

§8. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

8.1. ĐỒ THỊ CÓ TRỌNG SỐ

Đồ thị mà mỗi cạnh của nó được gán cho tương ứng với một số (nguyên hoặc thực) được gọi là đồ thị có trọng số. Số gán cho mỗi cạnh của đồ thị được gọi là trọng số của cạnh. Tương tự như đồ thị không trọng số, có nhiều cách biểu diễn đồ thị có trọng số trong máy tính. Đối với đơn đồ thị thì cách dễ dàng nhất là sử dụng ma trận trọng số:

Giả sử đồ thị $G = (V, E)$ có n đỉnh. Ta sẽ dùng ma trận vuông C kích thước $n \times n$. Ở đây:

- ❖ Nếu $(u, v) \in E$ thì $C[u, v] =$ trọng số của cạnh (u, v)
- ❖ Nếu $(u, v) \notin E$ thì tùy theo trường hợp cụ thể, $C[u, v]$ được gán một giá trị nào đó để có thể nhận biết được (u, v) không phải là cạnh (Chẳng hạn có thể gán bằng $+\infty$, hay bằng 0, bằng $-\infty$, v.v...)
- ❖ Quy ước $c[v, v] = 0$ với mọi đỉnh v .

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 phải tính bằng số cạnh đi qua, mà được tính bằng tổng trọng số của các cạnh đi qua.

8.2. BÀI TOÁN ĐƯỜNG ĐI NGẮN NHẤT

Trong các ứng dụng thực tế, chẳng hạn trong mạng lưới giao thông đường bộ, đường thuỷ hoặc đường không. 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 chi phí). Khi đó phát sinh yêu cầu tìm đường đi ngắn nhất giữa hai đỉnh của đồ thị. Bài toán đó phát biểu dưới dạng tổng quát như sau: Cho đồ thị có trọng số $G = (V, E)$, hãy tìm một đường đi ngắn nhất từ đỉnh xuất phát $s \in V$ đến đỉnh đích $f \in V$. Độ dài của đường đi này ta sẽ ký hiệu là $d[s, f]$ và gọi là **khoảng cách** (distance) từ s đến f . Nếu như không tồn tại đường đi từ s tới f thì ta sẽ đặt khoảng cách đó = $+\infty$.

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 cơ bản** (đường đi không có đỉnh lặp lại) ngắn nhất. Vấn đề đó là một vấn đề hết sức phức tạp mà ta sẽ không bàn tới ở đây.

Nếu như đồ thị không có chu trình âm thì ta có thể chứng minh được rằng một trong những đường đi ngắn nhất là đường đi cơ bản. Và nếu như biết được khoảng cách từ s tới tất cả những đỉnh khác thì đường đi ngắn nhất từ s tới f có thể tìm được một cách dễ dàng qua thuật toán sau:

Gọi $c[u, v]$ là trọng số của cạnh $[u, v]$. Qui ước $c[v, v] = 0$ với mọi $v \in V$ và $c[u, v] = +\infty$ nếu như $(u, v) \notin E$. Đặt $d[s, v]$ là khoảng cách từ s tới v . Để tìm đường đi từ s tới f , ta có thể nhận thấy rằng luôn tồn tại đỉnh $f_1 \neq f$ sao cho:

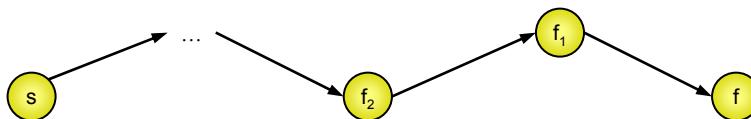
$$d[s, f] = d[s, f_1] + c[f_1, f]$$

(Độ dài đường đi ngắn nhất $s \rightarrow f$ = Độ dài đường đi ngắn nhất $s \rightarrow f_1$ + Chi phí đi từ $f_1 \rightarrow f$)

Đỉnh f_1 đó là đỉnh liền trước f trong đường đi ngắn nhất từ s tới f . Nếu $f_1 = s$ thì đường đi ngắn nhất là đường đi trực tiếp theo cung (s, f) . Nếu không thì vẫn đề trở thành tìm đường đi ngắn nhất từ s tới f_1 . Và ta lại tìm được một đỉnh f_2 khác f và f_1 để:

$$d[s, f_1] = d[s, f_2] + c[f_2, f_1]$$

Cứ tiếp tục như vậy, sau một số hữu hạn bước, ta suy ra rằng dãy f, f_1, f_2, \dots không chứa đỉnh lặp lại và kết thúc ở s . Lật ngược thứ tự dãy cho ta đường đi ngắn nhất từ s tới f .



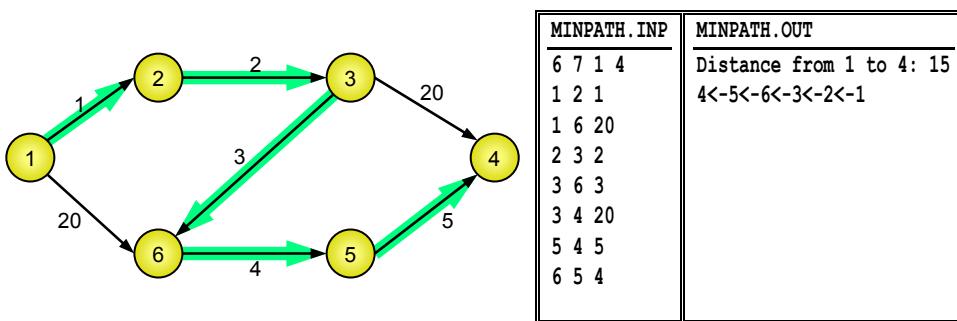
Tuy nhiên, người ta thường không sử dụng phương pháp này mà sẽ kết hợp lưu vết đường đi ngay trong quá trình tìm kiếm.

Dưới đây ta sẽ xét một số thuật toán tìm đường đi ngắn nhất từ đỉnh s tới đỉnh f trên đơn đồ thị có hướng $G = (V, E)$ có n đỉnh và m cung. Trong trường hợp đơn đồ thị vô hướng với trọng số không âm, bài toán tìm đường đi ngắn nhất có thể dẫn về bài toán trên đồ thị có hướng bằng cách thay mỗi cạnh của nó bằng hai cung có hướng ngược chiều nhau. Bạn có thể đưa vào một số sửa đổi nhỏ trong thủ tục nhập liệu để giải quyết bài toán trong trường hợp đa đồ thị

Input: file văn bản MINPATH.INP

- ❖ Dòng 1: Chứa số đỉnh n (≤ 1000), số cung m của đồ thị, đỉnh xuất phát s , đỉnh đích f cách nhau ít nhất 1 dấu cách
- ❖ m dòng tiếp theo, mỗi dòng có dạng ba số $u, v, c[u, v]$ cách nhau ít nhất 1 dấu cách, thể hiện (u, v) là một cung $\in E$ và trọng số của cung đó là $c[u, v]$ ($c[u, v]$ là số nguyên có giá trị tuyệt đối ≤ 1000)

