TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG
_____ * _____

BÀI TẬP MÔN HỌC
TOÁN RỜI RẠC

Đề tài 10 : Đường đi ngắn nhất trên đồ thị

Nguyễn Duy Thành-CNTT1 20102737
Vũ Văn Hiệp-CNTT2 20101545
Bạch Văn Hải-CNTT1 20101464

Giáo viên :

Huỳnh Thanh Bình

HÀ NỘI
Ngày 7 tháng 5 năm 2012

Mục lục

Lời nói đầu

Trong lý thuyết đồ thị, bài toán tìm đường đi ngắn nhất giữa hai đỉnh của một đồ thị liên thông là một trong những bài toán có ứng dụng thực tế to lớn nhất. Từ bài toán này, người ta đã phát triển thành nhiều bài toán khác như: Bài toán chọn hành trình tiết kiệm nhất về thời gian, chi phí hay khoảng cách trên một mạng lưới giao thông; Bài toán lập lịch thi công các công đoạn của một công trình lớn, Bài toán lực chọn con đường truyền tin với chi phí thấp nhất trong mạng lưới thông tin, Bài toán tìm lời dịch tốt nhất cho một câu nói . . . Đến nay, chúng ta đã có rất nhiều thuật toán để giải Bài toán tìm đường đi ngắn nhất với khả năng áp dụng và độ phức tạp khác nhau. Trong bài báo cáo này, chúng em xin trình bày về 6 phương pháp giải quyết bài toán này kèm theo project minh họa việc giải bài toán bằng giao diện đồ họa.

Dù đã rất cố gắng nhưng do khả năng còn hạn chế nên không thể không có sai sót, rất mong được cô tận tình chỉ bảo. Xin chân thành cảm ơn!

Nhóm thực hiện:

Nguyễn Duy Thành
Vũ Văn Hiệp
Bạch Văn Hải

Chương 1

Bài toán tìm đường đi ngắn nhất trên đồ thị và các thuật toán thông dụng

1.1 Phát biểu bài toán

1.1.1 Các khái niệm cơ bản

Xét đồ thị có hướng $G = (V, E)$ với $|V| = n$, $|E| = m$. Các cung (u, v) của đồ thị được gắn trọng số được xác định bởi hàm $w(u, v)$ có giá trị thực còn được gọi là độ dài của cạnh (u, v). Quy ước $w(u, v) = \infty$ nếu $(u, v)E$.

Nếu $P = v \to v_1 \to v_2 \to \ldots \to k$ là một đường đi trên G, thì độ dài của nó là tổng các trọng số trên các cung của nó:

$$w(P) = \sum_{i=0}^{k-1} w(v_i, v_{i+1})$$

Đường đi ngắn nhất từ đỉnh u đến đỉnh v là đường đi có độ dài ngắn nhất trong số các đường đi từ u đến v, còn được gọi là khoảng cách từ u đến v ký hiệu là $\partial(u, v)$.

1.1.2 Phát biểu bài toán

Bài toán tìm đường đi ngắn nhất trên đồ thị được phát biểu dưới dạng tổng quát:
Ïtrên đồ thị liên thông $G = (V, E)$, tìm đường đi có độ dài nhỏ nhất từ một đỉnh xuất phát $s\epsilon V$ đến một đỉnh đích $t\epsilon V$. Đường đi như vậy gọi là đường đi ngắn nhất từ s đến t, độ dài của nó ký hiệu là $d(s, t)$.

1.1.3 Nhận xét

- Nếu không tồn tại đường đi từ s đến t thì coi $d(s, t) = \infty$
- Nếu mọi chu trình trên đồ thị đều có độ dài dương, thì đường đi ngắn nhất không có đỉnh nào lặp lại (gọi là đường đi cơ bản).
- Nếu trong đồ thị có chu trình âm thì không thể xác định khoảng cách giữa một số cặp đỉnh, nên chúng ta không xét ở đây.
- Đường đi ngắn nhất luôn có thể tìm trong số các đường đi đơn.

