WIKIPEDIA

Thuật toán Bellman-Ford

Bách khoa toàn thư mở Wikipedia

Thuật toán Bellman-Ford là một thuật toán tính các đường đi ngắn nhất nguồn đơn trong một đồ thị có hướng có trọng số (trong đó một số cung có thể có trọng số âm). Thuật toán Dijkstra giải cùng bài toán này tuy nhiên Dijkstra có thời gian chạy nhanh hơn đơn giản là đòi hỏi trọng số của các cung phải có giá trị không âm.

Thuật toán Bellman Ford chạy trong thời gian $O(V{\cdot}E)$, trong đó V là số đỉnh và E là số cung của đồ thị.

Mục lục

Tư tưởng thuật toán

[1]

- Bước 1: Khởi tạo $\prod(0,x)=0$; $\prod(0,i)=+\infty$, $\forall i\neq x$ và k=1

- Bước 2: Với mỗi $i\in X$ ta đặt:

```
∏(k,i)=min({∏(k-1,i)}∪{∏(k-1,j)+L[j][i]})
```

- Bước 3: Nếu $\prod(k,i)=\prod(k-1,i)$ với $i\in X$ thì $\prod(k,i)$ chính là độ dài đường đi ngắn nhất từ x đến i. Ngược lại nếu k<n thì tăng k=k+1 và trở lại bước 2; nếu k=n thì dừng vì từ x đi tới được 1 mạch âm.

Ưu điểm:[2]

- Từ 1 đỉnh xuất phát nhìn hình ta có thể suy ra đường đi ngắn nhất từ đỉnh đó tới các đỉnh khác mà không cần làm lại từ đầu.

- Ví dụ: Từ đỉnh 1 ta có thể tìm đường đi ngắn nhất từ 1->3 và 1->4 mà không cần làm lại.

Nội dung thuật toán

```
function BellmanFord(danh_sách_đỉnh, danh_sách_cung, nguồn)
  // hàm yêu cầu đồ thị đưa vào dưới dạng một danh sách đỉnh, một danh sách cung
  // hàm tính các giá trị khoảng_cách và đỉnh_liền_trước của các đỉnh,
  // sao cho các giá trị đỉnh_liền_trước sẽ lưu lại các đường đi ngắn nhất.

  // bước 1: khởi tạo đồ thị
  for each v in danh_sách_đỉnh:
      if v is nguồn then khoảng_cách(v):= 0
      else khoảng_cách(v):= vô cùng
      đỉnh_liền_trước(v):= null

  // bước 2: kết nạp cạnh
  for i from 1 to size(danh_sách_đỉnh)-1:
      for each (u,v) in danh_sách_cung:
          if khoảng_cách(v) > khoảng_cách(u) + trọng_số(u,v):
              khoảng_cách(v):= khoảng_cách(u) + trọng_số(u,v)
              đỉnh_liền_trước(v):= u

  // bước 3: kiểm tra chu trình âm
  for each (u,v) in danh_sách_cung:
      if khoảng_cách(v) > khoảng_cách(u) + trọng_số(u,v):
          error "Đồ thị chứa chu trình âm"
```

Chứng minh tính đúng đắn

Tính đúng đắn của thuật toán có thể được chứng minh bằng quy nạp. Thuật toán có thể được phát biểu chính xác theo kiểu quy nạp như sau:

Bổ đề. Sau *i* lần lặp vòng *for*:

1. Nếu Khoảng_cách(u) không có giá trị vô cùng lớn, thì nó bằng độ dài của một đường đi nào đó từ *s* tới *u*;
2. Nếu có một đường đi từ *s* tới *u* qua nhiều nhất *i* cung, thì Khoảng_cách(u) có giá trị không vượt quá độ dài của đường đi ngắn nhất từ *s* tới *u* qua tối đa *i* cung.

Chứng minh.

Trường hợp cơ bản: Xét `i=0` và thời điểm trước khi vòng *for* được chạy lần đầu tiên. Khi đó, với đỉnh nguồn `khoảng_cách(nguồn) = 0`, điều này đúng. Đối với các đỉnh *u* khác, `khoảng_cách(u) = vô cùng`, điều này cũng đúng vì không có đường đi nào từ *nguồn* đến *u* qua 0 cung.

Trường hợp quy nạp:

Chứng minh câu 1. Xét thời điểm khi khoảng cách tới một đỉnh được cập nhật bởi công thức `khoảng_cách(v):= khoảng_cách(u) + trọng_số(u,v)`. Theo giả thiết quy nạp, `khoảng_cách(u)` là độ dài của một đường đi nào đó từ *nguồn* tới *u*. Do đó, `khoảng_cách(u) + trọng_số(u,v)` là độ dài của đường đi từ *nguồn* tới *u* rồi tới *v*.

