên tố.
- 2.14.** Một công trình lớn được chia làm n công đoạn. Công đoạn i phải thực hiện mất thời gian t_i . Quan hệ giữa các công đoạn được cho bởi bảng $A = \{a_{ij}\}_{n \times n}$ trong đó $a_{ij} = 1$ nếu công đoạn j chỉ được bắt đầu khi mà công đoạn i đã hoàn thành và $a_{ij} = 0$ trong trường hợp ngược lại. Mỗi công đoạn khi bắt đầu cần thực hiện liên tục cho tới khi hoàn thành, hai công đoạn độc lập nhau có thể tiến hành song song, hãy bố trí lịch thực hiện các công đoạn sao cho thời gian hoàn thành cả công trình là sớm nhất, cho biết thời gian sớm nhất đó.

Gợi ý: Dựng đồ thị có hướng $G = (V, E)$, mỗi đỉnh tương ứng với một công đoạn, đỉnh u có cung nối tới đỉnh v nếu công đoạn u phải hoàn thành trước khi công đoạn v bắt đầu. Thêm vào G một đỉnh s và cung nối từ s tới tất cả các đỉnh còn lại. Gán trọng số mỗi cung (u, v) của đồ thị bằng t_v .

Nếu đồ thị có chu trình, không thể có cách xếp lịch, nếu đồ thị không có chu trình (DAG) tìm đường đi dài nhất xuất phát từ s tới tất cả các đỉnh của đồ thị, khi đó nhãn khoảng cách $d[v]$ chính là thời điểm hoàn thành công đoạn v , ta chỉ cần xếp lịch để công đoạn v được bắt đầu vào thời điểm $d[v] - t_v$ là xong.

- 2.15.** Cho đồ thị $G = (V, E)$, các cạnh được gán trọng số không âm. Tìm thuật toán và viết chương trình tìm một chu trình có độ dài ngắn nhất trên G .

2. Cây khung nhỏ nhất

Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số. Với một cây khung T của G , ta gọi trọng số của cây T , ký hiệu $w(T)$, là tổng trọng số các cạnh trong T . Bài toán đặt ra là trong số các cây khung của G , chỉ ra cây khung có trọng số nhỏ nhất, cây khung như vậy được gọi là cây khung nhỏ nhất (*minimum*

spanning tree) của đồ thị. Sau đây ta sẽ xét hai thuật toán thông dụng để giải bài toán cây khung nhỏ nhất của đơn đồ thị vô hướng có trọng số, cả hai thuật toán này đều là thuật toán tham lam.

2.1. Phương pháp chung

Xét đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Cả hai thuật toán để tìm cây khung ngắn nhất đều dựa trên một cách làm chung: Nở dần cây khung. Cách làm này được mô tả như sau: Thuật toán quản lý một tập các cạnh $A \subseteq E$ và cố gắng duy trì tính chất sau (tính bát biến vòng lặp):

***A* luôn nằm trong tập cạnh của một cây khung nhỏ nhất.**

Tại mỗi bước lặp, thuật toán tìm một cạnh (u, v) để thêm vào tập A sao cho tính bát biến vòng lặp được duy trì, tức là $A \cup (u, v)$ phải nằm trong tập cạnh của một cây khung nhỏ nhất. Ta nói những cạnh (u, v) như vậy là *an toàn* (*safe*) đối với tập A .

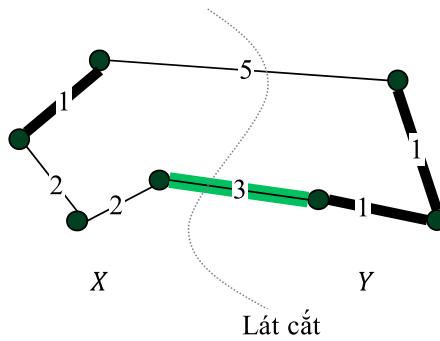
```

procedure FindMST; //Tìm cây khung ngắn nhất
begin
    A := ∅;
    while «A chưa phải cây khung» do
        begin
            «Tìm cạnh an toàn (u, v) đối với A»;
            A := A ∪ { (u, v) }; //Bổ sung (u, v) vào A
        end;
    end;

```

Vẫn đề còn lại là tìm một thuật toán hiệu quả để tìm cạnh an toàn đối với tập A . Chúng ta cần một số khái niệm để giải thích tính đúng đắn của những thuật toán sau này.

Một lát cắt (*cut*) trên đồ thị là một cách phân hoạch tập đỉnh V thành hai tập rời nhau $X, Y: X \cup Y = V; X \cap Y = \emptyset$. Ta nói một lát cắt $V = X \cup Y$ tương thích với tập A nếu không có cạnh nào của A nối giữa một đỉnh thuộc X và một đỉnh thuộc Y . Trong những cạnh nối X với Y , ta gọi những cạnh có trọng số nhỏ nhất là những cạnh nhẹ (*light edge*) của lát cắt $V = X \cup Y$.



Hình 2.3. Lát cắt và cạnh nhẹ

Định lý 2-1

Cho đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Gọi A là một tập con của tập cạnh của một cây khung nhỏ nhất và $V = X \cup Y$ là một lát cắt tương thích với A . Khi đó mỗi cạnh nhẹ của lát cắt $V = X \cup Y$ đều là cạnh an toàn đối với A .

Chứng minh

Gọi (u, v) là một cạnh nhẹ của lát cắt $V = X \cup Y$, gọi T là cây khung nhỏ nhất chứa tất cả các cạnh của A . Nếu T chứa cạnh (u, v) , ta có điều phải chứng minh. Nếu T không chứa cạnh (u, v) , ta thêm cạnh (u, v) vào T sẽ được một chu trình, trên chu trình này có đỉnh thuộc X và cũng có đỉnh thuộc Y , vì vậy sẽ phải có ít nhất hai cạnh trên chu trình nối X với Y . Ngoài cạnh (u, v) nối X với Y , ta gọi (u', v') là một cạnh khác nối X với Y trên chu trình, theo giả thiết (u, v) là cạnh nhẹ nên $w(u, v) \leq w(u', v')$. Ngoài ra do lát cắt $V = X \cup Y$ tương thích với A nên $(u', v') \notin A$.

Cắt bỏ cạnh (u', v') khỏi cây T , cây sẽ bị tách rời làm hai thành phần liên thông, sau đó thêm cạnh (u, v) vào cây nối lại hai thành phần liên thông đó để được cây T' . Ta có

$$\begin{aligned} w(T') &= w(T) - w(u', v') + w(u, v) \\ &\leq w(T) \end{aligned}$$

Do T là cây khung nhỏ nhất, T' cũng phải là cây khung nhỏ nhất. Ngoài ra cây T' chứa cạnh (u, v) và tất cả các cạnh của A . Ta có điều phải chứng minh.

Hệ quả

Cho đồ thị vô hướng liên thông có trọng số $G = (V, E, w)$. Gọi A là một tập con của tập cạnh của một cây khung nhỏ nhất. Gọi C là tập các đỉnh của một thành phần liên thông trên đồ thị $G_A = (V, A)$. Khi đó nếu $(u, v) \in E$ là cạnh trọng số

nhỏ nhất nối từ C tới một thành phần liên thông khác thì (u, v) là cạnh an toàn đối với A .

Chứng minh

Xét lát cắt $V = C \cup (V - C)$, lát cắt này tương thích với A và cạnh (u, v) là cạnh nhẹ của lát cắt này. Theo Định lý 3-17, (u, v) an toàn đối với A .

Chúng ta sẽ trình bày hai thuật toán tìm cây khung nhỏ nhất trên đơn đồ thị vô hướng và cài đặt chương trình với khuôn dạng Input/Output như sau:

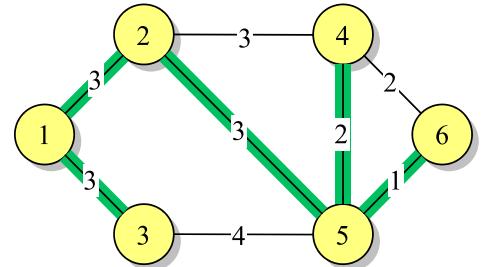
Input

- Dòng 1 chứa số đỉnh $n \leq 1000$ và số cạnh m của đồ thị
- m dòng tiếp theo, mỗi dòng chứa chỉ số hai đỉnh đầu mút và trọng số của một cạnh. Trọng số cạnh là số nguyên có giá trị tuyệt đối không quá 1000.

Output

Cây khung nhỏ nhất của đồ thị

Sample Input	Sample Output
6 8	Minimum Spanning Tree:
1 2 3	(5, 6) = 1
1 3 3	(4, 5) = 2
2 4 3	(1, 2) = 3
2 5 3	(2, 5) = 3
3 5 4	(1, 3) = 3
4 5 2	Weight = 12
4 6 2	
5 6 1	



2.2. Thuật toán Kruskal

Thuật toán Kruskal [27] dựa trên mô hình xây dựng cây khung bằng thuật toán hợp nhất, chỉ có điều thuật toán không phải xét các cạnh với thứ tự tùy ý mà xét các cạnh theo thứ tự đã sắp xếp: Để tìm cây khung ngắn nhất của đồ thị $G = (V, E, w)$, thuật toán khởi tạo cây T ban đầu không có cạnh nào. Duyệt danh sách cạnh của đồ thị từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn, mỗi khi xét tới một cạnh và việc thêm cạnh đó vào T không tạo thành chu trình đơn trong T thì kết nạp thêm cạnh đó vào T ... Cứ làm như vậy cho tới khi:

- Hoặc đã kết nạp được $|V| - 1$ cạnh vào trong T thì ta được T là cây khung nhỏ nhất
- Hoặc khi duyệt hết danh sách cạnh mà vẫn chưa kết nạp đủ $|V| - 1$ cạnh. Trong trường hợp này đồ thị G là không liên thông, việc tìm kiếm cây khung thất bại.

Như vậy cần làm rõ hai thao tác sau khi cài đặt thuật toán Kruskal:

- Làm thế nào để xét được các cạnh từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn.
- Làm thế nào kiểm tra xem việc thêm một cạnh có tạo thành chu trình đơn trong T hay không.

a) Duyệt danh sách cạnh

Vì các cạnh của đồ thị phải được xét từ cạnh có trọng số nhỏ tới cạnh có trọng số lớn. Ta có thể thực hiện một thuật toán sắp xếp danh sách cạnh rồi sau đó duyệt lại danh sách đã sắp xếp. Tuy nhiên khi cài đặt cụ thể, ta có thể khéo léo lồng thuật toán Kruskal vào QuickSort hoặc HeapSort để đạt hiệu quả cao hơn. Chẳng hạn với QuickSort, ý tưởng là sau khi phân đoạn danh sách cạnh bằng một cạnh chốt *Pivot*, ta được ba phân đoạn: Đoạn đầu gồm các cạnh có trọng số $\leq w(Pivot)$, tiếp theo là cạnh *Pivot*, đoạn sau gồm các cạnh có trọng số $\geq w(Pivot)$. Ta gọi đệ quy để sắp xếp và xử lý các cạnh thuộc đoạn đầu, tiếp theo xử lý cạnh *Pivot*, cuối cùng lại gọi đệ quy để sắp xếp và xử lý các cạnh thuộc đoạn sau. Để thấy rằng thứ tự xử lý các cạnh như vậy đúng theo thứ tự tăng dần của trọng số. Ngoài ra khi thấy đã có đủ $n - 1$ cạnh được kết nạp vào cây khung, ta có thể ngưng ngay QuickSort mà không cần xử lý tiếp nữa.

b) Kết nạp cạnh và hợp cây

Trong quá trình xây dựng cây khung, các cạnh trong T ở các bước sẽ tạo thành một rừng (đồ thị không có chu trình đơn), mỗi thành phần liên thông của rừng này là một cây khung. Muốn thêm một cạnh (u, v) vào T mà không tạo thành chu trình đơn thì (u, v) phải nối hai cây khác nhau của rừng T . Điều này làm chúng ta nghĩ đến cấu trúc dữ liệu biểu diễn các tập rời nhau: Ban đầu ta khởi tạo n tập S_1, S_2, \dots, S_n , mỗi tập chứa đúng một đỉnh của đồ thị. Khi xét tới cạnh

(u, v) , nếu u và v thuộc hai tập khác nhau S_u, S_v thì ta hợp nhất S_u, S_v lại thành một tập.

Vậy có hai thao tác cần phải cài đặt hiệu quả trong thuật toán Kruskal: phép kiểm tra hai đỉnh có thuộc hai tập khác nhau hay không và phép hợp nhất hai tập. Một trong những cấu trúc dữ liệu hiệu quả để cài đặt những thao tác này là *rừng các tập rời nhau* (*disjoint-set forest*). Cấu trúc dữ liệu này được cài đặt như sau:

Mỗi tập $S[.]$ được biểu diễn bởi một cây, trong đó mỗi đỉnh trong tập tương ứng với một nút trên cây. Cây được biểu diễn bởi mảng con trỏ tới nút cha: $lab[v]$ là nút cha của nút v . Trong trường hợp v là nút gốc của cây, ta đặt:

$$lab[v] := -\text{hạng của cây}$$

Hạng (*rank*) của một cây là một số nguyên không nhỏ hơn độ cao của cây. Ban đầu mỗi tập $S[.]$ chỉ gồm một đỉnh, nên họ các tập S_1, S_2, \dots, S_n được khởi tạo với các nhãn $lab[v] := 0, \forall v \in V$ tương ứng với một rừng gồm n cây độ cao 0.

Để xác định hai đỉnh u, v có thuộc 2 tập khác nhau hay không, ta chỉ cần xác định xem gốc của cây chứa u và gốc của cây chứa v có khác nhau hay không. Việc xác định gốc của cây chứa u được thực hiện bởi hàm *FindSet(u)*: Đi từ u lên nút cha, đến khi gặp nút gốc (nút r có $lab[r] \leq 0$) thì dừng lại. Đi kèm với hàm *FindSet(u)* là phép nén đường (*path compression*): Dọc trên đường đi từ u tới nút gốc r , đi qua đỉnh nào ta cho luôn đỉnh đó làm con của r :

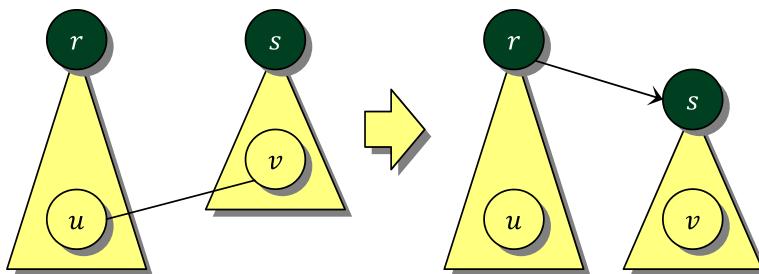
```
function FindSet(u: Integer): Integer;
//Xác định gốc cây chứa đỉnh u
begin
  if lab[u] <= 0 then Result := u
  //u là gốc của một tập S[.] nào đó, trả về chính u
  else //u không phải gốc
    begin
      Result := FindSet(lab[u]); //Gọi đệ quy tìm gốc
      lab[u] := Result; //Nén đường, cho u làm con của nút gốc luôn
    end;
end;
```

Việc hợp nhất hai tập tức là xây dựng cây mới chứa tất cả các phần tử trong hai cây ban đầu. Giả sử r và s là gốc của hai cây tương ứng với hai tập cần hợp nhất. Khi đó:

- Nếu cây gốc r có hạng cao hơn cây gốc s , ta cho s làm con của r , hạng của cây gốc r không thay đổi, tương tự cho trường hợp cây gốc r thấp hơn cây gốc s .
- Nếu hai cây ban đầu có cùng hạng, ta cho cây gốc r làm con của gốc s khi đó cây gốc s có thể sẽ bị tăng độ cao, do vậy để hợp lý hóa ta tăng hạng của s lên 1, tương đương với việc giảm $lab[s]$ đi 1.

```
procedure Union(r, s: Integer); //Hợp nhất hai tập r và s
begin
  if lab[r] < lab[s] then //hạng của r lớn hơn
    lab[s] := r //cho s làm con của r
  else
    begin
      if lab[r] = lab[s] then Dec(lab[s]);
      //Nếu hai tập bằng hạng, tăng hạng của s
      lab[r] := s; //Cho r làm con của s
    end;
end;
```

Hình 2.4 mô tả hai cây biểu diễn hai tập rời nhau, sau khi xét tới một cạnh (u, v) nối giữa hai tập, hai cây được hợp nhất lại bằng cách cho một cây làm cây con của gốc cây kia.



Hình 2.4. Hai tập rời nhau được hợp nhất lại khi xét tới một cạnh nối một đỉnh của tập này với một đỉnh của tập kia

KRUSKAL.PAS ✓ Thuật toán Kruskal

```
{$MODE OBJFPC}
program MinimumSpanningTree;
```

```

const
  maxN = 1000;
  maxM = maxN * (maxN - 1) div 2;

type
  TEdge = record //Cấu trúc cạnh
    x, y: Integer; //Hai đỉnh đầu mút
    w: Integer; //Trọng số
    Selected: Boolean; //Đánh dấu chọn/không chọn vào cây khung
  end;

var
  lab: array[1..maxN] of Integer; //Nhân của disjoint set forest
  e: array[1..maxM] of TEdge; //Danh sách cạnh
  n, m, k: Integer;
procedure Enter; //Nhập dữ liệu
var i: Integer;
begin
  ReadLn(n, m);
  for i := 1 to m do
    with e[i] do
      begin
        ReadLn(x, y, w);
        Selected := False; //Chưa chọn cạnh nào
      end;
  for i := 1 to n do lab[i] := 0;
  //Khởi tạo n tập rời nhau hàng của mỗi tập bằng 0
  k := 0; //Biến đếm số cạnh được kết nạp vào cây khung
end;
function FindSet(u: Integer): Integer; //Xác định tập chứa đỉnh u
begin
  if lab[u] <= 0 then Result := u //u là gốc của một tập S[.] nào đó
  else //u không phải gốc
    begin
      Result := FindSet(lab[u]); //Gọi đệ quy tìm gốc
      lab[u] := Result; //Nén đường
    end;
end;
procedure Union(r, s: Integer); //Hợp nhất hai tập r và s
begin

```

```

if lab[r] < lab[s] then //hạng của r lớn hơn
    lab[s] := r //cho s làm con của r
else
    begin
        if lab[r] = lab[s] then Dec(lab[s]);
        //Nếu hai tập bằng hạng, tăng hạng của s
        lab[r] := s; //Cho r làm con của s
    end;
end;
procedure ProcessEdge (var e: TEdge); //Xử lý một cạnh e
var r, s: Integer;
begin
    with e do
        begin
            r := FindSet(x);
            s := FindSet(y); //Xác định 2 tập tương ứng với 2 đầu mút
            if r <> s then //Hai đầu mút thuộc hai tập khác nhau
                begin
                    Selected := True; //Cạnh e sẽ được chọn vào cây khung nhỏ nhất
                    Inc(k); //Tăng biến đếm số cạnh được kết nạp
                    Union(r, s); //Hợp nhất hai tập thành một
                end;
        end;
    end;
end;
procedure QuickSort (L, H: Integer); //Xử lý danh sách cạnh e[L...H]
var
    i, j: Integer;
    pivot: TEdge;
begin
    //Nếu cây đã có đủ k cạnh hoặc danh sách cạnh rỗng thì thoát luôn
    if (L > H) or (k = n - 1) then Exit;
    //Chú ý L > H, không phải L ≥ H như trong QuickSort
    i := L + Random(H - L + 1);
    pivot := e[i];
    e[i] := e[L];
    i := L;
    j := H;
    repeat

```

```

while (e[j].w > pivot.w) and (i < j) do Dec(j);
if i < j then
    begin
        e[i] := e[j];
        Inc(i);
    end
else Break;
while (e[i].w < pivot.w) and (i < j) do Inc(i);
if i < j then
    begin
        e[j] := e[i];
        Dec(j);
    end
else Break;
until i = j;
QuickSort(L, i - 1);
//Các cạnh e[L...i - 1] có trọng số ≤ Pivot.w, gọi để quy xử lý trước
e[i] := Pivot;
if k < n - 1 then ProcessEdge(e[i]); //Xử lý tiếp cạnh e[i] = Pivot
QuickSort(i + 1, H);
//Các cạnh e[i + 1...H] có trọng số ≥ Pivot.w, gọi để quy xử lý sau
end;
procedure PrintResult;
var
    i, Weight: Integer;
begin
    if k < n - 1 then //Không kết nạp đủ n - 1 cạnh, đồ thị không liên thông
        WriteLn('Graph is not connected!')
    else //In ra cây khung nhỏ nhất
        begin
            WriteLn('Minimum Spanning Tree:');
            Weight := 0;
            for i := 1 to m do
                with e[i] do
                    if Selected then //In ra cách cạnh được đánh dấu chọn
                        begin
                            WriteLn('(' , x, ' , ' , y, ') = ' , w);
                            Inc(Weight, w);
                        end
        end

```

```

end;
    WriteLn ('Weight = ', Weight);
end;
end;
begin
    Enter;
    QuickSort (1, m); //Lồng thuật toán Kruskal vào QuickSort
    PrintResult;
end.

```

Tính đúng đắn của thuật toán Kruskal được suy ra từ Định lý 3-17: Để ý rằng các cạnh được kết nạp vào cây khung sau mỗi bước sẽ tạo thành một rừng (đô thị không có chu trình đơn). Mỗi khi cạnh (u, v) được xét đến, nó sẽ chỉ được kết nạp vào cây khung nếu như u và v thuộc hai cây (hai thành phần liên thông) T_u, T_v khác nhau. Ký hiệu A là tập cạnh của T_u , khi đó lát cắt $V = T_u \cup (V - T_u)$ là tương thích với tập A , (u, v) là cạnh nhẹ của lát cắt nên (u, v) cũng phải là một cạnh trên một cây khung nhỏ nhất.

c) Thời gian thực hiện giải thuật

Với hai số tự nhiên m, n , hàm Ackermann $A(m, n)$ được định nghĩa như sau:

$$A(m, n) = \begin{cases} n + 1, & \text{nếu } m = 0 \\ A(m - 1, 1), & \text{nếu } m > 0 \text{ và } n = 0 \\ A(m - 1, A(m, n - 1)), & \text{nếu } m > 0 \text{ và } n > 0 \end{cases}$$

Dưới đây là bảng một số giá trị hàm Ackermann:

$n:$	0	1	2	3	4
$m:$					
0	1	2	3	4	5
1	2	3	4	5	6
2	3	5	7	9	11
3	5	13	29	61	125
4	13	65533	$2^{65536} - 3$	$2^{2^{65536}} - 3$	$2^{2^{2^{65536}}} - 3$

Hàm $A(m, n)$ là một hàm tăng rất nhanh theo đối số n . Có thể chứng minh được

$$A(0, n) = n + 1$$

$$A(1, n) = n + 2$$

$$A(2, n) = 2n + 3$$

$$A(3, n) = 2^{n+3} - 3$$

$$A(4, n) = \underbrace{2^{2^{\dots^2}}}_{n+3 \text{ lũy thừa}} - 3$$

Chẳng hạn $A(4, 2)$ là một số có 19729 chữ số, $A(4, 4)$ là một số mà số chữ số của nó lớn hơn cả số nguyên tử trong phần vũ trụ mà con người biết đến.

Khi $n > 0$, xét hàm $\alpha(m, n)$, gọi là nghịch đảo của hàm Ackerman, định nghĩa như sau:

$$\alpha(m, n) = \min \left\{ k \geq 1 : A \left(k, \left\lfloor \frac{m}{n} \right\rfloor \right) \geq \lg n \right\}$$

Người ta đã chứng minh được rằng với cấu trúc dữ liệu rỗng các tập rời nhau, việc thực hiện m thao tác *FindSet* và *Union* mất thời gian $O(m\alpha(m, n))$. Ở đây $\alpha(m, n)$ là một hằng số rất nhỏ (trên tất cả các dữ liệu thực tế, không bao giờ $\alpha(m, n)$ vượt quá 4). Điều đó chỉ ra rằng ngoại trừ việc sắp xếp danh sách cạnh, thuật toán Kruskal ở trên có thời gian thực hiện $O(|E|\alpha(|E|, |V|))$.

Tuy nhiên nếu phải thực hiện sắp xếp danh sách cạnh, chúng ta cần cộng thêm thời gian thực hiện giải thuật sắp xếp $O(|E| \lg |E|)$ nữa.

2.3. Thuật toán Prim

a) Tư tưởng của thuật toán

Trong trường hợp đồ thị dày (có nhiều cạnh), có một thuật toán hiệu quả hơn để tìm cây khung ngắn nhất là thuật toán Prim [32]. Với một cây khung T và một đỉnh $v \notin T$, ta gọi khoảng cách từ v tới T , ký hiệu $d[v]$, là trọng số nhỏ nhất của một cạnh nối v với một đỉnh nằm trong T :

$$d[v] = \min_{u \in T} \{w(u, v)\}$$

Tư tưởng của thuật toán có thể trình bày như sau: Ban đầu khởi tạo một cây T chỉ gồm 1 đỉnh bất kỳ của đồ thị, sau đó ta cứ tìm đỉnh gần T nhất (có khoảng cách tới T ngắn nhất) kết nạp vào T và kết nạp luôn cạnh tạo ra khoảng cách gần nhất đó, cứ làm như vậy cho tới khi:

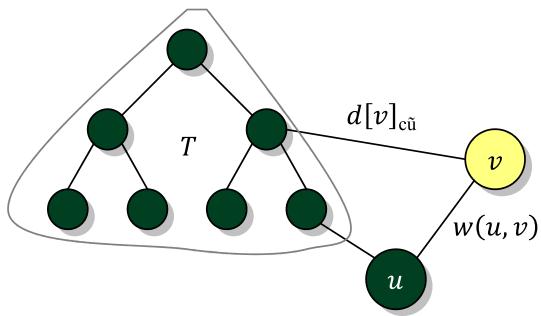
- Hoặc đã kết nạp đủ n đỉnh vào T , ta có một cây khung ngắn nhất.
- Hoặc chưa kết nạp đủ n đỉnh nhưng không còn cạnh nào nối một đỉnh trong T với một đỉnh ngoài T . Ta kết luận đồ thị không liên thông và không thể tồn tại cây khung.

b) Kỹ thuật cài đặt

Khi cài đặt thuật toán Prim, ta sử dụng các nhãn khoảng cách $d[v]$ để lưu khoảng cách từ v tới T tại mỗi bước. Mỗi khi cây T bổ sung thêm một đỉnh u , ta tính lại các nhãn khoảng cách theo công thức sau:

$$d[v]_{\text{mới}} := \min \{d[v]_{\text{cũ}}, w(u, v)\}$$

Tính đúng đắn của công thức có thể hình dung như sau: $d[v]$ là khoảng cách từ v tới cây T , theo định nghĩa là trọng số nhỏ nhất trong số các cạnh nối v với một đỉnh nằm trong T . Khi cây T “nở” ra thêm đỉnh u nữa mà đỉnh u này lại gần v hơn tất cả các đỉnh khác trong T , ta ghi nhận khoảng cách mới $d[v]$ là trọng số cạnh (u, v) , nếu không ta vẫn giữ khoảng cách cũ.



