**BỘ CÔNG THƯƠNG**

**TRƯỜNG ĐẠI HỌC CÔNG THƯƠNG TP.HỒ CHÍ MINH**

**KHOA CÔNG NGHỆ - THÔNG TIN**

**-----o0o-----**



BÁO CÁO TIỂU LUẬN

**BÀI TẬP NHÓM LÝ THUYẾT ĐỒ THỊ**

**Nhóm 2:**

**Thành viên:**

1. Lăng Minh Hải
2. Trần Trọng Quí
3. Nguyễn Minh Sang
4. Nguyễn Văn Thành
5. Nguyễn Hoành Thịnh

**Thành phố Hồ Chính Minh, tháng 11 ,năm 2023**

**Biên bảng phân công**

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **Họ và Tên** | **Phần trăm đóng góp** | **Nội dung phân công** |
| 23 | Lăng Minh Hải | 100% | Nhập xuất đồ thị |
| 53 | Trần Trọng Quí | 100% | Thuật toán BFS,DFS; ứng dụng BFS,DFS |
| 56 | Nguyễn Minh Sang | 100% | Thuật toán Dijkstra |
| 60 | Nguyễn Văn Thành  (nhóm trưởng) | 100% | Thuật toán Dijkstra; ứng dụng Dijkstra |
| 64 | Nguyễn Hoành Thịnh | 100% | Thuật toán Ploy |

MỤC LỤC

[1. Đồ thị nhập vào 2](#_Toc152011565)

[1.1. Đồ thị không có trọng số 2](#_Toc152011566)

[1.2. Đồ thị có trọng số 3](#_Toc152011567)

[2. Lý thuyết 4](#_Toc152011568)

[2.1. In ra màn hình ma trận kề 4](#_Toc152011569)

[2.2. Xuất danh sách kề của đỉnh i nhập từ bàn phím 4](#_Toc152011570)

[2.3. In tất cả danh sách kề của tất cả các đỉnh 5](#_Toc152011571)

[2.4. Thuật toán tìm kiếm trên đồ thị (DFS, BFS) 5](#_Toc152011572)

[2.4.1. Tìm kiếm theo chiều rộng (DFS - Depth First Search) 5](#_Toc152011573)

[2.4.2. Tìm kiếm theo chiều rộng (BFS - Breadth First Search) 7](#_Toc152011574)

[2.5. Thuật toán Ployed Warshall 10](#_Toc152011575)

[2.6. Thuật toán Dijkstra 13](#_Toc152011576)

[3. Ứng dụng 15](#_Toc152011577)

* 1. Đồ thị nhập vào
     1. Đồ thị không có trọng số

Đầu vào là 1 danh sách cạnh với 2 số đầu là số đỉnh và số cạnh

|  |  |
| --- | --- |
| 9: số đỉnh | 9: số cạnh |
| Đầu | Cuối |
| 1 | 2 |
| 1 | 3 |
| 1 | 5 |
| 2 | 4 |
| 3 | 6 |
| 3 | 7 |
| 3 | 9 |
| 5 | 8 |
| 8 | 9 |

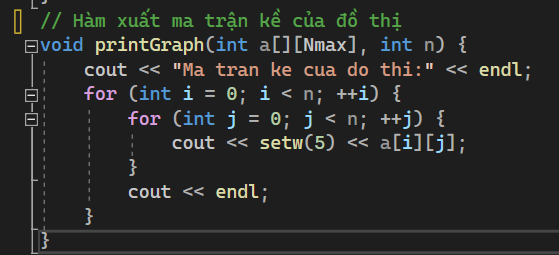


* + 1. Đồ thị có trọng số

|  |  |  |
| --- | --- | --- |
| Đầu | Cuối | Trọng số |
| 0 | 1 | 10 |
| 0 | 3 | 6 |
| 0 | 2 | 5 |
| 0 | 6 | 24 |
| 1 | 5 | 18 |
| 1 | 2 | 1 |
| 3 | 4 | 2 |
| 3 | 6 | 16 |