Output: file văn bản MINPATH.OUT ghi đường đi ngắn nhất từ s tới f và độ dài đường đi đó



8.3. TRƯỜNG HỢP ĐỒ THỊ KHÔNG CÓ CHU TRÌNH ÂM - THUẬT TOÁN FORD-BELLMAN

Thuật toán Ford-Bellman có thể phát biểu rất đơn giản:

Với đỉnh xuất phát s. Gọi $d[v]$ là khoảng cách từ s tới v với các giá trị khởi tạo là:

- ❖ $d[s] := 0$
- ❖ $d[v] := +\infty$ nếu $v \neq s$

Sau đó ta tối ưu hoá dần các $d[v]$ như sau: Xét mọi cặp đỉnh u, v của đồ thị, nếu có một cặp đỉnh u, v mà $d[v] > d[u] + c[u, v]$ thì ta đặt lại $d[v] := d[u] + c[u, v]$. Tức là nếu độ dài đường đi từ s tới v lại **lớn hơn** tổng độ dài đường đi từ s tới u cộng với chi phí đi từ u tới v thì ta sẽ huỷ bỏ đường đi từ s tới v đang có và coi đường đi từ s tới v chính là đường đi từ s tới u sau đó đi tiếp từ u tới v. Chú ý rằng ta đặt $c[u, v] = +\infty$ nếu (u, v) không là cung. Thuật toán sẽ kết thúc khi không thể tối ưu thêm bất kỳ một nhãn $d[v]$ nào nữa.

```

for ( $\forall v \in V$ ) do  $d[v] := +\infty$ ;
 $d[s] := 0$ ;
repeat
  Stop := True;
  for ( $\forall u \in V$ ) do
    for ( $\forall v \in V: (u, v) \in E$ ) do
      if  $d[v] > d[u] + c[u, v]$  then
        begin
           $d[v] := d[u] + c[u, v]$ ;
          Stop := False;
        end;
    until Stop;
  
```

Tính đúng của thuật toán:

Tại bước khởi tạo thì mỗi $d[v]$ chính là độ dài ngắn nhất của đường đi từ s tới v qua không quá 0 cạnh.

Giả sử khi bắt đầu bước lặp thứ i ($i \geq 1$), $d[v]$ đã bằng độ dài đường đi ngắn nhất từ s tới v qua không quá $i - 1$ cạnh. Bởi đường đi từ s tới v qua không quá $i - 1$ cạnh sẽ phải thành lập bằng cách: lấy một đường đi từ s tới một đỉnh u nào đó qua không quá $i - 1$ cạnh, rồi đi tiếp tới v bằng cung (u, v) , nên độ dài đường đi ngắn nhất từ s tới v qua không quá $i - 1$ cạnh sẽ được tính bằng giá trị nhỏ nhất trong các giá trị (Nguyên lý tối ưu Bellman):

- ❖ Độ dài đường đi ngắn nhất từ s tới v qua không quá $i - 1$ cạnh

- ❖ Độ dài đường đi ngắn nhất từ s tới u qua không quá i - 1 cạnh cộng với trọng số cạnh (u, v)
 $(\forall u)$

Vì vậy, sau bước lặp tối ưu các $d[v]$ bằng công thức

$$d[v]_{\text{bước } i} = \min(d[v]_{\text{bước } i-1}, d[u]_{\text{bước } i-1} + c[u, v]) \quad (\forall u)$$

thì các $d[v]$ sẽ bằng độ dài đường đi ngắn nhất từ s tới v qua không quá i cạnh.

Sau bước lặp tối ưu thứ n - 1, ta có $d[v] =$ độ dài đường đi ngắn nhất từ s tới v qua không quá n - 1 cạnh. Vì đồ thị không có chu trình âm nên sẽ có một đường đi ngắn nhất từ s tới v là đường đi cơ bản (qua không quá n - 1 cạnh). Tức là $d[v]$ sẽ là độ dài đường đi ngắn nhất từ s tới v.

Vậy thì số bước lặp tối ưu hoá sẽ không quá n - 1 bước.

Trong khi cài đặt chương trình, nếu mỗi bước lặp được mô tả dưới dạng:

```
for u := 1 to n do
  for v := 1 to n do
    d[v] := min(d[v], d[u] + c[u, v]);
```

Sự tối ưu bắc cầu (dùng $d[u]$ tối ưu $d[v]$ rồi lại có thể dùng $d[v]$ tối ưu $d[w]$ nữa...) chỉ làm tốc độ tối ưu các nhãn $d[.]$ tăng nhanh hơn nên số bước lặp vẫn sẽ không quá n - 1 bước

```
P_4_08_1.PAS * Thuật toán Ford-Bellman
{$MODE DELPHI} (*This program uses 32-bit Integer [-2^31..2^31 - 1]*)
program Finding_the_Shortest_Path_using_Ford_Bellman_Algorithm;
const
  InputFile = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 1000;
  maxEC = 1000;
  maxC = max * maxEC;
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Trace: array[1..max] of Integer;
  n, s, f: Integer;

procedure LoadGraph; {Nhập đồ thị, đồ thị không được có chu trình âm}
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, s, f);
  {Những cạnh không có trong đồ thị được gán trọng số +∞}
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure Init; {Khởi tạo}
var
  i: Integer;
begin
  for i := 1 to n do d[i] := MaxC;
  d[s] := 0;
```

```

end;

procedure Ford_Bellman; {Thuật toán Ford-Bellman}
var
  Stop: Boolean;
  u, v, CountLoop: Integer;
begin
  for CountLoop := 1 to n - 1 do
    begin
      Stop := True;
      for u := 1 to n do
        for v := 1 to n do
          if d[v] > d[u] + c[u, v] then {Nếu  $\exists u, v$  thoả mãn  $d[v] > d[u] + c[u, v]$  thì tối ưu lại  $d[v]$ }
            begin
              d[v] := d[u] + c[u, v];
              Trace[v] := u; {Lưu vết đường đi}
              Stop := False;
            end;
      if Stop then Break;
    end;
  {Thuật toán kết thúc khi không sửa nhãn các  $d[v]$  được nữa hoặc đã lặp đủ  $n - 1$  lần}
end;

procedure PrintResult; {In đường đi từ s tới f}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[f] = maxC then {Nếu  $d[f]$  vẫn là  $+\infty$  thì tức là không có đường}
    writeln(fo, 'There is no path from ', s, ' to ', f)
  else {Truy vết tìm đường đi}
    begin
      writeln(fo, 'Distance from ', s, ' to ', f, ': ', d[f]);
      while f <> s do
        begin
          write(fo, f, '-');
          f := Trace[f];
        end;
      writeln(fo, s);
    end;
  Close(fo);
end;

begin
  LoadGraph;
  Init;
  Ford_Bellman;
  PrintResult;
end.