- Mọi đường đi ngắn nhất trong đồ thị đều đi qua không quá $n-1$ cạnh, với n là số đỉnh.
- Ta có thể dùng tính chất sau để chứng minh tính đúng đắn của một thuật toán tìm đường đi ngắn nhất: Mọi đường đi con của đường đi ngắn nhất cũng là đường đi ngắn nhất.
- Giả sử P là đường đi ngắn nhất từ s đến v: $P = s \to \ldots \to u \to v$, khi đó ta có

$$\partial(s, v) = \partial(s, u) + w(u, v).$$

1.2 Các thuật toán giải bài toán tìm đường đi ngắn nhất trên đồ thị

1.2.1 Thuật toán tìm kiếm theo chiều rộng ứng dụng tìm đường đi ngắn nhất trên đồ thị theo số cạnh

Mô tả thuật toán

Xét đồ thị có hướng hoặc vô hướng không trọng số $G = (V, E)$ có n đỉnh và m cạnh, khi đó ta chỉ quan tâm xem giữa hai đỉnh bất kỳ của đồ thị có cạnh hay không. Ta quy ước $w(u, v) = 1$ nếu có cạnh nối u và v, ngược lại $w(u, v) = 0$. Bài toán trong trường hợp này trở thành tìm đường đi ngắn nhất trên đồ thị theo số cạnh. Khi đó, ta có thể dùng thuật toán tìm kiếm (duyệt) đồ thị theo chiều rộng (BFS) để giải bài toán vì $BFS(s)$ cho phép ta thăm tất cả các đỉnh thuộc cùng một thành phần liên thông với s. Phía dưới là thủ tục được viết dưới dạng mã giả mô tả thuật toán:

```
def BFS(s, V):
    Queue = deque([])

    # init
    for v in V:
    In_queue[v] = False;

    Queue.append(v)
    In_queue[v] = True;

    # bfs
    while Queue:
        v = Queue.popleft()
        for u in Adjacency[v]:
            if Not In_queue[u]:
                Queue.append(u)
                d[u] = d[u] + 1
                p[u] = v
                In_queue[u] = True;
```

Nhận xét

- Sau khi thực hiện thủ tục này, nếu $In\_queue[t] = True$ thì tồn tại đường đi từ s đến t, ngược lại không có đường đi.

- Biến $p[u]$ dùng để ghi nhận đỉnh đi trước đỉnh u trong đường đi từ s đến t.
- Nếu muốn chỉ tìm đường đi ngắn nhất từ s đến t thì có thể dừng thuật toán ngay khi có $u = t$ (ở dòng 16).
- Độ phức tạp tính toán của thuật toán là $O(n + m)$.

1.2.2 Thuật toán Ford-Bellman

Mô tả thuật toán

Thuật toán Ford-Bellman được sử dụng trong việc tìm đường đi ngắn nhất từ một đỉnh s đến tất cả các đỉnh còn lại. Nó có một ưu điểm rất hay đó là có thể làm việc với đồ thị có trọng số các cung tùy ý.
Thuật toán được xây dựng dựa trên tư tưởng sau: Xét đồ thị có hướng $G = (V, E)$ có n đỉnh với s là đỉnh xuất phát và $a[u, v]$, u, $v\epsilon V$ là ma trận trọng số. Giả thiết đồ thị không có chu trình âm, ta gọi $d[v]$ là khoảng cách từ s đến v. Ta khởi tạo $d[s] = 0$ và $d[v] = \infty$ với vs. Xét mọi cạnh (u, v) của đồ thị, nếu có cạnh nào thỏa mãn bất đẳng thức $d[u] + a[u, v] < d[v]$ thì ta gán $d[v] = d[u] + a[u, v]$ để tối ưu hóa $d[v]$. Tức là nếu độ dài đường đi từ s tới v lại lớn hơn tổng độ dài đường đi từ s đến u cộng với chi phí đi từ u đến v thì ta hủy bỏ đường đi từ s đến v đang có và thay vào đó là con đường từ s đến u và từ u đến v. Thuật toán sẽ kết thúc khi không thể tối ưu thêm bất kỳ $d[v]$ nào nữa. Dưới đây là mã giả mô tả thuật toán:

```
1  def Ford_Bellman(s):
2      for v in V:
3          d[v] = a[s, v];
4      p[v] = s;
5      d[s] = 0;
6      for k in range(1, n):
7          for v in V \ {s}:
8              for u in V:
9                  if d[v] > d[u] + a[u, v]
10                     d[v] = d[u] + a[u, v]
11                     p[v] = u
```