Chứng minh câu 2: Xét đường đi ngắn nhất từ *nguồn* tới *u* qua tối đa *i* cung. Giả sử *v* là đỉnh liền ngay trước *u* trên đường đi này. Khi đó, phần đường đi từ *nguồn* tới *v* là đường đi ngắn nhất từ *nguồn* tới *v* qua tối đa *i-1* cung. Theo giả thuyết quy nạp, `khoảng_cách(v)` sau *i-1* vòng lặp không vượt quá độ dài đường đi này. Do đó, `trọng_số(v,u) + khoảng_cách(v)` có giá trị không vượt quá độ dài của đường đi từ *s* tới *u*. Trong lần lặp thứ *i*, `khoảng_cách(u)` được lấy giá trị nhỏ nhất của `khoảng_cách(v) + trọng_số(v,u)` với mọi *v* có thể. Do đó, sau *i* lần lặp, `khoảng_cách(u)` có giá trị không vượt quá độ dài đường đi ngắn nhất từ *nguồn* tới *u* qua tối đa *i* cung.

Khi *i* bằng số đỉnh của đồ thị, mỗi đường đi tìm được sẽ là đường đi ngắn nhất toàn cục, trừ khi đồ thị có chu trình âm. Nếu tồn tại chu trình âm mà từ đỉnh nguồn có thể đi đến được thì sẽ không tồn tại đường đi nhỏ nhất (vì mỗi lần đi quanh chu trình âm là một lần giảm trọng số của đường).

Ứng dụng trong định tuyến

Một biến thể phân tán của thuật toán Bellman-Ford được dùng trong các giao thức định tuyến vector khoảng cách, chẳng hạn giao thức RIP (*Routing Information Protocol*). Đây là biến thể phân tán vì nó liên quan đến các nút mạng (các thiết bị định tuyến) trong một hệ thống tự chủ (*autonomous system*), ví dụ một tập các mạng IP thuộc sở hữu của một nhà cung cấp dịch vụ Internet (ISP).

Thuật toán gồm các bước sau:

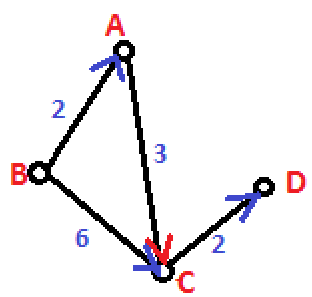
1. Mỗi nút tính khoảng cách giữa nó và tất cả các nút khác trong hệ thống tự chủ và lưu trữ thông tin này trong một bảng.
2. Mỗi nút gửi bảng thông tin của mình cho tất cả các nút lân cận.
3. Khi một nút nhận được các bảng thông tin từ các nút lân cận, nó tính các tuyến đường ngắn nhất tới tất cả các nút khác và cập nhật bảng thông tin của chính mình.

Nhược điểm chính của thuật toán Bellman-Ford trong cấu hình này là

- Không nhân rộng tốt
- Các thay đổi của tô-pô mạng không được ghi nhận nhanh do các cập nhật được lan truyền theo từng nút một.
- Đếm dần đến vô cùng (nếu liên kết hỏng hoặc nút mạng hỏng làm cho một nút bị tách khỏi một tập các nút khác, các nút này vẫn sẽ tiếp tục ước tính khoảng cách tới nút đó và tăng dần giá trị tính được, trong khi đó còn có thể xảy ra việc định tuyến thành vòng tròn)

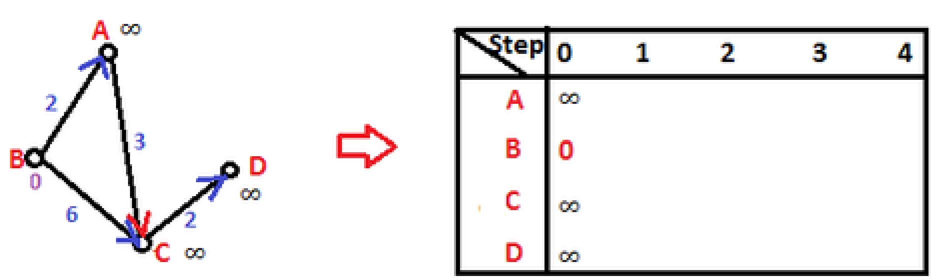
Minh họa bằng hình

Tìm đường đi ngắn nhất từ đỉnh B tới đỉnh D của đồ thị G [3]



Đồ thị G

- **Bước 0:** Ta đánh dấu đỉnh xuất phát = 0, các đỉnh còn lại bằng vô cực.

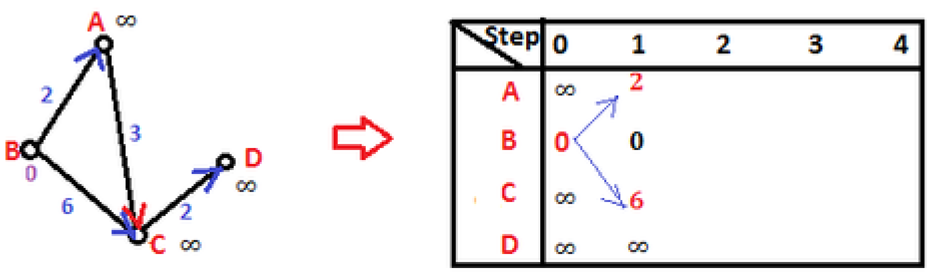


Bước 0

- **Bước 1:**

Tại đỉnh A có đỉnh B đi vào có chi phí hiện tại (2) < chi phí trước (∞) => cập nhật lại chi phí đỉnh A

Tại đỉnh C có đỉnh B đi vào có chi phí hiện tại (6) < chi phí trước (∞) => cập nhật lại chi phí đỉnh C



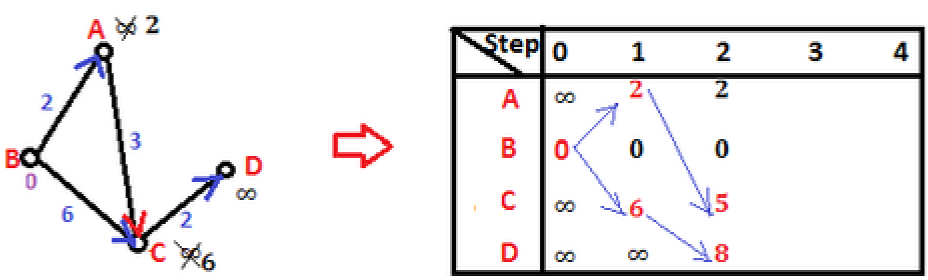
Step	0	1	2	3	4
A	∞	2			
B	0	0			
C	∞	6			
D	∞	∞			

Bước 1

- Bước 2:

Tại đỉnh C có đỉnh A đi vào có chi phí hiện tại (5) < chi phí trước (6) => cập nhật lại chi phí đỉnh C

Tại đỉnh D có đỉnh C đi vào có chi phí hiện tại (8) < chi phí trước (∞) => cập nhật lại chi phí đỉnh D

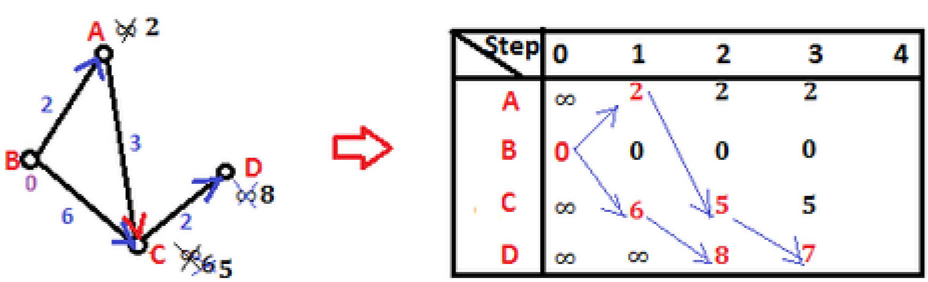


Step	0	1	2	3	4
A	∞	2	2		
B	0	0	0		
C	∞	6	5		
D	∞	∞	8		

Bước 2

- Bước 3:

Tại đỉnh D có đỉnh C đi vào có chi phí hiện tại (7) < chi phí trước (8) => cập nhật lại chi phí đỉnh D

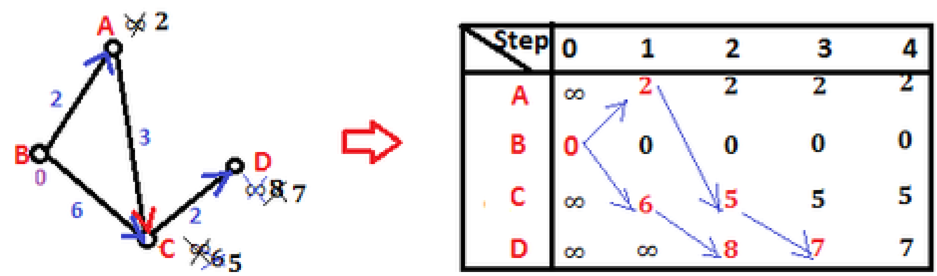


Step	0	1	2	3	4
A	∞	2	2	2	
B	0	0	0	0	
C	∞	6	5	5	
D	∞	∞	8	7	

Bước 3

- Bước 4:

Bước 4 giống bước 3 nên thuật toán dừng.



Step	0	1	2	3	4
A	∞	2	2	2	2
B	0	0	0	0	0
C	∞	6	5	5	5
D	∞	∞	8	7	7

Bước 4

- Kết luận:

Có đường đi ngắn nhất từ B->D: B->A->C->D

- Lưu ý:

- Nếu Bước 4 không giống bước 3 => kết luận không có đường đi ngắn nhất từ B->D

Cài đặt Bellman

Hàm khởi tạo (bước 0)

Không như khi cài đặt các thuật toán dijkstra, Floyd, do Bellman chấp nhận cạnh âm, việc sử dụng trị -1 không còn đúng nữa.Tạm thời, ta có thể sử dụng trị MAXINT (32767) cho giá trị inf, vì nếu như chi phí đạt đến ngưỡng này,có thể xem như tràn số.

step = 0; for {i...} {

```
mincost[step][i] = inf;
```

```
previous[step][i] = i;
```

} mincost[step][x] = 0;

Cài đặt hàm Bellman

Chú ý rằng để có thể kết luận được đồ thị có chu trình âm hay không. ta cần chạy đến bước thứ n (nghĩa là đi qua tối đa n+1 đỉnh).Do đó, cấu trúc dữ liệu để lưu cũng cần lưu ý khi khai báo.

bSuccess = false;

for (step = 1; step <=n; step ++) // dùng <=n thay vì <n

{

```
for(i...)
{
   mincost[step][i] = mincost[step-1][i]
   previous[step][i] = previous[step-1][i]
   // tìm các đỉnh j có đường nối từ j -->i
   // và chi phí bước step-1 của j khác vô cực
   for (j...)
   if (...&&...)
```

```
    {
       // cập nhật lại nếu chi phí bước step của i là vô cực
       // hoặc chi phí đi qua j: mincost[step-1][j]+a[j][i]
       //tối ưu hơn
       if (...||...)
       {
          // cập nhật lại chi phí và lưu đỉnh cha
       }
    }
 }
 // so sánh mincost[step] với mincost[step-1], nếu bằng nhau
 // kết thúc thành công
 int bSame = true;
 for (i...)
 if (mincost[step][i]!= mincost[step-1][i])
 {
    bSame = false;
    break;
  // đã giống nhau,đường đi đã tối ưu
  if (bSame)
     break;
```

}

Cấu trúc dữ liệu

int mincost [MAX+1][MAX];

int previous[MAX+1][MAX];

Hàm in kết quả

Nếu nStep = n+1, ta kết luận đồ thị có chu trình âm.

Ngược lại, ta sẽ dò chi phí ngược từ bước nStep-1 đến bước 0.(Do bước nStep có giá trị giống bước nStep-1)

k = y;

for (i=nStep-1;i>0;i--) // chừa lại bước cuối

{

```
printf("%d <---", k);
k = previous[i][k];    // đỉnh trước k
```

}

printf("%dn",k); // có thể thêm kiểm tra k == x

Chú thích

1. ^ C.Berge (1973). *Graphs and Hypergraph*. New York: Elsevier.
2. ^ B.Bollobás (1979). *Graph Theory: An Introductory Course*. New York: Springer-Verlag.
3. ^ W.T. Tutte (1966). *Connectivity in Graphs*. University of Toronto.

Tham khảo

- Tài liệu lý thuyết bộ môn Lý Thuyết Đồ Thị trường Đại Học Khoa Học tự Nhiên

Lê Đình Huy. “Một tập tài liệu nhỏ về Lý thuyết đồ thị (Graph Theory Ebooks)” (http://book.mathvn.com/2010/04/graph-theory-ebooks-vietnamese.html) (PDF) (bằng tiếng Việt Nam).

giảng viên Nguyễn Ngọc Trung. “Tuyển tập 95 bài tập về Lý thuyết đồ thị(95 exercises Graph Theory - Nguyen Ngoc Trung)” (http://book.mathvn.com/2010/04/95-exercises-graph-theory-nguyen-ngoc.html) (PDF) (bằng tiếng Việt Nam).

- Richard Bellman: *On a Routing Problem*, in Quarterly of Applied Mathematics, 16(1), pp. 87–90, 1958.
- Lestor R. Ford jr., D. R. Fulkerson: *Flows in Networks*, Princeton University Press, 1962.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 24.1: The Bellman-Ford algorithm, pp. 588–592.