```

8.4. TRƯỜNG HỢP TRỌNG SỐ TRÊN CÁC CUNG KHÔNG ÂM - THUẬT TOÁN DIJKSTRA

Trong trường hợp trọng số trên các cung không âm, thuật toán do Dijkstra đề xuất dưới đây hoạt động hiệu quả hơn nhiều so với thuật toán Ford-Bellman. Ta hãy xem trong trường hợp này, thuật toán Ford-Bellman thiếu hiệu quả ở chỗ nào:

Với đỉnh $v \in V$, Gọi $d[v]$ là độ dài đường đi ngắn nhất từ s tới v . Thuật toán Ford-Bellman khởi gán $d[s] = 0$ và $d[v] = +\infty$ với $\forall v \neq s$, sau đó tối ưu hoá dần các nhãn $d[v]$ bằng cách sửa

nhân theo công thức: $d[v] := \min(d[v], d[u] + c[u, v])$ với $\forall u, v \in V$. Như vậy nếu như ta dùng đỉnh u sửa nhân đỉnh v , sau đó nếu ta lại tối ưu được $d[u]$ thêm nữa thì ta cũng phải sửa lại nhân $d[v]$ dẫn tới việc $d[v]$ có thể phải chỉnh đi chỉnh lại rất nhiều lần. Vậy nên chăng, tại mỗi bước **không phải ta xét mọi cặp đỉnh (u, v)** để dùng đỉnh u sửa nhân đỉnh v mà sẽ **chọn đỉnh u là đỉnh mà không thể tối ưu nhân $d[u]$ thêm được nữa**.

Thuật toán Dijkstra (E.Dijkstra - 1959) có thể mô tả như sau:

Bước 1: Khởi tạo

Với đỉnh $v \in V$, gọi nhân $d[v]$ là độ dài đường đi ngắn nhất từ s tới v . Ban đầu $d[v]$ được khởi gán như trong thuật toán Ford-Bellman ($d[s] = 0$ và $d[v] = \infty$ với $\forall v \neq s$). Nhân của mỗi đỉnh có hai trạng thái tự do hay cố định, nhân tự do có nghĩa là có thể còn tối ưu hơn được nữa và nhân cố định tức là $d[v]$ đã bằng độ dài đường đi ngắn nhất từ s tới v nên không thể tối ưu thêm. Để làm điều này ta có thể sử dụng kỹ thuật đánh dấu: $\text{Free}[v] = \text{TRUE}$ hay FALSE tùy theo $d[v]$ tự do hay cố định. Ban đầu các nhân đều tự do.

Bước 2: Lặp

Bước lặp gồm có hai thao tác:

- ❖ Cố định nhân: Chọn trong các đỉnh có nhân tự do, lấy ra đỉnh u là đỉnh có $d[u]$ nhỏ nhất, và cố định nhân đỉnh u .
- ❖ Sửa nhân: Dùng đỉnh u , xét tất cả những đỉnh v và sửa lại các $d[v]$ theo công thức:

$$d[v] := \min(d[v], d[u] + c[u, v])$$

Bước lặp sẽ kết thúc khi mà đỉnh đích f được cố định nhân (tìm được đường đi ngắn nhất từ s tới f); hoặc tại thao tác cố định nhân, tất cả các đỉnh tự do đều có nhân là $+\infty$ (không tồn tại đường đi). Có thể đặt câu hỏi, ở thao tác 1, tại sao đỉnh u như vậy được cố định nhân, giả sử $d[u]$ còn có thể tối ưu thêm được nữa thì tất phải có một đỉnh t mang nhân tự do sao cho $d[u] > d[t] + c[t, u]$. Do trọng số $c[t, u]$ không âm nên $d[u] > d[t]$, trái với cách chọn $d[u]$ là nhỏ nhất. Tuy nhiên trong lần lặp đầu tiên thì s là đỉnh được cố định nhân do $d[s] = 0$.

Bước 3: Kết hợp với việc lưu vết đường đi trên từng bước sửa nhân, thông báo đường đi ngắn nhất tìm được hoặc cho biết không tồn tại đường đi ($d[f] = +\infty$).

```
for (v ∈ V) do d[v] := +∞;
d[s] := 0;
repeat
  u := arg min{d[v] | v ∈ V}; {Lấy u là đỉnh có nhân d[u] nhỏ nhất}
  if (u = f) or (d[u] = +∞) then Break; {Hoặc tìm ra đường đi ngắn nhất từ s tới f, hoặc kết luận không có đường}
  for (v ∈ V: (u, v) ∈ E) do {Dùng u tối ưu nhân những đỉnh v kề với u}
    d[v] := min (d[v], d[u] + c[u, v]);
until False;
```

P_4_08_2.PAS * Thuật toán Dijkstra

```
{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)
program Finding_the_Shortest_Path_using_Dijkstra_Algorithm;
const
  InputFile = 'MINPATH.INP';
```

```

OutputFile = 'MINPATH.OUT';
max = 1000;
maxEC = 1000;
maxC = max * maxEC;
var
  c: array[1..max, 1..max] of Integer;
  d: array[1..max] of Integer;
  Trace: array[1..max] of Integer;
  Free: array[1..max] of Boolean; {Free[u] = True ⇔ u có nhãn tự do}
  n, s, f: Integer;

procedure LoadGraph; {Nhập đồ thị, trọng số các cung phải là số không âm}
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, s, f);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure Init; {Khởi tạo các nhãn d[v], các đỉnh đều được coi là tự do}
var
  i: Integer;
begin
  for i := 1 to n do d[i] := MaxC;
  d[s] := 0;
  FillChar(Free, SizeOf(Free), True);
end;

procedure Dijkstra; {Thuật toán Dijkstra}
var
  i, u, v: Integer;
  min: Integer;
begin
  repeat
    {Tìm trong các đỉnh có nhãn tự do ra đỉnh u có d[u] nhỏ nhất}
    u := 0; min := maxC;
    for i := 1 to n do
      if Free[i] and (d[i] < min) then
        begin
          min := d[i];
          u := i;
        end;
  until False;
  {Thuật toán sẽ kết thúc khi các đỉnh tự do đều có nhãn +∞ hoặc đã chọn đến đỉnh f}
  if (u = 0) or (u = f) then Break;
  {Có định nhãn đỉnh u}
  Free[u] := False;
  {Dùng đỉnh u tối ưu nhãn những đỉnh tự do kề với u}
  for v := 1 to n do
    if Free[v] and (d[v] > d[u] + c[u, v]) then
      begin
        d[v] := d[u] + c[u, v];
        Trace[v] := u;
      end;
  until False;
end;

procedure PrintResult; {In đường đi từ s tới f}

```

```

var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[f] = maxC then
    WriteLn(fo, 'There is no path from ', s, ' to ', f)
  else
    begin
      begin
        WriteLn(fo, 'Distance from ', s, ' to ', f, ': ', d[f]);
        while f <> s do
          begin
            Write(fo, f, '<-');
            f := Trace[f];
          end;
        WriteLn(fo, s);
      end;
    Close(fo);
  end;

begin
  LoadGraph;
  Init;
  Dijkstra;
  PrintResult;
end.