Chứng minh tính đúng đắn của thuật toán:

- Sau khi khởi tạo $d[s] = 0$ và $d[v! = s] = \partial$, dãy $d[v]$ chính là độ dài ngắn nhất của đường đi từ s đến v không quá 0 cạnh.
- Tại lần lặp thứ k, $d[v]$ bằng độ dài đường đi ngắn nhất từ s đến v không quá k cạnh. Dễ thấy tính chất sau: đường đi từ s đến v qua không quá $k + 1$ cạnh sẽ phải được thành lập bằng cách lấy đường đi từ s đến một đỉnh u nào đó qua không quá k cạnh, rồi đi qua cung (u, v) để tới v. Vì vậy, độ dài đường đi ngắn nhất từ s đến v qua không quá $k + 1$ cạnh sẽ là nhỏ nhất (Nguyên lý tối ưu Bellman).
- Sau lần lặp thứ $n - 1$, ta có $d[v]$ bằng 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 đến v là đường đi cơ bản, tức là d[v] là độ dài đường đi ngắn nhất từ s đến v.

Nhận xét:

- Độ phức tạp của thuật toán là $O(n^3)$.
- Ta có thể tăng hiệu quả thuật toán bằng cách chấm dứt vòng lặp theo k khi phát hiện trong quá trình thực hiện hai vòng lặp trong không có biến d[v] nào bị thay đổi giá trị.
- Đối với đồ thị thưa, ta nên sử dụng ma trận kề $Adjacency[v]$ trong vòng lặp trong cùng theo u vì sẽ giúp độ phức tạp thuật toán chỉ còn là $O(n.m)$:

```
for u in Adjacency[v]:
    if d[v] > d[u] + a[u, v]:
        d[v] = d[u] + a[u, v]
        p[v] = u
```

- Thuật toán More Bellman: đây là một cải tiến của Ford-Bellman. Ý tưởng của nó là thay vì tìm kiếm cặp đỉnh (u, v) thỏa mãn bất đẳng thức để tối ưu đường đi từ s tới v, ta sẽ lưu các đỉnh u vào hàng đợi Queue, rồi lần lượt lấy các đỉnh trong Queue ra để tính $d[v]$. Từ đỉnh u ta tính $d[v]$ cho các đỉnh v là lân cận của u, nếu nhận được giá trị tốt hơn hiện tại thì cập nhật lại $d[v]$ và kiểm tra xem v có nằm trong Queue chưa. Nếu v không nằm trong Queue thì nạp nó vào còn không thì bỏ qua. Việc cài đặt thuật toán này khá phức tạp nên không tiện trình bày tại đây.

1.2.3 Thuật toán Dijkstra

Mô tả thuật toán

Trong trường hợp trọng số trên các cung là không âm, thuật toán Dijkstra để tìm đường đi ngắn nhất từ đỉnh s đến các đỉnh còn lại của đồ thị làm việc hiệu quả hơn so với thuật toán Ford-Bellman.
Ý tưởng của thuật toán là gán cho các đỉnh các nhãn tạm thời cho biết cân trên của đọ dài đường đi ngắn nhất từ s đến đỉnh đó. Các nhãn được biến đổi theo một thủ tục lặp. Mỗi bước lặp sẽ có một nhãn tạm thời trở thành nhãn cố định. Nếu nhãn của một đỉnh nào đó trở thành nhãn cố định thì nó sẽ cho ta không phải là cận trên mà là độ dài đường đi ngắn nhất từ đỉnh s đến đỉnh nó.
Sau đây là mã giả mô tả thuật toán:

```
def Dijkstra(s):
    for v in V:
        d[v] = a[s,v]
        p[v] = s

    d[s] = 0
    S = {s}
    T = V \ {s}

    while T:
        u = z : d[z] = min {d[z] | z in T}
        T =  T \ {u}
        S.add(u)
```

```
14
15          for v in T:
16              if d[v] > d[u] + a[u, v]:
17                  d[v] = d[u] + a[u, v]
18                  p[v] = u
```