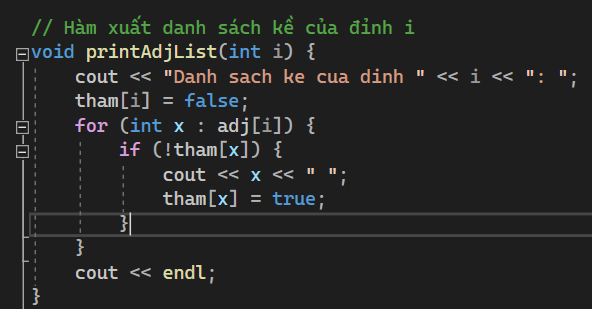
* 1. Lý thuyết
     1. In ra màn hình ma trận kề



Hàm printfGraph truyền vào 2 tham số: Mảng 2 chiều a cho ma trận kề của đồ thị; n là số đỉnh của đồ thị.

Dùng 2 vòng lặp for để lần lượt in từng giá trị trong ma trận kề. Hết mỗi dòng j sẽ xuống dòng và tăng i đến khi nhỏ hơn số đỉnh.

* + 1. Xuất danh sách kề của đỉnh i nhập từ bàn phím



Hàm printAdjList truyền vào tham số i (đỉnh) nhập từ bàn phím trong hàm main.

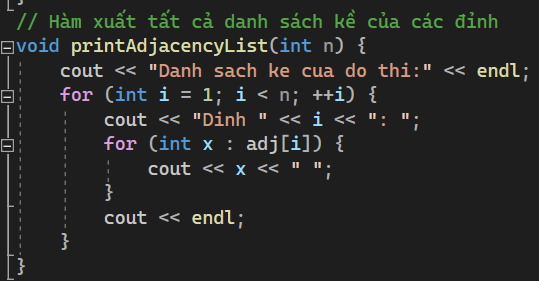
Gán tham[i]=false để đánh dấu điểm i là chưa thăm trong quá trình duyệt.

Vòng for duyệt qua từng phần tử x trong danh sách kề của đỉnh i. Xét điều kiện:

+ Nếu tham[x] = false tức là x chưa được thăm thì in ra màn hình giá trị của x và gán x là đã thăm.

+ Nếu đã thăm x thì tăng i cho đến hết x.

* + 1. In tất cả danh sách kề của tất cả các đỉnh



Cũng giống như hàm ở phần trên là in danh sách kề của đỉnh nhưng hàm printfAdjaccencyList sẽ xuất tất cả các đỉnh kề của từng đỉnh có trong ma trận.

Hàm truyền vào n là số đỉnh. Cout dòng danh sách kề của đồ thị. Duyệt mảng 2 chiều: i từ 1 đến n và in ra đỉnh i; tương ứng với từng đỉnh i sẽ duyệt đến từng đỉnh x trong danh sách kề của đỉnh i và in ra x.

* + 1. Thuật toán tìm kiếm trên đồ thị (DFS, BFS)
       1. Tìm kiếm theo chiều rộng (DFS - Depth First Search)

DFS là thuật toán được sử dụng để tìm kiếm hay duyệt qua (thăm) tất cả các đỉnh  
thuộc các thành phần liên thông của đồ thị một cách có hệ thống với kỹ thuật gọi là ưu  
tiên độ sâu. Thuật toán có nhiều ứng dụng giải các bài toán về liên thông, cây khung,  
đường đi...Thuật toán có thể được cài đặt bằng phương pháp đệ qui hay phương pháp  
lặp sử dụng cấu trúc stack để khử đệ qui. Độ phức tạp của thuật toán là O(n+m) với n  
là số đỉnh và m là số cạnh của đồ thị.