```

8.5. THUẬT TOÁN DIJKSTRA VÀ CẤU TRÚC HEAP

Nếu đồ thị thưa (có nhiều đỉnh, ít cạnh) ta có thể sử dụng danh sách kè kèm trọng số để biểu diễn đồ thị, tuy nhiên tốc độ của thuật toán Dijkstra vẫn khá chậm vì trong trường hợp xấu nhất, nó cần n lần cố định nhãn và mỗi lần tìm đỉnh để cố định nhãn sẽ mất một đoạn chương trình với độ phức tạp $O(n)$. Để thuật toán làm việc hiệu quả hơn, người ta thường sử dụng cấu trúc dữ liệu Heap (PHẦN 2, §8, 8.7.1) để lưu các đỉnh chưa cố định nhãn. Heap ở đây là một cây nhị phân hoàn chỉnh thoả mãn: Nếu u là đỉnh lưu ở nút cha và v là đỉnh lưu ở nút con thì $d[u] \leq d[v]$. (Đỉnh r lưu ở gốc Heap là đỉnh có $d[r]$ nhỏ nhất).

Tại mỗi bước lặp của thuật toán Dijkstra có hai thao tác: Tìm đỉnh cố định nhãn và Sửa nhãn.

- ❖ Với mỗi đỉnh v, gọi $Pos[v]$ là vị trí đỉnh v trong Heap, quy ước $Pos[v] = 0$ nếu v chưa bị đẩy vào Heap. Mỗi lần có thao tác sửa đổi vị trí các đỉnh trên cấu trúc Heap, ta lưu ý cập nhập lại mảng Pos này.
- ❖ Thao tác tìm đỉnh cố định nhãn sẽ lấy đỉnh lưu ở gốc Heap, cố định nhãn, đưa phần tử cuối Heap vào thế chỗ và thực hiện việc vun đồng (Adjust).
- ❖ Thao tác sửa nhãn, sẽ duyệt danh sách kè của đỉnh vừa cố định nhãn và sửa nhãn những đỉnh tự do kè với đỉnh này, mỗi lần sửa nhãn một đỉnh nào đó (nhãn trọng số $d[.]$ bị giảm đi), ta xác định đỉnh này nằm ở đâu trong Heap (dựa vào mảng Pos) và thực hiện việc chuyển đỉnh đó lên (UpHeap) phía gốc Heap nếu cần để bảo toàn cấu trúc Heap.

Cài đặt dưới đây có Input/Output giống như trên nhưng có thể thực hiện trên đồ thị 10000 đỉnh, 100000 cạnh, trọng số mỗi cạnh ≤ 100000 .

```

P_4_08_3.PAS * Thuật toán Dijkstra và cấu trúc Heap

{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)
program Finding_the_Shortest_Path_using_Dijkstra_Algorithm_with_Heap;
const
  InputFile = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 10000;
  maxE = 100000;
  maxEC = 100000;
  maxC = max * maxEC;
type
  TAdj = array[1..maxE] of Integer;
  TAdjCost = array[1..maxE] of Integer;
  THeader = array[1..max + 1] of Integer;
var
  adj: TAdj; {Danh sách kề dạng mảng}
  adjCost: TAdjCost; {Kèm trọng số}
  h: THeader; {Mảng đánh dấu các đoạn trong danh sách kề adj}
  d: array[1..max] of Integer;
  Trace: array[1..max] of Integer;
  Free: array[1..max] of Boolean;
  heap: array[1..max] of Integer; {heap[i] = đỉnh lưu tại nút i của heap}
  Pos: array[1..max] of Integer; {pos[v] = vị trí của nút v trong heap (tức là pos[heap[i]] = i)}
  n, s, f, nHeap: Integer;

procedure LoadGraph; {Nhập dữ liệu}
var
  i, m, u, v, c: Integer;
  fi: Text;
begin
  {Đọc file lần 1, để xác định các đoạn}
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, s, f);
  {Phép đếm phân phối (Distribution Counting)}
  FillChar(h, SizeOf(h), 0);
  for i := 1 to m do
    begin
      ReadLn(fi, u); {Ta chỉ cần tính bán bậc ra (deg+) của mỗi đỉnh nên không cần đọc đủ 3 thành phần}
      Inc(h[u]);
    end;
  for i := 2 to n do h[i] := h[i - 1] + h[i];
  Close(fi);
  {Đến đây, ta xác định được h[u] là vị trí cuối của danh sách kề đỉnh u trong adj}
  Reset(fi); {Đọc file lần 2, vào cấu trúc danh sách kề}
  ReadLn(fi); {Bỏ qua dòng đầu tiên Input file}
  for i := 1 to m do
    begin
      ReadLn(fi, u, v, c);
      adj[h[u]] := v; {Diễn v và c vào vị trí đúng trong danh sách kề của u}
      adjCost[h[u]] := c;
      Dec(h[u]);
    end;
  h[n + 1] := m;
  Close(fi);
end;

procedure Init; {Khởi tạo d[i] = độ dài đường đi ngắn nhất từ s tới i qua 0 cạnh, Heap rỗng}
var
  i: Integer;
begin
  for i := 1 to n do d[i] := maxC;
  d[s] := 0;
  FillChar(Free, SizeOf(Free), True);

```

```

FillChar(Pos, SizeOf(Pos), 0);
nHeap := 0;
end;

procedure Update(v: Integer); {Định v vừa được sửa nhẫn, cần phải chỉnh lại Heap}
var
  parent, child: Integer;
begin
  child := Pos[v]; {child là vị trí của v trong Heap}
  if child = 0 then {Nếu v chưa có trong Heap thì Heap phải bổ sung thêm 1 phần tử và coi child = nút lá cuối Heap}
    begin
      Inc(nHeap); child := nHeap;
    end;
  parent := child div 2; {parent là nút cha của child}
  while (parent > 0) and (d[heap[parent]] > d[v]) do
    begin {Nếu đỉnh lưu ở nút parent ưu tiên kém hơn v thì đỉnh đó sẽ bị đẩy xuống nút con child}
      heap[child] := heap[parent]; {Đẩy đỉnh lưu trong nút cha xuống nút con}
      Pos[heap[child]] := child; {Ghi nhận lại vị trí mới của đỉnh đó}
      child := parent; {Tiếp tục xét lên phía nút gốc}
      parent := child div 2;
    end;
  {Thao tác "kéo xuống" ở trên tạo ra một "khoảng trống" tại nút child của Heap, đỉnh v sẽ được đặt vào đây}
  heap[child] := v;
  Pos[v] := child;
end;

function Pop: Integer; {Lấy từ Heap ra đỉnh có nhẫn tự do nhỏ nhất}
var
  r, c, v: Integer;
begin
  r := heap[1]; {Nút gốc Heap chứa đỉnh có nhẫn tự do nhỏ nhất}
  v := heap[nHeap]; {v là đỉnh ở nút lá cuối Heap, sẽ được đảo lên đầu và vun đồng}
  Dec(nHeap);
  r := 1; {Bắt đầu từ nút gốc}
  while r * 2 <= nHeap do {Chừng nào r chưa phải là lá}
    begin
      {Chọn c là nút chứa đỉnh ưu tiên hơn trong hai nút con}
      c := r * 2;
      if (c < nHeap) and (d[heap[c + 1]] < d[heap[c]]) then Inc(c);
      {Nếu v ưu tiên hơn cả đỉnh chứa trong C, thì thoát ngay}
      if d[v] <= d[heap[c]] then Break;
      heap[r] := heap[c]; {Chuyển đỉnh lưu ở nút con c lên nút cha r}
      Pos[heap[r]] := r; {Ghi nhận lại vị trí mới trong Heap của đỉnh đó}
      r := c; {Gán nút cha := nút con và lặp lại}
    end;
  heap[r] := v; {Định v sẽ được đặt vào nút r để bảo toàn cấu trúc Heap}
  Pos[v] := r;
end;

procedure Dijkstra;
var
  i, u, iv, v, min: Integer;
begin
  Update(s); {Đưa đỉnh xuất phát về gốc Heap}
  repeat
    u := Pop; {Chọn đỉnh tự do có nhẫn nhỏ nhất}
    if u = f then Break; {Nếu đỉnh đó là f thì dừng ngay}
    Free[u] := False; {Cô định nhẫn đỉnh đó}
    for iv := h[u] + 1 to h[u + 1] do {Xét danh sách kề}
      begin
        v := adj[iv];
        if Free[v] and (d[v] > d[u] + adjCost[iv]) then
          begin
            d[v] := d[u] + adjCost[iv];
            Free[v] := True;
          end;
      end;
  until Break;
end;