Hình 2.5. Cơ chế cập nhật nhẫn khoảng cách

Tại mỗi bước, đỉnh ngoài cây có nhẫn khoảng cách nhỏ nhất sẽ được kết nạp vào cây, sau đó các nhẫn khoảng cách được cập nhật và lặp lại. Mô hình cài đặt của thuật toán có thể viết như sau:

```

u := «Một đỉnh bất kỳ»;
T := {u};
for  $\forall v \notin T$  do  $d[v] := +\infty$ ;
//Các đỉnh ngoài T được khởi tạo nhẫn khoảng cách  $+\infty$ 
for  $i := 1$  to  $n - 1$  do //Làm  $n - 1$  lần
    begin
        for ( $\forall v \notin T : (u, v) \in E$ ) do
             $d[v] := \min\{d[v], w(u, v)\}$ ;
        //Cập nhật nhẫn khoảng cách của các đỉnh kề u nằm ngoài T
         $u := \arg \min\{d[v] : v \notin T\}$ ;
        //Chọn u là đỉnh có nhẫn khoảng cách nhỏ nhất trong số các đỉnh nằm ngoài T
        if  $d[u] = +\infty$  then //Đò thị không liên thông
            begin
                Output  $\leftarrow$  "Không tồn tại cây khung";
                Break;
            end;
        T := T  $\cup$  {u}; //Bổ sung u vào T
    end;
    Output  $\leftarrow$  T;

```

Cài đặt dưới đây sử dụng ma trận trọng số để biểu diễn đồ thị. Kỹ thuật đánh dấu được sử dụng để biết một đỉnh v đang nằm ngoài ($outside[v] = True$) hay nằm trong cây T ($outside[v] = False$). Ngoài ra để tiện lợi hơn trong việc chỉ ra cây khung nhỏ nhất, với mỗi đỉnh v nằm ngoài T ta lưu lại $trace[v]$ là đỉnh u nằm trong T mà cạnh (u, v) tạo ra khoảng cách gần nhất từ v tới T : $w(u, v) =$

$d[v]$. Khi thuật toán kết thúc, các cạnh trong cây khung là những cạnh ($trace[v], v$).

PRIM.PAS ✓ Thuật toán Prim

```
{$MODE OBJFPC}
program MinimumSpanningTree;
const
  maxN = 1000;
  maxW = 1000;
var
  w: array[1..maxN, 1..maxN] of Integer; //Ma trận trọng số
  d: array[1..maxN] of Integer; //Các nhãn khoảng cách
  outside: array[1..maxN] of Boolean; //Đánh dấu các đỉnh ngoài cây
  trace: array[1..maxN] of Integer; //Vết
  n: Integer;
procedure Enter;
var
  i, m, u, v: Integer;
begin
  ReadLn(n, m);
  for u := 1 to n do
    for v := 1 to n do w[u, v] := maxW + 1;
    //Khởi tạo ma trận trọng số với các phần tử +∞
  for i := 1 to m do
    begin
      ReadLn(u, v, w[u, v]);
      //Chú ý: Đơn đồ thị mới có thể đọc dữ liệu thẻ này
      w[v, u] := w[u, v]; //Đồ thị vô hướng
    end;
  end;
function Prim: Boolean; //Thuật toán Prim, trả về True nếu tìm được cây khung
var
  u, v, dmin, i: Integer;
begin
  u := 1; //Cây ban đầu chỉ gồm đỉnh 1
  for v := 2 to n do d[v] := maxW + 1;
  //Nhãn khoảng cách cho các đỉnh ngoài cây khởi tạo bằng +∞
  FillChar(outside[2],
```

```

        (n - 1) * SizeOf(outside[2]),
        True); //Đánh dấu các đỉnh 2...n nằm ngoài cây
outside[1] := False; //Đỉnh 1 nằm trong cây
for i := 1 to n - 1 do //Làm n - 1 lần
begin
    //Trước hết tính lại các nhãn khoảng cách
    for v := 1 to n do
        if outside[v] and (d[v] > w[u, v]) then
            //Cạnh (u, v) tạo khoảng cách ngắn hơn khoảng cách cũ
            begin
                d[v] := w[u, v]; //Cập nhật nhãn khoảng cách
                trace[v] := u; //Lưu vết
            end;
    //Tim đỉnh u ngoài cây có nhãn khoảng cách nhỏ nhất
    dmin := maxW + 1;
    u := 0;
    for v := 1 to n do
        if outside[v] and (d[v] < dmin) then
            begin
                dmin := d[v];
                u := v;
            end;
        if u = 0 then Exit(False);
        //Cây không có cạnh nào nối ra ngoài, đồ thị không liên thông, thoát
        outside[u] := False; //Kết nạp u vào cây
    end;
    Result := True;
end;
procedure PrintResult; //In kết quả trong trường hợp tìm ra cây khung nhỏ nhất
var
    v, Weight: Integer;
begin
    WriteLn('Minimum Spanning Tree:');
    Weight := 0;
    for v := 2 to n do
        begin
            WriteLn('(' , trace[v], ', ', v, ') = ',
                    w[trace[v], v]);
            Inc(Weight, w[trace[v], v]);
        end;

```

```

end;
WriteLn('Weight = ', Weight);
end;
begin
Enter;
if Prim then PrintResult
else WriteLn('Graph is not connected! ');
end.

```

Tính đúng đắn của thuật toán Prim cũng dễ dàng suy ra được từ Định lý 3-17: Gọi A là tập cạnh của cây T tại mỗi bước, xét lát cắt tách tập đỉnh V làm 2 tập rời nhau, một tập gồm các đỉnh $\in T$ và tập còn lại gồm các đỉnh $\notin T$. Đỉnh v được kết nạp vào cây T tại mỗi bước tương ứng với cạnh $(Trace[v], v)$ là cạnh nhẹ của lát cắt nên nó là an toàn với tập A , việc bổ sung cạnh này vào A vẫn đảm bảo A là tập con của tập cạnh một cây khung ngắn nhất.

c) Thời gian thực hiện giải thuật

Chương trình cài đặt thuật toán Prim ở trên có thời gian thực hiện $\Theta(n^2)$, hiệu quả hơn thuật toán Kruskal trong trường hợp đồ thị dày nhưng lại kém hơn nếu đồ thị thưa. Trong trường hợp đồ thị thưa, ta có thể cải tiến mô hình cài đặt thuật toán Prim bằng cách kết hợp với một hàng đợi ưu tiên chứa các đỉnh ngoài cây có nhãn khoảng cách $< +\infty$. Hàng đợi ưu tiên cần hỗ trợ các thao tác sau:

- *Extract*: Lấy ra một đỉnh ưu tiên nhất (đỉnh u có $d[u]$ nhỏ nhất) khỏi hàng đợi ưu tiên.
- *Update(v)*: Thao tác này báo cho hàng đợi ưu tiên biết rằng nhãn $d[v]$ đã bị giảm đi, cần tổ chức lại (thêm v vào hàng đợi ưu tiên nếu v đang nằm ngoài).

Khi đó thuật toán Prim có thể viết theo mô hình mới:

```

T := {Một đỉnh bất kỳ u};
for  $\forall v \notin T$  do  $d[v] := +\infty$ ;
//Các đỉnh ngoài T được khởi tạo nhãn khoảng cách  $+\infty$ 
PQ :=  $\emptyset$ ; //Hàng đợi ưu tiên được khởi tạo rỗng
for i := 1 to |V| - 1 do //Làm n - 1 lần
    begin
        for ( $\forall v \in V : (u, v) \in E$ ) do //Cap nhật nhãn các đỉnh ngoài cây kề với u
    
```

```

if (v  $\notin$  T) and (d[v] > w(u, v)) then
    begin
        d[v] := w(u, v);
        Update(v);
    end;
if PQ =  $\emptyset$  then //Đồ thị không liên thông
    begin
        Output  $\leftarrow$  "Không tồn tại cây khung";
        Break;
    end;
    u := Extract;
    //Chọn u là đỉnh có nhãn khoảng cách nhỏ nhất trong số các đỉnh nằm ngoài T
    T := T  $\cup$  {u}; //Bổ sung u vào T
end;

```

Thời gian thực hiện giải thuật có thể ước lượng theo số lời gọi *Update* và *Extract*. Tương tự như thuật toán Dijkstra, có thể thấy rằng chúng ta cần sử dụng không quá $n - 1$ lời gọi *Extract* và không quá m lời gọi *Update* với n là số đỉnh và m là số cạnh của đồ thị.

Một số cấu trúc dữ liệu biểu diễn hàng đợi ưu tiên có thể sử dụng để cải thiện tốc độ thuật toán Prim trong trường hợp đồ thị thừa. Chẳng hạn nếu sử dụng Fibonacci Heap làm hàng đợi ưu tiên, thời gian thực hiện giải thuật là $O(|V| \lg |V| + |E|)$. Mặc dù vậy cần nhấn mạnh rằng trong các ứng dụng thực tế mà danh sách cạnh có thể được sắp xếp trong thời gian $O(|E|)$ (chẳng hạn dùng các thuật toán sắp xếp cơ sở hoặc đếm phân phôi), thuật toán Kruskal luôn là sự lựa chọn hợp lý hơn cả vì nó có thể tìm được cây khun trong thời gian $O(|E|\alpha(|E|, |V|))$.

Bài tập

- 2.16.** Cho T là cây khung nhỏ nhất của đồ thị G và (u, v) là một cạnh trong T .
 Chứng minh rằng nếu ta trù trọng số cạnh (u, v) đi một số dương thì T vẫn là cây khung nhỏ nhất của đồ thị G .

- 2.17.** Cho G là một đồ thị vô hướng liên thông, C là một chu trình trên G và e là cạnh trọng số lớn nhất của C . Chứng minh rằng nếu ta loại bỏ cạnh e khỏi đồ thị thì không ảnh hưởng tới trọng số của cây khung nhỏ nhất.
- 2.18.** Chứng minh rằng đồ thị có duy nhất một cây khung nhỏ nhất nếu với mọi lát cắt của đồ thị, có duy nhất một cạnh nhẹ nối hai tập của lát cắt. Cho một ví dụ để chỉ ra rằng điều ngược lại không đúng.
- 2.19.** Gọi T là một cây khung nhỏ nhất của đồ thị vô hướng liên thông G , ta giảm trọng số của một cạnh không nằm trong cây T , hãy tìm một thuật toán đơn giản để tìm cây khung của đồ thị mới.
- Gợi ý: Gọi cạnh bị giảm trọng số là (u, v) , thêm (u, v) vào T ta sẽ được đúng một chu trình đơn, loại bỏ cạnh trọng số lớn nhất trên chu trình đơn này sẽ được cây khung nhỏ nhất của đồ thị mới.*
- 2.20.** Giả sử rằng đồ thị vô hướng liên thông G có cây khung nhỏ nhất T , người ta thêm vào đồ thị một đỉnh mới và một số cạnh liên thuộc với đỉnh đó. Tìm thuật toán xác định cây khung nhỏ nhất của đồ thị mới.
- 2.21.** Giáo sư X đề xuất một thuật toán tìm cây khung ngắn nhất dựa trên ý tưởng chia để trị: Với đồ thị vô hướng liên thông $G = (V, E)$, phân hoạch tập đỉnh V làm hai tập rời nhau V_1, V_2 mà lực lượng của hai tập này hơn kém nhau không quá 1. Gọi E_1 là tập các cạnh chỉ liên thuộc với các đỉnh $\in V_1$ và E_2 là tập các cạnh chỉ liên thuộc với các đỉnh $\in V_2$. Tìm cây khung nhỏ nhất trên đồ thị $G_1 = (V_1, E_1)$ và $G_2 = (V_2, E_2)$ bằng thuật toán đệ quy, sau đó chọn cạnh trọng số nhỏ nhất nối V_1 với V_2 để nối hai cây khung tìm được thành một cây. Chứng minh tính đúng đắn của thuật toán hoặc chỉ ra một phản ví dụ cho thấy thuật toán sai.
- 2.22.** (Cây khung nhỏ thứ nhì) Cho $G = (V, E, w)$ là đồ thị vô hướng liên thông có trọng số, giả sử rằng $|E| \geq |V|$ và các trọng số cạnh là hoàn toàn phân biệt (w là đơn ánh). Gọi \mathcal{T} là tập tất cả các cây khung của G và A là cây khung nhỏ nhất của G , khi đó cây khung nhỏ thứ nhì được định nghĩa là cây khung $B \in \mathcal{T}$ thỏa mãn:

$$w(B) = \min_{T \in \mathcal{T} - \{A\}} \{w(T)\}$$

- Chỉ ra rằng đồ thị G có duy nhất một cây khung nhỏ nhất là A , nhưng có thể có nhiều cây khung nhỏ thứ nhì.
- Chứng minh rằng luôn tồn tại một cạnh $(u, v) \in A$ và $(x, y) \notin A$ để nếu ta loại bỏ cạnh (u, v) khỏi A rồi thêm cạnh (x, y) vào A thì sẽ được cây khung nhỏ thứ nhì.
- Với $\forall u, v \in V$, gọi $f[u, v]$ là cạnh mang trọng số lớn nhất trên đường đi duy nhất từ u tới v trên cây A . Tìm thuật toán $O(|V|^2)$ để tính tất cả các $f[u, v]$, $\forall u, v \in V$.
- Tìm thuật toán hiệu quả để tìm cây khung nhỏ thứ nhì của đồ thị.

2.23. Cho s và t là hai đỉnh của một đồ thị vô hướng có trọng số $G = (V, E, w)$. Tìm một đường đi từ s tới t thỏa mãn: Trọng số cạnh lớn nhất đi qua trên đường đi là nhỏ nhất có thể.

Gợi ý: Có rất nhiều cách làm: Kết hợp một thuật toán tìm kiếm trên đồ thị với thuật toán tìm kiếm nhị phân, hoặc sửa đổi thuật toán Dijkstra, hoặc sử dụng thuật toán tìm cây khung ngắn nhất.

2.24. (Euclidean Minimum Spanning Tree) Trong trường hợp các đỉnh của đồ thị đầy đủ được đặt trên mặt phẳng trực chuẩn và trọng số cạnh nối giữa hai đỉnh chính là khoảng cách hình học giữa chúng. Người ta có một phép tiền xử lý để giảm bớt số cạnh của đồ thị bằng thuật toán tam giác phân Delaunay ($O(n \lg n)$), đồ thị sau phép tam giác phân Delaunay sẽ còn không quá $3n$ cạnh, do đó sẽ làm các thuật toán tìm cây khung nhỏ nhất hoạt động hiệu quả hơn. Hãy tự tìm hiểu về phép tam giác phân Delaunay và cài đặt chương trình để tìm cây khung nhỏ nhất.

2.25. Trên một nền phẳng với hệ toạ độ trực chuẩn đặt n máy tính, máy tính thứ i được đặt ở toạ độ (x_i, y_i) . Đã có sẵn một số dây cáp mạng nối giữa một số cặp máy tính. Cho phép nối thêm các dây cáp mạng nối giữa từng cặp máy tính. Chi phí nối một dây cáp mạng tỉ lệ thuận với khoảng cách giữa hai máy cần nối. Hãy tìm cách nối thêm các dây cáp mạng để cho các máy tính trong toàn mạng là liên thông và chi phí nối mạng là nhỏ nhất.

2.26. Hệ thống điện trong thành phố được cho bởi n trạm biến thế và các đường dây điện nối giữa các cặp trạm biến thế. Mỗi đường dây điện e có độ an toàn là $p(e) \in (0, 1]$. Độ an toàn của cả lưới điện là tích độ an toàn trên các

đường dây. Hãy tìm cách bỏ đi một số dây điện để cho các trạm biến thế vẫn liên thông và độ an toàn của mạng là lớn nhất có thể.

Gợi ý: Bằng kỹ thuật lấy logarithm, độ an toàn trên lưới điện trở thành tổng độ an toàn trên các đường dây.

3. Luồng cực đại trên mạng

3.1. Các khái niệm và bài toán

a) Mạng

Mạng (*flow network*) là một bộ năm $G = (V, E, c, s, t)$, trong đó:

V và E lần lượt là tập đỉnh và tập cung của một đồ thị có hướng không có khuyên (cung nối từ một đỉnh đến chính nó).

s và t là hai đỉnh phân biệt thuộc V , s gọi là *đỉnh phát* (*source*) và t gọi là *đỉnh thu* (*sink*).

c là một hàm xác định trên tập cung E :

$$\begin{aligned} c: E &\rightarrow [0, +\infty) \\ e &\mapsto c(e) \end{aligned}$$

gán cho mỗi cung $e \in E$ một số không âm gọi là *sức chứa* (*capacity*)^{*} $c(e) \geq 0$.

Bằng cách thêm vào mạng một số cung có sức chứa 0, ta có thể giả thiết rằng mỗi cung $e = (u, v) \in E$ luôn có tương ứng duy nhất một cung ngược chiều, ký hiệu $-e = (v, u) \in E$, gọi là *cung đối* của cung e , ta cũng coi e là cung đối của cung $-e$ (tức là $-(-e) = e$). Có thể thấy rằng số cung cần thêm vào mạng là một đại lượng $O(E)$.

Chú ý rằng **mạng là một đa đồ thị**, tức là giữa hai đỉnh có thể có nhiều cung nối.

Để thuận tiện cho việc trình bày, ta quy ước các ký hiệu sau:

* Từ này còn có thể dịch là “khả năng thông qua” hay “lưu lượng”

Với X, Y là hai tập con của tập đỉnh V và $f: E \rightarrow \mathbb{R}$ là một hàm xác định trên tập cạnh E :

Ký hiệu $\{X \rightarrow Y\}$ là tập các cung nối một từ một đỉnh thuộc X tới một đỉnh thuộc Y :

$$\{X \rightarrow Y\} = \{e = (u, v) \in E : u \in X, v \in Y\}$$

Ký hiệu $f(X, Y)$ là tổng các giá trị hàm f trên các cung $e \in \{X \rightarrow Y\}$:

$$f(X, Y) = \sum_{e \in \{X \rightarrow Y\}} f(e)$$

b) Luồng

Luồng (*flow*) trên mạng G là một hàm:

$$\begin{aligned} f: E &\rightarrow \mathbb{R} \\ e &\mapsto f(e) \end{aligned}$$

gán cho mỗi cung e một số thực $f(e)$, gọi là luồng trên cung e , thỏa mãn ba ràng buộc sau đây:

- Ràng buộc về sức chứa (*Capacity constraint*): Luồng trên mỗi cung không được vượt quá sức chứa của cung đó: $\forall e \in E: f(e) \leq c(e)$.
- Ràng buộc về tính đối xứng lệch (*Skew symmetry*): Với $\forall e \in E$, luồng trên cung e và luồng trên cung đối $-e$ có cùng giá trị tuyệt đối nhưng trái dấu nhau: $\forall e \in E: f(e) = -f(-e)$.
- Ràng buộc về tính bảo tồn (*Flow conservation*): Với mỗi đỉnh v không phải đỉnh phát và cũng không phải đỉnh thu, tổng luồng trên các cung đi ra khỏi v bằng 0: $\forall v \in V - \{s, t\}: f(\{v\}, V) = 0$.

Từ ràng buộc về tính đối xứng lệch và tính bảo tồn, ta suy ra được: Với mọi đỉnh $v \in V - \{s, t\}$, tổng luồng trên các cung đi vào v bằng 0: $f(V, \{v\}) = 0$.

Giá trị của luồng f trên mạng G được định nghĩa bằng tổng luồng trên các cung đi ra khỏi đỉnh phát:

$$|f| = f(\{s\}, V) \tag{3.1}$$

Bài toán luồng cực đại trên mạng (*maximum-flow problem*): Cho một mạng G với đỉnh phát s và đỉnh thu t , hàm sức chứa c , hãy tìm một luồng có giá trị lớn nhất trên mạng G .

c) Luồng dương

Luồng dương (*positive flow*) trên mạng G là một hàm

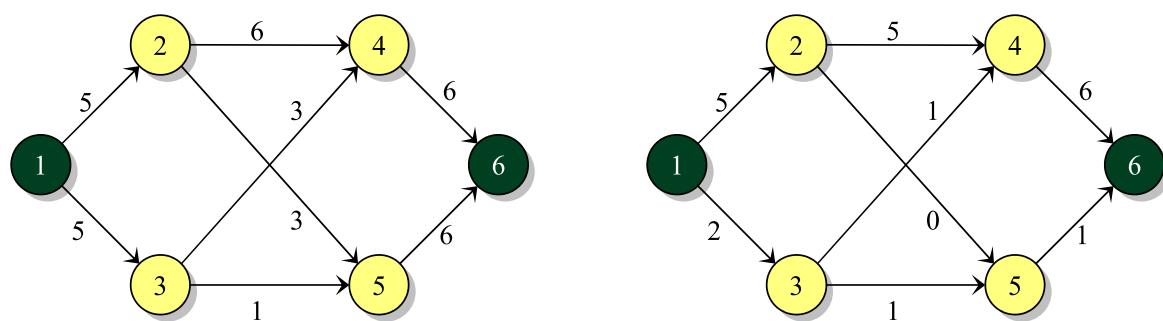
$$\begin{aligned}\varphi: E &\rightarrow [0, +\infty) \\ e &\mapsto \varphi(e)\end{aligned}$$

gán cho mỗi cung e một số thực không âm $\varphi(e)$ gọi là luồng dương trên cung e thỏa mãn hai ràng buộc sau đây:

- Ràng buộc về sức chứa (*Capacity constraint*): Luồng dương trên mỗi cung không được vượt quá sức chứa của cung đó: $\forall e \in E: 0 \leq \varphi(e) \leq c(e)$.
- Ràng buộc về tính bảo tồn (*Flow conservation*): Với mỗi đỉnh v không phải đỉnh phát và cũng không phải đỉnh thu, tổng luồng dương trên các cung đi vào v bằng tổng luồng dương trên các cung đi ra khỏi v : $\forall v \in V - \{s, t\}: \varphi(V, \{v\}) = \varphi(\{v\}, V)$.

Giá trị của một luồng dương được định nghĩa bằng tổng luồng dương trên các cung đi ra khỏi đỉnh phát trừ đi tổng luồng dương trên các cung đi vào đỉnh phát*:

$$|\varphi| = \varphi(\{s\}, V) - \varphi(V, \{s\}) \quad (3.2)$$



Hình 2.6. Mạng với các sức chứa trên cung (1 phát, 6 thu) và một luồng dương với giá trị 7

* Một số tài liệu khác đưa vào thêm ràng buộc: đỉnh phát s không có cung đi vào và đỉnh thu t không có cung đi ra. Khi đó giá trị luồng dương bằng tổng luồng dương trên các cung đi ra khỏi đỉnh phát. Cách hiểu này có thể quy về một trường hợp riêng của định nghĩa.

d) Mối quan hệ giữa luồng và luồng dương

Bố đề 3-1

Cho $\varphi: E \rightarrow \mathbb{R}$ là một luồng dương trên mạng $G = (V, E, c, s, t)$. Khi đó hàm

$$\begin{aligned} f: E &\rightarrow \mathbb{R} \\ e &\mapsto f(e) = \varphi(e) - \varphi(-e) \end{aligned}$$

là một luồng trên mạng G và $|f| = |\varphi|$

Chứng minh

Trước hết ta chứng minh f thỏa mãn tất cả các ràng buộc về luồng:

Ràng buộc về sức chứa: Với $\forall e \in E$:

$$f(e) = \varphi(e) - \underbrace{\varphi(-e)}_{\geq 0} \leq \varphi(e) \leq c(u, v)$$

Ràng buộc về tính đối xứng lệch: Với $\forall e \in E$

$$\begin{aligned} f(e) &= \varphi(e) - \varphi(-e) \\ &= -(\varphi(-e) - \varphi(e)) \\ &= -f(-e) \end{aligned}$$

Ràng buộc về tính bảo tồn: $\forall v \in V$, ta có:

$$\begin{aligned} f(\{v\}, V) &= \sum_{e \in \{\{v\} \rightarrow V\}} (\varphi(e) - \varphi(-e)) \\ &= \left(\sum_{e \in \{\{v\} \rightarrow V\}} \varphi(e) \right) - \left(\sum_{e \in \{\{v\} \rightarrow V\}} \varphi(-e) \right) \\ &= \left(\sum_{e \in \{\{v\} \rightarrow V\}} \varphi(e) \right) - \left(\sum_{-e \in \{V \rightarrow \{v\}\}} \varphi(-e) \right) \\ &= \varphi(\{v\}, V) - \varphi(V, \{v\}) \end{aligned}$$

Nếu $v \neq s$ và $v \neq t$, ta có:

$$f(\{v\}, V) = \varphi(\{v\}, V) - \varphi(V, \{v\}) = 0$$

(Do tổng luồng dương đi ra khỏi v bằng tổng luồng dương đi vào v)

Nếu $v = s$, xét giá trị luồng f

$$|f| = f(\{s\}, V) = \varphi(\{s\}, V) - \varphi(V, \{s\}) = |\varphi|$$

Bố đề 3-2

Cho $f: E \rightarrow \mathbb{R}$ là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó hàm:

$$\varphi: E \rightarrow [0, \infty)$$

$$e \mapsto \varphi(e) = \max\{f(e), 0\} = \begin{cases} f(e), & \text{nếu } f(e) \geq 0 \\ 0, & \text{nếu } f(e) < 0 \end{cases}$$

là một luồng dương trên mạng và $|\varphi| = |f|$

Chứng minh

Trước hết ta chứng minh φ thỏa mãn các ràng buộc về luồng dương:

Ràng buộc về sức chứa: $\forall e \in E$, rõ ràng $\varphi(e)$ là số không âm và $\varphi(e) = \max\{f(e), 0\} \leq c(e)$.

Ràng buộc về tính bảo tồn: $\forall v \in V$, tổng luồng dương ra khỏi v trừ tổng luồng dương đi vào v bằng:

$$\begin{aligned} \varphi(\{v\}, V) - \varphi(V, \{v\}) &= \sum_{\substack{e \in \{v\} \rightarrow V \\ f(e) \geq 0}} f(e) - \sum_{\substack{e \in V \rightarrow \{v\} \\ f(e) > 0}} f(e) \\ &= \sum_{\substack{e \in \{v\} \rightarrow V \\ f(e) \geq 0}} f(e) + \sum_{\substack{-e \in \{v\} \rightarrow V \\ f(-e) < 0}} f(-e) = f(\{v\}, V) \end{aligned}$$

Nếu $v \neq s$ và $v \neq t$, $f(\{v\}, V) = 0$ nên luồng dương đi vào v ($\varphi(\{v\}, V)$) được bảo tồn khi đi ra khỏi v ($\varphi(V, \{v\})$).