**Ý tưởng :**

- Xuất phát từ một đỉnh u cho trước chưa thăm

- Thăm đỉnh u và tìm đỉnh kề v chưa được thăm. Nếu có tiếp tục duyệt từ v lặp lại quá trình này cho đến khi không còn đỉnh kề nào chưa được thăm.

- Nếu tại một đỉnh u nào đó không còn đỉnh kề chưa thăm quay trở lại đỉnh trước của u (đỉnh kề của u trên đường đến u) và tìm các đỉnh kề chưa thăm của đỉnh này

**Cài đặt thuật toán**

//duyet thuat toan DFS Danhsachke

void DFS(int u) {

temp.push\_back(u);

tham[u] = true;

for (int v : adj[u])

{

if (!tham[v])

DFS(v);

}

}

- Đầu tiên, đỉnh u được thêm vào danh sách kết quả temp, đánh dấu đỉnh này đã được thăm.

tham[u] = true: Đánh dấu đỉnh u là đã được thăm để tránh việc duyệt lại.

- Duyệt qua các đỉnh kề của u:

Với mỗi đỉnh v kề với u (trong danh sách kề adj[u]):

Nếu đỉnh v chưa được thăm

Gọi đệ quy DFS(v): Tiến hành duyệt theo chiều sâu từ đỉnh v.

- Thuật toán sẽ tiếp tục duyệt sâu vào từng đỉnh kề của mỗi đỉnh đệ quy cho đến khi không còn đỉnh nào để duyệt nữa.

**Lưu ý:**

- Ở đây hàm DFS được triển khai bằng cách sử dụng để quy để duyệt qua các đỉnh kề. Dẫn đến việc với đồ thị có độ sâu lớn hoặc quá sâu có thể gây tràn bộ nhớ stack có thể thay bằng cách sử vòng lặp hoặc cấu trúc dữ liệu stack tự tạo.

- DFS cũng cần được gọi từ mỗi đỉnh chưa được thăm để xác định tất cả các thành phần liên thông của đồ thị. Nếu không, chỉ một phần của đồ thị có thể được duyệt và các thành phần khác nhau sẽ không được xét.

- DFS theo chiều sâu, duyệt hết một nhánh trước khi chuyển sang nhánh khác. Điều này có thể dẫn đến việc đầu tiên tìm được đường đi sâu nhất trước khi đi theo nhánh khác.

- Duyệt qua các đỉnh kề của một đỉnh chưa được thăm trước khi di chuyển đến đỉnh kề khác. Điều này đảm bảo việc duyệt hết một nhánh trước khi quay lại để xét các nhánh khác.

- DFS có thể được áp dụng cho cả đồ thị có hướng và vô hướng, nhưng cần xác định cách duyệt đỉnh kề phù hợp với từng loại đồ thị.

* + - 1. Tìm kiếm theo chiều rộng (BFS - Breadth First Search)

Thuật toán **BFS** dùng để duyệt và tìm kiếm trên đồ thị cũng tương tự như thuật toán DFS nhưng thay vì dùng cấu trúc stack thì BFS dùng cấu trúc hàng đợi để thực hiện,  
kỹ thuật này gọi là tìm kiếm ưu tiên chiều rộng. Thuật toán có độ phức tạp là **O(m+n).**

**Ý tưởng:**

**-** Bắt đầu từ một đỉnh **u** cho trước chưa được thăm.

- Thăm đỉnh **u** và tìm tất cả các đỉnh kề chưa được thăm sau đó thêm chúng vào **hàng đợi**.

- Lặp lại quá trình này cho đến khi không còn đỉnh nào trong hàng đợi(**hàng đợi rỗng**).