Chứng minh tính đúng đắn của thuật toán:

- Giả sử ở một bước lặp nào đó các nhãn cố định cho ta độ dài các đường đi ngắn nhất từ s đến các đỉnh có nhãn cố định, ta sẽ chứng minh rằng ở lần lặp tiếp theo nếu đỉnh u thu được nhãn cố định thì $d[u]$ chính là độ dài đường đi ngắn nhất từ s đến u.

- Nếu gọi S_1 là tập các đỉnh có nhãn cố định, S_2 là tập các nhãn tạm thời ở bước lặp đang xét. Kết thúc mỗi bước lập nhãn tạm thời $d[v]$ cho ta độ dài đường đi ngắn nhất từ s đến v chỉ qua những đỉnh nằm hoàn toàn trong tập S_1. Giả sử đường đi ngắn nhất từ s đến u không nằm trọn trong tập S_1, tức là nó đi qua ít nhất một dỉnh của tập S_2. Gọi $z \epsilon S_2$ là đỉnh đầu tiên như vậy trên đường đi này. Do trong số trên các cung là không âm nên đoạn đường từ z đến u có độ dài $L > 0$ và $d[z] < d[u]\breve{}L < d[u]$. Bất đẳng thức này mâu thuẫn với cách xác định đỉnh u* là đỉnh có nhãn tạm thời nhỏ nhất. Vậy đường đi ngắn nhất từ s đến u phải nằm trọn trong S_1 và $d[u^*]$ là độ dài đường đi.

- Tại lần lặp đầu tiên $S_1 = s$, tại các lần lặp tiếp theo ta lần lượt thêm vào S_1 một đỉnh $u*$. Do đó giả thiết $d[v]$ là đường đi ngắn nhất từ s đến v với mọi $v \epsilon S_1$ là đúng với lần lặp đầu tiên. Áp dụng quy nạp suy ra thuật toán Dijkstra cho ta đường đi ngắn nhất từ s đến mọi đỉnh của đồ thị.

Nhận xét:

- Đơn giản, dễ cài đặt.

- Trong trường hợp tìm u bằng tìm kiếm tuyến tính, độ phức tạp thuật toán là $O(n^2)$.

- Với đồ thị thưa, do phải liên tục lấy phần tử u có $d[u]$ nhỏ nhất trong tập T, nên tốt nhất T nên là một hàng đợi ưu tiên (priority queue). Nếu cài đặt hàng đợi ưu tiên bằng Binary Heap thì độ phức tạp tính toán chỉ còn $O((m+n)log(n))$, với hàng đợi ưu tiên cài đặt bằng Fibonacci Heap thì độ phức tạp tính toán còn tốt hơn: $O(m + nlog(n))$.

1.2.4 Thuật toán A^*

A^* là một thuật toán tìm kiếm trong đồ thị có trọng số không âm, tìm đường đi từ một đỉnh khởi đầu s đến một đỉnh đích t cho trước, sử dụng một hàm heuristic ước lượng khoảng cách từ nút hiện tại đến nút đích (trạng thái đích), và nó sẽ duyệt đồ thị theo thứ tự ước lượng Heuristic này.

Ý tưởng:

Xét bài toán tìm đường, A^* sẽ xây dựng tăng dần các tuyến đường từ đỉnh xuất phát đến khi nó tìm thấy đường đi chạm đến đích. Để xác định khả năng dẫn đến đích, A^* sử dụng một đánh giá heuristic về khoảng cách từ một điểm bất kì cho trước đến đích. Trong ví dụ

về bài toán tìm đường đi giao thông này thì đánh giá heuristic là khoảng cách đường chim bay từ điểm cho trước đến điểm đích.

Yêu cầu của hàm đánh giá $h(x)$ là nó phải *tối ưu (optimistic heuristic)*, nghĩa là giá trị của nó phải không vượt quá giá trị thực (độ dài đường đi từ x đến đích t). Tức là $h(x) \leq d[x,t], \forall x \in V$.

Mô tả thuật toán:

Thuật toán A^* được cài đặt dựa trên một tập hợp gọi là *tập mở (OPEN SET)*. Ban đầu *OPEN SET* chỉ chứa đỉnh nguồn s, những đỉnh khác ở trong tập này có nghĩa là nó đã được *phát hiện (discovered)* bởi thuật toán (do được nối với một đỉnh thuộc *OPEN SET* trước đó) nhưng chưa phân tích tiếp các đỉnh kề với nó. Khi đã phân tích hết thì đỉnh này được loại ra khỏi *OPEN SET*.
Ở mỗi lần lặp thì chọn một đỉnh v trong *OPEN SET* sao cho khoảng cách đến đích theo dự đoán của nó là nhỏ nhất. Tức $f(v) = g(v) + h(v)$ nhỏ nhất, trong đó $g(v)$ là chi phí nhỏ nhất từ nguồn s đến v (tính được cho đến thời điểm hiện tại), $h(v)$ là đánh giá heuristic về chi phí từ v đến đích t. Sau khi đã chọn được đỉnh v, tối ưu hoá đường đi ngắn nhất của các đỉnh kề với v (như trong thuật toán Dijkstra), nếu đỉnh kề đó không thuộc *OPEN SET* thì cho vào. Sau đó loại v ra khỏi *OPEN SET*.
Thuật toán dừng lại khi *OPEN SET* rỗng.

```
def AStar(s, t)
    #init
    for v in V:
        f[v] = d[v] = Infinity
        p[v] = v

    d[s] = 0
    f[s] = h(s)
    openset = {s}

    while Not openset.empty():
        v = openset.min()
        if v == t: #target found
            return
        for u in Adjacency[v]:
            if d[u] > d[v] + a[v,u]:
                d[u] = d[v] + a[v,u]
                p[u] = v
                f[u] = d[u] + h(u)
                if u in openset:
                    openset.insert(u)
        openset.remove(v)

    return # target not found
```

Nhận xét

- Để dễ dàng lấy ra đỉnh v có $f[v]$ nhỏ nhất từ *OPEN SET* thì *OPEN SET* nên được cài đặt dưới dạng *hàng đợi ưu tiên (priority queue)*.

- Có thể dùng *Bảng màu (COLOR MAP)* để ghi nhận trạng thái của mỗi đỉnh. Ban đầu, tất cả là *trắng (white)*, khi đưa vào *OPEN SET* thì có màu *xám (gray)*, đỉnh được đưa ra khỏi *OPEN SET* sẽ có màu *đen (black)*. Nhờ đó mà việc xác định xem đỉnh u có thuộc *openset* hay không (dòng 20) có thể thực hiện trong thời gian $O(1)$.
- Nếu $h()$ là hàm *đơn điệu (consistent heuristic function)* (tức là $h()$ có tính chất $h(x) \leq a[x,y] + h(y), \forall x,y \in V$) thì với mỗi đỉnh u bị loại khỏi *openset* ($color[u] = black$), $d[v]$ đã được tối ưu. Do đó nếu sau này gặp lại u thì ta không cần kiểm trả điều kiện ở dòng 16 nữa.
- Thuật toán Dijkstra là trường hợp đặc biệt của A^*, với $h(v) = 0, \forall vinV$. Ta thấy hàm $h()$ này vừa là *optimistic* vừa là *consistent*, nên ở phần cài đặt thuật toán Dijkstra, ta chỉ cần phải kiểm tra điều kiện tối ưu khi đỉnh v còn thuộc tập T.

1.2.5 Thuật toán tìm đường đi ngắn nhất trên đồ thị không có chu trình

Bài toán tìm đường đi ngắn nhất trên đồ thị không có chu trình với trọng số là số thực tùy ý hoàn toàn có thể giải được với thuật toán với độ phức tạp tốt hơn thuật toán Ford - Bellman.

Mô tả thuật toán

Bước 1: Với một đồ thị không có chu trình, rõ ràng nếu tồn tại đường đi từ i đến j thì sẽ không tồn tại đường đi từ j đến i. Do đó, không mất tính tổng quát, ta có thể *đánh số lại* ***(topological sorting)*** các đỉnh sao cho chỉ tồn tại những cung $(order[i], order[j])$ sao cho $i < j$. Khi đó bài toán tìm đường đi ngắn nhất sẽ trở nên đơn giản hơn.

Do mục đích của ta tìm đường đi ngắn nhất từ đỉnh nguồn s đến tất cả các đỉnh còn lại hoặc là đỉnh đích t nào đó nên ta chỉ cần đánh số cho đỉnh s và các đỉnh có thể tìm thấy đường đi từ s. Nếu dùng thuật toán *tìm kiếm theo chiều sâu* ***(DFS)*** để liệt kê các đỉnh có thể đi tới từ s vào một tập T thì chỉ số *order* mới của các đỉnh chính là vị trí của nó trong T theo thứ tự ngược lại. Để ý rằng $order[s]$ luôn là 0.

Thuật toán đánh số như sau:

```
1  def DFS(v, T):
2      for u in Adjacency[v]:
3          if Not u in T:
4              DFS(u, T)
5      T.add(v)
6
7  def TopologySort(s):
8      T = []
9      v = [0] * |V|
10
11     DFS(s, T)
12     order = 0
13     for u in reverse(T):
14         map[order] = u # map[order] la dinh duoc danh so lai la order
```

```
15          order = order + 1
16
17      return order
```

Nhận xét:

- Độ phức tạp tính toán là $O(m)$. Tuy nhiên với đồ thị không phải là liên thông mạnh thì thuật toán có thể kết thúc trước khi duyệt hết tất cả các đỉnh, cung.
- Nếu là bài toán tìm đường đi ngắn nhất từ s đến t thì có thể bỏ qua việc thực hiện DFS khi $v = t$. Nếu kết thúc thuật toán mà t chưa được đánh số thì có nghĩa là không tìm được đường đi từ s đến t.
- Nếu trong khi thực hiện DFS phát hiện được một *cạnh ngược (**back edge**)* thì đồ thị có chu trình, ta có thể kết thúc bài toán ngay tại đây.
- Thuật toán trả về số đỉnh đã được đánh số lại.

Bước 2: Sau khi đã đánh số lại các đỉnh, ta có thuật toán tìm đường đi ngắn nhất như sau:

```
1   def DAGShortestPath(s):
2       # Ban dau gan khoang cach tu s den tat ca cac dinh khac la Infinity
3       for v in V:
4           d[v] = Infinity
5           p[v] = -1
6
7       d[s] = 0
8       p[s] = 0
9
10      count = TopologySort(s)
11      for i in range(0, count):
12          v = map[i]
13          for u in Adjacency[v]:
14              if d[u] > d[v] + a[v, u]:
15                  d[u] = d[v] + a[v, u]
16                  p[u] = v
```

Chứng minh: Có thể chứng minh thuật toán bằng cách tương tự như phần chứng minh thuật toán Ford-Bellman, chỉ khác là đường đi từ s đến u nếu tồn tại thì nó chỉ đi qua các đỉnh $\{map[0], map[1], ..., map[order[u] - 1]\}$ (do ta đã đánh số lại). Nhờ đó giảm được một vòng lặp so với Ford-Bellman.

Nhận xét:

- Độ phức tạp tính toán của phần tìm đường đi ngắn nhất là $O(m+n)$ (do mỗi cung được duyệt tối đa 1 lần). Tổng hợp lại, độ phức tạp tính toán của cả hai bước là $O(m + n)$.
- Kết thúc thuật toán, nếu đỉnh v nào đó có $d[v] = Infinity$ thì có nghĩa là không có đường đi nào từ s đến v.
- Bài toán tìm đường đi từ s đến t có thể kết thúc khi tìm thấy $v = t$ (dòng 12).

1.2.6 Thuật toán Floyd-Warshall

Ta có thể giải bài toán tìm đường đi ngắn nhất giữa tất cả các đỉnh của đồ thị bằng cách sử dụng n lần thuật toán Ford-Bellman (độ phức tạp trở thành $O(n^4)$) hoặc thuật toán Dijkstra (độ phức tạp trở thành $O(n^3)$). Tuy nhiên, với trường hợp tổng quát, ta phải sử dụng thuật toán Floyd-Warshall.

Mô tả thuật toán

Tư tưởng của thuật toán này là với mọi đỉnh k của đồ thị được xét theo thứ tự từ 1 đến n, xét mọi cặp đỉnh (u, v), ta làm cực tiểu hóa $d[u, v]$ theo công thức: $d[u, v] = min(d[u, v], d[u, k] + d[k, v])$. Tức là nếu đường đi từ u đến v đang có lại dài hơn đường đi từ u đến k cộng với đường đi từ k đến v thì ta hủy bỏ đường đi từ u đến v hiện tại và thay vào đó là hai đường từ u đến k và từ k đến v. Sau đây là mã giả mô tả thuật toán:

```
procedure Floyd_Warshall
    begin
        for i = 1 to n do
           for j = 1 to n do
              begin
                d[i, j] = a[i, j];
                Before[i, j] = I;
              end;
        for k = 1 to n do
            for i = 1 to n do
                for j = 1 to n do
                    if  (d[i, j] > d[i, k] + d[k, j]) then
                        begin
                          d[i, j] = d[i, k] + d[k, j];
                          Before[i, j] = Before[k, j];
                        end;
    end
```

Chứng minh tính đúng đắn của thuật toán:

- Gọi $a^k[u, v]$ là độ dài đường đi ngắn nhất từ u đến v mà chỉ đi qua các đỉnh trung gian thuộc tập $1, 2, \ldots, k$, khi đó với $k = 0$ thì $a^0[u, v] = a[u, v]$.
- Giả sử ta đã tính được các $-a^{k-1}[u, v]$ thì $a^k[u, v]$ sẽ được xây dựng như sau:
  - Nếu không đi qua đỉnh k tức là chỉ đi qua các đỉnh trung gian thuộc tập $1, 2, \ldots, k-1$, khi đó $a^k[u, v] = a^{k-1}[u, v]$.
  - Nếu có đi qua đỉnh k thì đường đi đó sẽ bao gồm hai đường đi từ u đến k và từ k đến v, hai đường này chỉ đi qua các đỉnh trung gian thuộc tập $1, 2, \ldots, k-1$, khi đó ta có $a^k[u, v] = a^{k-1}[u, k] + a^{k-1}[k, v]$.
  - Vì ta muốn $a^k[u, v]$ là cực tiểu nên suy ra $a^k[u, v] = min(a^{k-1}[u, v], a^{k-1}[u, k] + a^{k-1}[k, v])$.
- Cuối cùng, ta có $a^n[u, v]$ là đường đi ngắn nhất từ u đến v mà chỉ đi qua các đỉnh trung gian thuộc tập $1, 2, \ldots, n$.

Nhận xét:

- Khi cài đặt thuật toán, ta không có khái niệm $a^k[u, v]$ mà sẽ thao tác trực tiếp với $a[u, v].a[u, v]$ tại bước tối ưu thứ k sẽ được tính toán để bằng $min(a[u, v]), a^{k-1}[u, k] + a^{k-1}[k, v])$.
- Độ phức tạp thuật toán là $O(n^3)$.

Chương 2

Giới thiệu về Project minh họa các thuật toán tìm đường đi ngắn nhất trên đồ thị bằng giao diện đồ họa

Chương 3

Kết luận