Nếu $v = s$, ta có:

$$|\varphi| = \varphi(\{s\}, V) - \varphi(V, \{s\}) = f(\{s\}, V) = |f|$$

Bổ đề 3-1 và Bổ đề 3-2 cho ta một mối tương quan giữa luồng và luồng dương. Khái niệm về luồng dương dễ hình dung hơn so với khái niệm luồng, tuy nhiên những định nghĩa về luồng tổng quát lại thích hợp hơn cho việc trình bày và chứng minh các thuật toán trong bài. Ta sẽ **sử dụng luồng dương trong các hình vẽ và output** (chỉ quan tâm tới các giá trị luồng dương $\varphi(e)$), còn các khái niệm về luồng sẽ được dùng để diễn giải các thuật toán trong bài.

Trong quá trình cài đặt thuật toán, các hàm c và f sẽ được xác định bởi tập các giá trị $\{c[e]\}_{e \in E}$ và $\{f[e]\}_{e \in E}$ nên ta có thể dùng lẩn các ký hiệu $c(e), f(e)$ (nếu muốn đề cập tới giá trị hàm) hoặc $c[e], f[e]$ (nếu muốn đề cập tới các biến số).

e) Một số tính chất cơ bản

Cho mạng $G = (V, E, c, s, t)$ và một luồng f trên G . Gọi $c(X, Y)$ là lưu lượng từ X sang Y và $f(X, Y)$ là giá trị luồng từ X sang Y .

Định lý 3-3

Cho f là một luồng trên mạng $G = (V, E, c, s, t)$, khi đó:

- a) $\forall X \subseteq V$, ta có $f(X, X) = 0$.
- b) $\forall X, Y \subseteq V$, ta có $f(X, Y) = -f(Y, X)$.
- c) $\forall X, Y, Z \subseteq V$ và $X \cap Y = \emptyset$, ta có $f(X, Z) + f(Y, Z) = f(X \cup Y, Z)$.
- d) $\forall X \subseteq V - \{s, t\}$, ta có $f(X, V) = 0$.

Chứng minh

- a) $\forall X \subseteq V$, ta có:

$$f(X, X) = \sum_{e \in \{X \rightarrow X\}} f(e)$$

như vậy $f(e)$ xuất hiện trong tổng nếu và chỉ nếu $f(-e)$ cũng xuất hiện trong tổng. Theo tính đối xứng lệch của luồng: $f(e) = -f(-e)$, ta có $f(X, X) = 0$.

- b) $\forall X, Y \subseteq V$, ta có :

$$f(X, Y) = \sum_{e \in \{X \rightarrow Y\}} f(e) = - \sum_{-e \in \{Y \rightarrow X\}} f(-e) = -f(Y, X)$$

- c) $\forall X, Y, Z \subseteq V$ và $X \cap Y = \emptyset$, ta có:

$$\begin{aligned} f(X \cup Y, Z) &= \sum_{e \in \{X \cup Y \rightarrow Z\}} f(e) \\ &= \underbrace{\sum_{e \in \{X \rightarrow Z\}} f(e)}_{f(X, Z)} + \underbrace{\sum_{e \in \{Y \rightarrow Z\}} f(e)}_{f(Y, Z)} \end{aligned}$$

- d) $\forall X \subseteq V - \{s, t\}$, do

$$X = \bigcup_{u \in X} \{u\}$$

Nên theo chứng minh phần c):

$$f(X, V) = \sum_{u \in X} f(\{u\}, V)$$

Mỗi hạng tử của tổng: $f(\{u\}, V)$ chính là tổng luồng trên các cung đi ra khỏi đỉnh u , do tính bảo tồn luồng và u không phải đỉnh phát cũng không phải đỉnh thu, hạng tử này phải bằng 0, suy ra $f(X, V) = 0$. Từ chứng minh phần b), ta còn suy ra $f(V, X) = 0$ nữa.

Định lý 3-4

Giá trị luồng trên mạng bằng tổng luồng trên các cung đi vào đỉnh thu

Chứng minh

Giả sử f là một luồng trên mạng $G = (V, E, c, s, t)$, ta có:

$$\begin{aligned}
 |f| &= f(\{s\}, V) \\
 &= f(V, V) - f(V - \{s\}, V) \\
 &= -f(V - \{s\}, V) \\
 &= f(V, V - \{s\}) \\
 &= f(V, \{t\}) + f(V, V - \{s, t\}) \\
 &= f(V, \{t\})
 \end{aligned}$$

Hệ quả

Giá trị luồng dương trên mạng bằng tổng luồng dương đi vào đỉnh thu trừ tổng luồng dương ra khỏi đỉnh thu.

f) Mạng thặng dư

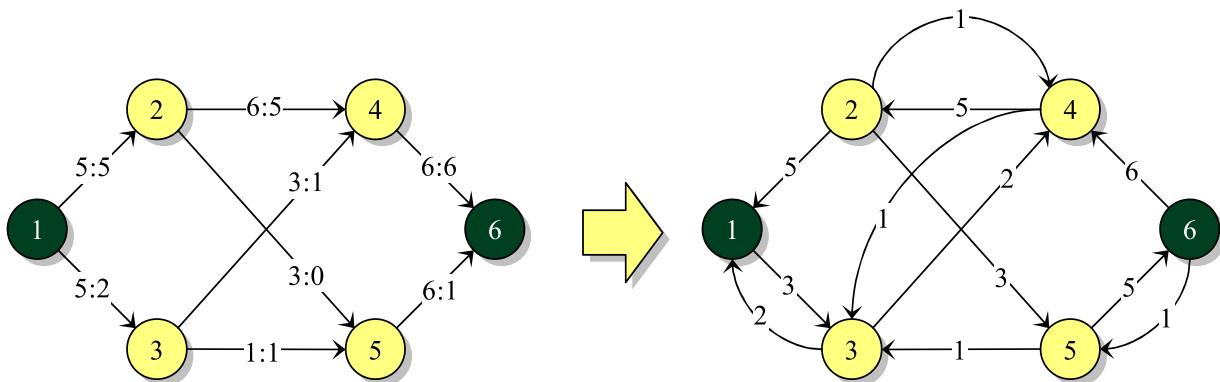
Với f là một luồng trên mạng $G = (V, E, c, s, t)$. Ta xét mạng G_f cũng là mạng G nhưng với hàm sức chứa mới cho bởi:

$$\begin{aligned}
 c_f: E &\rightarrow [0, +\infty) \\
 e &\mapsto c_f(e) = c(e) - f(e)
 \end{aligned} \tag{3.3}$$

Mạng G_f xây dựng như vậy được gọi là *mạng thặng dư* (*residual network*) của mạng G sinh ra bởi luồng f . Sức chứa của cung (e) trên G_f thực chất là lượng luồng tối đa chúng ta có thể đẩy thêm vào luồng $f(e)$ mà không làm vượt quá sức chứa $c(e)$.

Một cung trên G gọi là *cung bão hòa* (*saturated edge*) nếu luồng trên cung đó đúng bằng sức chứa, ngược lại cung đó gọi là *cung thặng dư* (*residual edge*). Ký hiệu E_f là tập các cung thặng dư trên mạng thặng dư G_f . Một đường đi chỉ qua các cung thặng dư trên G_f gọi là *đường thặng dư* (*residual path*).

Cung bão hòa của mạng G trên mạng thặng dư sẽ có sức chứa 0, cung này ít có ý nghĩa trong thuật toán nên chúng ta sẽ chỉ vẽ các cung thặng dư ($\in E_f$) trong các hình vẽ.



Hình 2.7. Một luồng trên mạng (số ghi trên các cung là: sức chứa:luồng dương) và mạng thặng dư tương ứng.

Hình 2.7 là một ví dụ về mạng thặng dư. Như đã quy ước, chúng ta chỉ vẽ các luồng dương. Đồ thị có cung $(2,4)$ với sức chứa $c(2,4) = 6$, tức là phải có cung đối $(4,2)$ với $c(4,2) = 0$. Luồng dương trên cung $(2,4)$ là $\varphi(2,4) = f(2,4) = 5$, điều này cũng cho biết luồng trên cung $(4,2)$ là $f(4,2) = -5$ theo tính đối xứng lệch. Vậy trên mạng thặng dư, ta có cung $(2,4)$ với sức chứa $c(2,4) - f(2,4) = 6 - 5 = 1$ đồng thời có cung $(4,2)$ với sức chứa $c(4,2) - f(4,2) = 0 - (-5) = 5$.

Định lý 3-5

Cho f là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó nếu f' là một luồng trên G_f thì hàm:

$$\begin{aligned} f + f' : E &\rightarrow \mathbb{R} \\ e &\mapsto (f + f')(e) = f(e) + f'(e) \end{aligned}$$

là một luồng trên mạng G với giá trị luồng $|f + f'| = |f| + |f'|$.

Chứng minh

Ta chứng minh $(f + f')$ thỏa mãn ba tính chất của luồng:

Ràng buộc về sức chứa: Với $\forall e \in E$:

$$\begin{aligned} (f + f')(e) &= f(e) + f'(e) \\ &\leq f(e) + (c(e) - f(e)) \\ &= c(e) \end{aligned}$$

Tính đối xứng lệch: Với $\forall e \in E$:

$$\begin{aligned} (f + f')(e) &= f(e) + f'(e) \\ &= -f(-e) - f'(-e) \end{aligned}$$

$$\begin{aligned}
&= -(f(-e) + f'(-e)) \\
&= -(f + f')(-e)
\end{aligned}$$

Tính bảo tồn: Với $\forall u \in V$, tổng luồng $f + f'$ đi ra khỏi u bằng:

$$\begin{aligned}
(f + f')(\{u\}, V) &= \sum_{e \in \{u\} \rightarrow V} (f(e) + f'(e)) \\
&= \sum_{e \in \{u\} \rightarrow V} f(e) + \sum_{e \in \{u\} \rightarrow V} f'(e) \\
&= f(\{u\}, V) + f'(\{u\}, V)
\end{aligned}$$

Nếu $u \neq s$ và $u \neq t$, ta có $(f + f')(\{u\}, V) = f(\{u\}, V) + f'(\{u\}, V) = 0$.

Thay $u = s$, ta có

$$|f + f'| = (f + f')(\{s\}, V) = f(\{s\}, V) + f'(\{s\}, V) = |f| + |f'|$$

Định lý 3-6

Cho f và f' là hai luồng trên mạng $G = (V, E, c, s, t)$ khi đó hàm:

$$\begin{aligned}
f' - f: E &\rightarrow \mathbb{R} \\
e &\mapsto (f' - f)(e) = f'(e) - f(e)
\end{aligned}$$

là một luồng trên mạng thặng dư G_f với giá trị luồng $|f' - f| = |f'| - |f|$.

Chứng minh

Ta chứng minh rằng $f' - f$ thỏa mãn ba tính chất của luồng

Ràng buộc về sức chứa: Với $\forall e \in E$:

$$\begin{aligned}
(f' - f)(e) &= f'(e) - f(e) \\
&\leq c(e) - f(e) \\
&= c_f(e)
\end{aligned}$$

Tính đối xứng lệch: Với $\forall e \in E$:

$$\begin{aligned}
(f' - f)(e) &= f'(e) - f(e) \\
&= -(f'(-e) - f(-e)) \\
&= -(f' - f)(-e)
\end{aligned}$$

Tính bảo tồn: Với $\forall v \in V$

$$\begin{aligned}
(f' - f)(\{v\}, V) &= \sum_{e \in \{v\} \rightarrow V} (f'(e) - f(e)) \\
&= \sum_{e \in \{v\} \rightarrow V} f'(e) - \sum_{e \in \{v\} \rightarrow V} f(e) \\
&= f'(\{v\}, V) - f(\{v\}, V)
\end{aligned}$$

Nếu $u \neq s$ và $u \neq t$, ta có $(f' - f)(\{v\}, V) = f'(\{v\}, V) - f(\{v\}, V) = 0$.

Thay $u = s$, ta có

$$|f' - f| = (f' - f)(\{s\}, V) = f'(\{s\}, V) - f(\{s\}, V) = |f'| - |f|$$

3.2. Thuật toán Ford-Fulkerson

a) Đường tăng luồng

Với f là một luồng trên mạng $G = (V, E, c, s, t)$. Gọi P là một đường đi đơn từ s tới t trên mạng thặng dư G_f . Giá trị thặng dư (residual capacity) của đường P , ký hiệu Δ_P , được định nghĩa bằng sức chứa nhỏ nhất của các cung dọc trên đường P (xét trên G_f):

$$\Delta_P = \min\{c_f(e) : (e) nằm trên P\}$$

Vì các sức chứa $c_f(e)$ là số không âm nên Δ_P luôn là số không âm. Nếu $\Delta_P > 0$ tức là đường đi P là một đường thặng dư, khi đó đường đi P gọi là một *đường tăng luồng* (augmenting path) tương ứng với luồng f .

Định lý 3-7

Cho f là một luồng trên mạng $G = (V, E, c, s, t)$, P là một đường tăng luồng trên G_f . Khi đó hàm $f_P: E \rightarrow \mathbb{R}$ định nghĩa như sau:

$$f_P(e) = \begin{cases} +\Delta_P, & \text{nếu } e \in P \\ -\Delta_P, & \text{nếu } -e \in P \\ 0, & \text{trường hợp khác} \end{cases} \quad (3.4)$$

là một luồng trên G_f với giá trị luồng $|f_P| = \Delta_P > 0$.

Chứng minh

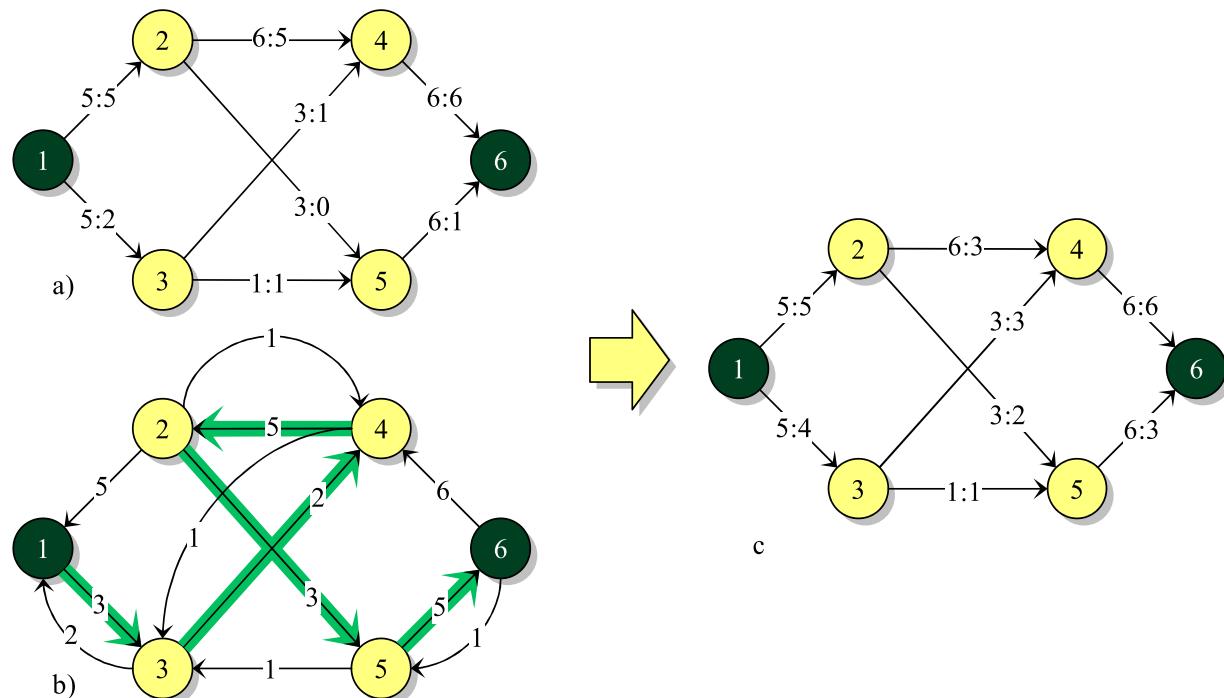
Chúng ta sẽ không chứng minh cụ thể vì việc kiểm chứng f_P thỏa mãn ba tính chất của luồng khá dễ dàng. Bản chất của luồng f_P là đẩy một giá trị luồng Δ_P từ s tới t dọc theo các cung trên đường P , đồng thời kéo một giá trị luồng $-\Delta_P$ từ t về s theo hướng ngược lại*.

* Có thể hình dung cơ chế này như một quá trình điện phân: Bao nhiêu ion dương (cation) chuyển đến cực âm (anot) t thì cũng phải có bấy nhiêu ion âm (anion) chuyển đến cực dương (anot) s .

Định lý 3-5 và Định lý 3-7 cho ta một hệ quả sau:

Hệ quả 3-8

Cho f là một luồng trên mạng $G = (V, E, c, s, t)$ và P là một đường tăng luồng trên G_f , gọi f_P là luồng trên G_f định nghĩa như trong công thức (3.4). Khi đó $f + f_P$ là một luồng mới trên G với giá trị $|f + f_P| = |f| + |f_P| = |f| + \Delta_P$.



Hình 2.8. Tăng luồng dọc đường tăng luồng.

Hình 2.8 là ví dụ về cơ chế tăng luồng trên mạng với đỉnh phát 1, đỉnh thu 6 và luồng f giá trị 7 (hình a) (chú ý rằng ta chỉ vẽ các luồng dương cho đỡ rối). Với mạng thặng dư G_f (hình b), giả sử ta chọn đường đi $P = \langle 1, 3, 4, 2, 5, 6 \rangle$ làm đường tăng luồng, giá trị thặng dư của P bằng $\Delta_P = 2$ (sức chứa của cung $(3, 4)$). Luồng f_P trên G_f sẽ có các giá trị sau:

$$\begin{aligned} f_P(1,3) &= f_P(3,4) = f_P(4,2) = f_P(2,5) = f_P(5,6) = 2 \\ f_P(3,1) &= f_P(4,3) = f_P(2,4) = f_P(5,2) = f_P(6,5) = -2 \end{aligned}$$

Cộng các giá trị này vào luồng f đang có, ta sẽ được một luồng mới trên G với giá trị 9 (hình c).

Cơ chế cộng luồng f_P vào luồng f hiện có gọi là *tăng luồng dọc theo đường tăng luồng P* .

b) Thuật toán Ford-Fulkerson

Thuật toán Ford-Fulkerson [14] để tìm luồng cực đại trên mạng dựa trên cơ chế tăng luồng dọc theo đường tăng luồng. Bắt đầu từ một luồng f bất kỳ trên mạng (chẳng hạn luồng trên mọi cung đều bằng 0), thuật toán tìm đường tăng luồng P trên mạng thặng dư, gán $f := f + f_P$ để tăng giá trị luồng f và lặp lại cho tới khi không tìm được đường tăng luồng nữa.

```
f := «Một luồng bất kỳ»;  
while «Tìm được đường tăng luồng P» do  
    f := f + fP;  
    Output ← f;
```

c) Cài đặt

Chúng ta sẽ cài đặt thuật toán Ford-Fulkerson để tìm luồng cực đại trên mạng với khuôn dạng Input/Output như sau:

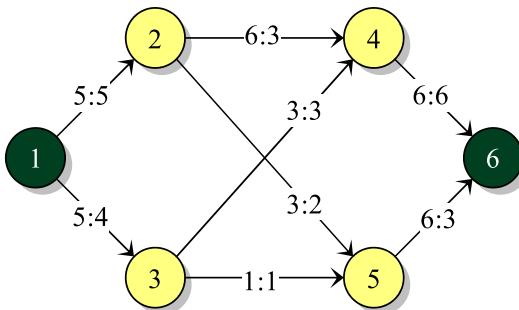
Input

- Dòng 1 chứa số đỉnh $n \leq 10^3$, số cung $m \leq 10^5$ của mạng, đỉnh phát s , đỉnh thu t .
- m dòng tiếp theo, mỗi dòng chứa ba số nguyên dương u, v, c tương ứng với một cung nối từ u tới v với sức chứa $c \leq 10^4$.

Output

Luồng cực đại trên mạng (như đã quy ước, chỉ đưa ra các luồng dương trên các cung).

Sample Input	Sample Output
6 8 1 6	Maximum flow:
5 6 6	$e[1] = (5, 6): c = 6, f = 3$
4 6 6	$e[2] = (4, 6): c = 6, f = 6$
3 5 1	$e[3] = (3, 5): c = 1, f = 1$
3 4 3	$e[4] = (3, 4): c = 3, f = 3$
2 5 3	$e[5] = (2, 5): c = 3, f = 2$
2 4 6	$e[6] = (2, 4): c = 6, f = 3$
1 3 5	$e[7] = (1, 3): c = 5, f = 4$
1 2 5	$e[8] = (1, 2): c = 5, f = 5$
	Value of flow: 9



Để cài đặt thuật toán được hiệu quả cần có một cơ chế tổ chức dữ liệu hợp lý. Chúng ta cần lưu trữ luồng f trên các cung, tìm đường tăng luồng P trên G_f và cộng luồng f_P vào luồng f hiện có. Việc tìm đường tăng luồng P trên G_f sẽ được thực hiện bằng một thuật toán tìm kiếm trên đồ thị còn việc tăng luồng dọc trên đường P đòi hỏi phải tăng giá trị luồng trên các cung dọc trên đường đi đồng thời giảm giá trị luồng trên các cung đối. Vậy cấu trúc dữ liệu cần tổ chức để tạo điều kiện thuận lợi cho thuật toán tìm đường tăng luồng cũng như dễ dàng chỉ ra cung đối của một cung cho trước.

Đồ thị được biểu diễn bởi danh sách liên thuộc. Tất cả m cung của mạng được chứa trong danh sách $e[1 \dots m]$. Ngoài ra ta thêm m cung đối của chúng với sức chứa 0. Các cung đối này được lưu trữ trong danh sách $e[-m \dots -1]$, cung đối của cung $e[i]$ là cung $e[-i]$, cung $e[0]$ được sử dụng với vai trò phần tử cầm cánh và không được tính đến.

Mỗi phần tử của danh sách e là một bản ghi gồm 4 trường (x, y, c, f) trong đó x , y là đỉnh đầu và đỉnh cuối của cung, c là sức chứa và f là luồng trên cung. Danh

sách liên thuộc được xây dựng bởi hai mảng $head[1 \dots n]$ và $link[-m \dots m]$, trong đó:

- $head[u]$ là chỉ số cung đầu tiên trong danh sách liên thuộc các cung đi ra khỏi u , trường hợp u không có cung đi ra, $head[u]$ được gán bằng 0.
- $link[i]$ là chỉ số cung kế tiếp cung $e[i]$ trong cùng danh sách liên thuộc các cung đi ra khỏi một đỉnh. Trường hợp $e[i]$ là cung cuối cùng của một danh sách liên thuộc, $link[i]$ được gán bằng 0

Việc duyệt các cung đi ra khỏi đỉnh u sẽ được thực hiện theo cách sau:

```
i := head[u]; //i là chỉ số cung đầu tiên trong danh sách liên thuộc các cung ra khỏi u
while i ≠ 0 do //Chừng nào chưa duyệt qua cung cuối danh sách liên thuộc
    begin
        «Xử lý cung e[i]»;
        i := link[i]; //Nhảy sang xét cung kế tiếp trong danh sách liên thuộc
    end;
```

Tại mỗi bước, ta dùng thuật toán BFS để tìm đường đi từ s tới t trên G_f , mỗi đỉnh v trên đường đi được lưu vết $trace[v]$ là chỉ số cung đi vào v trên đường đi P tìm được. Dựa vào vết này, ta sẽ liệt kê được tất cả các cung trên đường đi, tăng luồng trên các cung này lên Δ_P đồng thời giảm luồng trên các cung đối đi Δ_P .

Edmonds và Karp [12] đã đề xuất mô hình cài đặt thuật toán Ford-Fulkerson trong đó thuật toán BFS được sử dụng để tìm đường tăng luồng nên người ta còn gọi thuật toán Ford-Fulkerson với kỹ thuật sử dụng BFS tìm đường tăng luồng là thuật toán Edmonds-Karp.

EDMONDSKARP.PAS ✓ Thuật toán Edmonds-Karp

```
{$MODE OBJFPC}
program MaximumFlow;
const
    maxN = 1000;
    maxM = 100000;
    maxC = 10000;
type
    TEdge = record //Cấu trúc một cung
        x, y: Integer; //Hai đỉnh đầu mút
```

```

c, f: Integer; //Sốc chứa và luồng
end;
TQueue = record //Hàng đợi dùng cho BFS
    items: array[1..maxN] of Integer;
    front, rear: Integer;
end;
var
e: array[-maxM..maxM] of TEdge; //Danh sách các cung
link: array[-maxM..maxM] of Integer;
//Móc nối trong danh sách liên thuộc
head: array[1..maxN] of Integer;
//head[u]: Chỉ số cung đầu tiên trong danh sách liên thuộc các cung ra khỏi u
trace: array[1..maxN] of Integer; //Vết đường đi
n, m, s, t: Integer;
FlowValue: Integer;
Queue: TQueue;
procedure Enter; //Nhập dữ liệu
var i, u, v, capacity: Integer;
begin
ReadLn(n, m, s, t);
FillChar(head[1], n * SizeOf(head[1]), 0);
for i := 1 to m do
    begin
        ReadLn(u, v, capacity);
        with e[i] do //Thêm cung e[i] = (u, v) vào danh sách liên thuộc của u
            begin
                x := u;
                y := v;
                c := capacity;
                link[i] := head[u];
                head[u] := i;
            end;
        with e[-i] do //Thêm cung e[-i] = (v, u) vào danh sách liên thuộc của v
            begin
                x := v;
                y := u;
                c := 0;
                link[-i] := head[v];
            end;
    end;

```

```

        head[v] := -i;
    end;
end;
procedure InitZeroFlow; //Khởi tạo luồng 0
var i: Integer;
begin
    for i := -m to m do e[i].f := 0;
    FlowValue := 0;
end;
function FindPath: Boolean; //Tìm đường tăng luồng bằng BFS
var u, v, i: Integer;
begin
    FillChar(trace[1], n * SizeOf(trace[1]), 0);
    trace[s] := 1; //trace[s] ≠ 0: đỉnh đã thăm, có thể dùng bắt cứ hằng số nào khác 0
    with Queue do
        begin
            items[1] := s;
            front := 1;
            rear := 1; //Hàng đợi chỉ gồm đỉnh s
            repeat
                u := items[front];
                Inc(front); //Lấy một đỉnh u khỏi hàng đợi
                i := head[u];
                while i <> 0 do //Duyệt danh sách liên thuộc của u
                    begin
                        v := e[i].y; //nút e[i] chứa một cung đi từ u tới v
                        if (trace[v] = 0)
                            and (e[i].f < e[i].c) then
                            //v chưa thăm và e[i] là cung thăng dư
                            begin
                                trace[v] := i; //Lưu vết
                                if v = t then Exit(True);
                                //Tìm thấy đường tăng luồng, thoát
                                Inc(rear);
                                items[rear] := v; //Đẩy v vào hàng đợi
                            end;
                        i := link[i]; //Nhảy sang nút kế tiếp trong danh sách liên thuộc
                    end;
            end;

```

```

        end;
until front > rear;
Result := False; //Không tìm thấy đường tăng luồng
end;
end;

procedure AugmentFlow; //Tăng luồng dọc đường một tăng luồng
var Delta, v, i: Integer;
begin
//Trước hết xác định Delta bằng sức chứa nhỏ nhất của các cung trên đường tăng luồng
v := t; //Bắt đầu từ t
Delta := maxC;
repeat
    i := trace[v];
    //e[i] là một cung trên đường tăng luồng với sức chứa e[i].c - e[i].f
    if e[i].c - e[i].f < Delta then
        Delta := e[i].c - e[i].f;
    v := e[i].x; //Di dời về s
until v = s;
//Tăng luồng thêm Delta
v := t; //Bắt đầu từ t
repeat
    i := trace[v]; //e[i] là một cung trên đường tăng luồng
    Inc(e[i].f, Delta); //Tăng luồng trên e[i] lên Delta
    Dec(e[-i].f, Delta); //Giảm luồng trên cung đối tương ứng đi Delta
    v := e[i].x; //Di dời về s
until v = s;
Inc(FlowValue, Delta); //Giá trị luồng f được tăng lên Delta
end;
procedure PrintResult; //In kết quả
var i: Integer;
begin
WriteLn('Maximum flow: ');
for i := 1 to m do
    with e[i] do
        if f > 0 then //Chỉ cần in ra các cung có luồng > 0
            WriteLn('e[ ', i, ' ] = ( ', x, ', ', ' , y, ', ') : c
                    = ', c, ', f = ', f);
    WriteLn('Value of flow: ', FlowValue);
end;

```

```

begin
    Enter; //Nhập dữ liệu
    InitZeroFlow; //Khởi tạo luồng 0
    while FindPath do //Thuật toán Ford-Fulkerson
        AugmentFlow;
    PrintResult; //In kết quả
end.