- Duyệt qua các đỉnh kề chưa thăm của mỗi đỉnh trong **hàng đợi**, đánh dấu và thêm chúng vào hàng đợi và tiếp tục thăm các đỉnh kề chưa được thăm của đỉnh đó. **Cài đặt thuật toán**

//duyet thuat toan BFS Danhsachke

void BFS(int u) {

queue<int> q;

q.push(u);

tham[u] = true;

while (!q.empty())

{

int v = q.front();

q.pop();

temp.push\_back(v);

for (int x : adj[v])

{

if (tham[x] == false)

{

q.push(x);

tham[x] = true;

}

}

}

}

- Đầu tiên, khởi tạo một **hàng đợi** q và thêm đỉnh u vào hàng đợi. Đỉnh này được coi là đỉnh bắt đầu.

queue<int> q;

q.push(u);

tham[u] = **true:** Đánh dấu đỉnh u là đã được thăm để tránh việc lặp lại.

- **Duyệt qua hàng đợi:**

Khi hàng đợi không rỗng:

Lấy đỉnh đầu tiên từ hàng đợi ( **int v = q.front()** ).

Sau đó loại bỏ đỉnh vừa lấy ra khỏi hàng đợi.

Thêm đỉnh v vào danh sách kết quả **temp**.

**- Duyệt qua các đỉnh kề của v:**

Với mỗi đỉnh x kề với v (trong danh sách kề adj[v]):

Nếu đỉnh x chưa được thăm (!tham[x]):

Thêm đỉnh x vào hàng đợi.

Đánh dấu đỉnh x là đã được thăm.

- Thuật toán BFS này duyệt qua từng đỉnh kề của các đỉnh đã được thăm, lưu trữ các đỉnh đã duyệt theo thứ tự rộng trước khi tiến hành duyệt đỉnh tiếp theo.

**Lưu ý:**

- Ở đây BFS được triển khai sử dụng một hàng đợi (**queue**) để duyệt qua các đỉnh kề theo chiều rộng. Việc này đảm bảo các đỉnh được thăm theo cấp bậc và không lặp lại.

- Nếu đồ thị không liên thông, cần gọi thuật toán BFS (**hoặc DFS**) từ mỗi đỉnh chưa được thăm để xác định các thành phần liên thông khác nhau của đồ thị.

- BFS có thể được áp dụng cho cả đồ thị có hướng và vô hướng, nhưng cần xác định cách duyệt đỉnh kề phù hợp với từng loại đồ thị.

- BFS duyệt theo chiều rộng, nghĩa là các đỉnh cùng cấp bậc sẽ được duyệt trước, sau đó mới duyệt các đỉnh cấp bậc cao hơn. Điều này tạo ra việc duyệt theo cấp bậc của đồ thị, từ đỉnh gốc đến các đỉnh xa hơn theo từng tầng.

**3. Ứng dụng 2 thuật toán BFS và DFS để kiểm tra tính liên thông của đồ thị**

//ktr tính lien thong

int ktrTinhLienThong(int n, int m) {

int cnt = 0;

for (int i = 1; i <= n; i++)

{

if (!tham[i]) {

++cnt;

BFS(i);

}

}

return cnt;

}

**Ở đây,** ta sẽ sử dụng BFS để kiểm tra tính liên thông.

- Đầu tiên khởi tạo biến cnt để đếm số lượng thành phần liên thông trong đồ thị.

- Duyệt các đỉnh từ 1 đến n

* + Nếu đỉnh i chưa được thăm (!tham[i]) tức là chưa thuộc bất kỳ thành phần liên thông nào.
  + Tăng biến cnt để đếm số lượng thành phần liên thông
  + Gọi hàm BFS(i) để duyệt từ đỉnh i.Mục tiêu là tìm tất cả các đỉnh thuộc cùng một thành phần liên thông với i.
  + Cuối cùng là trả về kết quả cuối là số lượng thành phần liên thông của đồ thị.

- Nếu số lượng thành phần liên thông là 1, tức là chỉ cần một lần duyệt từ một đỉnh bất kỳ để có thể duyệt hết các đỉnh khác, đồ thị là liên thông.