```

```

d[v] := d[u] + adjCost[iv]; {Tối ưu hóa nhãn của các đỉnh tự do kề với u}
Trace[v] := u; {Lưu vết đường đi}
Update(v); {Tổ chức lại Heap}
end;
end;
until nHeap = 0; {Không còn đỉnh nào mang nhãn tự do}
end;

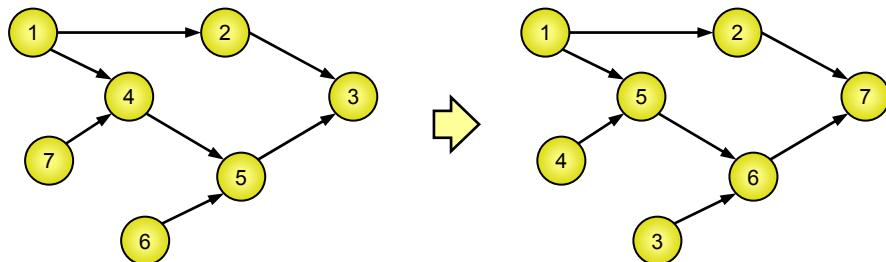
procedure PrintResult;
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[f] = maxC then
    WriteLn(fo, 'There is no path from ', s, ' to ', f)
  else
    begin
      WriteLn(fo, 'Distance from ', s, ' to ', f, ': ', d[f]);
      while f <> s do
        begin
          Write(fo, f, '<-');
          f := Trace[f];
        end;
      WriteLn(fo, s);
    end;
  Close(fo);
end;

begin
  LoadGraph;
  Init;
  Dijkstra;
  PrintResult;
end.

```

8.6. TRƯỜNG HỢP ĐỒ THỊ KHÔNG CÓ CHU TRÌNH - SẮP XẾP TÔ PÔ

Xét trường hợp đồ thị có hướng, không có chu trình (Directed Acyclic Graph - DAG), ta có một thuật toán hiệu quả 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 76: Phép đánh lại chỉ số theo thứ tự tốpô

Để đánh số lại các đỉnh theo điều kiện trên, ta có thể dùng thuật toán tìm kiếm theo chiều sâu và đánh số ngược lại với thứ tự duyệt xong, thuật toán có thể viết:

```

procedure Number;
  procedure Visit(u: Integer);
  var

```

```

v: Integer;
begin
  {Đánh dấu u đã thăm};
  for (forall v in V: v chưa thăm kè với u) do Visit(v); {Thăm tiếp những đỉnh chưa thăm kè với u}
  {Đã duyệt xong nhánh DFS gốc u, đánh số mới cho đỉnh u là count};
  count := count - 1;
end;

begin
  {Đánh dấu mọi đỉnh ∈ V đều chưa thăm};
  count := n; {Biến đánh số được khởi tạo bằng n để đếm lùi}
  for u := 1 to n do
    if (u chưa thăm) then Visit(u);
end;

```

Việc kiểm tra đồ thị không được có chu trình cũng rất đơn giản bằng cách thêm vào đoạn chương trình trên vài dòng lệnh, tôi sẽ không viết dài để tập trung vào thuật toán, các bạn có thể tham khảo các kỹ thuật trình bày trong thuật toán Tarjan (§4, 4.4.3).

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 mô tả rất đơn giản:

Gọi $d[v]$ là độ dài đường đi ngắn nhất từ s tới v . Khởi tạo $d[s] = 0$ và $d[v] = +\infty$ với $\forall v \neq s$. Ta sẽ tính các $d[v]$ như sau:

```

for u := 1 to n - 1 do
  for v := u + 1 to n do
    d[v] := min(d[v], d[u] + c[u, v]);

```

(Giả thiết rằng $c[u, v] = +\infty$ nếu như (u, v) không là cung).

Tức là dùng đỉnh u , tối ưu nhãn $d[v]$ của những đỉnh v nối từ u , với u được xét lần lượt từ 1 tới $n - 1$. Có thể làm tốt hơn nữa bằng cách chỉ cần cho u chạy từ đỉnh xuất phát tới đỉnh kết thúc. Bởi lẽ u chạy tới đâu thì nhãn $d[u]$ là không thể cực tiểu hoá thêm nữa.

```

P_4_08_4.PAS * Đường đi ngắn nhất trên đồ thị không có chu trình
{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)
program The_Critical_Path;
const
  InputFile = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 1000;
  maxEC = 1000;
  maxC = max * maxEC;
var
  c: array[1..max, 1..max] of Integer;
  List, d, Trace: array[1..max] of Integer; {List chứa danh sách các đỉnh theo thứ tự đánh số mới}
  n, s, f: Integer;

procedure LoadGraph; {Nhập dữ liệu}
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, s, f);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);

```

```

end;

procedure Numbering; {Thuật toán đánh số các đỉnh}
var
  Free: array[1..max] of Boolean;
  u, count: Integer;

procedure Visit(u: Integer);
var
  v: Integer;
begin
  Free[u] := False;
  for v := 1 to n do
    if Free[v] and (c[u, v] <> maxC) then Visit(v);
  List[count] := u;
  Dec(count);
end;

begin
  FillChar(Free, SizeOf(Free), True);
  count := n;
  for u := 1 to n do
    if Free[u] then Visit(u);
end;

procedure Init; {Khởi tạo các nhãn trọng số}
var
  i: Integer;
begin
  for i := 1 to n do d[i] := maxC;
  d[s] := 0;
end;

procedure FindPath; {Thuật toán quy hoạch động tối ưu hoá các d[.]}
var
  i, j, u, v: Integer;
begin
  for i := 1 to n - 1 do
    for j := i + 1 to n do
      begin
        u := List[i]; v := List[j]; {Ánh xạ ngược i, j sang chỉ số cũ u, v}
        if d[v] > d[u] + c[u, v] then
          begin
            d[v] := d[u] + c[u, v];
            Trace[v] := u;
          end
      end;
end;

procedure PrintResult; {In kết quả}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if d[f] = maxC then
    WriteLn(fo, 'There is no path from ', s, ' to ', f)
  else
    begin
      WriteLn(fo, 'Distance from ', s, ' to ', f, ': ', d[f]);
      while f <> s do
        begin
          Write(fo, f, '<-');
          f := Trace[f];
        end;
    end;
end;

```

```

    end;
    WriteLn(fo, s);
  end;
  Close(fo);
end;

begin
  LoadGraph;
  Numbering;
  Init;
  FindPath;
  PrintResult;
end.