```

d) Tính đúng của thuật toán

Trước hết dễ thấy rằng thuật toán Ford-Fulkerson trả về một luồng, tức là kết quả mà thuật toán trả về thỏa mãn các tính chất của luồng. Việc chứng minh luồng đó là cực đại đã xây dựng một định lý quan trọng về mối quan hệ giữa luồng cực đại và lát cắt hẹp nhất.

Ta gọi một lát cắt (X, Y) là một cách phân hoạch tập đỉnh V làm hai tập rời nhau: $X \cap Y = \emptyset$ và $X \cup Y = V$. Lát cắt có $s \in X$ và $t \in Y$ được gọi là một lát cắt $s - t$.

Lưu lượng từ X sang Y ($c(X, Y)$) và luồng từ X sang Y ($f(X, Y)$) được gọi là lưu lượng và luồng thông qua lát cắt.

Bố đề 3-9

Với f là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó luồng thông qua một lát cắt $s - t$ bất kỳ bằng $|f|$.

Chứng minh

Với $V = X \cup Y$ là một lát cắt $s - t$ bất kỳ, theo Định lý 3-3

$$f(X, Y) = f(X, V) - f(X, V - Y) = f(X, V) - f(X, X) = f(X, V)$$

Cũng theo định lý này ta có:

$$f(X, V) = f(s, V) + \underbrace{f(X - \{s\}, V)}_0 = f(s, V) = |f|$$

Bố đề 3-10

Với f là một luồng trên mạng $G = (V, E, c, s, t)$. Khi đó luồng thông qua một lát cắt $s - t$ bất kỳ không vượt quá lưu lượng của lát cắt đó.

Chứng minh

Với $V = X \cup Y$ là một lát cắt $s - t$ bất kỳ ta có

$$f(X, Y) = \sum_{e \in \{X \rightarrow Y\}} f(e) \leq \sum_{e \in \{X \rightarrow Y\}} c(e) = c(X, Y)$$

Định lý 3-11 (mối quan hệ giữa luồng cực đại, đường tăng luồng và lát cắt hẹp nhất)

Nếu f là một luồng trên mạng $G = (V, E, c, s, t)$, khi đó ba mệnh đề sau là tương đương:

- a) f là luồng cực đại trên mạng G .
- b) Mạng thặng dư G_f không có đường tăng luồng.
- c) Tồn tại $V = X \cup Y$ là một lát cắt $s - t$ để $f(X, Y) = c(X, Y)$

Chứng minh

“a⇒b”

Giả sử phản chứng rằng mạng thặng dư G_f có đường tăng luồng P thì $f + f_P$ cũng là một luồng trên G với giá trị luồng lớn hơn f , trái giả thiết f là luồng cực đại trên mạng.

“b⇒c”

Nếu G_f không tồn tại đường tăng luồng thì ta đặt X là tập các đỉnh đến được từ s bằng một đường thặng dư và Y là tập các đỉnh còn lại:

$$X = \{v: \exists \text{ đường thặng dư } s \rightsquigarrow v\}; Y = V - X$$

Rõ ràng $X \cap Y = \emptyset$, $X \cup Y = V$ và $s \in X$, $t \in Y$ (t không thể đến được từ s bởi một đường thặng dư bởi nếu không thì đường đi đó sẽ là một đường tăng luồng).

Các cung $e \in \{X \rightarrow Y\}$ chắc chắn phải là cung bão hòa, bởi nếu có cung thặng dư $e = (u, v) \in \{X \rightarrow Y\}$ thì từ s sẽ tới được v bằng một đường thặng dư. Tức là $v \in X$, trái với cách xây dựng lát cắt. Từ $f(e) = c(e)$ với $\forall e \in \{X \rightarrow Y\}$, ta có

$$f(X, Y) = \sum_{e \in \{X \rightarrow Y\}} f(e) = \sum_{e \in \{X \rightarrow Y\}} c(e) = c(X, Y)$$

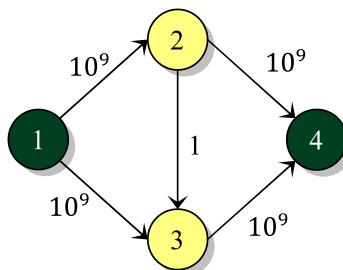
“c⇒a”

Bở đề 16.9 và Định lí 16.17 cho thấy giá trị của một luồng trên mạng không thể vượt quá lưu lượng của một lát cắt $s - t$ bất kỳ. Nếu tồn tại một lát cắt $s - t$ mà luồng thông qua lát cắt đúng bằng lưu lượng thì luồng đó chắc chắn phải là luồng cực đại.

Lát cắt $s - t$ có lưu lượng nhỏ nhất (bằng giá trị luồng cực đại trên mạng) gọi là *Lát cắt hẹp nhất* của mạng G .

e) Tính dừng của thuật toán

Thuật toán Ford-Fulkerson có thời gian thực hiện phụ thuộc vào thuật toán tìm đường tăng luồng tại mỗi bước. Có thể chỉ ra được ví dụ mà nếu dùng DFS để tìm đường tăng luồng thì thời gian thực hiện giải thuật không bị chặn bởi một hàm đa thức của số đỉnh và số cạnh.Thêm nữa, nếu sức chứa của các cung là số thực, người ta còn chỉ ra được ví dụ mà với thuật toán tìm đường tăng luồng không tốt, giá trị luồng sau mỗi bước vẫn tăng nhưng **không bao giờ đạt luồng cực đại**. Tức là nếu có thể cài đặt chương trình tính toán số thực với độ chính xác tuyệt đối, thuật toán sẽ chạy mãi không dừng.



Hình 2.8. Mạng với 4 đỉnh (1 phát, 4 thu), thuật toán Ford-Fulkerson có thể mất 2 lần tìm đường tăng luồng nếu luôn phiên chọn hai đường $\langle 1, 2, 3, 4 \rangle$ và $\langle 1, 3, 2, 4 \rangle$ làm đường tăng luồng, mỗi lần tăng giá trị luồng lên 1 đơn vị.

Chính vì vậy nên trong một số tài liệu người ta gọi là “phương pháp Ford-Fulkerson” để chỉ một cách tiếp cận chung, còn từ “thuật toán” được dùng để chỉ một cách cài đặt phương pháp Ford-Fulkerson trên một cấu trúc dữ liệu cụ thể, với một thuật toán tìm đường tăng luồng cụ thể. Ví dụ phương pháp Ford-Fulkerson cài đặt với thuật toán tìm đường tăng luồng bằng BFS như trên được gọi là thuật toán Edmonds-Karp. Tính dừng của thuật toán Edmonds-Karp sẽ được chỉ ra khi chúng ta đánh giá thời gian thực hiện giải thuật.

Xét G_f là mạng thặng dư của một mạng G ứng với luồng f nào đó, ta gán trọng số 1 cho các cung thặng dư của G_f và gán trọng số $+\infty$ cho các cung bão hòa của G_f . Để thấy rằng thuật toán tìm đường tăng luồng bằng BFS sẽ trả về một đường đi ngắn nhất từ s tới t tương ứng với hàm trọng số đã cho. Ký hiệu $\delta_f(u, v)$ là độ dài đường đi ngắn nhất từ u tới v (khoảng cách từ u tới v) trên mạng thặng dư.

Bố đề 3-12

Nếu ta khởi tạo luồng 0 và thực hiện thuật toán Edmonds-Karp trên mạng $G = (V, E)$ có đỉnh phát s và đỉnh thu t . Khi đó với mọi đỉnh $v \in V$, khoảng cách từ s tới v trên mạng thặng dư không giảm sau mỗi bước tăng luồng.

Chứng minh

Khi $v = s$, rõ ràng khoảng cách từ s tới chính nó luôn bằng 0 từ khi bắt đầu tới khi kết thúc thuật toán. Ta chỉ cần chứng minh bù đê đúng với những đỉnh $v \neq s$.

Giả sử phản chứng rằng tồn tại một đỉnh $v \in V - \{s\}$ mà khi thuật toán Edmonds-Karp tăng luồng f lên thành f' sẽ làm cho $\delta_{f'}(s, v)$ nhỏ hơn $\delta_f(s, v)$. Nếu có nhiều đỉnh v như vậy ta chọn đỉnh v có $\delta_{f'}(s, v)$ nhỏ nhất. Gọi $P = s \rightsquigarrow u \rightarrow v$ là đường đi ngắn nhất từ s tới v trên $G_{f'}$, ta có (u, v) là cung thặng dư trên $G_{f'}$ và

$$\delta_{f'}(s, u) = \delta_{f'}(s, v) - 1$$

Bởi cách chọn đỉnh v , độ dài đường đi ngắn nhất từ s tới u không thể bị giảm đi sau phép tăng luồng, tức là

$$\delta_{f'}(s, u) \geq \delta_f(s, u)$$

Ta chứng minh rằng (u, v) phải là cung bão hòa trên G_f . Thật vậy, nếu (u, v) là cung thặng dư (có trọng số 1) trên G_f thì:

$$\begin{aligned} \delta_f(s, v) &\leq \delta_f(s, u) + 1 \text{ (bất đẳng thức tam giác)} \\ &\leq \delta_{f'}(s, u) + 1 \text{ (khoảng cách từ } s \text{ tới } u \text{ không giảm)} \\ &= \delta_{f'}(s, v) \end{aligned}$$

Trái với giả thiết rằng khoảng cách từ s tới v phải giảm đi sau phép tăng luồng.

Làm thế nào để (u, v) là cung bão hòa trên G_f nhưng lại là cung thặng dư trên $G_{f'}$? Câu trả lời duy nhất là do phép tăng luồng từ f lên f' làm giảm luồng trên cung (u, v) , tức là cung đối (v, u) phải là một cung trên đường tăng luồng tìm được. Vì đường tăng luồng tại mỗi bước luôn là đường đi ngắn nhất nên (v, u) phải là cung cuối cùng trên đường đi ngắn nhất từ s tới u của G_f . Từ đó suy ra:

$$\begin{aligned} \delta_f(s, v) &= \delta_f(s, u) - 1 \\ &\leq \delta_{f'}(s, u) - 1 \text{ (khoảng cách từ } s \text{ tới } u \text{ không giảm)} \\ &= \delta_{f'}(s, v) - 2 \text{ (theo cách chọn } u \text{ và } v) \end{aligned}$$

Mâu thuẫn với giả thuyết khoảng cách từ s tới v phải giảm đi sau khi tăng luồng. Ta có điều phải chứng minh: Với $\forall v \in V$, khoảng cách từ s tới v trên mạng thặng dư không giảm sau mỗi bước tăng luồng.

Bố đề 3-13

Nếu thuật toán Edmonds-Karp thực hiện trên mạng $G = (V, E, c, s, t)$ với luồng khởi tạo là luồng 0 thì số lượt tăng luồng được sử dụng trong thuật toán là $O(|V||E|)$.

Chứng minh

Ta chia quá trình thực hiện thuật toán Edmonds-Karp thành các pha. Mỗi pha tìm một đường tăng luồng P và tăng luồng thêm một giá trị thặng dư Δ_P . Giá trị thặng dư này theo định nghĩa sẽ phải bằng sức chứa của một cung thặng dư e nào đó trên đường P :

$$\exists e \in P: \Delta_P = c(e) - f(e)$$

Khi tăng luồng dọc trên được P thì cung e sẽ trở thành bão hòa. Những cung thặng dư trở nên bão hòa sau khi tăng luồng gọi là *cung tới hạn (critical edge)* tại mỗi pha. Mỗi pha có ít nhất một cung tới hạn.

Ta đánh giá xem mỗi cung của mạng có thể trở thành cung tới hạn bao nhiêu lần. Với một cung $e = (u, v)$, ta xét pha A đầu tiên làm e trở thành cung tới hạn và f_A là luồng khi bắt đầu pha A . Do e nằm trên đường tăng luồng ngắn nhất trên G_{f_A} nên khi pha này bắt đầu:

$$\delta_{f_A}(s, u) + 1 = \delta_{f_A}(s, v)$$

Pha A sau khi tăng luồng sẽ làm cung e sẽ trở nên bão hòa.

Để e có thể trở thành cung tới hạn một lần nữa thì tiếp theo pha A phải có một pha B giảm luồng trên cung e để biến e thành cung thặng dư, tức là cung $-e = (v, u)$ phải là một cung trên đường tăng luồng của pha B . Gọi f_B là luồng khi pha B bắt đầu, cũng vì tính chất của đường đi ngắn nhất, ta có

$$\delta_{f_B}(s, v) + 1 = \delta_{f_B}(s, u)$$

Bố đề 16.12 đã chứng minh rằng khoảng cách từ s tới v trên mạng thặng dư không giảm đi sau mỗi pha, nên $\delta_{f_B}(s, v) \geq \delta_{f_A}(s, v)$. Suy ra:

$$\begin{aligned} \delta_{f_B}(s, u) &= \delta_{f_B}(s, v) + 1 \\ &\geq \delta_{f_A}(s, v) + 1 \\ &= \delta_{f_A}(s, u) + 2 \end{aligned}$$

Như vậy nếu một cung (u, v) là cung tới hạn trong k pha thì khi pha thứ k bắt đầu, khoảng cách từ s tới u trên mạng thặng dư đã tăng lên ít nhất $2(k - 1)$ đơn vị so với thời điểm trước pha thứ nhất. Khoảng cách $\delta_f(s, u)$ ban đầu là số không âm và chừng nào còn đường thặng dư đi từ s tới u , khoảng cách $\delta_f(s, u)$ không thể vượt quá $|V| - 1$. Điều đó cho thấy $k \leq \frac{|V|+1}{2} = O(|V|)$.

Tổng hợp lại, ta có:

- Mạng có tất cả $|E|$ cung.
- Mỗi pha có ít nhất một cung tới hạn
- Một cung có thể trở thành tới hạn trong $O(|V|)$ pha

Vậy tổng số pha được thực hiện trong thuật toán Edmonds-Karp là một đại lượng $O(|V||E|)$

Định lý 3-14

Có thể cài đặt thuật toán Edmonds-Karp để tìm luồng cực đại trên mạng $G = (V, E, c, s, t)$ trong thời gian $O(|V||E|^2)$.

Chứng minh

Bở đè 3-15 đã chứng minh rằng thuật toán Edmonds-Karp cần thực hiện $O(|V||E|)$ lượt tăng luồng. Tại mỗi lượt thuật toán tìm đường tăng luồng bằng BFS và tăng luồng dọc đường này có thời gian thực hiện $O(|E|)$. Suy ra thời gian thực hiện giải thuật Edmonds-Karp là $O(|V||E|^2)$.

Nếu khả năng thông qua trên các cung của mạng là số nguyên thì còn có một cách đánh giá khác dựa trên giá trị luồng cực đại, nếu ta khởi tạo luồng 0 thì sau mỗi lượt tăng luồng, giá trị luồng được tăng lên ít nhất 1 đơn vị. Suy ra thời gian thực hiện giải thuật khi đó là $O(|f||E|)$ với $|f|$ là giá trị luồng cực đại.

3.3. Thuật toán đầy tiền luồng

Thuật toán Ford-Fulkerson không những là một cách tiếp cận thông minh mà việc chứng minh tính đúng đắn của nó cho ta nhiều kết quả thú vị về mối liên hệ giữa luồng cực đại và lát cắt hẹp nhất. Tuy vậy với những đồ thị kích thước rất lớn thì tốc độ của chương trình tương đối chậm.

Trong phần này ta sẽ trình bày một lớp các thuật toán nhanh nhất cho tới nay để giải bài toán luồng cực đại, tên chung của các thuật toán này là thuật toán *đầy tiền luồng* (*preflow-push*).

Hãy hình dung mạng như một hệ thống đường ống dẫn nước từ với điểm phát s tới điểm thu t , các cung là các đường ống, súc chứa là lưu lượng đường ống có thể tải. Nước chảy theo nguyên tắc từ chỗ cao về chỗ thấp. Với một lượng nước lớn phát ra từ s tới một đỉnh v , nếu có cách chuyển lượng nước đó sang địa điểm khác thì không có vấn đề gì, nếu không thì có hiện tượng “tràn” xảy ra tại v , ta “dâng cao” điểm v để lượng nước đó đổ sang điểm khác (có thể đổ ngược về s).

Cứ tiếp tục quá trình như vậy cho tới khi không còn hiện tượng tràn ở bất cứ điểm nào. Cách tiếp cận này hoàn toàn khác với thuật toán Ford-Fulkerson: thuật toán Ford-Fulkerson cố gắng tìm một dòng chảy phụ từ s tới t và thêm dòng chảy này vào luồng hiện có đến khi không còn dòng chảy phụ nữa.

a) Tiền luồng

Cho một mạng $G = (V, E, c, s, t)$. Một *tiền luồng* (*preflow*) trên G là một hàm:

$$\begin{aligned} f: E &\rightarrow \mathbb{R} \\ e &\mapsto f(e) \end{aligned}$$

gán cho mỗi cung $e \in E$ một số thực $f(e)$ thỏa mãn ba ràng buộc:

- Ràng buộc về sức chứa (capacity constraint): tiền luồng trên mỗi cung không được vượt quá sức chứa của cung đó: $\forall e \in E: f(e) \leq c(e)$.
- Ràng buộc về tính đối xứng lệch (skew symmetry): Với $\forall e \in E$, tiền luồng trên cung e và cung đối $-e$ có cùng giá trị tuyệt đối nhưng trái dấu nhau: $f(e) = -f(-e)$.
- Ràng buộc về tính dư: Với mọi đỉnh không phải đỉnh phát, tổng tiền luồng trên các cung đi vào đỉnh đó là số không âm: $\forall v \in V - \{s\}: f(V, \{v\}) = \sum_{e \in \{V \rightarrow \{v\}\}} f(e) \geq 0$.

Với $\forall v \in V$, ta gọi lượng tràn tại v , ký hiệu $excess[v]$, là tổng tiền luồng trên các cung đi vào đỉnh v :

$$excess[v] = f(V, \{v\}) = \sum_{e \in \{V \rightarrow \{v\}\}} f(e)$$

Đỉnh $v \in V - \{s, t\}$ gọi là *đỉnh tràn* (*overflowing vertex*) nếu $excess[v] > 0$. Khái niệm đỉnh tràn chỉ có nghĩa với các đỉnh không phải đỉnh phát cũng không phải đỉnh thu.

```
function Overflow(v ∈ V) : Boolean;
begin
    Result := (v ≠ s) and (v ≠ t) and (excess[u] > 0);
end;
```

Định nghĩa về tiền luồng tương tự như định nghĩa luồng, chỉ khác nhau ở ràng buộc thứ ba. Vì vậy chúng ta cũng có khái niệm mạng thặng dư, cung thặng dư, đường thặng dư... ứng với tiền luồng tương tự như đối với luồng.

b) Khởi tạo

Cho f là một tiền luồng trên mạng $G = (V, E, c, s, t)$. Ta gọi $h: V \rightarrow \mathbb{N}$ là một hàm độ cao ứng với f nếu h gán cho mỗi đỉnh $v \in V$ một số tự nhiên $h(v)$ thỏa mãn ba điều kiện:

- $h(s) = |V|$.
- $h(t) = 0$.
- $h(u) \leq h(v) + 1$ với mọi cung thặng dư (u, v) .

Những ràng buộc này gọi là **ràng buộc độ cao**.

Hàm độ cao h khi cài đặt sẽ được xác định bởi tập các giá trị $\{h[v]\}_{v \in V}$ nên tùy theo từng trường hợp, ta có thể sử dụng ký hiệu $h(v)$ (nếu muốn nói tới giá trị hàm) hoặc $h[v]$ (nếu muốn nói tới một biến số).

Thao tác khởi tạo *Init* chịu trách nhiệm khởi tạo một tiền luồng và một hàm độ cao tương ứng. Một cách khởi tạo là đặt tiền luồng trên mỗi cung e đi ra khỏi s đúng bằng sức chứa $c(e)$ của cung đó (dĩ nhiên sẽ phải đặt cả tiền luồng trên cung đối $-e$ bằng $-c(e)$ để thỏa mãn tính đối xứng lệch), còn tiền luồng trên các cung khác bằng 0. Khi đó tất cả các cung đi ra khỏi s là bão hòa.

$$f(e) = \begin{cases} c(e), & \text{nếu } e \in E^+(s) \\ -c(e), & \text{nếu } -e \in E^+(s) \\ 0, & \text{trường hợp khác} \end{cases}$$

Ta khởi tạo hàm độ cao $h: V \rightarrow \mathbb{N}$ như sau:

$$h(v) = \begin{cases} |V|, & \text{nếu } v = s \\ 0, & \text{nếu } v = t \\ 1, & \text{nếu } v \neq \{s, t\} \end{cases}$$

Rõ ràng mọi cung thặng dư (u, v) không thể là cung đi ra khỏi s ($u \neq s$) nên ta có $h(u) \leq 1 \leq h(v) + 1$. Hàm độ cao trên là thích ứng với tiền luồng f .

Việc cuối cùng là khởi tạo các giá trị *excess*[.] ứng với tiền luồng f .

procedure <i>Init</i> ;

```

begin
    //Khởi tạo tiền luồng
    for  $\forall e \in E$  do  $f[e] := 0$ ;
    for  $\forall v \in V$  do  $excess[v] := 0$ ;
    for  $\forall e = (s, v) \in E^+(s)$  do
        begin
             $f[e] := c(e)$ ;
             $f[-e] := -c(e)$ ;
             $excess[v] := excess[v] + c(e)$ ;
        end;
    //Khởi tạo hàm độ cao
    for  $\forall v \in V$  do  $h[v] := 1$ ;
     $h[s] := |V|$ ;
     $h[t] := 0$ ;
end;

```

c) Phép đẩy luồng

Phép đẩy luồng $Push(e)$ có thể thực hiện trên cung $e = (u, v)$ nếu các điều kiện sau được thỏa mãn:

- u là đỉnh tràn: $u \in V - \{s, t\}$ và $excess[u] > 0$
- e là cung thặng dư trên G_f : $c_f(e) = c(e) - f(e) > 0$
- u cao hơn v : $h(u) > h(v)$

Ràng buộc $h(u) > h(v)$ kết hợp với ràng buộc độ cao: $h(u) \leq h(v) + 1$ có thể viết thành $h(u) = h(v) + 1$.

Phép $Push(e = (u, v))$ sẽ tính lượng luồng tối đa có thể thêm vào cung e : $\Delta = \min\{excess[u], c_f(e)\}$, thêm lượng luồng này vào cung e và bớt một lượng luồng Δ từ v về u theo cung $-e$ để giữ tính đối xứng lệch của tiền luồng. Việc cuối cùng là cập nhật lại $excess[u]$ và $excess[v]$ theo tiền luồng mới. Bản chất của phép $Push(e = (u, v))$ là chuyển một lượng luồng tràn Δ từ đỉnh u sang đỉnh v . Để thấy rằng các tính chất của tiền luồng vẫn được duy trì sau phép $Push$:

```

procedure Push ( $e = (u, v)$ );
begin
     $\Delta := \min(excess[u], c_f(u, v))$ ; //Tính lượng luồng tối đa có thể đẩy

```

```

f[e] := f[e] + Δ; f[-e] := f[-e] - Δ; //Đẩy luồng
excess[u] := excess[u] - Δ;
excess[v] := excess[v] + Δ; //Cập nhật mức tràn
end;

```

Phép *Push* bảo tồn tính chất của hàm độ cao. Thật vậy, khi thao tác $Push(e = (u, v))$ được thực hiện, nó chỉ có thể sinh ra thêm một cung thăng dư $-e = (v, u)$ mà thôi. Phép *Push* không làm thay đổi các độ cao, tức là trước khi *Push*, $h[u] > h[v]$ thì sau khi *Push*, $h[v]$ vẫn nhỏ hơn $h[u]$, tức là ràng buộc độ cao $h[v] \leq h[u] + 1$ vẫn được duy trì trên cung thăng dư $-e = (v, u)$.

Phép $Push(e = (u, v))$ đẩy một lượng luồng $\Delta = \min\{excess[u], c_f(e)\}$ tràn từ u sang v . Nếu Δ đúng bằng $c_f(e) = c(e) - f(e)$ có nghĩa là khi phép *Push* tăng $f(e)$ lên Δ thì cung e sẽ bão hòa và không còn là cung thăng dư trên G_f nữa, ta gọi phép đẩy luồng này là *đẩy bão hòa (saturating push)*, ngược lại phép đẩy luồng đó gọi là *đẩy không bão hòa (non-saturating push)*, sau phép đẩy không bão hòa thì $excess[u] = 0$, tức là u không còn là đỉnh tràn nữa.

d) Phép nâng

Phép nâng $Lift(u)$ thực hiện trên đỉnh u nếu các điều kiện sau được thỏa mãn:

- u là đỉnh tràn: $(u \neq s), (u \neq t)$ và $excess[u] > 0$.
- u không chuyển được luồng xuống nơi nào thấp hơn: Với mọi cung thăng dư $e = (u, v) \in E_f$: $h(u) \leq h(v)$.

Khi đó phép $Lift(u)$ nâng đỉnh u lên bằng cách đặt $h[u]$ bằng độ cao thấp nhất của một đỉnh v nó có thể chuyển tải sang cộng thêm 1:

$$h[u] := \min\{h[v] : \exists (u, v) \in E_f\} + 1$$

```

procedure Lift (u∈V);
begin
    minH := +∞;
    for ∀v: (u, v) ∈ Ef do
        if h[v] < minH then minH := h[v];
        h[u] := minH + 1;
    end;

```

Nếu u là đỉnh tràn thì ít nhất phải có một cung thặng dư đi ra khỏi u , điều này đảm bảo cho phép lấy $\min\{h[v]: (u, v) \in E_f\}$ được thực hiện trên một tập khác rỗng. Thật vậy, do u là đỉnh tràn, ta có $excess[u] = \sum_{e \in \{V \rightarrow \{u\}\}} f(e) > 0$ tức là ít nhất có một cung $e \in \{V \rightarrow \{u\}\}$ để $f(e) > 0$. Cung đối $-e$ chắc chắn là một cung thặng dư đi ra khỏi u bởi:

$$c_f(-e) = c(-e) - f(-e) = c(-e) + f(e) > 0$$

Phép *Lift* không động chạm gì đến tiền luồng f . Ngoài ra phép ***Lift chỉ tăng độ cao của một đỉnh*** và bảo tồn ràng buộc độ cao: Với một cung thặng dư (v, u) đi vào u , ràng buộc độ cao $h(v) \leq h(u) + 1$ không bị vi phạm nếu ta nâng độ cao $h(u)$ của đỉnh u . Mặt khác, với một cung thặng dư (u, v) đi ra khỏi u thì việc đặt $h[u] := \min\{h[v]: \exists (u, v) \in E_f\} + 1$ cũng đảm bảo rằng $h(u) \leq h(v) + 1$.

e) Mô hình chung và thuật toán FIFO Preflow-Push

□ ***Mô hình chung***

Thuật toán đầy tiền luồng có mô hình cài đặt chung khá đơn giản: Khởi tạo tiền luồng f và hàm độ cao, sau đó nếu thấy phép nâng (*Lift*) hay đầy luồng (*Push*) nào thực hiện được thì thực hiện nga... Cho tới khi không còn phép nâng hay đầy nào có thể thực hiện được nữa thì tiền luồng f sẽ trở thành luồng cực đại trên mạng.

Chính vì thứ tự các phép *Push* và *Lift* được thực hiện không ảnh hưởng tới tính đúng đắn của thuật toán nên người ta đã đề xuất rất nhiều cơ chế chọn thứ tự thực hiện nhằm giảm thời gian thực hiện giải thuật.

Bồ đề 3-15

Cho mạng $G = (V, E, c, s, t)$ có tiền luồng f và hàm độ cao h . Với một đỉnh tràn u , luôn có thể thực hiện được thao tác *Push*(e) trên một cung e đi ra khỏi u hoặc thực hiện được thao tác *Lift*(u)

Chứng minh

Nếu thao tác *Push* không thể áp dụng được cho cung thặng dư nào đi ra khỏi u tức là với mọi cung thặng dư $(u, v) \in E_f$, $h(u)$ không cao hơn $h(v)$, điều đó chính là điều kiện hợp lệ để thực hiện thao tác *Lift*(u).

□ Thuật toán FIFO Preflow-Push

Định lý 3-17 là cơ sở cho thuật toán FIFO Preflow-Push. Thuật toán được Goldberg đề xuất [18] dựa trên cơ chế xử lý đỉnh tràn lấy ra từ một hàng đợi.

Tại thao tác khởi tạo, các đỉnh tràn sẽ được lưu trữ trong một hàng đợi *Queue* hỗ trợ hai thao tác: *PushToQueue(v)* để đẩy một đỉnh tràn v vào hàng đợi và *PopFromQueue* để lấy một đỉnh tràn khỏi hàng đợi. Thuật toán sẽ xử lý từng đỉnh tràn z lấy ra khỏi hàng đợi theo cách sau: Trước hết cố gắng đẩy luồng trên các cung thặng dư đi ra khỏi z bằng phép *Push*. Nếu đẩy được hết lượng tràn ($excess[z] = 0$) thì xong, nếu không ta nâng cao đỉnh z bằng phép *Lift(z)* và đẩy lại z vào hàng đợi chờ xử lý sau. Thuật toán sẽ tiếp tục với đỉnh tràn tiếp theo trong hàng đợi và kết thúc khi hàng đợi rỗng, bởi khi mạng không còn đỉnh tràn thì không còn thao tác *Push* hay *Lift* nào có thể thực hiện được nữa.

Giả sử rằng chúng ta có một đỉnh tràn u và một cung $e = (u, v)$ không thể đẩy luồng được, tức là ít nhất một trong hai điều kiện sau đây được thỏa mãn:

- (u, v) là cung bão hòa $c(e) = f(e)$.
- u không cao hơn v : $h(u) \leq h(v)$.

Khi đó:

- Sau bất kỳ phép *Push* nào, chúng ta vẫn không thể đẩy luồng được trên cung $e = (u, v)$. Thật vậy, nếu u không cao hơn v , phép *Push* không làm thay đổi hàm độ cao nên sau phép *Push* thì u vẫn không cao hơn v . Nếu u cao hơn v thì e phải là cung bão hòa, lệnh *Push* duy nhất có thể biến nó thành cung thặng dư là lệnh *Push*($-e$) làm giảm $f(e)$. Nhưng lệnh *Push*($-e$) không thể thực hiện được vì cung $-e = (v, u)$ có v thấp hơn u .
- Sau bất kỳ phép *Lift* nào ngoại trừ *Lift(u)*, chúng ta cũng không thể đẩy luồng được trên cung $e = (u, v)$. Bởi phép *Lift* không làm thay đổi tiền luồng trên các cung, tính bão hòa hay thặng dư của các cung được giữ nguyên. Như vậy nếu (u, v) đang bão hòa thì sau phép *Lift* nó vẫn bão hòa và không thể đẩy luồng được. Nếu (u, v) là cung thặng dư thì u đang không cao hơn v , lệnh *Lift* duy nhất có thể khiến u cao hơn v là lệnh *Lift(u)*.

Hai nhận định trên cho phép ta xây dựng một cấu trúc dữ liệu hiệu quả để cài đặt thuật toán:

Tương tự như chương trình cài đặt thuật toán Edmonds-Karp, ta sử dụng mảng $e[-m \dots m]$ chứa các cung, mảng $link[m \dots m]$ chứa mốc nối trong danh sách liên thuộc và mảng $head[1 \dots n]$ chứa chỉ số cung đầu tiên của các danh sách liên thuộc. Ngoài ra thuật toán duy trì một mảng chỉ số $current[1 \dots n]$, ở đây $current[v]$ là chỉ số của một cung nào đó trong danh sách liên thuộc các cung đi ra khỏi v , ban đầu $current[v]$ được gán bằng $head[v]$ với mọi đỉnh $v \in V$.

```

type
    TEdge = record //Cấu trúc một cung
        x, y: Integer; //Hai đỉnh đầu mút
        c, f: Integer; //Sức chứa và luồng
    end;
var
    e: array[-maxM..maxM] of TEdge; //Danh sách các cung
    link: array[-maxM..maxM] of Integer;
    //Mốc nối trong danh sách liên thuộc
    head, current: array[1..maxN] of Integer;

```

Trên cấu trúc dữ liệu này, danh sách mốc nối các nút chứa các cung đi ra khỏi z là:

$$e[i_1], e[i_2], e[i_3], \dots$$

Trong đó $i_1 = head[z]$, $i_2 = link[i_1]$, $i_3 = link[i_2], \dots$

Thuật toán FIFO Preflow-Push sẽ xử lý lần lượt từng đỉnh tràn lấy ra khỏi hàng đợi. Với mỗi đỉnh tràn z lấy khỏi hàng đợi, cung $e[current[z]]$ là một cung đi ra khỏi z, giả sử cung đó là (z, v) . Nếu phép đẩy luồng (Push) trên cung đó có thể thực hiện được thì thực hiện ngay, đồng thời đẩy v vào hàng đợi nếu v chưa có trong hàng đợi. Nếu phép đẩy luồng này làm z hết tràn thì chuyển sang xử lý đỉnh tràn kế tiếp trong hàng đợi, ngược lại nếu z vẫn còn là đỉnh tràn (tức là không thể đẩy luồng trên cung $e[current[z]]$ nữa), ta dịch chỉ số $current[z]$ sang cung kế tiếp trong danh sách liên thuộc ($current[z] := link[current[z]]$) để chuyển sang xét một cung khác... Khi dịch chỉ số $current[z]$ đến hết danh sách liên thuộc mà z vẫn tràn, đỉnh z sẽ được nâng lên bằng phép $Lift(z)$, chỉ số $current[x]$ được đặt trở lại bằng $head[z]$ để nó trở lại về đầu danh sách liên thuộc. Đỉnh z sau đó được đẩy lại vào hàng đợi chờ xử lý sau...

Tính hợp lý của thuật toán nằm ở chỗ : khi đỉnh tràn z bắt đầu được xử lý, tất cả những cung đứng trước cung $e[current[z]]$ đều không thể đẩy luồng được. Tức là nếu muốn đẩy luồng ra khỏi z thì chỉ cần xét các cung từ $e[current[z]]$ trở đi là đủ, không cần duyệt từ đầu danh sách liên thuộc.

```

procedure FIFOPreflowPush;
begin
    Init; //Khởi tạo tiền luồng, độ cao, hàng đợi Queue chứa các đỉnh tràn
    while Queue ≠ Ø do
        begin
            z := PopFromQueue; //Xử lý đỉnh tràn x lấy ra từ hàng đợi
            while current[z] <> 0 do //Có gắng đẩy luồng khỏi z
                begin //Xét cung (z, v) chưa trong nút e[current[z]]
                    v := e[current[z]].y;
                    if «Có thẻ đẩy luồng trên cung (z, v)» then
                        begin
                            NeedQueue := (v ≠ s) and (v ≠ t)
                                and (excess[v] = 0);
                            Push(z, v); //Đẩy luồng
                            if NeedQueue then
                                //Sau phép đẩy, v đang không tràn trở thành tràn
                                PushToQueue(v); //Đẩy v vào hàng đợi chờ xử lý
                            if excess[z] = 0 then Break;
                                //Sau phép đẩy mà z hết tràn thì dừng đẩy
                            end;
                            current[z] := link[current[z]];
                            //z chưa hết tràn, chuyển sang xét cung liên thuộc tiếp theo
                        end;
                    if excess[z] > 0 then //Duyệt hết danh sách liên thuộc mà x vẫn tràn
                        begin
                            Lift(z); //Dâng cao z
                            current[z] := head[z];
                            //Đặt lại chỉ số current[z] về nút đầu danh sách liên thuộc
                            PushToQueue(z); //Đẩy z vào hàng đợi chờ xử lý sau
                        end;
                    end;
                end;

```

Từ nhận xét trên, có thể nhận thấy rằng những phép *Push* và *Lift* trong mô hình cài đặt đảm bảo được gọi tại những thời điểm mà những điều kiện cần để thực thi chúng được thỏa mãn.

f) Tính đúng của thuật toán

Sau mỗi bước của vòng lặp chính, hàng đợi *Queue* luôn chứa danh sách các đỉnh tràn và thuật toán sẽ kết thúc khi không còn đỉnh tràn nào trên mạng. Với $\forall v \in V - \{s, t\}$, ta có:

$$f(\{v\}, V) = -f(V, \{v\}) = -\text{excess}[v] = 0$$

Tức là với $\forall v \in V - \{s, t\}$ thì tổng luồng trên các cung đi ra khỏi v bằng 0, điều này chỉ ra rằng khi thuật toán kết thúc, tiền luồng chúng ta duy trì trên mạng trở thành một luồng.

Định lý 3-16

Cho f là một tiền luồng trên mạng $G = (V, E, c, s, t)$, nếu tồn tại một hàm độ cao $h: V \rightarrow \mathbb{N}$ ứng với f thì mạng thặng dư G_f không có đường tăng luồng.

Chứng minh

Nhắc lại về ràng buộc độ cao: $h(s) = |V|$, $h(t) = 0$ và với mọi cung thặng dư (u, v) thì $h(u) \leq h(v) + 1$. Giả sử phản chứng rằng có đường tăng luồng $\langle s = v_0, v_1, \dots, v_k = t \rangle$ trên mạng thặng dư G_f đi qua k cung thặng dư. Khi đó:

$$h(v_0) \leq h(v_1) + 1 \leq h(v_2) + 2 \leq \dots \leq h(v_k) + k$$

hay

$$\underbrace{h(s)}_{|V|} \leq \underbrace{h(t)}_0 + k$$

Ta có $|V| \leq k$, nhưng đường tăng luồng phải là đường đi đơn, tức là qua không quá $|V| - 1$ cạnh, vậy $k \leq |V| - 1$. Điều này mâu thuẫn, nghĩa là không thể tồn tại đường tăng luồng trên G_f .

Định lý 3-17 và Định lý 3-11 (mối quan hệ giữa luồng cực đại, đường tăng luồng và lát cắt hẹp nhất) chỉ ra rằng: thuật toán đầy tiền luồng trả về một luồng và một hàm độ cao ứng với luồng đó nên luồng trả về chắc chắn là luồng cực đại.

g) Tính dừng của thuật toán

Tính dừng của thuật toán đầy tiền luồng ở trên sẽ được suy ra khi chúng ta phân tích thời gian thực hiện giải thuật. Tương tự như thuật toán Ford-Fulkerson, chúng ta sẽ không phân tích thời gian thực hiện trên mô hình tổng quát mà chỉ phân tích thời gian thực hiện giải thuật FIFO Preflow-Push mà thôi.

Định lý 3-17

Cho f là một tiền luồng trên mạng $G = (V, E, c, s, t)$, khi đó với mọi đỉnh tràn u , tồn tại một đường thặng dư đi từ u tới s .

Chứng minh

Với một đỉnh tràn u bất kỳ, xét tập X là tập các đỉnh có thể đến được từ u bằng một đường thặng dư. Đặt $Y = V - X$ là tập những đỉnh nằm ngoài X . Trước hết ta chỉ ra rằng tiền luồng trên các cung thuộc $\{Y \rightarrow X\}$ không thể là số dương. Thật vậy nếu có $e \in \{Y \rightarrow X\}$ mà $f(e) > 0$ thì $-e \in \{X \rightarrow Y\}$ và $f(-e) < 0$. Suy ra có cung thặng dư $-e$ nối một đỉnh thuộc X với một đỉnh y nào đó thuộc Y . Theo cách xây dựng tập X , y sẽ phải là đỉnh thuộc X . Mâu thuẫn.

Tiền luồng trên các cung thuộc $\{Y \rightarrow X\}$ không thể là số dương thì $f(Y, X) \leq 0$. Ta xét tổng mức tràn của các đỉnh $\in X$:

$$\text{excess}(X) = f(V, X) = \underbrace{f(X, X)}_0 + \underbrace{f(Y, X)}_{\leq 0} \leq 0$$

Lượng tràn tại mỗi đỉnh không phải đỉnh phát đều là số không âm, ngoài ra u là đỉnh tràn $\in X$ nên $\text{excess}[u] > 0$, điều này cho thấy chắc chắn đỉnh phát s phải thuộc X để $\text{excess}(X) \leq 0$. Nói cách khác từ u đến được s bằng một đường thặng dư.

Hệ quả

Cho mạng $G = (V, E, c, s, t)$. Giả sử chúng ta thực hiện thuật toán đầy tiền luồng với hàm độ cao $h: V \rightarrow \mathbb{N}$ thì độ cao của các đỉnh trong quá trình thực hiện giải thuật không vượt quá $2|V| - 1$.

Chứng minh

Mạng phải có ít nhất một đỉnh phát và một đỉnh thu nén $|V| \geq 2$. Ban đầu, $h(s) = |V|$, $h(t) = 0$ và $h(v) = 1, \forall v \notin \{s, t\}$ nên độ cao của các đỉnh đều nhỏ hơn $2|V| - 1$.

Độ cao của s và t không bao giờ bị thay đổi và với mỗi đỉnh $u \in V - \{s, t\}$ thì chỉ phép $\text{Lift}(u)$ có thể làm tăng độ cao của đỉnh u . Điều kiện để thực hiện phép

$Lift(u)$ là u phải là đỉnh tràn. Phép $Lift$ không thay đổi tiền luồng nên sau phép $Lift(u)$ thì u vẫn tràn. Áp dụng kết quả của Định lý 3-17, tồn tại đường đi đơn từ u tới s $\langle u = v_0, v_1, \dots, v_k = s \rangle$ chỉ đi qua k cung thăng dư $(v_0, v_1), (v_1, v_2), \dots, (v_{k-1}, v_k)$. Từ ràng buộc độ cao ta có:

$$h(u) = h(v_0) \leq h(v_1) + 1 \leq h(v_2) + 2 \leq \dots \leq h(v_k) + k \leq |V| + k$$

Đường đi đơn thì không qua nhiều hơn $|V| - 1$ cạnh nên ta có $k \leq |V| - 1$, kết hợp lại có $h(u) \leq 2|V| - 1$. ĐPCM.

Định lý 3-18 (thời gian thực hiện giải thuật FIFO Preflow-Push)

Có thể cài đặt giải thuật FIFO Preflow-Push để tìm luồng cực đại trên mạng $G = (V, E, c, s, t)$ trong thời gian $O(|V|^3 + |V||E|)$.

Chứng minh

Ta sẽ chứng minh mô hình cài đặt thuật toán FIFO Preflow-Push ở trên có thời gian thực hiện là $O(|V|^3 + |V||E|)$. Vòng lặp chính của thuật toán mỗi lượt lấy một đỉnh tràn z khỏi hàng đợi và cố gắng tháo luồng cho đỉnh z bằng các phép $Push$ theo các cung đi ra khỏi z . Nếu z chưa hết tràn thì thực hiện phép $Lift(z)$ và đẩy lại z vào hàng đợi. Như vậy thuật toán FIFO Preflow-Push sẽ thực hiện một dãy các phép $Lift$ và $Push$:

$$Lift(.), Push(.), Push(.), \dots, Push(.), Lift(.), Push(.), \dots$$

Trước hết ta chứng minh rằng số phép $Lift$ trong dãy thao tác trên là $O(|V|^2)$ và tổng thời gian thực hiện chúng là $O(|V||E|)$. Thật vậy, Mỗi phép $Lift$ sẽ nâng độ cao của một đỉnh lên ít nhất 1, ngoài ra độ cao của mỗi đỉnh không vượt quá $2|V| - 1$ (theo hệ quả của định lý Định lý 3-17). Cứ cho là mọi đỉnh $\in V - \{s, t\}$ khi kết thúc thuật toán đều có độ cao $2|V| - 1$ đi nữa thì do chúng được khởi tạo bằng 1, tổng số phép $Lift$ cần thực hiện cũng không vượt quá:

$$(|V| - 2)(2|V| - 2) = O(|V|^2)$$

Mỗi cung (u, v) sẽ được xét đến đúng một lần trong phép $Lift(u)$, phép $Lift(u)$ lại được gọi không quá $2|V| - 2$ lần. Vậy tổng cộng trong tất cả các phép $Lift$ thì mỗi cung sẽ được xét không quá $2|V| - 2$ lần, mạng có $|E|$ cung suy ra tổng thời gian thực hiện của các phép $Lift$ trong giải thuật là $|E|(2|V| - 2) = O(|V||E|)$.

Tiếp theo ta chứng minh rằng số phép đẩy bão hòa cũng như tổng thời gian thực hiện chúng là $O(|V||E|)$. Sau phép đẩy bão hòa $Push(e)$, nếu muốn thực hiện tiếp phép đẩy $Push(e)$ nữa thì trước đó chắc chắn phải có phép đẩy $Push(-e)$ để làm giảm $f(e)$ và biến e trở lại thành cung thăng dư. Giả sử $e = (u, v)$ và $-e = (v, u)$ thì để thực hiện phép $Push(e)$, ta phải có $h(u) > h(v)$. Để thực hiện $Push(-e)$, ta phải có $h(v) > h(u)$ và để thực hiện tiếp $Push(e)$ nữa ta lại phải có $h(u) >$

$h(v)$. Bởi độ cao của các đỉnh không bao giờ giảm đi nên sau phép $Push(e)$ thứ hai, độ cao $h(u)$ lớn hơn ít nhất 2 đơn vị so với $h(u)$ ở phép $Push(e)$ thứ nhất. Vậy nếu một cung $e = (u, v)$ của mạng được đầy bão hòa k lần thì độ cao của đỉnh u sẽ tăng lên ít nhất là $2(k - 1)$. Vì độ cao của các đỉnh không vượt quá $2|V| - 1$ nên số phép đầy bão hòa trên mỗi cung e là $k \leq |V|$. Mạng có $|E|$ cung và thời gian thực hiện phép $Push$ là $O(1)$ nên số phép đầy bão hòa là $O(|V||E|)$ và thời gian thực hiện chúng cũng là $O(|V||E|)$.

Đối với các phép đầy không bão hòa, việc đánh giá thời gian thực hiện giải thuật được thực hiện bằng *hàm tiềm năng* (*potential function*). Định nghĩa hàm tiềm năng Φ là độ cao lớn nhất của các đỉnh tràn:

$$\Phi = \max\{h[v] : v \text{ là đỉnh tràn}\}$$

Trong trường hợp mạng không còn đỉnh tràn thì ta quy ước $\Phi = 0$. Vậy $\Phi \leq 1$ khi khởi tạo tiền luồng và trở lại bằng 0 khi thuật toán kết thúc.

Chia dãy các thao tác *Lift* và *Push* làm các pha liên tiếp. Pha thứ nhất bắt đầu khi hàng đợi được khởi tạo gồm các đỉnh tràn và kết thúc khi tất cả các đỉnh đó (và chỉ những đỉnh đó thôi) đã được lấy ra khỏi hàng đợi và xử lý. Pha thứ hai tiếp tục với hàng đợi gồm những đỉnh được đẩy vào trong pha thứ nhất và kết thúc khi tất cả các đỉnh này được lấy ra khỏi hàng đợi và xử lý, pha thứ ba, thứ tư... được chia ra theo cách tương tự như vậy.

Nhận xét rằng phép *Push* chỉ đẩy luồng từ đỉnh cao xuống đỉnh thấp, vậy nên những đỉnh được đẩy vào hàng đợi sau phép *Push* luôn thấp hơn đỉnh đang xét vừa lấy ra khỏi hàng đợi. Suy ra nếu một pha chỉ chứa phép *Push* thì giá trị hàm tiềm năng Φ sau pha đó giảm đi ít nhất 1 đơn vị.

Giá trị hàm tiềm năng Φ chỉ **có thể** tăng lên sau một pha nếu pha đó có chứa phép *Lift* và giá trị Φ tăng lên phải bằng một độ cao của một đỉnh v nào đó sau phép *Lift*(v) trong pha. Xét mức tăng của Φ sau pha đang xét:

$$\Phi_{\text{mới}} - \Phi_{\text{cũ}} = h(v)_{\text{mới}} - \Phi_{\text{cũ}} \leq h(v)_{\text{mới}} - h(v)_{\text{cũ}}$$

Tức là sau mỗi pha làm Φ tăng lên, luôn tồn tại một đỉnh v mà mức tăng độ cao của v lớn hơn mức tăng của Φ . Xét trên toàn bộ giải thuật, độ cao của mỗi đỉnh $v \in V - \{s, t\}$ được khởi tạo bằng 0 và được nâng lên tối đa bằng $2|V| - 1$ nên tổng toàn bộ mức tăng của các đỉnh không vượt quá $(|V| - 2)(2|V| - 1) = O(|V|^2)$.

Vậy nếu ta xét các pha làm Φ tăng thì tổng mức tăng của Φ trên các pha này là $O(|V|^2)$, tức là số các pha làm Φ giảm cũng phải là $O(|V|^2)$. Nói cách khác, sẽ chỉ có $O(|V|^2)$ pha có chứa phép *Lift* và $O(|V|^2)$ pha không chứa phép *Lift*. Cộng lại ta có số pha cần thực hiện trong toàn bộ giải thuật là $O(|V|^2)$.

Một pha sẽ phải lấy khỏi hàng đợi tối đa $|V| - 2$ đỉnh để xử lý. Với mỗi đỉnh lấy từ hàng đợi, việc tháo luồng sẽ chỉ sử dụng tối đa 1 phép đẩy không bão hòa vì sau phép đẩy này thì đỉnh sẽ hết tràn và quá trình xử lý sẽ chuyển sang đỉnh tiếp theo trong hàng đợi. Vậy trong mỗi pha có không quá $|V| - 2$ phép đẩy không bão hòa. Vì tổng số pha là $O(|V|^2)$, ta có số phép đẩy không bão hòa trong cả giải thuật là $O(|V|^3)$ và tổng thời gian thực hiện chúng cũng là $O(|V|^3)$.

Cuối cùng, ta đánh giá thời gian thực hiện những thao tác duyệt danh sách liên thuộc bằng các chỉ số $current[.]$ trong thuật toán FIFO Preflow-Push. Với mỗi đỉnh z , chỉ số $current[z]$ ban đầu sẽ ứng với nút đầu danh sách liên thuộc và chuyển dần đến hết danh sách gồm $\text{deg}^+(z)$ nút. Khi duyệt hết danh sách liên thuộc mà z vẫn tràn thì sẽ có một phép $Lift(z)$ và con trỏ $current[z]$ được đặt lại về đầu danh sách liên thuộc. Số phép $Lift(z)$ trong toàn bộ giải thuật không vượt quá $2|V| - 1$, nên số lượt dịch chỉ số $current[z]$ không vượt quá $2|V| \text{deg}^+(z)$. Suy ra nếu xét tổng thể, số phép dịch các chỉ số $current[.]$ trên tất cả các danh sách liên thuộc phải nhỏ hơn:

$$(2|V|) \sum_{z \in V} \text{deg}^+(z) = 2|V||E| = O(|V||E|)$$

Kết luận:

Tổng thời gian thực hiện các phép nâng: $O(|V||E|)$.

Tổng thời gian thực hiện các phép đẩy bão hòa: $O(|V||E|)$.

Tổng thời gian thực hiện các phép đẩy không bão hòa: $O(|V|^3)$.

Tổng thời gian thực hiện các phép duyệt danh sách liên thuộc bằng chỉ số $current[.]$: $O(|V||E|)$.

Thời gian thực hiện giải thuật FIFO Preflow-Push: $O(|V|^3 + |V||E|)$.

h) Một số kỹ thuật tăng tốc độ giải thuật

Ta đã chứng minh rằng thuật toán Edmonds-Karp có thời gian thực hiện $O(|V||E|^2)$ và thuật toán FIFO Preflow-Push có thời gian thực hiện $O(|V|^3 + |V||E|)$. Những đại lượng này thoạt nhìn làm chúng ta có cảm giác như thuật toán FIFO Preflow-Push thực hiện nhanh hơn thuật toán Edmonds-Karp, đặc biệt trong trường hợp đồ thị dày ($|E| \gg |V|$).

Tuy vậy, những đánh giá này chỉ là cận trên của thời gian thực hiện giải thuật trong trường hợp xấu nhất. Hiện tại chưa có các đánh giá chặt chẽ cận trên và cận dưới trong trường hợp trung bình. Những thử nghiệm bằng chương trình cụ thể cũng cho thấy rằng các thuật toán đẩy tiền luồng như FIFO Preflow-Push, Lift-

to-Front Preflow-Push, Highest-Label Preflow-Push... không có cải thiện gì về tốc độ so với thuật toán Edmonds-Karp (thậm chí còn chậm hơn) nếu không sử dụng những mẹo cài đặt (*heuristics*).

Chưa có đánh giá lý thuyết chặt chẽ nào về tác động của những mẹo cài đặt lên thời gian thực hiện giải thuật nhưng hầu hết các thử nghiệm đều cho thấy việc sử dụng những mẹo cài đặt trên thực tế gần như là bắt buộc đối với các thuật toán đẩy tiền luồng.

□ *Bản chất của hàm độ cao*

Nhắc lại về ràng buộc độ cao: Xét một tiền luồng f trên mạng $G = (V, E, c, s, t)$, hàm độ cao $h: V \rightarrow \mathbb{N}$ gọi là tương ứng với tiền luồng f nếu $h(s) = |V|$; $h(t) = 0$; và với mọi cung thặng dư (u, v) thì $h(u) \leq h(v) + 1$.

Nếu ta gán trọng số cho các cung của mạng thặng dư G_f theo quy tắc: Cung thặng dư có trọng số 1 và cung bão hòa có trọng số $+\infty$. Ký hiệu $\delta_f(u, v)$ là độ dài đường đi ngắn nhất từ u tới v trên G_f với cách gán trọng số này. Khi đó không khó khăn kiểm chứng được rằng với $\forall v \in V - \{s, t\}$:

- $h(v) \leq \delta_f(v, t)$, tức là $h(v)$ luôn là cận dưới của độ dài đường đi ngắn nhất từ v tới đỉnh thu.
- Trong trường hợp $h(v) > |V|$, từ v chắc chắn không có đường thặng dư đi tới t và $h(v) - |V| \leq \delta_f(v, s)$, tức là $h(v) - |V|$ trong trường hợp này là cận dưới của độ dài đường đi ngắn nhất từ v về đỉnh phát.

Những mẹo cài đặt dưới đây nhằm đẩy nhanh các độ cao $h(v)$ trong tiến trình thực hiện giải thuật dựa vào những nhận xét trên.

□ *Gán nhãn lại toàn bộ*

Nội dung của phương pháp gán nhãn lại toàn bộ (*global relabeling heuristic*) được tóm tắt như sau: Xét lát cắt chia tập V làm hai tập rời nhau X và Y : Tập Y gồm những đỉnh đến được t bằng một đường thặng dư và tập X gồm những đỉnh còn lại. Chắc chắn không có cung thặng dư nối từ X sang Y , ta có $s \in X, t \in Y$. Phép gán nhãn lại toàn bộ sẽ đặt:

- Với $\forall v \in Y$, ta gán lại độ cao $h[v] := \delta_f(v, t)$.
- Với $\forall u \in X$ và $\delta_f(u, s) < +\infty$, ta gán lại độ cao $h[u] := |V| + \delta_f(u, s)$

- Với $\forall u \in X$ và $\delta_f(u, s) = +\infty$, ta gán lại độ cao $h[u] := 2|V| - 1$

Không khó khăn để kiểm chứng tính hợp lý của hàm độ cao mới. Có thể thấy rằng các độ cao mới ít ra là không thấp hơn các độ cao cũ.

Các giá trị $\delta_f(v, t)$ cũng như $\delta_f(u, s)$ có thể được xác định bằng hai lượt thực hiện thuật toán BFS từ t và s . Bởi ta cần thời gian $O(|E|)$ cho hai lượt BFS và gán lại các độ cao, nên phép gán nhãn lại toàn bộ thường được gọi thực hiện sau một loạt chỉ thị sơ cấp để không làm ảnh hưởng tới đánh giá O lớn của thời gian thực hiện giải thuật (chẳng hạn sau mỗi $|V|$ phép *Lift*). Chú ý là khi nâng độ cao $h[z]$ của một đỉnh z nào đó, cần cập nhật lại $current[z] := head[z]$.

□ **Đẩy nhãn theo khe**

Phép đẩy nhãn theo khe (*gap heuristic*) được thực hiện nhò quan sát sau:

Giả sử ta có một số nguyên $0 < gap < |V|$ mà không đỉnh nào có độ cao gap (số nguyên gap này được gọi là “khe”), khi đó mọi đỉnh z có $h[z] > gap$ đều không có đường thặng dư đi đến t .

Nhận định trên có thể chứng minh bằng phản chứng: Giả sử từ z có đường thặng dư đi đến t , với một cung (u, v) trên đường đi ta có $h(u) \leq h(v) + 1$, tức là trên đường đi này, từ một đỉnh u ta chỉ có thể đi sang một đỉnh v không thấp hơn hoặc thấp hơn u đúng một đơn vị. Từ $h(z) > gap > 0$ và $h(t) = 0$, chắc chắn trên đường thặng dư từ x tới t phải có một đỉnh độ cao gap . Mâu thuẫn với giả thuyết phản chứng.

Phép đẩy nhãn theo khe nếu phát hiện khe $0 < gap < |V|$ sẽ xét tất cả những đỉnh $z \in V - \{s\}$ có $gap < h(z) \leq |V|$ và đặt lại $h[z] := |V| + 1$.

Ta sẽ chỉ ra rằng phép đẩy độ cao này vẫn đảm bảo ràng buộc độ cao của hàm h . Độ cao của đỉnh phát và đỉnh thu không bị động chạm đến, tức là $h[s] = |V|$ và $h[t] = 0$. Trước khi thực hiện phép đẩy theo khe, ta chia tập đỉnh V thành hai tập rời nhau: Tập X gồm những đỉnh cao hơn gap và tập Y gồm những đỉnh thấp hơn gap . Do ràng buộc độ cao $h(u) \leq h(v) + 1$ với mọi cung thặng dư (u, v) , không tồn tại cung thặng dư nối từ X tới Y . Phép đẩy theo khe chỉ tăng độ cao của một vài đỉnh $x \in X$ và như vậy ràng buộc độ cao nếu bị vi phạm thì chỉ bị vi phạm trên những cung thặng dư đi ra khỏi x . Như lập luận trên, cung thặng dư đi

ra khỏi x chắc chắn phải đi vào một đỉnh $x' \in X$ có độ cao ít nhất là $|V|$ sau phép đẩy theo khe. Từ $h(x) = |V| + 1$ ta có $h(x) \leq h(x') + 1$.

Phép đẩy nhãn theo khe sử dụng mảng $count[0 \dots 2|V| - 1]$ để đếm $count[k]$ là số đỉnh có độ cao k . Mỗi khi có sự thay đổi độ cao, ta phải đồng bộ lại mảng $count$ theo tình trạng hàm độ cao mới. Sau mỗi phép $Lift(u)$, độ cao cũ của đỉnh u được lưu trữ lại trong biến $OldH$ và phép $Lift$ thực hiện như bình thường. Sau đó nếu $0 < OldH < |V|$ và $count[OldH] = 0$, phép đẩy theo khe $OldH$ sẽ được gọi và thực hiện trong thời gian $O(|V|)$. Bởi số phép $Lift$ cần thực hiện trong toàn bộ giải thuật là $O(|V|^2)$, tổng thời gian thực hiện các phép đẩy theo khe sẽ là $O(|V|^3)$ nên không ảnh hưởng tới đánh giá 0-lớn của thời gian thực hiện giải thuật FIFO Preflow-Push.

Dưới đây là bảng so sánh tốc độ của các chương trình cài đặt cụ thể trên một số bộ dữ liệu. Với một cặp số n, m , 100 đồ thị với n đỉnh, m cung được sinh ngẫu nhiên với sức chứa là số nguyên trong khoảng từ 0 tới 10^4 . Có 4 chương trình được thử nghiệm: A: Thuật toán Edmonds-Karp, B: thuật toán FIFO Preflow-Push, C: thuật toán FIFO Preflow-Push với phép gán nhãn lại toàn bộ và D: thuật toán FIFO Preflow-Push với phép đẩy nhãn theo khe. Mỗi chương trình được thử trên cả 100 đồ thị và đo thời gian thực hiện trung bình (tính bằng giây):

	$n = 100$ $m = 10000$	$n = 200$ $m = 30000$	$n = 500$ $m = 40000$	$n = 800$ $m = 90000$	$n = 1000$ $m = 100000$
A	0.0688	0.5925	0.6598	1.7158	2.7629
B	0.0983	0.7395	3.4377	9.9014	25.0723
C	0.0313	0.0624	0.0857	0.1809	0.2433
D	0.0282	0.0577	0.0828	0.1575	0.1889

☐ Cài đặt

Dưới đây là chương trình cài đặt thuật toán FIFO Preflow-Push kết hợp với kỹ thuật đẩy nhãn theo khe, việc cài đặt và đánh giá hiệu suất của phép gán nhãn lại toàn bộ chúng ta coi như bài tập. Các bạn có thể thử cài đặt kết hợp cả hai kỹ thuật tăng tốc này để xác định xem việc đó có thực sự cần thiết không.

Input/Output có khuôn dạng giống như ở chương trình cài đặt thuật toán Edmonds-Karp. Hàng đợi chứa các đỉnh tràn được tổ chức dưới dạng danh sách vòng: Các chỉ số đầu/cuối hàng đợi sẽ chạy xuôi trong một mảng và khi chạy đến hết mảng sẽ tự động quay về đầu mảng.

FIFOPREFLOWPUSH.PAS ✓ Thuật toán FIFO Preflow-Push

```

{$MODE OBJFPC}
program MaximumFlow;
const
  maxN = 1000;
  maxM = 100000;
  maxC = 10000;
type
  TEdge = record //Cấu trúc một cung
    x, y: Integer; //Hai đỉnh đầu mút
    c, f: Integer; //Sức chứa và luồng
  end;
  TQueue = record //Cấu trúc hàng đợi
    items: array[0..maxN - 1] of Integer; //Danh sách vòng
    front, rear, nItems: Integer;
  end;
var
  e: array[-maxM..maxM] of TEdge; //Mảng chứa các cung
  link: array[-maxM..maxM] of Integer;
  //Môc nối trong danh sách liên thuộc
  head, current: array[1..maxN] of Integer;
  //con trỏ tới đầu và vị trí hiện tại của danh sách liên thuộc
  excess: array[1..maxN] of Integer; //mức tràn của các đỉnh
  h: array[1..maxN] of Integer; //hàm độ cao
  count: array[0..2 * maxN - 1] of Integer;
  //count[k] = số đỉnh có độ cao k
  Queue: TQueue; //Hàng đợi chứa các đỉnh tràn
  n, m, s, t: Integer;
  FlowValue: Integer;
procedure Enter; //Nhập dữ liệu
var
  i, u, v, capacity: Integer;
begin

```

```

ReadLn(n, m, s, t);
FillChar(head[1], n * SizeOf(head[1]), 0);
for i := 1 to m do
  begin
    ReadLn(u, v, capacity);
    with e[i] do //Thêm cung e[i] = (u, v) vào danh sách liên thuộc của u
      begin
        x := u;
        y := v;
        c := capacity;
        link[i] := head[u];
        head[u] := i;
      end;
    with e[-i] do //Thêm cung e[-i] = (v, u) vào danh sách liên thuộc của v
      begin
        x := v;
        y := u;
        c := 0;
        link[-i] := head[v];
        head[v] := -i;
      end;
    end;
    for v := 1 to n do current[v] := head[v];
  end;
procedure PushToQueue(v: Integer); //Đẩy một đỉnh v vào hàng đợi
begin
  with Queue do
    begin
      rear := (rear + 1) mod maxN;
      //Dịch chỉ số cuối hàng đợi, rear = maxN - 1 sẽ trở lại thành 0
      items[rear] := v; //Đặt v vào vị trí cuối hàng đợi
      Inc(nItems); //Tăng biến đếm số phần tử trong hàng đợi
    end;
  end;
function PopFromQueue: Integer; //Lấy một đỉnh khỏi hàng đợi
begin
  with Queue do
    begin

```

```

Result := items[front]; //Trả về phần tử ở đầu hàng đợi
front := (front + 1) mod maxN;
//Dịch chỉ số đầu hàng đợi, front = maxN - 1 sẽ trở lại thành 0
Dec(nItems); //Giảm biến đếm số phần tử trong hàng đợi
end;
end;

procedure Init; //Khởi tạo
var v, sf, i: Integer;
begin
//Khởi tạo tiền luồng
for i := -m to m do e[i].f := 0;
FillChar(excess[1], n * SizeOf(excess[1]), 0);
i := head[s];
while i <> 0 do
//Duyệt các cung đi ra khỏi đỉnh phát và đầy bão hòa các cung đó, cập nhật các mức tràn excess[].
begin
sf := e[i].c;
e[i].f := sf;
e[-i].f := -sf;
Inc(excess[e[i].y], sf);
Dec(excess[s], sf);
i := link[i];
end;
//Khởi tạo hàm độ cao
for v := 1 to n do h[v] := 1;
h[s] := n;
h[t] := 0;
//Khởi tạo các biến đếm: count[k] là số đỉnh có độ cao k
FillChar(count[0], (2 * n) * SizeOf(count[0]), 0);
count[n] := 1;
count[0] := 1;
count[1] := n - 2;
//Khởi tạo hàng đợi chứa các đỉnh tràn
Queue.front := 0;
Queue.rear := -1;
Queue.nItems := 0; //Hàng đợi rỗng
for v := 1 to n do //Duyệt tập đỉnh
  if (v <> s) and (v <> t) and (excess[v] > 0) then
    //v tràn

```

```

PushToQueue (v) ; //đẩy v vào hàng đợi
end;
procedure Push (i: Integer) ; //Phép đẩy luồng theo cung e[i]
var Delta: Integer;
begin
  with e[i] do
    if excess[x] < c - f then Delta := excess[x]
    else Delta := c - f;
    Inc(e[i].f, Delta);
    Dec(e[-i].f, Delta);
  with e[i] do
    begin
      Dec(excess[x], Delta);
      Inc(excess[y], Delta);
    end;
  end;
procedure SetH(u: Integer; NewH: Integer);
//Đặt độ cao của u thành NewH, đồng bộ hóa mảng count
begin
  Dec(count[h[u]] );
  h[u] := NewH;
  Inc(count[NewH]);
end;
procedure PerformGapHeuristic(gap: Integer);
//Đẩy nhän theo khe gap
var v: Integer;
begin
  if (0 < gap) and (gap < n) and (count[gap] = 0) then
    //gap đúng là khe thật
    for v := 1 to n do
      if (v <> s) and (gap < h[v]) and (h[v] <= n) then
        begin
          SetH(v, n + 1);
          current[v] := head[v];
          //Nâng độ cao của v cần phải cập nhật lại con trỏ current[v]
        end;
    end;
  procedure Lift(u: Integer); //Phép nâng đỉnh u
  var minH, OldH, i: Integer;

```

```

begin
    minH := 2 * maxN;
    i := head[u];
    while i <> 0 do //Duyệt các cung đi ra khỏi u
        begin
            with e[i] do
                if (c > f) and (h[y] < minH) then
                    //Gặp cung tăng dư (u, v), ghi nhận đỉnh v thấp nhất
                    minH := h[y];
                    i := link[i];
            end;
            OldH := h[u]; //Nhớ lại h[u] cũ
            SetH(u, minH + 1); //nâng cao đỉnh u
            PerformGapHeuristic(OldH); //Có thể tạo ra khe OldH, đẩy nhãn theo khe
        end;
    procedure FIFOPreflowPush; //Thuật toán FIFO Preflow-Push
    var
        NeedQueue: Boolean;
        z: Integer;
    begin
        while Queue.nItems > 0 do //Chừng nào hàng đợi vẫn còn đỉnh tràn
            begin
                z := PopFromQueue; //Lấy một đỉnh tràn x khỏi hàng đợi
                while current[z] <> 0 do //Xét một cung đi ra khỏi x
                    begin
                        with e[current[z]] do
                            begin
                                if (c > f) and (h[x] > h[y]) then
                                    //Nếu có thể đẩy luồng được theo cung (u, v)
                                    begin
                                        NeedQueue := (y <> s) and (y <> t)
                                            and (excess[y] = 0);
                                        Push(current[z]); //Đẩy luồng luôn
                                        if NeedQueue then
                                            //v đang không tràn sau phép đẩy trở thành tràn
                                            PushToQueue(y); //Đẩy v vào hàng đợi
                                        if excess[z] = 0 then Break;
                                        //x hết tràn thì chuyển qua xét đỉnh khác ngay
                                    end;
                            end;
                    end;
            end;
    
```

```

end;
    current[z] := link[current[z]];
    //x chưa hết tràn thì chuyển sang xét cung liên thuộc tiếp theo
end;
if excess[z] > 0 then //Duyệt hết danh sách liên thuộc mà x vẫn tràn
    begin
        Lift(z); //Nâng cao x
        current[z] := head[z];
        //Đặt con trỏ current[x] trở lại về đầu danh sách liên thuộc
        PushToQueue(z); //Đẩy lại x vào hàng đợi chờ xử lý sau
    end;
end;
FlowValue := excess[t];
//Thuật toán kết thúc, giá trị luồng bằng tổng luồng đi vào đỉnh thu (= - excess[s])
end;
procedure PrintResult; //In kết quả
var i: Integer;
begin
    WriteLn('Maximum flow: ');
    for i := 1 to m do
        with e[i] do
            if f > 0 then //Chỉ cần in ra các cung có luồng > 0
                WriteLn('e[ ', i, ' ] = ( ', x, ', ', ' , y, ', ' ) : c
                        = ', c, ', f = ', f);
    WriteLn('Value of flow: ', FlowValue);
end;
begin
    Enter; //Nhập dữ liệu
    Init; //Khởi tạo
    FIFOPreflowPush; //Thực hiện thuật toán đẩy tiền luồng
    PrintResult; //In kết quả
end.

```

Định lý 3-19 (định lý về tính nguyên)

Nếu tất cả các sút chứa là số nguyên thì thuật toán Ford-Fulkerson cũng như thuật toán đẩy tiền luồng luôn tìm được luồng cực đại với luồng trên cung là các số nguyên.

Chứng minh

Đối với thuật toán Ford-Fulkerson, ban đầu ta khởi tạo luồng 0 thì luồng trên các cung là nguyên. Mỗi lần tăng luồng dọc theo đường tăng luồng P , luồng trên mỗi cung hoặc giữ nguyên, hoặc tăng/giảm một lượng Δ_P cũng là số nguyên. Vậy nên cuối cùng luồng cực đại phải có giá trị nguyên trên tất cả các cung.

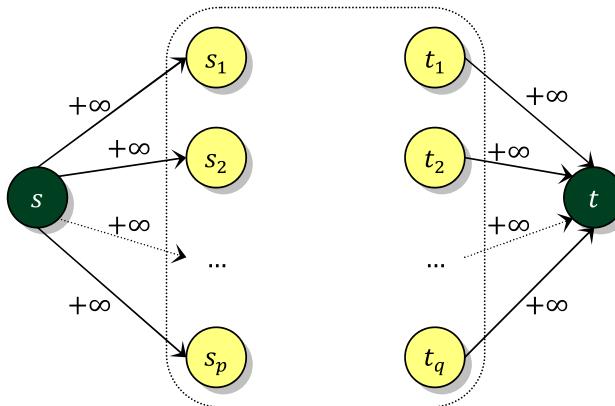
Đối với thuật toán đẩy tiền luồng, ban đầu ta khởi tạo một tiền luồng trên các cung là số nguyên. Phép Lift và Push không làm thay đổi tính nguyên của tiền luồng trên các cung. Vậy nên khi thuật toán kết thúc, tiền luồng trở thành luồng cực đại với giá trị luồng trên các cung là số nguyên.

3.4. Một số mở rộng và ứng dụng của luồng

a) Mạng với nhiều đỉnh phát và nhiều đỉnh thu

Ta mở rộng khái niệm mạng bằng cách cho phép mạng G có p đỉnh phát: s_1, s_2, \dots, s_p và q đỉnh thu t_1, t_2, \dots, t_q , các đỉnh phát và các đỉnh thu hoàn toàn phân biệt. Hàm sức chứa và luồng trên mạng được định nghĩa tương tự như trong trường hợp mạng có một đỉnh phát và một đỉnh thu. Giá trị của luồng được định nghĩa bằng tổng luồng trên các cung đi ra khỏi các đỉnh phát. Bài toán đặt ra là tìm luồng cực đại trên mạng có nhiều đỉnh phát và nhiều đỉnh thu.

Thêm vào mạng hai đỉnh: một siêu đỉnh phát s và siêu đỉnh thu t . Thêm các cung nối từ s tới các đỉnh s_i có sức chứa $+\infty$, thêm các cung nối từ các đỉnh t_j tới t với sức chứa $+\infty$. Ta được một mạng mới $G' = (V, E')$ (h.2.9).



Hình 2.9. Mạng với nhiều đỉnh phát và nhiều đỉnh thu

Có thể thấy rằng nếu f là một luồng cực đại trên G' , thì f hạn chế trên G cũng là luồng cực đại trên G . Vậy để tìm luồng cực đại trên G , ta sẽ tìm luồng cực đại

trên G' rồi loại bỏ siêu đỉnh phát s , siêu đỉnh thu t và tất cả những cung giả mới thêm vào.

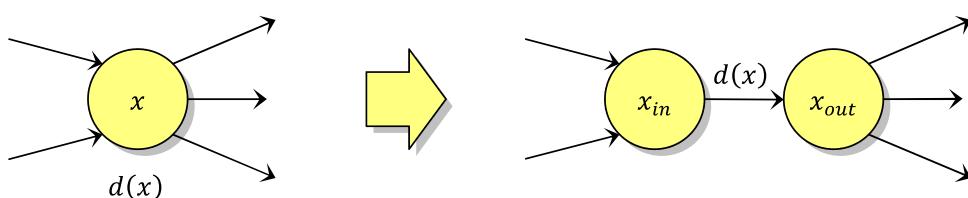
Một cách khác có thể thực hiện để tìm luồng trên mạng có nhiều đỉnh phát và nhiều đỉnh thu là loại bỏ tất cả các cung đi vào các đỉnh phát cũng như các cung đi ra khỏi các đỉnh thu. Chập tất cả các đỉnh phát thành một siêu đỉnh s và chập tất cả các đỉnh thu lại thành một siêu đỉnh thu t , mạng không còn các đỉnh s_1, s_2, \dots, s_p và t_1, t_2, \dots, t_q nữa mà chỉ có thêm đỉnh phát s và đỉnh thu t . Trên mạng ban đầu, mỗi cung đi vào/ra s_i được chỉnh lại đầu mút để nó đi vào/ra đỉnh s , mỗi cung đi vào/ra t_j cũng được chỉnh lại đầu mút để nó đi vào/ra đỉnh t , ta được một mạng mới G'' .

Khi đó ta có thể tìm f là một luồng cực đại trên G'' và khôi phục lại đầu mút của các cung như cũ để f trở thành luồng cực đại trên G .

b) Mạng với sức chứa trên cả các đỉnh và các cung

Cho mạng $G = (V, E, c, s, t)$, mỗi đỉnh $v \in V - \{s, t\}$ được gán một số không âm $d(v)$ gọi là sức chứa của đỉnh đó. Luồng dương φ trên mạng này được định nghĩa với tất cả các ràng buộc của luồng dương và thêm một điều kiện: Tổng luồng dương trên các cung đi vào mỗi đỉnh $v \in V - \{s, t\}$ không được vượt quá $d(v)$: $\sum_{e \in E^-(v)} \varphi(e) \leq d(v)$. Bài toán đặt ra là tìm luồng dương cực đại trên mạng có ràng buộc sức chứa trên cả các đỉnh và các cung.

Tách mỗi đỉnh $x \in V - \{s, t\}$ thành 2 đỉnh mới x_{in}, x_{out} và một cung (x_{in}, x_{out}) với sức chứa $d(x)$. Các cung đi vào x được chỉnh lại đầu mút để đi vào x_{in} và các cung đi ra khỏi x được chỉnh lại đầu mút để đi ra khỏi x_{out} (h.2.10). Ta xây dựng được mạng $G' = (V', E')$ với đỉnh phát s và đỉnh thu t .



Hình 2.10. Tách đỉnh

Khi đó việc tìm luồng dương cực đại trên mạng G có thể thực hiện bằng cách tìm luồng dương cực đại trên mạng G' , sau đó chập tất cả các cặp (x_{in}, x_{out}) trở lại thành đỉnh x ($\forall x \in V - \{s, t\}$) để khôi phục lại mạng G ban đầu.

c) Mạng với ràng buộc luồng dương bị chặn hai phía

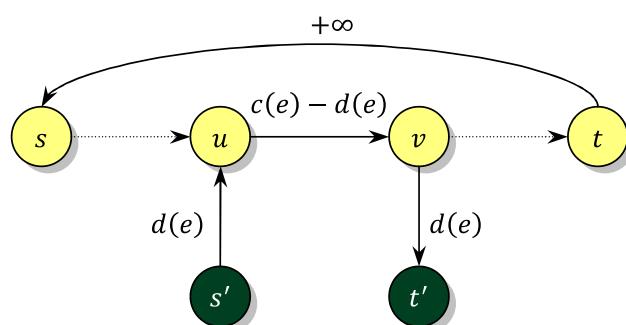
Cho mạng $G = (V, E, c, s, t)$ trong đó mỗi cung $e \in E$ ngoài sức chứa (lưu lượng) tối đa $c(e)$ còn được gán một số không âm $d(e) \leq c(e)$ gọi là lưu lượng tối thiểu. Một *luồng dương tương thích* φ trên G được định nghĩa với tất cả các ràng buộc của luồng dương và thêm một điều kiện: Luồng dương trên mỗi cung $e \in E$ không được nhỏ hơn sức chứa tối thiểu của cung đó:

$$d(e) \leq \varphi(e) \leq c(e)$$

Bài toán đặt ra là kiểm chứng sự tồn tại của luồng dương tương thích trên mạng với ràng buộc luồng dương bị chặn hai phía.

Xây dựng một mạng $G' = (V', E')$ từ mạng G theo quy tắc:

- Tập đỉnh V' có được từ tập V thêm vào đỉnh phát giả s' và đỉnh thu giả t' : $V' = V + \{s', t'\}$.
- Mỗi cung $e = (u, v) \in E$ sẽ tương ứng với ba cung trên E' : cung $e_1 = (u, v)$ có sức chứa $c(e) - d(e)$, cung $e_2 = (s', v)$ và cung $e_3 = (u, t')$ có sức chứa $d(e)$. Ngoài ra thêm vào cung $(t, s) \in E'$ với sức chứa $+\infty$



Hình 2.11.

Gọi $D = \sum_{e \in E} d(e)$ là tổng sức chứa tối thiểu của các cung trên mạng G . Khi đó trên mạng G' , tổng sức chứa các cung đi ra khỏi s' cũng như tổng sức chứa các cung đi vào t' bằng D . Vì vậy với mọi luồng dương trên G' thì giá trị luồng đó không thể vượt quá D .

Từ đó suy ra rằng nếu tồn tại một luồng dương φ' trên G' có giá trị luồng $|\varphi'| = D$ thì φ' bắt buộc là luồng dương cực đại trên G' .

Bổ đề 3-20 cho phép ta kiểm chứng sự tồn tại luồng dương tương thích trên G bằng việc đo giá trị luồng cực đại trên G' .

Bổ đề 3-20

Điều kiện cần và đủ để tồn tại luồng dương tương thích φ trên mạng G là tồn tại luồng dương cực đại φ' trên G' với giá trị luồng $|\varphi'| = D$.

Chứng minh

Giả sử luồng dương cực đại φ' trên G' có $|\varphi'| = D = \sum_{e \in E} d(e)$. Ta xây dựng luồng φ trên G bằng cách cộng thêm vào luồng φ' trên mỗi cung e một lượng $d(e)$:

$$\begin{aligned}\varphi: E &\rightarrow [0, +\infty) \\ e &\rightarrow \varphi(e) = \varphi'(e) + d(e)\end{aligned}$$

Khi đó có thể dễ dàng kiểm chứng được φ thỏa mãn tất cả các ràng buộc của luồng dương tương thích trên mạng G .

Ngược lại nếu φ là một luồng dương tương thích trên G . Ta xây dựng luồng dương φ' trên G' bằng cách trừ luồng φ trên mỗi cung e đi một lượng $d(e)$, đồng thời đặt luồng φ' trên các cung đi ra khỏi s' cũng như trên các cung đi vào t' đúng bằng sức chứa của cung đó. Khi đó cũng dễ dàng kiểm chứng được φ' là luồng dương cực đại và $|\varphi'| = D$.

d) Mạng với sức chứa âm

Cho mạng $G = (V, E, c, w, s, t)$ trong đó ta mở rộng khái niệm sức chứa bằng cách cho phép cả những sức chứa âm trên một số cung. Khái niệm luồng được định nghĩa như bình thường.

Nếu như có thể khởi tạo được một luồng thì thuật toán Ford-Fulkerson vẫn hoạt động đúng để tìm luồng cực đại trên mạng có sức chứa âm. Vấn đề khởi tạo một luồng bất kỳ trên mạng không phải đơn giản vì chúng ta không thể khởi tạo bằng luồng 0, bởi nếu như vậy, ràng buộc sức chứa tối đa sẽ bị vi phạm trên các cung có sức chứa âm.

Giả sử một cung $e \in E$ có sức chứa $c(e) < 0$. Theo tính đối xứng lệch của luồng $f(e) = -f(-e)$ và ràng buộc sức chứa tối đa $f(e) \leq c(e)$, ta có:

$$f(-e) = -f(e) \geq -c(e) > 0 \quad (3.5)$$

Như vậy ràng buộc sức chứa tối đa $f(e) \leq c(e)$ tương đương với ràng buộc về sức chứa tối thiểu $-c(e)$ trên cung đối $-e$. Việc chỉ ra một luồng bất kỳ trên G có thể thực hiện bằng cách tìm luồng dương trên mạng với ràng buộc luồng dương bị chặn hai phía sau đó biến đổi luồng dương này thành luồng cần tìm.

e) Lát cắt hẹp nhất

Ta quan tâm tới đồ thị vô hướng liên thông $G = (V, E)$ với hàm trọng số (hay lưu lượng) $c: E \rightarrow [0, +\infty)$. Giả sử $|V| \geq 2$, người ta muốn bỏ đi một số cạnh để đồ thị mất tính liên thông và yêu cầu tìm phương án sao cho tổng trọng số các cạnh bị loại bỏ là nhỏ nhất.

Bài toán cũng có thể phát biểu dưới dạng: hãy phân hoạch tập đỉnh V thành hai tập khác rỗng rời nhau X và Y sao cho tổng lưu lượng các cạnh nối giữa X và Y là nhỏ nhất có thể. Cách phân hoạch này gọi là lát cắt tổng quát hẹp nhất của G , ký hiệu $\text{MinCut}(G)$.

$$c(X, Y) \rightarrow \min$$

$$X \neq \emptyset; Y \neq \emptyset; X \cap Y = \emptyset; X \cup Y = V;$$

Một cách tệ nhất có thể thực hiện là thử tất cả các cặp đỉnh s, t . Với mỗi lần thử ta cho s làm đỉnh phát và t làm đỉnh thu trên mạng G , sau đó tìm luồng cực đại và lát cắt $s - t$ hẹp nhất. Cuối cùng là chọn lát cắt $s - t$ có lưu lượng nhỏ nhất trong tất cả các lần thử. Phương pháp này cần $\binom{n}{2} = \frac{n \times (n-1)}{2}$ lần tìm luồng cực đại, có tốc độ chậm và không khả thi với dữ liệu lớn.

Bố đề 3-21

Với s và t là hai đỉnh bất kỳ. Từ đồ thị G , ta xây dựng đồ thị G_{st} bằng cách chập hai đỉnh s và t thành một đỉnh duy nhất, ký hiệu st , các cạnh nối s với t bị hủy bỏ, các cạnh liên thuộc với chỉ s hoặc t được chỉnh lại đầu mút để trở thành cạnh liên thuộc với st . Khi đó $\text{MinCut}(G)$ có thể thu được bằng lấy lát cắt có lưu lượng nhỏ nhất trong hai lát cắt:

- Lát cắt $s - t$ hẹp nhất: Coi s là đỉnh phát và t là đỉnh thu, lát cắt $s - t$ hẹp nhất có thể xác định bằng việc giải quyết bài toán luồng cực đại trên mạng G .
- Lát cắt tổng quát hẹp nhất trên G_{st} : $MinCut(G_{st})$.

Chứng minh

Xét lát cắt tổng quát hẹp nhất trên G có thể đưa s và t vào hai thành phần liên thông khác nhau hoặc đưa chúng vào cùng một thành phần liên thông. Trong trường hợp thứ nhất, $MinCut(G)$ là lát cắt $s - t$ hẹp nhất. Trong trường hợp thứ hai, $MinCut(G)$ là $MinCut(G_{st})$.

Bố đề 3-21 cho phép chúng ta xây dựng một thuật toán tốt hơn: Nếu đồ thị chỉ gồm 2 đỉnh thì chỉ việc cắt rời hai đỉnh vào hai tập. Nếu không, ta chọn hai đỉnh bất kỳ s, t làm đỉnh phát và đỉnh thu, tìm luồng cực đại và ghi nhận lát cắt $s - t$ hẹp nhất. Tiếp theo ta chập hai đỉnh s, t thành một đỉnh st và lặp lại với đồ thị $G_{st} \dots$ Cuối cùng là chỉ ra lát cắt $s - t$ hẹp nhất trong số tất cả các lát cắt được ghi nhận. Phương pháp này đòi hỏi phải thực hiện $|V| - 1$ lần tìm luồng cực đại, tuy đã có sự cải thiện về tốc độ nhưng chưa phải thật tốt.

Nhận xét rằng tại mỗi bước của cách giải trên, chúng ta có thể chọn hai đỉnh s, t bất kỳ miễn sao $s \neq t$. Vì vậy người ta muốn tìm một cách chọn cặp đỉnh s, t một cách hợp lý tại mỗi bước để có thể chỉ ra ngay lát cắt $s - t$ hẹp nhất mà không cần tìm luồng cực đại. Thuật toán dưới đây [37] là một trong những thuật toán hiệu quả dựa trên ý tưởng đó.

Với A là một tập con của tập đỉnh V và x là một đỉnh không thuộc A . Định nghĩa *lực hút* của A đối với x là tổng trọng số các cạnh nối x với các đỉnh thuộc A :

$$c(A, \{x\}) = \sum_{\substack{e=(x,y) \in E \\ y \in A}} c(e)$$

Bố đề 3-22

Bắt đầu từ tập A chỉ gồm một đỉnh bất kỳ $a \in V$, ta cứ tìm một đỉnh bị A hút chặt nhất kết nạp thêm vào A cho tới khi $A = V$. Gọi s và t là hai đỉnh được kết nạp cuối cùng theo cách này. Khi đó lát cắt $(V - \{t\}, \{t\})$ là lát cắt $s - t$ hẹp nhất.

Chứng minh

Xét một lát cắt $s - t$ bất kỳ κ , ta sẽ chứng minh rằng lưu lượng của lát cắt $(V - \{t\}, \{t\})$ không lớn hơn lưu lượng của lát cắt κ .

Một đỉnh v được gọi là *đỉnh hoạt tính* nếu v và đỉnh được đưa vào A liền trước v bị rơi vào hai phía của lát cắt κ . Gọi A_v là tập các đỉnh được kết nạp vào A trước đỉnh v , κ_v là lát cắt κ hạn chế trên $A_v \cup \{v\}$ (Lát cắt κ_v dùng đúng cách phân hoạch của lát cắt κ nhưng chỉ quan tâm tới tập đỉnh $A_v \cup \{v\}$). Gọi $c(\kappa)$ là lưu lượng của lát cắt κ , $c(\kappa_v)$ là lưu lượng của lát cắt κ_v .

Trước hết ta sử dụng phép quy nạp để chỉ ra rằng nếu u là đỉnh hoạt tính thì:

$$c(A_u, \{u\}) \leq c(\kappa_u) \quad (3.6)$$

Nếu u là đỉnh hoạt tính đầu tiên được kết nạp vào A , lát cắt κ_u sẽ chia tập $A_u \cup \{u\}$ làm hai tập, một tập là A_u và một tập là $\{u\}$, khi đó ta có $c(A_u, \{u\})$ cũng chính là $c(\kappa_u)$. Giả thiết rằng bất đẳng thức (3.6) đúng với đỉnh hoạt tính u , ta sẽ chứng nó cũng đúng với những đỉnh hoạt tính v được kết nạp vào A sau u . Thật vậy,

$$c(A_v, \{v\}) = c(A_u, \{v\}) + c(A_v - A_u, \{v\}) \quad (3.7)$$

Do A_u phải hút u mạnh hơn v , kết hợp với giả thiết quy nạp, ta có:

$$c(A_u, \{v\}) \leq c(A_u, \{u\}) \leq c(\kappa_u)$$

Hạng tử $c(A_v - A_u, \{v\})$ là tổng trọng số các cạnh nối giữa v và $A_v - A_u$. Do u và v là hai đỉnh hoạt tính liên tiếp, các cạnh này sẽ nối giữa hai phía của lát cắt κ_v và có đóng góp trong phép tính $c(\kappa_v)$, mặt khác do $v \notin A_u \cup \{u\}$ nên những cạnh này không đóng góp trong phép tính $c(\kappa_u)$. Vậy từ công thức (3.7), ta suy ra:

$$\begin{aligned} c(A_v, \{v\}) &= c(A_u, \{v\}) + c(A_v - A_u, \{v\}) \\ &\leq c(\kappa_u) + c(A_v - A_u, \{v\}) \\ &\leq c(\kappa_v) \end{aligned} \quad (3.8)$$

Vì κ là một lát cắt $s - t$ nên chắc chắn s và t nằm ở hai phía khác nhau của lát cắt κ , hay nói cách khác, t là đỉnh hoạt tính. Bất đẳng thức (3.6) chứng minh ở trên cho ta kết quả:

$$\begin{aligned} c(V - \{t\}, \{t\}) &= c(A_t, \{t\}) \\ &\leq c(\kappa_t) \\ &= c(\kappa) \end{aligned} \quad (3.9)$$

Ta chứng minh được lát cắt $(V - \{t\}, \{t\})$ là lát cắt $s - t$ hẹp nhất.

Định lý 3-23

Việc tìm lát cắt tổng quát trên đồ thị vô hướng liên thông với hàm trọng số không âm có thể được thực hiện bằng thuật toán trong thời gian $O(|V|^2 \log|V| + |V||E|)$.

Chứng minh

Bắt đầu từ tập A chỉ gồm một đỉnh bất kỳ, ta mở rộng A bằng cách lần lượt kết nạp vào A đỉnh bị hút chặt nhất cho tới khi $A = V$. Việc này được thực hiện với kỹ thuật tương tự như thuật toán Prim: với $\forall v \notin A$, ta ký hiệu nhãn $d[v]$ là lực hút của A đối với đỉnh v . Khi A được kết nạp thêm một u thì các nhãn lực hút của những đỉnh v khác sẽ được cập nhật lại theo công thức:

$$d[v]_{\text{mới}} := d[v]_{\text{cũ}} + c(u, v), \forall (u, v) \in E$$

Bằng việc tổ chức các đỉnh ngoài A trong một hàng đợi ưu tiên dạng Fibonacci Heap, việc mở rộng tập A cho tới khi $A = V$ được thực hiện trong thời gian $O(|V| \log|V| + |E|)$. Trong quá trình đó, s và t là hai đỉnh cuối cùng được kết nạp vào A cũng được xác định và $\text{MinCut}(G)$ được cập nhật theo lát cắt $s - t$ hẹp nhất. Sau đó hai đỉnh s, t được chèn vào và thuật toán lặp lại với đồ thị G_{st} . Tổng cộng ta có $|V| - 1$ lần lặp, suy ra lát cắt tổng quát hẹp nhất có thể tìm được trong thời gian $O(|V|^2 \log|V| + |V||E|)$.

Mặc dù tính đúng đắn của thuật toán được chứng minh dựa vào lý thuyết về luồng cực đại và lát cắt hẹp nhất, việc cài đặt thuật toán lại khá đơn giản và không động chạm gì đến luồng cực đại.

Bài tập

- 2.27.** Cho f_1 và f_2 là hai luồng trên mạng $G = (V, E, c, s, t)$ và α là một số thực nằm trong đoạn $[0,1]$. Xét ánh xạ:

$$\begin{aligned} f_\alpha: E &\rightarrow \mathbb{R} \\ e &\mapsto f_\alpha(e) = \alpha f_1(e) + (1 - \alpha) f_2(e) \end{aligned}$$

Chứng minh rằng f_α cũng là một luồng trên mạng G với giá trị luồng:

$$|f_\alpha| = \alpha|f_1| + (1 - \alpha)|f_2|$$

- 2.28.** Cho f là một luồng trên mạng $G = (V, E, c, s, t)$, chứng minh rằng với $\forall e \in E$, ta có:

$$c_f(e) + c_f(-e) = c(e) + c(-e)$$

- 2.29.** Cho f là luồng cực đại trên mạng $G = (V, E, c, s, t)$, gọi Y là tập các đỉnh đến được t bằng một đường thặng dư trên G_f và $X = V - Y$. Chứng minh rằng (X, Y) là lát cắt $s - t$ hẹp nhất của mạng G .
- 2.30.** Viết chương trình nhận vào một đồ thị có hướng $G = (V, E)$ với hai đỉnh phân biệt s và t và tìm một tập gồm nhiều đường đi nhất từ s tới t sao cho các đường đi trong tập này đôi một không có cạnh chung.

Gợi ý

Coi s là đỉnh phát và t là đỉnh thu, các cung đều có sức chứa 1. Tìm luồng cực đại trên mạng bằng thuật toán Ford-Fulkerson, theo Định lý 3-19 (định lý về tính nguyên), luồng trên các cung chỉ có thể là 0 hoặc 1. Loại bỏ các cung có luồng 0 và chỉ giữ lại các cung có luồng 1. Tiếp theo ta tìm một đường đi từ s tới t , chọn đường đi này vào tập hợp, loại bỏ tất cả các cung dọc trên đường đi này khỏi đồ thị và lặp lại..., thuật toán sẽ kết thúc khi đồ thị không còn cạnh nào (không còn đường đi từ s tới t).

Về kỹ thuật cài đặt, ta có thể tìm một đường đi từ s tới t trên đồ thị G , đảo chiều tất cả các cung trên đường đi này và lặp lại cho tới khi không còn đường đi từ s tới t nữa. Có thể thấy rằng đồ thị G tại mỗi bước chính là đồ thị các cung thặng dư và đường đi tìm được ở mỗi bước chính là đường tăng luồng.

Đồ thị G giờ đây không còn đường đi từ s tới t , ta tìm một đường đi từ t về s , kết nạp đường đi theo chiều ngược lại (từ s tới t) vào tập hợp, xóa bỏ tất cả các cung trên đường đi và cứ tiếp tục như vậy cho tới khi không còn đường đi từ t về s nữa.

- 2.31.** Tương tự như Bài tập 2.29 nhưng yêu cầu thực hiện trên đồ thị vô hướng.
- 2.32.** (Hệ đại diện phân biệt) Một lớp học có n bạn nam và n bạn nữ. Nhân ngày 8/3, lớp có mua m món quà để các bạn nam tặng các bạn nữ. Mỗi món quà có thể thuộc sở thích của một số bạn trong lớp.

Hãy lập chương trình tìm cách phân công tặng quà thỏa mãn:

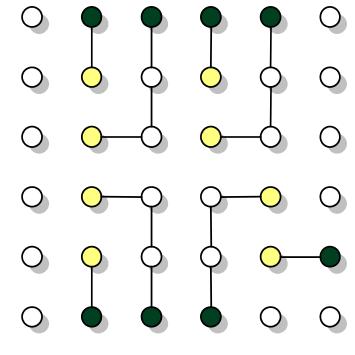
- Mỗi bạn nam phải tặng quà cho đúng một bạn nữ và mỗi bạn nữ phải nhận quà của đúng một bạn nam. Món quà được tặng phải thuộc sở thích của cả hai người.
- Món quà nào đã được một bạn nam chọn để tặng thì bạn nam khác không được chọn nữa.

Gợi ý: Xây dựng một mạng trong đó tập đỉnh V gồm 3 lớp đỉnh S , X và T :

- Lớp đỉnh phát $S = \{s_1, s_2, \dots, s_n\}$, mỗi đỉnh tương ứng với một bạn nam.
- Lớp đỉnh $X = (x_1, x_2, \dots, x_n)$ mỗi đỉnh tương ứng với một món quà.
- Lớp đỉnh thu $T = \{t_1, t_2, \dots, t_n\}$ mỗi đỉnh tương ứng với một bạn nữ.

Nếu bạn nam i thích món quà k , ta cho cung nối từ s_i tới x_k , nếu bạn nữ j thích món quà k , ta cho cung nối từ x_k tới t_j . Sức chứa của các cung đặt bằng 1 và sức chứa của các đỉnh v_1, v_2, \dots, v_n cũng đặt bằng 1. Tìm luồng nguyên cực đại trên mạng G có n đỉnh phát, n đỉnh thu, đồng thời có cả ràng buộc sức chứa trên các đỉnh, những cung có luồng 1 sẽ nối giữa một món quà và người tặng/nhận tương ứng.

- 2.33.** Cho mạng điện gồm $m \times n$ điểm nằm trên một lưới m hàng, n cột. Một số điểm nằm trên biên của lưới là nguồn điện, một số điểm trên biên của lưới là các thiết bị sử dụng điện. Người ta chỉ cho phép nối dây điện giữa hai điểm nằm cùng hàng hoặc cùng cột. Hãy tìm cách đặt các dây điện nối các thiết bị sử dụng điện với nguồn điện sao cho hai đường dây bất kỳ nối hai thiết bị sử dụng điện với nguồn điện tương ứng của chúng không được có điểm chung.



- 2.34.** (Kỹ thuật giãn sức chứa) Cho mạng $G = (V, E, c, w, s, t)$ với sức chứa nguyên: $c: E \rightarrow \mathbb{N}$. Gọi $C := \max_{e \in E} c(e)$.

a) Chứng minh rằng lát cắt $s - t$ hẹp nhất của G có lưu lượng không vượt quá $C|E|$

b) Với một số nguyên k , tìm thuật toán xác định đường tăng luồng có giá trị thặng dư $\geq k$ trong thời gian $O(|E|)$.

c) Chứng minh rằng thuật toán sau đây tìm được luồng cực đại trên mạng G :

```

procedure MaxFlowByScaling;
begin
    f := «Luồng 0»;
    k := C; //k là sức chứa lớn nhất của một cung trong E
    while k ≥ 1 do
        begin
            while «Tìm được đường tăng luồng P
                có giá trị thặng dư ≥ k» do
                «Tăng luồng dọc theo đường P»;
            k := k div 2;
        end;
    end;

```

d) Chứng minh rằng khi bước vào mỗi lượt lặp của vòng lặp:

```
while k ≥ 1 do...
```

Lưu lượng của lát cắt hẹp nhất trên mạng thặng dư G_f không vượt quá $2k|E|$.

e) Chứng minh rằng trong mỗi lượt lặp của vòng lặp:

```
while k ≥ 1 do...
```

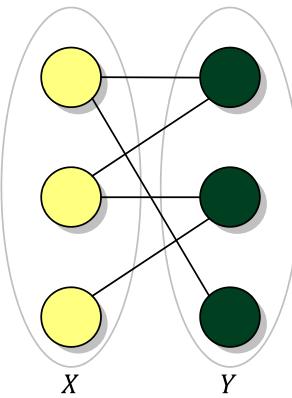
Vòng lặp while bên trong thực hiện $O(|E|)$ lần với mỗi giá trị của k .

f) Chứng minh rằng thuật toán trên (*maximum flow by scaling*) có thể cài đặt để tìm luồng cực đại trên G trong thời gian $O(|E|^2 \log C)$.

4. Bộ ghép cực đại trên đồ thị hai phía

4.1. Đồ thị hai phía

Đồ thị vô hướng $G = (V, E)$ được gọi là đồ thị hai phía nếu tập đỉnh V của nó có thể chia làm hai tập con rời nhau: X và Y sao cho mọi cạnh của đồ thị đều nối một đỉnh thuộc X với một đỉnh thuộc Y . Khi đó người ta còn ký hiệu $G = (X \cup Y, E)$. Để thuận tiện trong trình bày, ta gọi các đỉnh thuộc X là các $X_đỉnh$ và các đỉnh thuộc Y là các $Y_đỉnh$.



Hình 2.12. Đồ thị hai phía

Một đồ thị vô hướng là đồ thị hai phia nếu và chỉ nếu từng thành phần liên thông của nó là đồ thị hai phia. Để kiểm tra một đồ thị vô hướng liên thông có phải đồ thị hai phia hay không, ta có thể sử dụng một thuật toán tìm kiếm trên đồ thị (BFS hoặc DFS) bắt đầu từ một đỉnh s bất kỳ. Đặt:

$$X := \{\text{tập các đỉnh đến được từ } s \text{ qua một số chẵn cạnh}\}$$

$$Y := \{\text{tập các đỉnh đến được từ } s \text{ qua một số lẻ cạnh}\}$$

Nếu tồn tại cạnh của đồ thị nối hai đỉnh $\in X$ hoặc hai đỉnh $\in Y$ thì đồ thị đã cho không phải đồ thị hai phia, ngược lại đồ thị đã cho là đồ thị hai phia với cách phân hoạch tập đỉnh thành hai tập X, Y ở trên.

Đồ thị hai phia gặp rất nhiều mô hình trong thực tế. Chẳng hạn quan hệ hôn nhân giữa tập những người đàn ông và tập những người đàn bà, việc sinh viên chọn trường, thầy giáo chọn tiết dạy trong thời khoá biểu v.v...

4.2. Bài toán tìm bộ ghép cực đại trên đồ thị hai phia

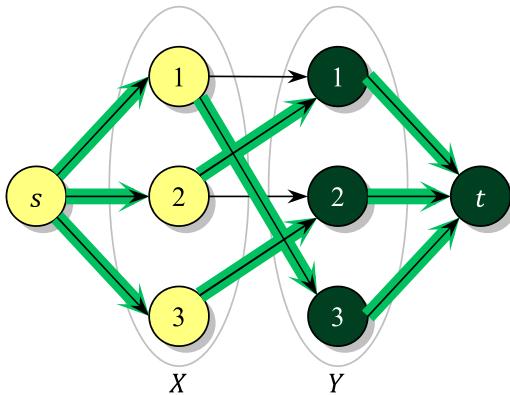
Cho đồ thị hai phia $G = (X \cup Y, E)$. Một *bộ ghép* (*matching*) của G là một tập các cạnh đôi một không có đỉnh chung. Có thể coi một bộ ghép là một tập $M \subseteq E$ sao cho trên đồ thị $(X \cup Y, M)$, mỗi đỉnh có bậc không quá 1.

Vấn đề đặt ra là tìm *một bộ ghép lớn nhất* (*maximum matching*) (có nhiều cạnh nhất) trên đồ thị hai phia cho trước.

4.3. Mô hình luồng

Định hướng các cạnh của G thành cung từ X sang Y . Thêm vào đỉnh phát giả s và các cung nối từ s tới các X _đỉnh, thêm vào đỉnh thu giả t và các cung nối từ

các Y _đỉnh tới t . Sức chứa của tất cả các cung được đặt bằng 1, ta được mạng G' . Xét một luồng trên mạng G' có luồng trên các cung là số nguyên, khi đó có thể thấy rằng những cung có luồng bằng 1 từ X sang Y sẽ tương ứng với một bộ ghép trên G . Bài toán tìm bộ ghép cực đại trên G có thể giải quyết bằng cách tìm luồng nguyên cực đại trên G' .



Hình 2.13. Mô hình luồng của bài toán tìm bộ ghép cực đại trên đồ thị hai phía.

Chúng ta sẽ phân tích một số đặc điểm của đường tăng luồng trong trường hợp này để tìm ra một cách cài đặt đơn giản hơn.

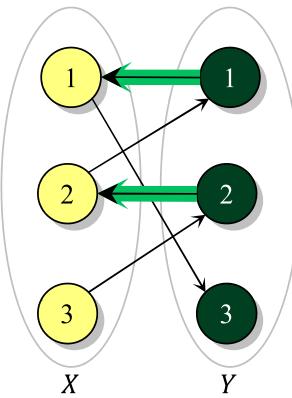
Xét đồ thị hai phía $G = (X \cup Y, E)$ và một bộ ghép M trên G .

- Những đỉnh thuộc M gọi là những *đỉnh đã ghép* (*matched vertices*), những đỉnh không thuộc M gọi là những *đỉnh chưa ghép* (*unmatched vertices*).
- Những cạnh thuộc M gọi là những *cạnh đã ghép*, những cạnh không thuộc M được gọi là những *cạnh chưa ghép*.
- Nếu định hướng lại những cạnh của đồ thị thành cung: Những cạnh chưa ghép định hướng từ X sang Y , những cạnh đã ghép định hướng ngược lại từ Y về X . Trên đồ thị định hướng đó, một đường đi được gọi là *đường pha* (*alternating path*) và một đường đi từ một X _đỉnh chưa ghép tới một Y _đỉnh chưa ghép gọi là một *đường mở* (*augmenting path*).

Đọc trên một đường pha, các cạnh đã ghép và chưa ghép xen kẽ nhau. Đường mở cũng là một đường pha, đi qua một số lẻ cạnh, trong đó số cạnh chưa ghép nhiều hơn số cạnh đã ghép đúng một cạnh.

Ví dụ với đồ thị hai phía trong hình 2.14 và một bộ ghép $\{(x_1, y_1), (x_2, y_2)\}$.

Đường đi $\langle x_3, y_2, x_2, y_1 \rangle$ là một đường pha, đường đi $\langle x_3, y_2, x_2, y_1, x_1, y_3 \rangle$ là một đường mở.



Hình 2.14. Đồ thị hai phía và các cạnh được định hướng theo một bộ ghép

Đường mở thực chất là đường tăng luồng với giá trị thặng dư 1 trên mô hình luồng. Định lý 3-11 (mỗi quan hệ giữa luồng cực đại, đường tăng luồng và lát cắt hẹp nhất) đã chỉ ra rằng điều kiện cần và đủ để một bộ ghép M là bộ ghép cực đại là không tồn tại đường mở ứng với M .

Nếu tồn tại đường mở P ứng với bộ ghép M , ta mở rộng bộ ghép bằng cách: đọc trên đường P loại bỏ những cạnh đã ghép khỏi M và thêm những cạnh chưa ghép vào M . Bộ ghép mới thu được sẽ có lực lượng nhiều hơn bộ ghép cũ đúng một cạnh. Đây thực chất là phép tăng luồng đọc trên đường P trên mô hình luồng.

4.4. Thuật toán đường mở

Từ mô hình luồng của bài toán, chúng ta có thể xây dựng được thuật toán tìm bộ ghép cực đại dựa trên cơ chế tìm đường mở và tăng cặp: Thuật toán khởi tạo một bộ ghép bất kỳ trước khi bước vào vòng lặp chính. Tại mỗi bước lặp, đường mở (thực chất là một đường đi từ một $X_đỉnh$ chưa ghép tới một $Y_đỉnh$ chưa ghép) được tìm bằng BFS hoặc DFS và bộ ghép sẽ được mở rộng dựa trên đường mở tìm được.

```

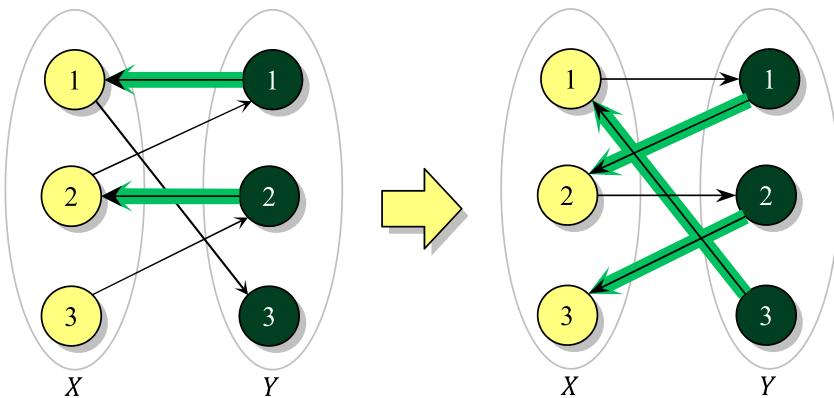
M := «Một bộ ghép bất kỳ, chẳng hạn: Ø»;
while «Tìm được đường mở P» do
    begin
        «Đọc trên đường P:
            - Loại bỏ những cạnh đã ghép khỏi M
            - Thêm những cạnh chưa ghép vào M
        »
    end;

```

Ví dụ với đồ thị trong hình 2.14 và bộ ghép $M = \{(x_1, y_1), (x_2, y_2)\}$, thuật toán sẽ tìm được đường mở:

$$x_3 \rightsquigarrow y_2 \xrightarrow[\in M]{} x_2 \rightsquigarrow y_1 \xrightarrow[\in M]{} x_1 \rightsquigarrow y_3$$

Đọc trên đường mở này, ta loại bỏ hai cạnh (y_2, x_2) và (y_1, x_1) khỏi bộ ghép và thêm vào bộ ghép ba cạnh $(x_3, y_2), (x_2, y_1), (x_1, y_3)$, được bộ ghép mới 3 cạnh. Đồ thị với bộ ghép mới không còn đỉnh chưa ghép (không còn đường mở) nên đây chính là bộ ghép cực đại (h.2.15).



Hình 2.15. Mở rộng bộ ghép

4.5. Cài đặt

Chúng ta sẽ cài đặt thuật toán tìm bộ ghép cực đại trên đồ thị hai phía $G = (X \cup Y, E)$, trong đó $|X| = p$, $|Y| = q$ và $|E| = m$. Các X _đỉnh được đánh số từ 1 tới p và các Y _đỉnh được đánh số từ 1 tới q . Khuôn dạng Input/Output như sau:

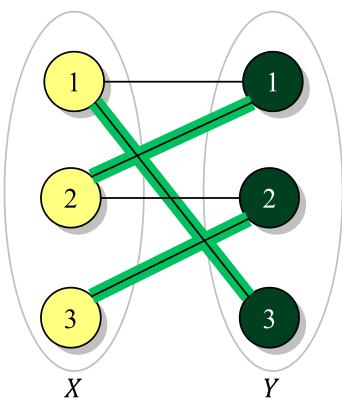
Input

- Dòng 1 chứa ba số nguyên dương p, q, m lần lượt là số X _đỉnh, số Y _đỉnh và số cạnh của đồ thị hai phía. ($p, q \leq 10^4; m \leq 10^6$).
- m dòng tiếp theo, mỗi dòng chứa hai số nguyên dương i, j tương ứng với một cạnh (x_i, y_j) của đồ thị.

Output

Bộ ghép cực đại trên đồ thị.

Sample Input	Sample Output
3 3 5	1: $x[2] - y[1]$
3 2	2: $x[3] - y[2]$
2 2	3: $x[1] - y[3]$
2 1	
1 3	
1 1	



a) Biểu diễn đồ thị hai phía và bộ ghép

Đồ thị hai phía $G = (X \cup Y, E)$ sẽ được biểu diễn bằng cách danh sách kè của các X _đỉnh. Cụ thể là ta sẽ sử dụng mảng $head[1 \dots p]$ với các phần tử ban đầu được khởi tạo bằng 0, mảng $adj[1 \dots m]$ và mảng $link[1 \dots m]$. Danh sách kè được xây dựng ngay trong quá trình đọc danh sách cạnh: mỗi khi đọc một cạnh $e_i = (x, y)$ ta gán $adj[i] := y$, đặt $link[i] := head[x]$ sau đó cập nhật lại $head[x] := i$. Khi đọc xong danh sách cạnh thì danh sách kè cũng được xây dựng xong, khi đó để duyệt các Y _đỉnh kè với một đỉnh $x \in X$, ta có thể sử dụng thuật toán sau:

```
i := head[x]; //Từ đầu danh sách móc nối các đỉnh kè x
while i ≠ 0 do
    begin
        «Xử lý đỉnh adj[i]»;
        i := link[i]; //Nhảy sang phần tử kế tiếp trong danh sách móc nối
    end;
```

Bộ ghép trên đồ thị hai phía được biểu diễn bởi mảng $match[1 \dots ny]$, trong đó $match[j]$ là chỉ số của X _đỉnh ghép với đỉnh y_j . Nếu y_j là đỉnh chưa ghép, ta gán $match[j] := 0$.

b) Tìm đường mở

Đường mở thực chất là một đường đi từ một X _đỉnh chưa ghép tới một Y _đỉnh chưa ghép trên đồ thị định hướng. Ta sẽ tìm đường mở tại mỗi bước bằng thuật toán DFS:

Bắt đầu từ một đỉnh $x \in X$ chưa ghép, trước hết ta đánh dấu các Y _đỉnh bằng mảng $avail[1 \dots q]$ trong đó $avail[j] = \text{True}$ nếu đỉnh $y_j \in Y$ chưa thăm và $avail[j] = \text{False}$ nếu đỉnh $y_j \in Y$ đã thăm (chỉ cần đánh dấu các Y _đỉnh).

Thuật toán DFS để tìm đường mở xuất phát từ x được thực hiện bằng một thủ tục đệ quy $Visit(x)$, thủ tục này sẽ quét tất cả những đỉnh $y \in Y$ chưa thăm nối từ X (dĩ nhiên qua một cạnh chưa ghép), với mỗi khi xét đến một đỉnh $y \in Y$, trước hết ta đánh dấu thăm y . Sau đó:

- Nếu y đã ghép, dựa vào sự kiện từ y chỉ đi đến được $match[y]$ qua một cạnh đã ghép hướng từ Y về X , lời gọi đệ quy $Visit(match[y])$ được thực hiện để thăm luôn đỉnh $match[y] \in X$ (thăm liền hai bước).
- Ngược lại nếu y chưa ghép, tức là thuật toán DFS tìm được đường mở kết thúc ở y , ta thoát khỏi dây chuyền đệ quy. Quá trình thoát dây chuyền đệ quy thực chất là lần ngược đường mở, ta sẽ lợi dụng quá trình này để mở rộng bộ ghép dựa trên đường mở.

Để thuật toán hoạt động hiệu quả hơn, ta sử dụng liên tiếp các pha xử lý lô: Ký hiệu X^* là tập các X _đỉnh chưa ghép, mỗi pha sẽ cố gắng mở rộng bộ ghép dựa trên không chỉ một mà nhiều đường mở không có đỉnh chung xuất phát từ các đỉnh khác nhau thuộc X^* . Cụ thể là một pha sẽ khởi tạo mảng đánh dấu $avail[1 \dots q]$ bởi giá trị True, sau đó quét tất cả những đỉnh $x \in X^*$, thử tìm đường mở xuất phát từ x và mở rộng bộ ghép nếu tìm ra đường mở. Trong một pha có thể có nhiều X _đỉnh được ghép thêm.

```

procedure Visit (x ∈ X) ; //Thuật toán DFS
begin
    for ∀y: (x, y) ∈ E do //Quét các Y_đỉnh kề x
        if avail[y] then //y chưa thăm, chú ý (x, y) chắc chắn là cạnh chưa ghép
            begin
                avail[y] := False; //Đánh dấu thăm y
                if match[y] = 0 then Found := True
                    //y chưa ghép, dựng cờ báo tìm thấy đường mở
                else Visit(match[y]); //y đã ghép, gọi đệ quy tiếp tục DFS
                if Found then //Ngay khi đường mở được tìm thấy
                    begin
                        match[y] := x; //Chỉnh lại bộ ghép theo đường mở
                    
```

```

        Exit; //Thoát luôn, lệnh Exit đặt ở đây sẽ thoát cả dây chuyền để quy
    end;
end;
begin //Thuật toán tìm bộ ghép cực đại trên đồ thị hai phía
    «Khởi tạo một bộ ghép bất kỳ, chẳng hạn  $\emptyset$ »;
     $X^*$  := «Tập các đỉnh chưa ghép»;
repeat //Lặp các pha xử lý theo lô
    Old :=  $|X^*|$ ; //Lưu số đỉnh chưa ghép khi bắt đầu pha
    for  $\forall y \in Y$  do avail[y] := True; //Đánh dấu mọi  $y$  đỉnh chưa thăm
    for  $\forall x \in X^*$  do
        begin
            Found := False; //Cờ báo chưa tìm thấy đường mở
            Visit(x); //Tìm đường mở bằng DFS
            if Found then  $X^*$  :=  $X^* - \{x\}$ ;
            //x đã được ghép, loại bỏ x khỏi  $X^*$ 
        end;
    until  $|X^*| = Old$ ; //Lặp cho tới khi không thể ghép thêm
end;

```

BMATCH.PAS ✓ Tìm bộ ghép cực đại trên đồ thị hai phía

```

{$MODE OBJFPC}
program MaximumBipartiteMatching;
const
    maxN = 10000;
    maxM = 1000000;
var
    p, q, m: Integer;
    adj: array[1..maxM] of Integer;
    link: array[1..maxM] of Integer;
    head: array[1..maxN + 1] of Integer;
    match: array[1..maxN] of Integer;
    avail: array[1..maxN] of Boolean;
    List: array[1..maxN] of Integer;
    nList: Integer;
procedure Enter; //Nhập dữ liệu
var i, x, y: Integer;
begin

```

```

ReadLn(p, q, m);
FillChar(head[1], p * SizeOf(head[1]), 0);
for i := 1 to m do
  begin
    ReadLn(x, y); //Đọc một cạnh (x, y), đưa y vào danh sách kề của x
    adj[i] := y;
    link[i] := head[x];
    head[x] := i;
  end;
end;
procedure Init; //Khởi tạo bộ ghép rỗng
var i: Integer;
begin
  FillChar(match[1], q * SizeOf(match[1]), 0);
  for i := 1 to p do List[i] := i;
  //Mảng List chứa nList X_đỉnh chưa ghép
  nList := p;
end;
procedure SuccessiveAugmentingPaths;
var
  Found: Boolean;
  Old, i: Integer;
procedure Visit(x: Integer); //Thuật toán DFS từ x ∈ X
var i, y: Integer;
begin
  i := head[x]; //Từ đầu danh sách kề của x
  while i <> 0 do
    begin
      y := adj[i]; //Xét một đỉnh y ∈ Y kề x
      if avail[y] then //y chưa thăm, hiển nhiên (x, y) là cạnh chưa ghép
        begin
          avail[y] := False; //Đánh dấu thăm y
          if match[y] = 0 then Found := True
          //y chưa ghép thì báo hiệu tìm thấy đường mở
          else Visit(match[y]);
          //Thăm luôn match[y] ∈ X (thăm liền 2 bước)
          if Found then //Tìm thấy đường mở
            begin
              match[y] := x; //Chỉnh lại bộ ghép
            
```

```

        Exit; //Thoát dây chuyền đê quy
    end;
end;
i := link[i]; //Chuyển sang đỉnh kế tiếp trong danh sách các đỉnh kè x
end;
end;
begin
repeat
    Old := nList; //Lưu lại số X_đỉnh chưa ghép
    FillChar(avail[1], q * SizeOf(avail[1]), True);
    for i := nList downto 1 do
        begin
            Found := False;
            Visit(List[i]); //Có ghép List[i]
            if Found then //Nếu ghép được
                begin //Xóa List[i] khỏi danh sách các X_đỉnh chưa ghép
                    List[i] := List[nList];
                    Dec(nList);
                end;
            end;
        until Old = nList; //Không thể ghép thêm X_đỉnh nào nữa
    end;
procedure PrintResult; //In kết quả
var j, k: Integer;
begin
    k := 0;
    for j := 1 to q do
        if match[j] <> 0 then
            begin
                Inc(k);
                WriteLn(k, ': x[' , match[j], '] - y[' , j, ']');
            end;
    end;
begin
    Enter;
    Init;
    SuccessiveAugmentingPath;
    PrintResult;

```

end.

Nếu đồ thị có n đỉnh ($n = p + q$) và m cạnh, do mảng đánh dấu $avail[1 \dots q]$ chỉ được khởi tạo một lần trong pha, thời gian thực hiện của một pha sẽ bằng $O(n + m)$ (suy ra từ thời gian thực hiện giải thuật của DFS).

Các pha sẽ được thực hiện lặp cho tới khi $X^* = \emptyset$ hoặc khi một pha thực hiện xong mà không ghép thêm được đỉnh nào. Thuật toán cần không quá p lần thực hiện pha xử lý lô, nên thời gian thực hiện giải thuật tìm bộ ghép cực đại trên đồ thị hai phía là $O(n^2 + nm)$ trong trường hợp xấu nhất. Còn trong trường hợp tốt nhất, ta có thể tìm được bộ ghép cực đại chỉ qua một lượt thực hiện pha xử lý lô, tức là bằng thời gian thực hiện giải thuật DFS. Cần lưu ý rằng đây chỉ là những đánh giá O lớn về cận trên của thời gian thực hiện. Thuật toán này chạy rất nhanh trên thực tế nhưng hiện tại chưa có đánh giá nào chặt hơn.

Ý tưởng tìm một lúc nhiều đường mở không có đỉnh chung đã được nghiên cứu trong bài toán luồng cực đại bởi Dinic[10]. Dựa trên ý tưởng này, Hopcroft và Karp[21] đã tìm ra thuật toán tìm bộ ghép cực đại trên đồ thị hai phía trong thời gian $O(\sqrt{|V||E|})$. Thuật toán Hopcroft-Karp trước hết sử dụng BFS để phân lớp các đỉnh theo độ dài đường đi ngắn nhất sau đó mới sử dụng DFS trên từng cây BFS để xử lý lô tương tự như cách làm của chúng ta ở trên.

Bài tập

2.35. Có p thợ và q việc. Mỗi thợ cho biết mình có thể làm được những việc nào, và mỗi việc khi giao cho một thợ thực hiện sẽ được hoàn thành xong trong đúng 1 đơn vị thời gian. Tại một thời điểm, mỗi thợ chỉ thực hiện không quá một việc.

Hãy phân công các thợ làm các công việc sao cho:

- Mỗi việc chỉ giao cho đúng một thợ thực hiện.
- Thời gian hoàn thành tất cả các công việc là nhỏ nhất. Chú ý là các thợ có thể thực hiện song song các công việc được giao, việc của ai người này làm, không ảnh hưởng tới người khác.

3.36. Một bộ ghép M trên đồ thị hai phía gọi là tối đai nếu việc bổ sung thêm bất cứ cạnh nào vào M sẽ làm cho M không còn là bộ ghép nữa.

- a) Chỉ ra một ví dụ về bộ ghép tối đai nhưng không là bộ ghép cực đai trên đồ thị hai phía
- b) Tìm thuật toán $O(|E|)$ để xác định một bộ ghép tối đai trên đồ thị hai phía
- c) Chứng minh rằng nếu A và B là hai bộ ghép tối đai trên cùng một đồ thị hai phía thì $|A| \leq 2|B|$ và $|B| \leq 2|A|$. Từ đó chỉ ra rằng nếu thuật toán đường mở được khởi tạo bằng một bộ ghép tối đai thì số lượt tìm đường mở giảm đi ít nhất một nửa so với việc khởi tạo bằng bộ ghép rỗng.

3.37. (Phủ đỉnh – Vertex Cover) Cho đồ thị hai phía $G = (X \cup Y, E)$. Bài toán đặt ra là hãy chọn ra một tập C gồm ít nhất các đỉnh sao cho mọi cạnh $\in E$ đều liên thuộc với ít nhất một đỉnh thuộc C .

Bài toán tìm phủ đỉnh nhỏ nhất trên đồ thị tổng quát là NP-đầy đủ, hiện tại chưa có thuật toán đa thức để giải quyết. Tuy vậy trên đồ thị hai phía, phủ đỉnh nhỏ nhất có thể tìm được dựa trên bộ ghép cực đai.

Dựa vào mô hình luồng của bài toán bộ ghép cực đai, giả sử các cung (X, Y) có sức chứa $+\infty$, các cung (s, X) và (Y, t) có sức chứa 1. Gọi (S, T) là lát cắt hẹp nhất của mạng. Đặt $C = \{x \in T\} \cup \{y \in S\}$.

- a) Chứng minh rằng C là một phủ đỉnh
- b) Chứng minh rằng C là phủ đỉnh nhỏ nhất
- c) Giả sử ta tìm được M là bộ ghép cực đai trên đồ thị hai phía, khi đó chắc chắn không còn tồn tại đường mở tương ứng với bộ ghép M . Đặt:

$$Y^* = \{y \in Y : \exists x \in X \text{ chưa ghép, } x \text{ đến được } y \text{ qua một đường pha}\}$$

$$X^* = \{x \in X : x \text{ đã ghép và đỉnh ghép với } x \text{ không thuộc } Y^*\}$$

Chứng minh rằng (X^*, Y^*) là lát cắt hẹp nhất.

- d) Xây dựng thuật toán tìm phủ đỉnh nhỏ nhất trên đồ thị hai phía dựa trên thuật toán tìm bộ ghép cực đai.

3.38. Cho M là một bộ ghép trên đồ thị hai phía $G = (X \cup Y, E)$. Gọi k là số X -đỉnh chưa ghép. Chứng minh rằng ba mệnh đề sau đây là tương đương:

- M là bộ ghép cực đại.
- G không có đường mở tương ứng với bộ ghép M .
- Tồn tại một tập con A của X sao cho $|N(A)| = |A| - k$. Ở đây $N(A)$ là tập các Y _đỉnh kề với một đỉnh nào đó trong A (Gợi ý: Chọn A là tập các X _đỉnh đến được từ một X _đỉnh chưa ghép bằng một đường pha)

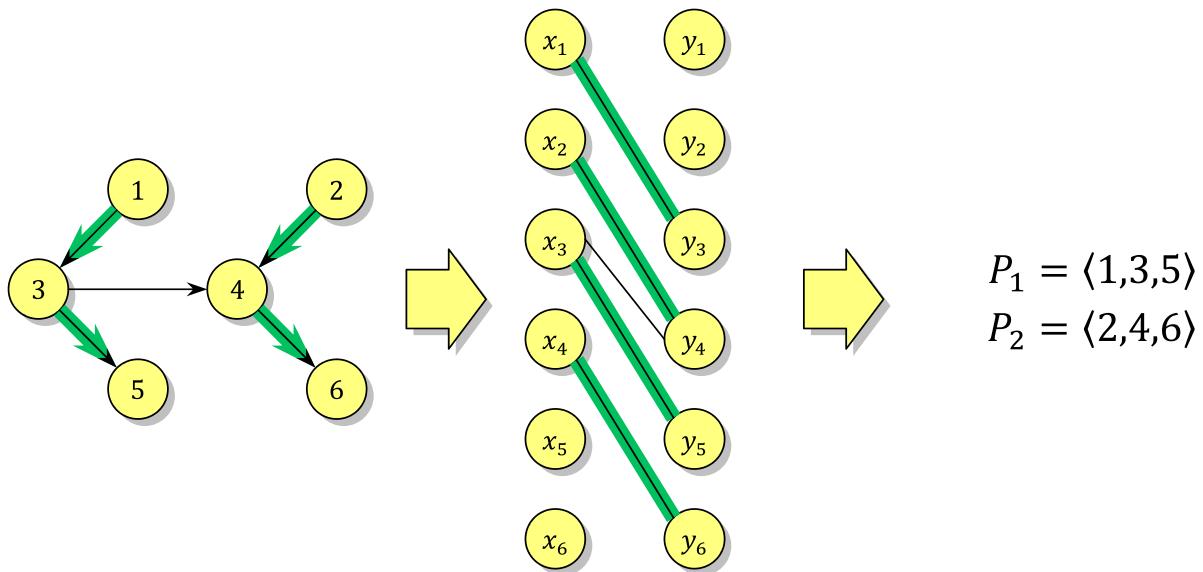
2.39. (Định lý Hall) Cho $G = (X \cup Y, E)$ là đồ thị hai phía có $|X| = |Y|$. Chứng minh rằng G có bộ ghép đầy đủ (bộ ghép mà mọi đỉnh đều được ghép) nếu và chỉ nếu $|A| \leq |N(A)|$ với mọi tập $A \subseteq X$.

2.40. (Phủ đường tối thiểu) Cho $G = (V, E)$ là đồ thị có hướng không có chu trình. Một *phủ đường* (*path cover*) là một tập P các đường đi trên G thỏa mãn: Với mọi đỉnh $v \in V$, tồn tại duy nhất một đường đi trong P chứa v . Đường đi có thể bắt đầu và kết thúc ở bất cứ đâu, tính cả đường đi độ dài 0 (chỉ gồm một đỉnh). Bài toán đặt ra là tìm *phủ đường tối thiểu* (*minimum path cover*): Phủ đường gồm ít đường đi nhất.

Gọi n là số đỉnh của đồ thị, ta đánh số các đỉnh thuộc V từ 1 tới n . Xây dựng đồ thị hai phía $G' = (X \cup Y, E')$ trong đó:

$$\begin{aligned} X &= \{x_1, x_2, \dots, x_n\} \\ Y &= \{y_1, y_2, \dots, y_n\} \end{aligned}$$

Tập cạnh E' được xây dựng như sau: Với mỗi cung $(i, j) \in E$, ta thêm vào một cạnh $(x_i, y_j) \in E'$ (h.2.16)



Hình 2.16. Bài toán tìm phủ đường tối thiểu trên DAG có thể quy về bài toán bộ ghép cực đại trên đồ thị hai phía.

Gọi M là một bộ ghép trên G' . Khởi tạo P là tập n đường đi, mỗi đường đi chỉ gồm một đỉnh trong G , khi đó P là một phủ đường. Xét lần lượt các cạnh của bộ ghép, mỗi khi xét tới cạnh (x_i, y_j) ta đặt cạnh (i, j) nối hai đường đi trong P thành một đường... Khi thuật toán kết thúc, P vẫn là một phủ đường.

- Chứng minh tính bất biến vòng lặp: Tại mỗi bước khi xét tới cạnh $(x_i, y_j) \in M$, cạnh $(i, j) \in E$ chắc chắn sẽ nối hai đường đi trong P : một đường đi kết thúc ở i và một đường đi khác bắt đầu ở j . Từ đó chỉ ra tính đúng đắn của thuật toán. (Gợi ý: mỗi khi xét tới cạnh $(x_i, y_j) \in M$ và đặt cạnh (i, j) nối hai đường đi của P thành một đường thì $|P|$ giảm 1. Vậy khi thuật toán trên kết thúc, $|P| = n - |M|$, tức là muốn $|P| \rightarrow \min$ thì $|M| \rightarrow \max$).
- Viết chương trình tìm phủ đường cực tiểu trên đồ thị có hướng không có chu trình.
- Chỉ ra ví dụ để thấy rằng thuật toán trên không đúng trong trường hợp G có chu trình.
- Chứng minh rằng nếu tìm được thuật toán giải bài toán tìm phủ đường cực tiểu trên đồ thị tổng quát trong thời gian đa thức thì có thể tìm được đường đi Hamilton trên đồ thị đó (nếu có) trong thời gian đa thức. (Lý

thuyết về độ phức tạp tính toán đã chứng minh được rằng trên đồ thị tổng quát, bài toán tìm đường đi Hamilton là NP-đầy đủ và bài toán tìm phủ đường cực tiểu là NP-khó. Có nghĩa là một thuật toán với độ phức tạp đa thức để giải quyết bài toán phủ đường cực tiểu trên đồ thị tổng quát sẽ là một phát minh lớn và đáng ngạc nhiên).

- 2.41.** Tự tìm hiểu về thuật toán Hopcroft-Karp. Cài đặt và so sánh tốc độ thực tế với thuật toán trong bài.
- 2.42.** (Bộ ghép cực đại trên đồ thị chính quy hai phía) Một đồ thị vô hướng $G = (V, E)$ gọi là *đồ thị chính quy bậc k* (k -regular graph) nếu bậc của mọi đỉnh đều bằng k . Đồ thị chính quy bậc 0 là đồ thị không có cạnh nào, đồ thị chính quy bậc 1 thì các cạnh tạo thành bộ ghép đầy đủ, đồ thị chính quy bậc 2 có các thành phần liên thông là các chu trình đơn.
- Chứng minh rằng đồ thị hai phía $G = (X \cup Y, E)$ là đồ thị chính quy thì $|X| = |Y|$.
 - Chứng minh rằng luôn tồn tại bộ ghép đầy đủ trên đồ thị hai phía chính quy bậc k ($k \geq 1$).
 - Tìm thuật toán $O(|E| \log|E|)$ để tìm một bộ ghép đầy đủ trên đồ thị chính quy bậc $k \geq 1$.