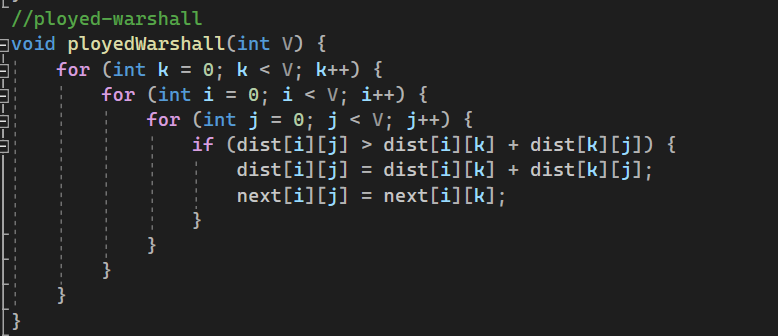
- Ngược lại, nếu có nhiều hơn một thành phần liên thông, đồ thị không liên thông và sẽ có số thành phần liên thông được xuất ra để thể hiện.

* + 1. Thuật toán Ployed Warshall

Thuật toán Floyd-Warshall là một thuật toán quan trọng trong lý thuyết đồ thị, được đặt theo tên của hai nhà khoa học đã đóng góp vào việc phát triển nó, Robert Floyd và Stephen Warshall. Thuật toán này được sử dụng để tìm khoảng cách ngắn nhất giữa mọi cặp đỉnh trong một đồ thị có hướng hoặc không hướng, có trọng số hoặc không trọng số.

Dưới đây là mã giả cho thuật toán Floyd-Warshall và giải thích về các hàm trong chương trình:

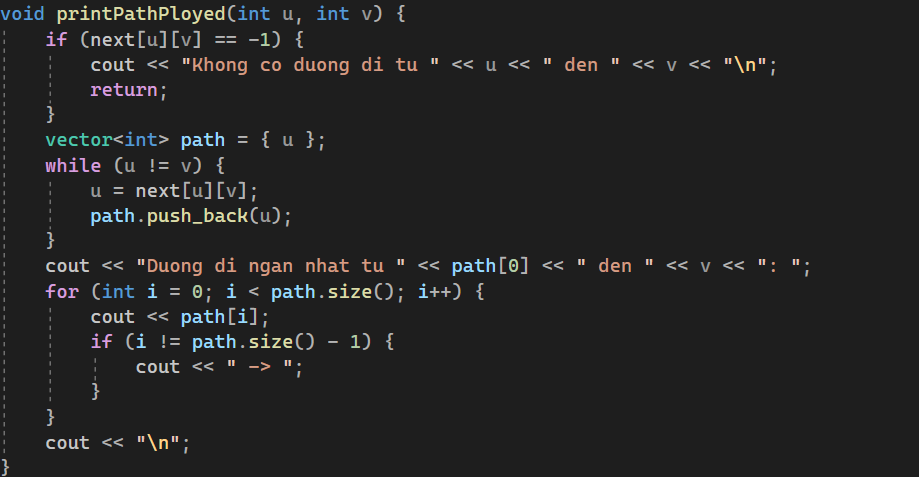
*Hàm ployedWarshall(V):*



1. Duyệt qua mọi cặp đỉnh (i, j) với k là đỉnh trung gian.

2. Nếu dist[i][j] > dist[i][k] + dist[k][j] thì cập nhật dist[i][j] = dist[i][k] + dist[k][j] và next[i][j] = next[i][k].

*Hàm printPathPloyed(u, v):*



1. Nếu không có đường đi từ u đến v (next[u][v] == -1), in ra thông báo và thoát hàm.

2. Khởi tạo một vector để lưu đường đi.

3. Trong khi u khác v, thêm u vào đường đi và cập nhật u = next[u][v].

4. In ra đường đi từ đỉnh u đến v.

Hàm nhapDoThiTrongSo(a, n, m):

1. Mở tệp "inputDijkstra.txt".

2. Đọc số lượng đỉnh n và cạnh m.