```

8.7. ĐƯỜNG ĐI NGẮN NHẤT GIỮA MỌI CẶP ĐỈNH - THUẬT TOÁN FLOYD

Cho đơn đồ thị có hướng, có trọng số $G = (V, E)$ với n đỉnh và m cạnh. Bài toán đặt ra là hãy tính tất cả các $d(u, v)$ là khoảng cách từ u tới v . Rõ ràng là ta có thể áp dụng thuật toán tìm đường đi ngắn nhất xuất phát từ một đỉnh với n khả năng chọn đỉnh xuất phát. Nhưng ta có cách làm gọn hơn nhiều, cách làm này rất giống với thuật toán Warshall mà ta đã biết: Từ ma trận trọng số c , thuật toán Floyd tính lại các $c[u, v]$ thành độ dài đường đi ngắn nhất từ u tới v : Với mọi đỉnh k của đồ thị được xét theo thứ tự từ 1 tới n , xét mọi cặp đỉnh u, v . Cực tiêu hoá $c[u, v]$ theo công thức:

$$c[u, v] := \min(c[u, v], c[u, k] + c[k, v])$$

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 (Chú ý rằng ta còn có việc lưu lại vết):

```

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

```

Tính đúng của thuật toán:

Gọi $c_k[u, v]$ 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 $c^0[u, v] = c[u, v]$ (đường đi ngắn nhất là đường đi trực tiếp).

Giả sử ta đã tính được các $c^{k-1}[u, v]$ thì $c^k[u, v]$ sẽ được xây dựng như sau:

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 thì tức là chỉ qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k-1\}$ thì

$$c^k[u, v] = c^{k-1}[u, v]$$

❖ Có đi qua đỉnh k thì đường đi đó sẽ là nối của một đường đi từ u tới k và một đường đi từ k tới v , hai đường đi này chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, k-1\}$.

$$c^k[u, v] = c^{k-1}[u, k] + c^{k-1}[k, v].$$

Vì ta muốn $c^k[u, v]$ là cực tiểu nên suy ra: $c^k[u, v] = \min(c^{k-1}[u, v], c^{k-1}[u, k] + c^{k-1}[k, v]).$

Và cuối cùng, ta quan tâm tới $c^n[u, v]$: Độ dài đường đi ngắn nhất từ u tới v mà chỉ đi qua các đỉnh trung gian thuộc tập $\{1, 2, \dots, n\}$.

Khi cài đặt, thì ta sẽ không có các khái niệm $c^k[u, v]$ mà sẽ thao tác trực tiếp trên các trọng số $c[u, v]$. $c[u, v]$ tại bước tối ưu thứ k sẽ được tính toán để tối ưu qua các giá trị $c[u, v]$; $c[u, k]$ và $c[k, v]$ tại bước thứ $k - 1$. Tính chính xác của cách cài đặt dưới dạng ba vòng lặp for lồng nhau trên có thể thấy được do sự tối ưu bắc cầu chỉ làm tăng tốc độ tối ưu các $c[u, v]$ trong mỗi bước

```

P_4_08_5.PAS * Thuật toán Floyd
{$MODE DELPHI} (*This program uses 32-bit Integer [-231..231 - 1]*)
program Finding_the_Shorest_Path_using_Floyd_Algorithm;
const
  InputFile = 'MINPATH.INP';
  OutputFile = 'MINPATH.OUT';
  max = 1000;
  maxEC = 1000;
  maxC = max * maxEC;
var
  c: array[1..max, 1..max] of Integer;
  Trace: array[1..max, 1..max] of Integer; {Trace[u, v] = Điểm liền sau u trên đường đi từ u tới v}
  n, s, f: Integer;

procedure LoadGraph; {Nhập dữ liệu, đồ thị không được có chu trình âm}
var
  i, m, u, v: Integer;
  fi: Text;
begin
  Assign(fi, InputFile); Reset(fi);
  ReadLn(fi, n, m, s, f);
  for u := 1 to n do
    for v := 1 to n do
      if u = v then c[u, v] := 0 else c[u, v] := maxC;
  for i := 1 to m do ReadLn(fi, u, v, c[u, v]);
  Close(fi);
end;

procedure Floyd;
var
  k, u, v: Integer;
begin
  for u := 1 to n do
    for v := 1 to n do
      if c[u, v] > c[u, k] + c[k, v] then {Đường đi từ qua k tốt hơn}
        begin
          c[u, v] := c[u, k] + c[k, v]; {Ghi nhận đường đi đó thay cho đường cũ}
          Trace[u, v] := Trace[u, k]; {Lưu vết đường đi}
        end;
end;

procedure PrintResult; {In đường đi từ s tới f}
var
  fo: Text;
begin
  Assign(fo, OutputFile); Rewrite(fo);
  if c[s, f] = maxC
    then WriteLn(fo, 'There is no path from ', s, ' to ', f)
  else
    begin
      WriteLn(fo, 'Distance from ', s, ' to ', f, ': ', c[s, f]);
    end;
end;

```

```

repeat
    Write(fo, s, '->');
    s := Trace[s, f]; {Nhắc lại rằng Trace[s, f] là đỉnh liền sau s trên đường đi tới f}
    until s = f;
    WriteLn(fo, f);
end;
Close(fo);
end;

begin
    LoadGraph;
    Floyd;
    PrintResult;
end.

```

Khác biệt rõ ràng của thuật toán Floyd là khi cần tìm đường đi ngắn nhất giữa một cặp đỉnh khác, chương trình chỉ việc in kết quả chứ không phải thực hiện lại thuật toán Floyd nữa.

8.8. NHẬN XÉT

Bài toán đường đi dài nhất trên đồ thị trong một số trường hợp có thể giải quyết bằng cách đổi dấu trọng số tất cả các cung rồi tìm đường đi ngắn nhất, nhưng hãy cẩn thận, có thể xảy ra trường hợp có chu trình âm.

Trong tất cả các cài đặt trên, vì sử dụng ma trận trọng số chứ không sử dụng danh sách cạnh hay danh sách kề có trọng số, nên ta đều đưa về đồ thị đầy đủ và đem trọng số $+\infty$ gán cho những cạnh không có trong đồ thị ban đầu. Trên máy tính thì không có khái niệm trừu tượng $+\infty$ nên ta sẽ phải chọn một số dương đủ lớn để thay. Như thế nào là đủ lớn? số đó phải đủ lớn hơn tất cả trọng số của các đường đi cơ bản để cho dù đường đi thật có tồi tệ đến đâu vẫn tốt hơn đường đi trực tiếp theo cạnh tương tự ra đó.

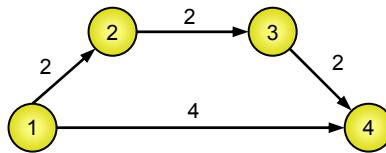
Xét về độ phức tạp tính toán, nếu cài đặt như trên, thuật toán Ford-Bellman có độ phức tạp là $O(n^3)$, thuật toán Dijkstra là $O(n^2)$, thuật toán tối ưu nhẫn theo thứ tự tópô là $O(n^2)$ còn thuật toán Floyd là $O(n^3)$. Tuy nhiên nếu sử dụng danh sách kề, thuật toán tối ưu nhẫn theo thứ tự tópô sẽ có độ phức tạp tính toán là $O(m)$. Thuật toán Dijkstra kết hợp với cấu trúc dữ liệu Heap có độ phức tạp $O(\max(n, m).log n)$.

Khác với một bài toán đại số hay hình học có nhiều cách giải thì chỉ cần nắm vững một cách cũng có thể coi là đạt yêu cầu, 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 yêu cầu trước tiên là 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 trường hợp cụ thể. Những bài tập sau đây cho ta thấy rõ điều đó.

Bài tập

Bài 1

Giải thích tại sao đối với đồ thị sau, cần tìm đường đi dài nhất từ đỉnh 1 tới đỉnh 4 lại không thể dùng thuật toán Dijkstra được:

**Bài 2**

Trên mặt phẳng cho n đường tròn ($n \leq 2000$), đường tròn thứ i được cho bởi bộ ba số thực $(x[i], y[i], R[i])$, $(x[i], y[i])$ là tọa 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, f cho trước với chi phí ít nhất.

Bài 3

Cho một dãy n số nguyên $a[1..n]$ ($n \leq 10000$; $1 \leq a[i] \leq 10000$). 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.

Hướng dẫn: Dựng đồ thị $G = (V, E)$, $V = \{1, \dots, n\}$, $(i, j) \in E$ nếu $i < j$ và $a[i] + a[j]$ là số nguyên tố, tìm đường đi dài nhất trên đồ thị G không có chu trình

Bài 4

Một công trình lớn được chia làm n công đoạn đánh số 1, 2, ..., n . Công đoạn i phải thực hiện mất thời gian $t[i]$. Quan hệ giữa các công đoạn được cho bởi bảng $a[i, j]$: $a[i, j] = \text{TRUE} \Leftrightarrow$ công đoạn j chỉ được bắt đầu khi mà công việc i đã xong. 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.

Hướng dẫn: Dựng đồ thị $G = (V, E)$, trong đó:

$$V = \{0, 1, \dots, n\}$$

$$E = \{(0, u) | 1 \leq u \leq n\} \cup \{(u, v) | a[u, v] = \text{TRUE}\}. \text{ Trọng số mỗi cung } (i, j) \text{ đặt bằng } t[j]$$

Tìm đường đi dài nhất trên đồ thị không có chu trình, thời điểm thích hợp để thực hiện công việc i chính là $d[i] - t[i]$.

Bài 5

Cho một bảng các số tự nhiên kích thước $m \times n$ ($1 \leq m, n \leq 100$). 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 cực tiểu.

Hướng dẫn: Dùng thuật toán Dijkstra với cấu trúc Heap

Bài 6: Arbitrage

Arbitrage là một cách sử dụng sự bất hợp lý trong hối đoái tiền tệ để kiếm lời.

Ví dụ:

Giả sử

1\$ mua được 0.7£

1£ mua được 190¥

1¥ mua được 0.009\$

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 1, 2, ..., n. Và một bảng R kích thước nxn cho biết tỉ lệ hối đoái. Một đơn vị tiền i đổi được $R[i, j]$ đơn vị tiền j.

a) 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?

b) Giả sử rằng nhà băng đủ thông minh để vô hiệu hóa Arbitrage, tính tỉ lệ tốt nhất có thể hoán đổi giữa hai loại tiền tệ bất kỳ.

Hướng dẫn:

Lấy logarithm của các tỉ giá hối đoái, đặt $R[i, j] := -\ln(R[i, j])$. Dụng đồ thị $G = (V, E)$ có n đỉnh và ma trận trọng số là R. Thêm vào G một đỉnh đánh số 0 và nối nó tới tất cả những đỉnh còn lại bằng cung có trọng số 0.

a) Dùng thuật toán Ford-Bellman tìm đường đi ngắn nhất xuất phát từ 0. Sau khi thuật toán Ford-Bellman kết thúc, xét tất cả các cặp đỉnh. Nếu tồn tại một cặp (u, v) mà $d[v] > d[u] + R[u, v]$ thì thông báo tồn tại Arbitrage. Thuật toán này dựa trên nhận xét: Tỉ giá hối đoái đã cho có thể dùng Arbitrage nếu và chỉ nếu đồ thị có chu trình âm.

b) Nếu tỉ giá hối đoái không cho phép Arbitrage tức là đồ thị G không có chu trình âm. Dùng thuật toán Floyd tìm đường đi ngắn nhất giữa mọi cặp đỉnh. Tỉ lệ tốt nhất có thể hoán đổi từ loại tiền i sang loại tiền j chính là $e^{-d[i, j]}$, trong đó $d[i, j]$ là độ dài đường đi ngắn nhất giữa hai đỉnh i và j trên G.

Bài 7:

Cho đồ thị vô hướng $G = (V, E)$ gồm các cạnh được gán trọng số không âm. Cho hai đỉnh A và B. Hãy chỉ ra hai đường đi từ A tới B thỏa mãn:

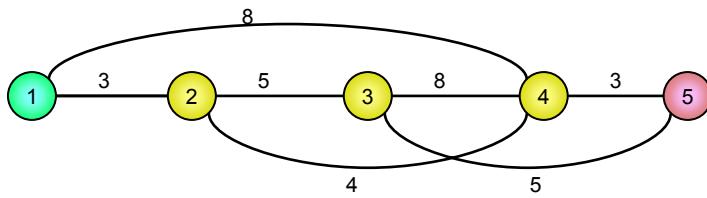
- ❖ Hai đường đi này không có cạnh chung
- ❖ Tổng độ dài hai đường đi là nhỏ nhất có thể

Hướng dẫn:

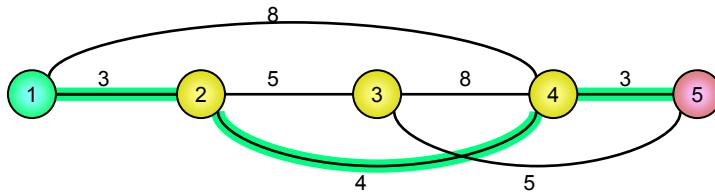
Coi mỗi cạnh của đồ thị tương đương với hai cung có hướng ngược chiều nhau. Dùng thuật toán Dijkstra tìm đường đi ngắn nhất từ A tới B: $\langle A=v_0, v_1, \dots, v_p=B \rangle$. Đọc trên đường đi Dijkstra, bỏ đi tất cả các cung (v_{i-1}, v_i) , và đổi dấu trọng số của cung (v_i, v_{i-1}) này. Phép biến đổi này có thể tạo ra những cung trọng số âm nhưng không tạo thành chu trình âm. Tiếp theo, dùng thuật toán Ford-Bellman tìm đường đi ngắn nhất từ A tới B: $\langle A=u_0, u_1, \dots, u_q=B \rangle$. Đọc trên đường đi Ford-Bellman, bỏ đi tất cả các cung (u_{i-1}, u_i) . Với mỗi cung còn lại trên đồ thị, nếu cả cung ngược hướng với nó cũng được duy trì đến bước này thì loại bỏ luôn cả hai. Cuối cùng, những cung còn lại chỉ ra hai đường đi từ B về A, bằng cách lật ngược chiều đường đi, ta có lời giải.