3. Đọc danh sách các cạnh và trọng số tương ứng, cập nhật ma trận a.

4. Đóng tệp.

*Giải thích các hàm:*

`ployedWarshall`: Hàm này thực hiện thuật toán Floyd-Warshall để tính toán khoảng cách ngắn nhất giữa mọi cặp đỉnh trong đồ thị.

`printPathPloyed`: Hàm này in ra đường đi ngắn nhất từ đỉnh u đến đỉnh v. Nếu không có đường đi, hàm sẽ in ra thông báo không có đường đi.

`nhapDoThiTrongSo`: Hàm này đọc dữ liệu đầu vào từ tệp "inputDijkstra.txt", bao gồm số lượng đỉnh, số lượng cạnh và danh sách các cạnh cùng với trọng số tương ứng.

*Trình bày thuật toán:*

Thuật toán Floyd-Warshall bắt đầu bằng việc khởi tạo ma trận khoảng cách `dist` và ma trận `next`. Ma trận `dist` lưu khoảng cách ngắn nhất giữa mọi cặp đỉnh, trong khi ma trận `next` lưu đỉnh kế tiếp trên đường đi ngắn nhất từ một đỉnh đến một đỉnh khác.

Sau đó, thuật toán sẽ duyệt qua mọi cặp đỉnh `(i, j)` với `k` là đỉnh trung gian. Nếu khoảng cách từ `i` đến `j` thông qua `k` ngắn hơn khoảng cách hiện tại từ `i` đến `j`, thuật toán sẽ cập nhật khoảng cách và đỉnh kế tiếp.

Cuối cùng, thuật toán sẽ in ra đường đi ngắn nhất từ đỉnh bắt đầu đến đỉnh kết thúc. Nếu không có đường đi giữa hai đỉnh, thuật toán sẽ thông báo không có đường đi. Chú ý rằng, bạn cần nhập cặp đỉnh bắt đầu và kết thúc sau khi nhập danh sách các cạnh.

* + 1. Thuật toán Dijkstra

Thuật toán Dijkstra có thể giải quyết bài toán tìm đường đi ngắn nhất trên đồ thị vô hướng lẫn có hướng miễn là trọng số không âm.

Được đặt tên theo nhà toán học và nhà máy tính người Hà Lan Edsger W. Dijkstra, thuật toán này thường được áp dụng trong các ứng dụng mạng và địa lý.

Độ phức tạp: O((E+V) logV)

Ý tưởng cơ bản của thuật toán như sau:

*Bước 1:* Từ đỉnh gốc, khởi tạo khoảng cách tới chính nó là (0,0), khởi tạo khoảng cách nhỏ nhất ban đầu tới các đỉnh khác là (-,∞). Ta được danh sách các khoảng cách tới các đỉnh.

*Bước 2:* Chọn đỉnh a có khoảng cách nhỏ nhất trong danh sách này và ghi nhận. Các lần sau sẽ không xét tới đỉnh này nữa.

*Bước* *3:* Lần lượt xét các đỉnh kề b của đỉnh a. Nếu khoảng cách từ đỉnh gốc tới đỉnh b nhỏ hơn khoảng cách hiện tại đang được ghi nhận thì cập nhật giá trị và đỉnh kề a vào khoảng cách hiện tại của b.

*Bước 4:* Sau khi xét tất cả đỉnh kề b của đỉnh a. Lúc này ta được danh sách khoảng cách tới các điểm đã được cập nhật. Quay lại Bước 2 với danh sách này. Thuật toán kết thúc khi chọn được khoảng cách nhỏ nhất từ tất cả các điểm.

**Code mẫu thuật toán dijkstra**

Sử dụng mảng 2 chiều để lưu đường đi trọng số của bài toán.

1. Đặt khoảng cách nguồn là 0

2. Duyệt qua tất cả các đỉnh của đồ thị

Khởi tạo biến u để lưu đỉnh có khoảng cách ngắn nhất

Vòng lặp chọn đỉnh chưa thăm có khoảng cách tới nguồn ngắn nhất

Điều kiện: Đỉnh thứ j chưa thăm và khoảng cách ngắn hơn đỉnh u hoặc u chưa được gán giá trị

Gán u = j

Đánh dấu đỉnh u đã thăm

Vòng lặp cập nhật khoảng cách từ đỉnh nguồn tới đỉnh kề của u

Điều kiện: “Đỉnh v chưa xét, nối với u, khoảng cách từ s tới u không phải là vô cùng, và khoảng cách mới từ s tới v ngắn hơn khoảng cách hiện tại”

Cập nhật khoảng cách từ nguồn tới đỉnh v bằng cách lấy khoảng cách của u cộng với trọng số của cạnh uv.

void dijkstra(int s, int n) {

khoangCach[s] = 0;

for (int i = 0; i < n; i++)

{

//Chon dinh chua tham co khoang cach toi nguon ngan nhat

int u = -1;//Dinh kc ngan nhat

for (int j = 0; j < n; j++)

{

if (!tham[j] && (u == -1 || khoangCach[j] < khoangCach[u]))

{

u = j;

}

}

tham[u] = true;

//cap nhat khoang cach tu dinh nguon toi dinh ke cua u

for (int v = 0; v < n; v++)

{

if (!tham[v] && a[u][v] && khoangCach[u] != INF && khoangCach[u] + a[u][v] < khoangCach[v])

{

khoangCach[v] = khoangCach[u] + a[u][v];

pre[v] = u;

}

}

}

}

* 1. Ứng dụng

Mục tiêu: Cho người dùng nhập điểm đi và điểm đến chạy thuật toán để tìm đường ngắn nhất đến các điểm đó.

Tìm đường đi ngắn nhắt từ …. đến …. (tên tỉnh thành)

Bắt các chuyến xe đi gần nhất

Trọng số 10km = 1

Table 1: Mã vùng

|  |  |
| --- | --- |
| 0 : Sài Gòn | 1 : Vũng Tàu |
| 2 : Cần Giờ | 3 : Tiền Giang |
| 4 : Bến Tre | 5 : Tây Ninh |
| 6 : Sóc Trăng |  |



void dijkstra(int s, int n) {

khoangCach[s] = 0;

for (int i = 0; i < n; i++)

{

//Chon dinh chua tham co khoang cach toi nguon ngan nhat

int u = -1;//Dinh kc ngan nhat

for (int j = 0; j < n; j++)

{

if (!tham[j] && (u == -1 || khoangCach[j] < khoangCach[u]))

{

u = j;

}

}

tham[u] = true;

//cap nhat khoang cach tu dinh nguon toi dinh ke cua u

for (int v = 0; v < n; v++)

{

if (!tham[v] && a[u][v] && khoangCach[u] != INF && khoangCach[u] + a[u][v] < khoangCach[v])

{

khoangCach[v] = khoangCach[u] + a[u][v];

pre[v].ma = u;

string tmp = tenVung(u, n);

pre[v].ten = tmp;

}

}

}

}

//in đường đi

void printPathUngDung(int s, int dest) {

if (khoangCach[dest] == INF) {

cout << "Khong co duong nao tu " << maVungArr[s].ten << " den " << maVungArr[dest].ten << endl;

return;

}

cout << "Duong ngan nhat tu " << maVungArr[s].ten << " den " << maVungArr[dest].ten << ": ";

ma\_Vung current = maVungArr[dest];

string tmp[Nmax]; int i = 0;

string den = current.ten;

while (pre[current.ma].ma != INF) {

tmp[i] = pre[current.ma].ten;

current = pre[current.ma];

i++;

}

while (i != 0)

{

cout << tmp[i - 1] << " -> ";

i--;

}

cout << den;

cout << endl;

}