Ví dụ:

Đồ thị dưới đây, A = 1, B = 5:

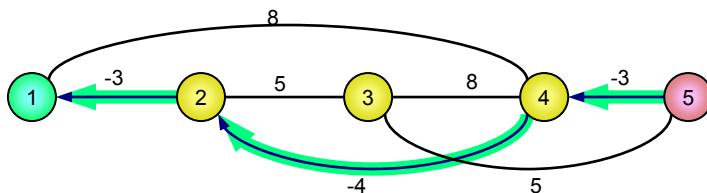


Đường đi Dijkstra $\langle 1, 2, 4, 5 \rangle$ (Độ dài 10):

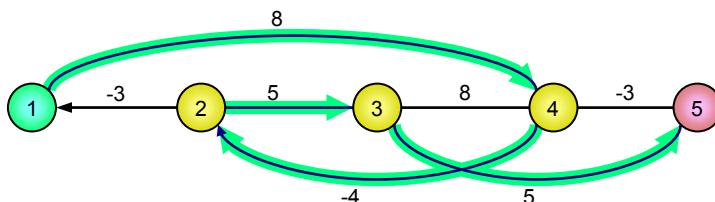


Bỏ đi các cung $(1, 2), (2, 4), (4, 5)$. Đặt lại trọng số các cung ngược chiều đường đi:

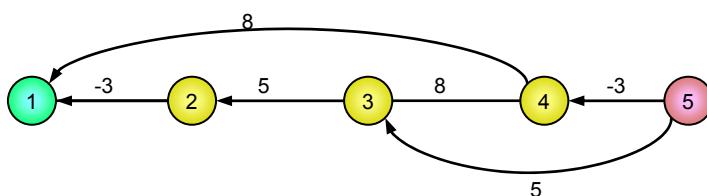
$$c[2, 1] := -3; c[4, 2] := -4; c[5, 4] := -3;$$



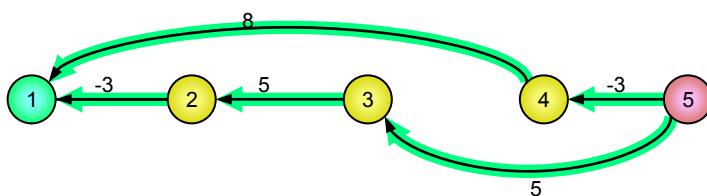
Đường đi Ford-Bellman $\langle 1, 4, 2, 3, 5 \rangle$ (Độ dài 14):



Bỏ đi các cung $(1, 4), (4, 2), (2, 3), (3, 5)$:



Bỏ nốt cung $(3, 4)$ và $(4, 3)$ ta có 2 đường đi từ 5 về 1: $\langle 5, 4, 1 \rangle$ và $\langle 5, 3, 2, 1 \rangle$.



Đổi chiều lên đồ thị ban đầu, tổng độ dài 2 đường đi tìm được = Độ dài đường đi Dijkstra + Độ dài đường đi Ford-Bellman = 24. Lật ngược thứ tự các đỉnh trong hai đường đi trên, ta được cặp đường đi từ 1 tới 5 có tổng độ dài nhỏ nhất cần tìm.

§9. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

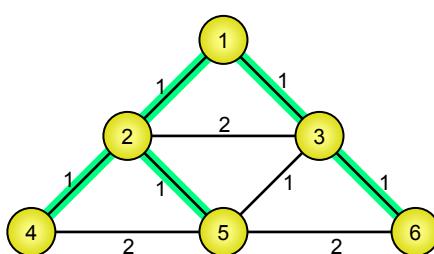
9.1. BÀI TOÁN CÂY KHUNG NHỎ NHẤT

Cho $G = (V, E)$ là đồ thị vô hướng liên thông có trọng số, với một cây khung T của G , ta gọi trọng số của cây 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 của đồ thị (minimum spanning tree) và bài toán đó gọi là bài toán cây khung nhỏ nhất. 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. Bạn có thể đưa vào một số sửa đổi nhỏ trong thủ tục nhập liệu để giải quyết bài toán trong trường hợp đa đồ thị.

Input: file văn bản MINTREE.INP:

- ❖ Dòng 1: Ghi hai số số đỉnh n (≤ 1000) và số cạnh m của đồ thị cách nhau ít nhất 1 dấu cách
- ❖ m dòng tiếp theo, mỗi dòng có dạng 3 số $u, v, c[u, v]$ cách nhau ít nhất 1 dấu cách thể hiện đồ thị có cạnh (u, v) và trọng số cạnh đó là $c[u, v]$. ($c[u, v]$ là số nguyên có giá trị tuyệt đối không quá 1000).

Output: file văn bản MINTREE.OUT ghi các cạnh thuộc cây khung và trọng số cây khung



MINTREE.INP	MINTREE.OUT
6 9	Minimum spanning tree:
1 2 1	(2, 4) = 1
1 3 1	(3, 6) = 1
2 4 1	(2, 5) = 1
2 3 2	(1, 3) = 1
2 5 1	(1, 2) = 1
3 5 1	Weight = 5
3 6 1	
4 5 2	
5 6 2	

9.2. THUẬT TOÁN KRUSKAL (JOSEPH KRUSKAL - 1956)

Thuật toán Kruskal dựa trên mô hình xây dựng cây khung bằng thuật toán hợp nhất (§5), 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: Với đồ thị vô hướng $G = (V, E)$ có n đỉnh. Khởi tạo cây T ban đầu không có cạnh nào. Xét tất cả các cạnh của đồ thị từ cạnh có trọng số nhỏ đến cạnh có trọng số lớn, nếu 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 $n - 1$ cạnh vào trong T thì ta được T là cây khung nhỏ nhất
- ❖ Hoặc chưa kết nạp đủ $n - 1$ cạnh nhưng hễ cứ kết nạp thêm một cạnh bất kỳ trong số các cạnh còn lại thì sẽ tạo thành chu trình đơn. Trong trường hợp này đồ thị G là không liên thông, việc tìm kiếm cây khung thất bại.

Như vậy có hai vấn đề quan trọng khi cài đặt thuật toán Kruskal: