

## MỤC LỤC

ĐÁNH GIÁ NHẬN XÉT CỦA GIẢNG VIÊN HƯỚNG DẪN.....	3
GIỚI THIỆU .....	4
I/ Lý do chọn đề tài .....	4
II/Sinh viên thực hiện .....	4
CHƯƠNG 1: CƠ SỞ LÝ THUYẾT .....	5
1.1 Định nghĩa đồ thị.....	5
1.2 Định nghĩa đồ thị có hướng .....	5
1.3 Định nghĩa đồ thị đầy đủ .....	5
1.4 Định nghĩa đồ thị liên thông .....	5
1.5 Định nghĩa đồ thị có hướng liên thông mạnh.....	5
1.6 Định nghĩa liên kết và cạnh liên thuộc đỉnh.....	5
1.7 Định nghĩa đường đi.....	5
1.8 Định nghĩa đường đi ngắn nhất .....	5
1.9 Định nghĩa độ phức tạp thuật toán về thời gian.....	6
1.10 Định nghĩa độ phức tạp thuật toán về không gian .....	6
CHƯƠNG 2: PHÁT BIỂU VẤN ĐỀ .....	7
2.1 Phát biểu bài toán .....	7
2.1.1 Bài toán.....	7
2.1.2 Nhận xét bài toán .....	7
2.1.3 Phép biến đổi từ đồ thị sang ma trận trọng số .....	7
2.2 Thuật toán Floyd-warshall .....	9
2.2.1 Thuật toán .....	9
2.2.2 Đánh giá độ phức tạp thời gian và không gian .....	10
2.2.3 Chương trình cài đặt.....	10
2.2.4 Minh họa thuật toán với bộ dữ liệu mẫu: .....	13
2.3 Thuật toán Dijkstra và cải tiến thuật toán Dijkstra: .....	15
2.3.1 Thuật toán .....	15
2.3.2 Đánh giá độ phức tạp thời gian và không gian .....	16
2.3.3 Cài đặt chương trình.....	17

2.3.4 Minh họa với dữ liệu mẫu.....	19
2.3.5 Ý tưởng cải tiến.....	20
2.3.6 Chương trình cải tiến.....	21
2.4 Thuật toán Bellman-Ford và cải tiến.....	25
2.4.1 Thuật toán .....	25
2.4.2 Đánh giá độ phức tạp thời gian và không gian.....	27
2.4.3 Cài đặt chương trình.....	27
2.4.4 Minh họa với dữ liệu mẫu.....	30
CHƯƠNG 3: KẾT QUẢ VÀ ĐÁNH GIÁ .....	31
3.1 Chương trình sinh test.....	31
3.2 Kết quả, đánh giá và ứng dụng.....	32
KẾT LUẬN .....	37
Tài liệu tham khảo .....	38

[illegible]

Chữ ký của giáo viên hướng dẫn:

TS. Trần Văn Hưng

## GIỚI THIỆU

### I/ Lý do chọn đề tài

Với nhu cầu phát triển công nghệ hiện tại, con người dần tiếp nhận và chuyển đổi công nghệ, việc tìm kiếm các địa điểm trên mạng xã hội ngày càng tăng cao. Việc tìm kiếm các địa điểm một phần dựa trên các thuật toán tìm đường đi ngắn nhất trong đồ thị. Vì vậy nhu cầu tìm hiểu và nắm bắt các thuật toán tìm kiếm đường đi ngắn nhất là điều hết sức cần thiết cho sinh viên trong thời kỳ hiện tại.

Trong các thuật toán về tìm đường đi ngắn nhất, lần này chúng ta chỉ tìm hiểu về thuật toán Floyd-warshall, Dijkstra và cải tiến Dijkstra, Bellman-Ford cải tiến

### II/Sinh viên thực hiện

Bùi Trọng Nghĩa

## CHƯƠNG 1: CƠ SỞ LÝ THUYẾT

Lý thuyết đồ thị là một ngành khoa học được phát triển từ lâu và đã có nhiều ứng dụng trong các ngành khoa học hiện đại. Những ý tưởng cơ bản của lý thuyết đồ thị đã được nhà toán học Thụy Sĩ tên là Leonhard Euler đề xuất từ thế kỷ 18, Ông đã dùng đồ thị để giải bài toán nổi tiếng về 7 chiếc cầu ở thành phố Königsberg. Đồ thị cũng được dùng để giải nhiều bài toán trong nhiều lĩnh vực khác nhau như: xác định xem hai máy tính có được nối với nhau bằng một đường truyền thông hay không, tìm đường đi ngắn nhất giữa hai thành phố trong một mạng giao thông, lập lịch thi và phân chia kênh cho các trạm phát thanh, truyền hình...

### 1.1 Định nghĩa đồ thị

Một đồ thị  $G = (V, E)$  gồm một tập khác rỗng  $V$  mà các phần tử của nó gọi là các đỉnh và một tập  $E$  mà các phần tử của nó gọi là các cạnh, đó là các cặp không có thứ tự của các đỉnh phân biệt.

### 1.2 Định nghĩa đồ thị có hướng

Một đồ thị có hướng  $G = (V, E)$  gồm một tập khác rỗng  $V$  mà các phần tử của nó gọi là các đỉnh và một tập  $E$  mà các phần tử của nó gọi là các cung, đó là các cặp có thứ tự của các phần tử thuộc  $V$ .

### 1.3 Định nghĩa đồ thị đầy đủ

Đồ thị đầy đủ: Đồ thị đầy đủ  $n$  đỉnh, ký hiệu là  $K_n$ , là đơn đồ thị mà hai đỉnh phân biệt bất kỳ của nó luôn liên kề. Như vậy,  $K_n$  có  $n(n-1)/2$  cạnh và mỗi đỉnh của  $K_n$  có bậc là  $n - 1$ .

### 1.4 Định nghĩa đồ thị liên thông

Một đồ thị (vô hướng) được gọi là liên thông nếu có đường đi giữa mọi cặp đỉnh phân biệt của đồ thị.

### 1.5 Định nghĩa đồ thị có hướng liên thông mạnh

Đồ thị có hướng  $G$  được gọi là liên thông mạnh nếu với hai đỉnh phân biệt bất kỳ  $u$  và  $v$  của  $G$  đều có đường đi từ  $u$  tới  $v$  và đường đi từ  $v$  tới  $u$ .

### 1.6 Định nghĩa liên kề và cạnh liên thuộc đỉnh

Hai đỉnh  $u$  và  $v$  trong đồ thị (vô hướng)  $G=(V,E)$  được gọi là liên kề nếu  $(u,v)$  thuộc  $E$ . Nếu  $e = (u,v)$  thì  $e$  gọi là cạnh liên thuộc với các đỉnh  $u$  và  $v$ . Cạnh  $e$  cũng được gọi là cạnh nối các đỉnh  $u$  và  $v$ . Các đỉnh  $u$  và  $v$  được gọi là các điểm đầu mút của cạnh  $e$ .

### 1.7 Định nghĩa đường đi

Đường đi độ dài  $n$  (nguyên dương) từ đỉnh  $u$  đến đỉnh  $v$  trong đồ thị  $G=(V,E)$  là một dãy các cạnh (hoặc cung)  $e_1, e_2, \dots, e_n$  của đồ thị sao cho  $e_1=(x_0,x_1), e_2=(x_1,x_2), \dots, e_n=(x_{n-1},x_n)$ , với  $x_0=u$  và  $x_n=v$ . Khi đồ thị không có cạnh (hoặc cung) bội, ta ký hiệu đường đi này bằng dãy các đỉnh  $x_0, x_1, \dots, x_n$ .

### 1.8 Định nghĩa đường đi ngắn nhất

Đường đi ngắn nhất từ đỉnh  $u$  sang đỉnh  $v$  trong đồ thị  $G(V,E)$  là đường đi độ dài

là tổng trọng số các cạnh đi từ  $u$  sang  $v$  sao cho nhỏ nhất.

### **1.9 Định nghĩa độ phức tạp thuật toán về thời gian**

Thời gian mà máy tính khi thực hiện một thuật toán không chỉ phụ thuộc vào bản thân thuật toán đó, ngoài ra còn tùy thuộc từng máy tính. Để đánh giá hiệu quả của một thuật toán, có thể xét số các phép tính phải thực hiện khi thực hiện thuật toán này. Thông thường số các phép tính được thực hiện phụ thuộc vào cỡ của bài toán, tức là độ lớn của đầu vào. Vì thế độ phức tạp thuật toán là một hàm phụ thuộc đầu vào. Tuy nhiên trong những ứng dụng thực tiễn, chúng ta không cần biết chính xác hàm này mà chỉ cần biết một ước lượng đủ tốt của chúng.

Vì vậy độ phức tạp về thời gian của thuật toán dùng để đánh giá gần đúng về thời gian chạy của một thuật toán.

### **1.10 Định nghĩa độ phức tạp thuật toán về không gian**

Cũng tương tự độ phức tạp về thời gian nhưng lần này đánh giá về không gian lưu trữ bộ nhớ của một thuật toán.

Độ phức tạp không gian và thời gian là hai độ phức tạp cực kỳ quan trọng đối với một thuật toán. Dùng để đánh giá một thuật toán có thật sự tốt tối ưu về mặt lý thuyết. Trên thực tế thời gian chạy và bộ nhớ cũng có sai lệch bởi vì các độ phức tạp này chỉ dùng để ước lượng, và so sánh đánh giá các thuật toán theo một tiêu chuẩn trên mặt lý thuyết.

## CHƯƠNG 2: PHÁT BIỂU VẤN ĐỀ

### 2.1 Phát biểu bài toán

#### 2.1.1 Bài toán

Cho đồ thị có hướng  $G=(V, E)$  và hai đỉnh  $u, v$  thuộc  $V$ . Hãy tìm đường đi ngắn nhất từ  $u$  sang  $v$ . Nếu không tồn tại đường đi thì thông báo không tồn tại đường đi.

#### 2.1.2 Nhận xét bài toán

Bài toán được phát biểu cho đồ thị có hướng có trọng số, nhưng các thuật toán sẽ trình bày đều có thể áp dụng cho các đồ thị vô hướng có trọng số, bằng cách xem mỗi cạnh của đồ thị vô hướng như hai cạnh có cùng trọng lượng nối cùng một cặp đỉnh nhưng có chiều ngược nhau.

Khi tìm đường đi ngắn nhất có thể bỏ bớt đi các cạnh song song và chỉ chừa lại một cạnh có trọng số nhỏ nhất.

Đối với các khuyên có trọng số không âm thì cũng có thể bỏ đi mà không làm ảnh hưởng đến kết quả của bài toán. Đối với các khuyên có trọng lượng âm thì có thể đưa đến bài toán đường đi ngắn nhất không có lời giải.

#### 2.1.3 Phép biến đổi từ đồ thị sang ma trận trọng số

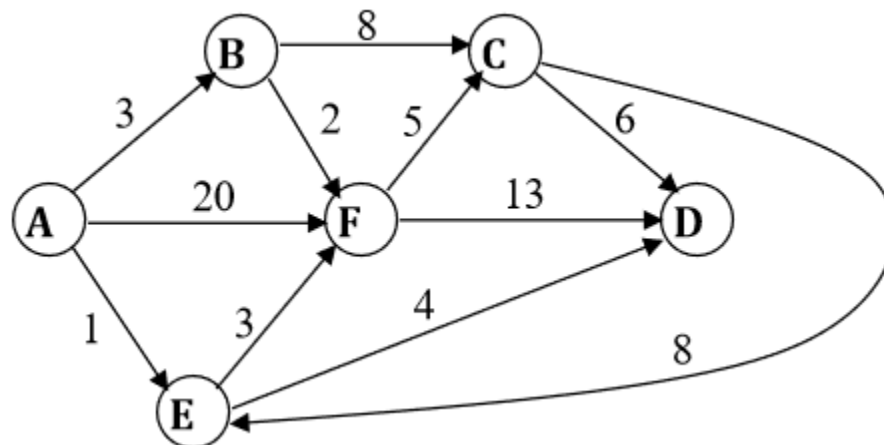
Ma trận trọng số  $Ar_{ij}$  là ma trận trọng số của đồ thị  $G(V,E)$  có các quy tắc sau:

Nếu giữa 2 đỉnh  $(i,j)$  có cạnh nối thì  $Ar_{ij} = w_{ij}$

Nếu giữa 2 đỉnh  $(i,j)$  có nhiều cạnh nối thì  $Ar_{ij} = \min(w_{ij1}, w_{ij2}, \dots)$

Nếu giữa 2 đỉnh  $(i,j)$  không có cạnh nối thì  $Ar_{ij} = \infty$ .

Ví dụ:



Ma trận trọng số:

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
<b>A</b>	$\infty$	3	$\infty$	$\infty$	1	20
<b>B</b>	$\infty$	$\infty$	8	$\infty$	$\infty$	2
<b>C</b>	$\infty$	$\infty$	$\infty$	6	8	$\infty$
<b>D</b>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
<b>E</b>	$\infty$	$\infty$	$\infty$	4	$\infty$	3
<b>F</b>	$\infty$	$\infty$	5	13	$\infty$	$\infty$



## 2.2 Thuật toán Floyd-warshall

**Thuật toán Floyd-Warshall** còn được gọi là thuật toán Floyd được Robert Floyd tìm ra năm 1962. Thuật toán Floyd là một thuật toán giải quyết bài toán đường đi ngắn nhất trong một đồ thị có hướng có cạnh mang trọng số dương dựa trên khái niệm các Đỉnh Trung Gian.

Floyd hoạt động được trên đồ thị có hướng, có thể có trọng số âm, tuy nhiên không có chu trình âm. Chính vì lẽ đó Floyd còn có thể được dùng để phát hiện chu trình âm.

### 2.2.1 Thuật toán

Bản chất của thuật toán là xét mọi bộ 3 các đỉnh  $(k, i, j)$  để tìm đường đi ngắn nhất từ  $i$  sang  $j$ . Cụ thể giả sử  $V = \{v_1, v_2, \dots, v_n\}$  và có ma trận trọng số là  $W = W_0$ . Thuật toán Floyd xây dựng dãy các ma trận vuông cấp  $n$  là  $W_k$  ( $0 \leq k \leq n$ ).

Chứng minh: Ta chứng minh bằng quy nạp theo  $k$  mệnh đề sau:  $W_k[i, j]$  là chiều dài đường đi ngắn nhất trong những đường đi nối đỉnh  $i$  với đỉnh  $j$  đi qua các đỉnh trung gian trong  $\{v_1, v_2, \dots, v_k\}$ .

Trước hết mệnh đề hiển nhiên đúng với  $k=0$ .

Giả sử mệnh đề đúng với  $k-1$ . Xét  $W_k[i, j]$ . Có hai trường hợp:

1) Trong các đường đi chiều dài ngắn nhất nối  $i$  với  $j$  và đi qua các đỉnh trung gian trong  $\{v_1, v_2, \dots, v_k\}$ , có một đường đi  $\gamma$  sao cho  $v_k$  không có trong  $\gamma$ . Khi đó  $\gamma$  cũng là đường đi ngắn nhất nối  $i$  với  $j$  đi qua các đỉnh trung gian trong  $\{v_1, v_2, \dots, v_{k-1}\}$ , nên theo giả thiết quy nạp:

$$W_{k-1}[i, j] = \text{chiều dài } \gamma \leq W_{k-1}[i, k] + W_{k-1}[k, j].$$

Do đó theo định nghĩa của  $W_k$  thì  $W_k[i, j] = W_{k-1}[i, j]$ .

2) Mọi đường đi chiều dài ngắn nhất nối  $i$  với  $j$  và đi qua các đỉnh trung gian trong  $\{v_1, v_2, \dots, v_k\}$ , đều chứa  $v_k$ . Gọi  $\gamma = v_i \dots v_k \dots v_j$  là một đường đi ngắn nhất như thế thì  $v_i \dots v_k$  và  $v_k \dots v_j$  cũng là những đường đi ngắn nhất đi qua các đỉnh trung gian trong  $\{v_1, v_2, \dots, v_{k-1}\}$  và

$$\begin{aligned} W_{k-1}[i, k] + W_{k-1}[k, j] &= \text{chiều dài}(v_i \dots v_k) + \text{chiều dài}(v_k \dots v_j) \\ &= \text{chiều dài } \gamma < W_{k-1}[i, j]. \end{aligned}$$

Do đó theo định nghĩa của  $W_k$  thì ta có:  $W_k[i, j] = W_{k-1}[i, k] + W_{k-1}[k, j]$ .

Các bước thuật toán:

Bước 1: Duyệt qua mỗi  $k \leq n$

Bước 2: Ứng với mỗi  $k$  duyệt từng cặp đỉnh  $(i, j)$

Bước 3: Xét xem đường đi hiện tại từ  $i$  sang  $j$  có lớn hơn từ  $i$  sang  $k$  rồi từ  $k$  sang  $j$  hay không.

Bước 4: Nếu có gán lại đường đi ngắn nhất từ  $i$  sang  $j$  là qua  $k$ .

Bước 5: Tăng  $k$  lên 1 đơn vị, nếu  $k \leq n$  quay về bước 1

Bước 6: kết thúc ta nhận được ma trận  $W^* = W_n$

Minh họa cơ bản thuật toán được xử lý qua hàm:

```
void Floyd ( int vertical )
{
    for(int k=1; k<=vertical; k++)
    {
        for(int i=1; i<=vertical; i++)
        {
            for(int j=1; j<=vertical; j++)
            {
                if(arr[i][j]>arr[i][k]+arr[k][j])
                {
                    arr[i][j]=arr[i][k]+arr[k][j];
                }
            }
        }
    }
}
```

### 2.2.2 Đánh giá độ phức tạp thời gian và không gian

**Độ phức tạp thời gian:** Vì thuật toán sử dụng 3 vòng lặp lồng nhau với mỗi vòng lặp lặp lại  $n$  lần nên độ phức tạp thuật toán này là  $O(n^3)$ .

**Độ phức tạp không gian:** Trong thuật toán này chúng ta cần một mảng 2 chiều lưu trữ thông tin của các đỉnh là lớn nhất, bỏ qua độ phức tạp không gian của các biến vì độ phức tạp không gian của các biến rời rạc đó là  $O(1)$ . Vì vậy độ phức tạp về mặt không gian là  $O(n^2)$

### 2.2.3 Chương trình cài đặt

Dữ liệu của chương trình được đọc từ file cụ thể file có cấu trúc:

Dòng đầu là 4 số vertical egde vertical, verticaly biểu thị số đỉnh đồ thị số cạnh và mục tiêu cần tìm đường đi ngắn nhất từ  $x$  đến  $y$ .

Egde dòng tiếp theo chứa bộ ba numberx, numbery và values biểu thị có đường nối giữa 2 đỉnh  $x$  và  $y$  và có trọng số là values.

Từ bộ dữ liệu mẫu như vậy ta có thể xây dựng ma trận  $W_0$ .

Chương trình cụ thể được viết trên c++:

```
#include< iostream>
#include< fstream >
using namespace std;
long long** arr;
```

```
void init_memory(int vertical)

{
    arr=new long long*[vertical+1];
    for(int i=1; i<=vertical; i++)
    {
        arr[i]=new long long[vertical+1];
    }
}

void init_valuesarr(int vertical)
{
    for(int i=1; i<=vertical; i++)
    {
        for(int j=1; j<= vertical; j++)arr[i][j]=10e17;
    }
}

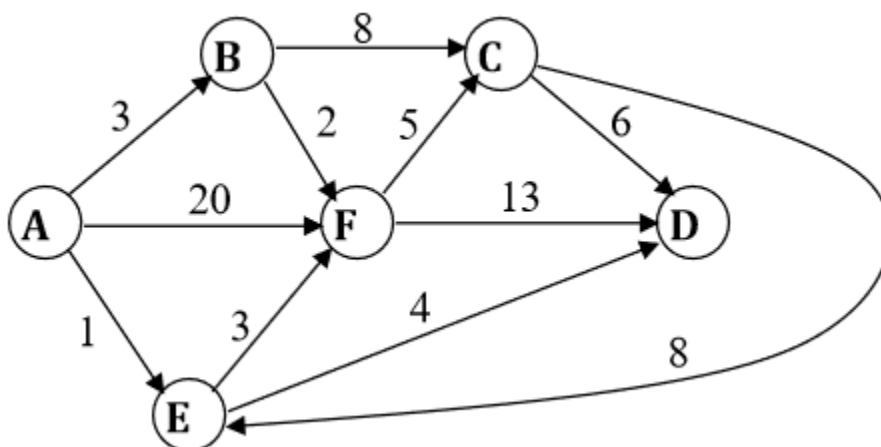
void print_arr(int vertical)
{
    for(int i=1; i<=vertical; i++)
    {
        for(int j=1; j<= vertical; j++)
        {
            if(arr[i][j]==10e17)cout<<"oo\t";
            else cout<<arr[i][j]<<"\t";
        }
        cout<<endl;
    }
    cout<<"-----"<<endl;
}

void input_file(int &vertical, int &verticalx, int &verticaly)
{
    Ifstream fi("test.inp");
    int edge;
    fi>>vertical>>edge>>verticalx>>verticaly;
    int numberx, numbery;
    long long values;
```

```
init_memory(vertical);
init_valuesarr(vertical);
for(int i=1; i<=edge; i++)
{
    fi>>numberx>>numbery>>values;
    arr[numberx][numbery]=min(arr[numberx][numbery],values);
}
}
void floyd(int vertical)
{
    for(int k=1; k<=vertical; k++)
    {
        for(int i=1; i<=vertical; i++)
        {
            for(int j=1; j<=vertical; j++)
            {
                if(arr[i][j]>arr[i][k]+arr[k][j])
                {
                    arr[i][j]=arr[i][k]+arr[k][j];
                }
            }
        }
    }
}
main()
{
    int vertical, edge, verticalx, verticaly;
    input_file(vertical, verticalx, verticaly);
    print_arr(vertical);
    floyd(vertical);

    print_arr(vertical);
    if(arr[verticalx][verticaly]==10e17)cout<<"khong ton tai duong di tu dinh
    "<<verticalx<<" sang dinh "<<verticaly;
    else cout<<"duong di ngan nhat tu dinh "<<verticalx<<" sang dinh
    "<<verticaly<<" la: "<<arr[verticalx][verticaly];
}
```

### 2.2.4 Minh họa thuật toán với bộ dữ liệu mẫu:



$W_0$ :

	A	B	C	D	E	F
A	$\infty$	3	$\infty$	$\infty$	1	20
B	$\infty$	$\infty$	8	$\infty$	$\infty$	2
C	$\infty$	$\infty$	$\infty$	6	8	$\infty$
D	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
E	$\infty$	$\infty$	$\infty$	4	$\infty$	3
F	$\infty$	$\infty$	5	13	$\infty$	$\infty$

$W_1$ :

	A	B	C	D	E	F
A	$\infty$	3	$\infty$	$\infty$	1	20
B	$\infty$	$\infty$	8	$\infty$	$\infty$	2
C	$\infty$	$\infty$	$\infty$	6	8	$\infty$
D	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
E	$\infty$	$\infty$	$\infty$	4	$\infty$	3
F	$\infty$	$\infty$	5	13	$\infty$	$\infty$

$W_2$ :

	A	B	C	D	E	F
A	$\infty$	3	11	$\infty$	1	5
B	$\infty$	$\infty$	8	$\infty$	$\infty$	2
C	$\infty$	$\infty$	$\infty$	6	8	$\infty$
D	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
E	$\infty$	$\infty$	$\infty$	4	$\infty$	3

<b>F</b>	$\infty$	$\infty$	5	13	$\infty$	$\infty$
----------	----------	----------	---	----	----------	----------

W<sub>3</sub>:

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
<b>A</b>	$\infty$	3	11	17	1	5
<b>B</b>	$\infty$	$\infty$	8	14	16	2
<b>C</b>	$\infty$	$\infty$	$\infty$	6	8	$\infty$
<b>D</b>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
<b>E</b>	$\infty$	$\infty$	$\infty$	4	$\infty$	3
<b>F</b>	$\infty$	$\infty$	5	11	13	$\infty$

W<sub>4</sub>:

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
<b>A</b>	$\infty$	3	11	17	1	5
<b>B</b>	$\infty$	$\infty$	8	14	16	2
<b>C</b>	$\infty$	$\infty$	$\infty$	6	8	$\infty$
<b>D</b>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
<b>E</b>	$\infty$	$\infty$	$\infty$	4	$\infty$	3
<b>F</b>	$\infty$	$\infty$	5	11	13	$\infty$

W<sub>5</sub>:

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
<b>A</b>	$\infty$	3	11	5	1	4
<b>B</b>	$\infty$	$\infty$	8	14	16	2
<b>C</b>	$\infty$	$\infty$	$\infty$	6	8	11
<b>D</b>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
<b>E</b>	$\infty$	$\infty$	$\infty$	4	$\infty$	3
<b>F</b>	$\infty$	$\infty$	5	11	13	16

W<sub>6</sub>:

	<b>A</b>	<b>B</b>	<b>C</b>	<b>D</b>	<b>E</b>	<b>F</b>
<b>A</b>	$\infty$	3	9	5	1	4
<b>B</b>	$\infty$	$\infty$	7	13	15	2
<b>C</b>	$\infty$	$\infty$	16	6	8	11
<b>D</b>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
<b>E</b>	$\infty$	$\infty$	8	4	16	3
<b>F</b>	$\infty$	$\infty$	5	11	13	16

## 2.3 Thuật toán Dijkstra và cải tiến thuật toán Dijkstra:

**Thuật toán Dijkstra**, mang tên của nhà khoa học máy tính người Hà Lan Edsger Dijkstra vào năm 1956 và ấn bản năm 1959<sup>[1]</sup>, là một thuật toán giải quyết bài toán đường đi ngắn nhất nguồn đơn trong một đồ thị có hướng không có cạnh mang trọng số âm. Thuật toán thường được sử dụng trong định tuyến với một chương trình con trong các thuật toán đồ thị hay trong công nghệ Hệ thống định vị toàn cầu(GPS).

Dijkstra cải tiến là một biến thể của thuật toán Dijkstra trong đó thuật toán này tối ưu hóa việc tìm kiếm đỉnh có đường đi nhỏ nhất. Nhưng để giảm chi phí về mặt thời gian tính toán thì buộc chúng ta phải trả thêm chi phí về mặt không gian nhớ. Nên tùy thuộc vào điều kiện của bài toán mà ta lựa chọn giải thuật Dijkstra cải tiến hay Dijkstra cho phù hợp.

### 2.3.1 Thuật toán

Ý tưởng của thuật toán Dijkstra: xác định tuần tự đỉnh có khoảng cách đến  $u_0$  từ nhỏ đến lớn.

- Khoảng cách  $d(u_0, u_0) = 0$ .

- Trong các đỉnh  $v$  khác  $u_0$ , tìm đỉnh có khoảng cách  $k_1$  đến  $u_0$  là nhỏ nhất.

Đỉnh này phải là một trong các đỉnh kề với  $u_0$ . Giả sử đó là  $u_1$ . Ta có:  $d(u_0, u_1) = k_1$ .

- Trong các đỉnh  $v$  khác  $u_0$  và  $v$  khác  $u_1$ , tìm đỉnh có khoảng cách  $k_2$  đến  $u_0$  là nhỏ nhất.

Đỉnh này phải là một trong các đỉnh kề với  $u_0$  hoặc với  $u_1$ . Giả sử đó là  $u_2$ . Ta có:  $d(u_0, u_2) = k_2$ .

Lặp lại quá trình trên cho đến khi tìm được khoảng cách từ  $u_0$  đến mọi đỉnh  $v$  của  $G$ . Nếu  $V = \{u_0, u_1, \dots, u_n\}$  thì:  $0 = d(u_0, u_0) < d(u_0, u_1) < d(u_0, u_2) < \dots < d(u_0, u_n)$ .

Cụ thể chúng ta đi theo từng bước sau:

Bước 1: xác định các đỉnh liền kề với đỉnh đang xét, đánh dấu đỉnh đang xét là đã xét.

Bước 2: Cập nhật tất cả các giá trị với các đỉnh liền kề ấy nếu đường đi hiện tại của đỉnh liền kề lớn hơn đường đi đến đi đang xét cộng với trọng số của cạnh nối giữa đỉnh đang xét và đỉnh liền kề.

Bước 3: Tìm trên tập đỉnh chưa xét có đường đi nhỏ nhất. Nếu tồn tại thì lấy đỉnh được chọn thành đỉnh xét và quay về bước 1. Nếu tập đỉnh chưa xét là rỗng thì qua bước 4.

Bước 4: kết thúc.

Đoạn minh code minh họa ý tưởng chính:

```
void Dijkstra(int vertical, int verticalx)
{
    map<int, long long>::iterator
    it; long long values=0;
    bool_arr[verticalx]=1;
```

```

while(true)
{
    for(it=arr[verticalx].begin(); it!=arr[verticalx].end(); it++)
    {
        if(!bool_arr[it->first])
        {
            if(values+it->second<values_arr[it->first])
            {
                values_arr[it->first]=values+it->second;
            }
        }
    }
    long long temp_vertical=verticalx,temp_values=10e17;
    for(int i=1; i<=vertical; i++)
    {
        if(!bool_arr[i] && temp_values>values_arr[i])
        {
            temp_vertical=i;
            temp_values=values_arr[i];
        }
    }
    if(temp_vertical==verticalx)return;
    else
    {
        verticalx=temp_vertical;
        values=temp_values;
        bool_arr[verticalx]=1;
    }
}
}

```

### 2.3.2 Đánh giá độ phức tạp thời gian và không gian

**Độ phức tạp về thời gian:** Với mức độ bình thường thì thuật toán trên phải duyệt qua hết  $m$  cạnh của đồ thị và chạy vòng lặp  $n$  đỉnh lồng với vòng lặp duyệt qua  $n$  đỉnh tìm kiếm đỉnh nhỏ nhất, bỏ qua các độ phức tạp  $O(1)$  thì độ phức tạp của thuật toán này là  $O(n^2 + m)$ .

**Độ phức tạp về mặt không gian:** Việc lưu trữ các đỉnh và giá trị đường đi ngắn



nhất của các đỉnh là  $O(n)$  nhưng ta cần lưu  $m$  cạnh vào tập dữ liệu lần này các cạnh được tách lưu chứ không thông qua ma trận trọng số nữa nên độ phức tạp để lưu  $m$  cạnh là  $O(m)$ . Vậy độ phức tạp về mặt không gian của thuật toán này là  $O(m+n)$ . Trong trường hợp xấu cây có thể có tới số cạnh  $m=n(n-1)$  cạnh nên độ phức tạp về mặt không gian trong trường hợp xấu là  $O(n^2-n+n)=O(n^2)$ .

### 2.3.3 Cài đặt chương trình

Chương trình minh họa được viết bằng ngôn ngữ C++ sử dụng thêm cấu trúc dữ liệu cây là map để tối ưu hóa bộ nhớ.

```
#include<iostream>
#include<vector>
#include<map>
#include<fstream>
using namespace std;
map<int,long long>* arr;
long long* values_arr;
bool* bool_arr;

void init_arr(int vertical)
{
    arr=new map<int,long long> [vertical+1];
    values_arr=new long long [vertical+1];
    bool_arr=new bool [vertical+1];
    for(int i=1; i<=vertical; i++)
    {
        bool_arr[i]=0;
        values_arr[i]=10e17;
    }
}

void input_file(int &vertical,int &verticalx,int &verticaly)
{
    ifstream fi("test.inp");
    int edge;
    fi>>vertical>>edge>>verticalx>>verticaly;
    int numberx, numbery;
    long long values;
    init_arr(vertical);
    for(int i=1; i<=edge; i++)
```

```

        {
            fi>>numberx>>numbery>>values;
            if(arr[numberx][numbery]!=0)arr[numberx][numbery]=min(arr[num
            berx][numbery],values);
            else arr[numberx][numbery]=values;
        }
    }
void printf_arr(int vertical)
{
    for(int i=1; i<=vertical; i++)
    {
        if(values_arr[i]==10e17)cout<<"oo\t";
        else cout<<values_arr[i]<<"\t";
    }
    cout<<endl<<"-----"<<endl;
}
void Dijkstra(int vertical,int verticalx)
{
    map<int,long long>::iterator it;
    long long values=0;
    bool_arr[verticalx]=1;
    while(true)
    {
        for(it=arr[verticalx].begin(); it!=arr[verticalx].end(); it++)
        {
            if(!bool_arr[it->first])
            {
                if(values+it->second<values_arr[it->first])
                {
                    values_arr[it->first]=values+it->second;
                }
            }
        }
        long long temp_vertical=verticalx, temp_values=10e17;
        for(int i=1; i<=vertical; i++)
        {

```

```

        if(!bool_arr[i] && temp_values>values_arr[i])
        {
            temp_vertical=i;
            temp_values=values_arr[i];
        }
    }
    if(temp_vertical==verticalx)return;
    else
    {
        verticalx=temp_vertical;
        values=temp_values;
        bool_arr[verticalx]=1;
    }
}
}
main()
{
    int vertical,edge,verticalx,verticaly;
    input_file(vertical, verticalx,verticaly);
    Dijkstra(vertical,verticalx);
    if(values_arr[verticaly]==10e17)cout<<"khong ton tai duong di tu dinh
    "<<verticalx<<" sang dinh "<<verticaly;
    else cout<<"duong di ngan nhat tu dinh "<<verticalx<<" sang dinh
    "<<verticaly<<" la: "<<values_arr[verticaly];

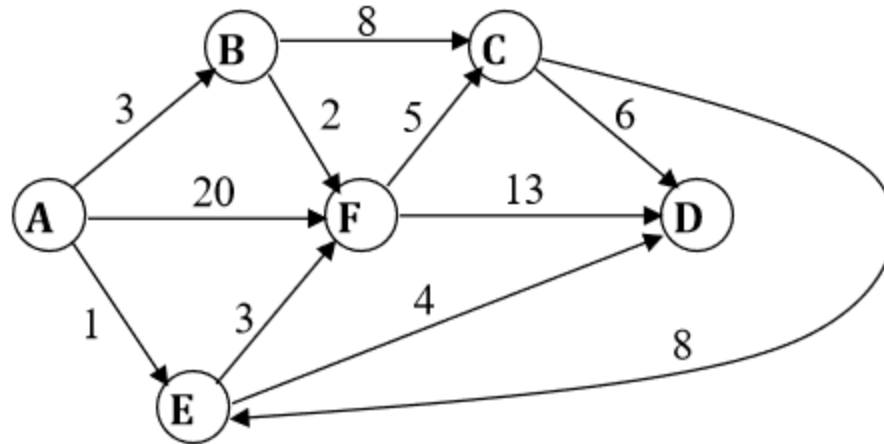
}

```

Hiện nhiên nếu muốn tăng tốc độ có thể tìm thấy kết quả rồi thoát ngay không cần duyệt hết tất cả các đỉnh. Chỉ cần thêm 1 lệnh break khi verticalx=verticaly.

#### **2.3.4 Minh họa với dữ liệu mẫu**

Ứng với bộ dữ liệu mẫu sau:



	A	B	C	D	E	F
<b>Ban đầu</b>	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
<b>Lần lặp 1</b>	$\infty$	3	$\infty$	$\infty$	1	20
<b>Lần lặp 2</b>	$\infty$	3	$\infty$	5	1	4
<b>Lần lặp 3</b>	$\infty$	3	11	5	1	4
<b>Lần lặp 4</b>	$\infty$	3	9	5	1	4
<b>Lần lặp 5</b>	$\infty$	3	9	5	1	4
<b>Lần lặp 6</b>	$\infty$	3	9	5	1	4

### 2.3.5 Ý tưởng cải tiến

Việc tính toán điểm nhỏ nhất ở thuật toán hiện tại là  $O(n)$  vậy ta có thể sài một số cấu trúc dữ liệu để có thể tính toán được điểm nhỏ nhất trong  $O(1)$  nhưng chi phí bỏ ra để sắp xếp chèn là  $O(\log n) < O(n)$ . Vậy ta cải tiến bước tính toán điểm nhỏ nhất.

Ở trong lần cải tiến lần này sẽ dùng cấu trúc dữ liệu với multiset. Việc xóa một phần tử, chèn một phần tử trong set theo iterator là  $\log n$  việc này cho phép chúng ta cải tiến được tìm phần tử nhỏ nhất vì nó luôn nằm ở đầu 1 lớp multiset. Vậy sau khi cải tiến thuật toán còn độ phức tạp là  $O((n+m)\log n)$  Cụ thể ta cải tiến lại hàm Dijkstra như sau:

```

void Dijkstra(int vertical,int verticalx)
{
    map<int,long long>::iterator it;
    set<pair<long long,int> >::iterator iit;
    long long values=0;
    bool_arr[verticalx]=1;
    while(true)
    {
        for(it=arr[verticalx].begin(); it!=arr[verticalx].end(); it++)
        {

```

```
        if(!bool_arr[it->first])
        {
            if(values+it->second<values_arr[it->first])
            {
                iit=multiset_values.find(make_pair(values_arr[it->first],it->first));
                multiset_values.erase(iit);
                values_arr[it->first]=values+it->second;
                multiset_values.insert(make_pair(values_arr[it->first],it->first));
            }
        }
    }
    iit=multiset_values.begin();
    if(iit==multiset_values.end())return;
    else
    {
        verticalx=iit->second;
        values=iit->first;
        bool_arr[verticalx]=1;
        multiset_values.erase(iit);
    }

}
}
```

### **2.3.6 Chương trình cải tiến**

```
#include<iostream>
#include<vector>
#include<map>
#include<set>
#include<fstream>

using namespace std;
map<int,long long>* arr;
long long* values_arr;
multiset<pair<long long,int> >multiset_values;
bool* bool_arr;
```

```
void init_arr(int vertical)
{
    arr=new map<int,long long> [vertical+1];
    values_arr=new long long [vertical+1];
    bool_arr=new bool [vertical+1];
    for(int i=1; i<=vertical; i++)
    {
        bool_arr[i]=0;
        values_arr[i]=10e17;
    }
}

void input_file(int &vertical,int &verticalx,int &verticaly)
{
    ifstream fi("test.inp");
    int edge;
    fi>>vertical>>edge>>verticalx>>verticaly;
    int numberx, numbery;
    long long values;
    init_arr(vertical);
    for(int i=1; i<=vertical; i++)
    {
        if(i!=verticalx)multiset_values.insert(make_pair<long long, int>(10e17,i));
    }
    for(int i=1; i<=edge; i++)
    {
        fi>>numberx>>numbery>>values;

        if(arr[numberx][numbery]!=0)arr[numberx][numbery]=min(arr[numberx][
numbery],values);
        else arr[numberx][numbery]=values;
    }
}

void printf_arr(int vertical)
{
    for(int i=1; i<=vertical; i++)
    {
```

```
        if(values_arr[i]==10e17)cout<<"oo\t";
        else cout<<values_arr[i]<<"\t";
    }
    cout<<endl<<"-----"<<endl;
}
void Dijkstra(int vertical,int verticalx)
{
    map<int,long long>::iterator it;
    set<pair<long long,int> >::iterator iit;
    long long values=0;
    bool_arr[verticalx]=1;
    while(true)
    {
        for(it=arr[verticalx].begin(); it!=arr[verticalx].end(); it++)
        {
            if(!bool_arr[it->first])
            {
                if(values+it->second<values_arr[it->first])
                {
                    iit=multiset_values.find(make_pair(values_arr[it->first],it->first));
                    multiset_values.erase(iit);
                    values_arr[it->first]=values+it->second;
                    multiset_values.insert(make_pair(values_arr[it->first],it->first));
                }
            }
        }
        iit=multiset_values.begin();
        if(iit==multiset_values.end())return;
        else
        {
            verticalx=iit->second;
            values=iit->first;
            bool_arr[verticalx]=1;
            multiset_values.erase(iit);
        }
    }
}
```

```
    }  
}  
main()  
{  
  
    int vertical, edge, verticalx, verticaly;  
    input_file(vertical, verticalx, verticaly);  
    Dijkstra(vertical, verticalx);  
    if(values_arr[verticaly]==10e17)cout<<"khong ton tai duong di tu dinh"  
    "<<verticalx<<" sang dinh "<<verticaly;  
    else cout<<"duong di ngan nhat tu dinh "<<verticalx<<" sang dinh"  
    "<<verticaly<<" la: "<<values_arr[verticaly];  
  
}
```



## 2.4 Thuật toán Bellman-Ford và cải tiến

**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.

### 2.4.1 Thuật toán

Ý tưởng thuật toán Bellman-Ford có tính tham lam:

Ta thực hiện duyệt  $V$  lần, với  $V$  là số đỉnh của đồ thị.

Với mỗi lần duyệt, ta tìm tất cả các cạnh mà đường đi qua cạnh đó sẽ rút ngắn đường đi ngắn nhất từ đỉnh gốc tới một đỉnh khác.

Ở lần duyệt thứ  $V$ , nếu còn bất kỳ cạnh nào có thể rút ngắn đường đi, điều đó chứng tỏ đồ thị có chu trình âm, và ta kết thúc thuật toán.

Tính đúng đắn của ý tưởng này được chứng minh quy nạp.

**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( $u,v$ ) + 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 trọng số( $u,v$ ) + khoảng cách( $v$ ) 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).

Cụ thể ta làm theo các bước sau:

Bước 1: Gán giá trị của các đỉnh bằng vô cùng riêng đỉnh được chọn gán  $=0$ .

Bước 2: Duyệt qua tất cả các cạnh của đồ thị xét xem nếu cặp cạnh  $(u,v)$  làm thay đổi đường đi ngắn nhất tới  $v$  thì cập nhật tại  $v$ .

Bước 3: Duyệt lại vòng lặp bước 2 với  $V-1$  lần. lần thứ  $V$  dùng để kiểm tra có chu trình âm hay không. Có thể thoát vòng duyệt sớm hơn  $V$  lần nếu không có cạnh nào trong tập cạnh làm thay đổi đường đi ngắn nhất. Nếu đã duyệt đủ  $V$  lần hoặc thoát khỏi vòng lặp sớm qua bước 4.

Bước 4: Kết thúc giá trị cần tìm nằm các ô.

Cụ thể phần code Bellman-Flord cải tiến:

```
void Bellman_Flrod(int vertical)
{
    map<pair<int,int>, long long>::iterator it;
    for(int i=1; i<vertical; i++)
    {
        for(it=arr.begin(); it!=arr.end(); it++)
        {
            if(values_arr[it->first.first]+it->second<values_arr[it->first.second])
            {
                values_arr[it->first.second]=values_arr[it->first.first]+it->second;
            }
        }
    }
    for(it=arr.begin(); it!=arr.end(); it++)
    {
        if(values_arr[it->first.first]+it->second<values_arr[it->first.second])
        {
```

```
        values_arr[it->first.second]=values_arr[it->first.first]+it->second;
        bool_arr[it->first.second]=1;
    }
}
}
```

#### 2.4.2 Đánh giá độ phức tạp thời gian và không gian

**Độ phức tạp về mặt thời gian:** Vì chương trình lặp 2 vòng lặp lồng nhau với mỗi vòng lặp vòng lặp V lần 1 vòng lặp E lần với V và E là số đỉnh và số cạnh của đồ thị. Vì vậy độ phức tạp thời gian là  $O(V \cdot E)$ .

**Độ phức tạp về mặt không gian:** Cần nhớ để nhớ danh sách đường đi với các đỉnh và M cặp bộ 3 giá trị để nhớ E cạnh của đồ thị. Vì vậy độ phức tạp về mặt không gian là  $O(V+E)$  với V là số đỉnh của đồ thị và E là số bộ 3 của đồ thị.

Nhưng sau khi cải tiến thuật toán thì dùng cấu trúc dữ liệu map làm cho các cặp đỉnh nhỏ được đưa xuống dưới thì thuật toán này có độ phức tạp thời gian không đến  $O(V \cdot E)$  vì số lượng V lặp lại rất nhỏ nên thời gian chạy sẽ có khác biệt so với thuật toán từ ban đầu. Trong trường hợp đặc biệt nếu đồ thị tồn tại chu trình âm thì độ phức tạp quay về  $O(V \cdot E)$ .

#### 2.4.3 Cài đặt chương trình

Chương trình cài đặt được viết trên c++:

```
#include<iostream>
#include<vector>
#include<map>
#include<set>
#include<fstream>
using namespace std;
map<pair<int,int>,long long> arr;
long long* values_arr;
bool* bool_arr;
void init_arr(int vertical)
{
    values_arr=new long long [vertical+1];
    bool_arr=new bool[vertical+1];
    for(int i=1; i<=vertical; i++)
    {
        values_arr[i]=10e17;
        bool_arr[i]=0;
    }
}
```

```

    }
}
void input_file(int &vertical,int &verticalx,int &verticaly)
{
    ifstream
    fi("test.inp");
    int edge;
    fi>>vertical>>edge>>verticalx>>verticaly;
    int numberx,numbery;
    long long values;
    init_arr(vertical);
    values_arr[verticalx]=0;
    for(int i=1; i<=edge; i++)
    {
        fi>>numberx>>numbery>>values;

        if(arr[make_pair(numberx,numbery)]!=0)
            arr[make_pair(numberx,number)]=
            min(arr[make_pair(numberx,numbery)],values);
        else arr[make_pair(numberx,numbery)]=values;
    }

}
void printf_arr(int vertical)
{
    for(int i=1; i<=vertical; i++)
    {
        if(values_arr[i]==10e17)cout<<"oo\t";
        else if(bool_arr[i])cout<<"chu trình
        âm\t"; else cout<<values_arr[i]<<"\t";
    }
    cout<<endl<<"-----"<<endl;
}
void Bellman_Flrod(int vertical)
{
    map<pair<int,int>, long long>::iterator it;
    bool check=1;

```

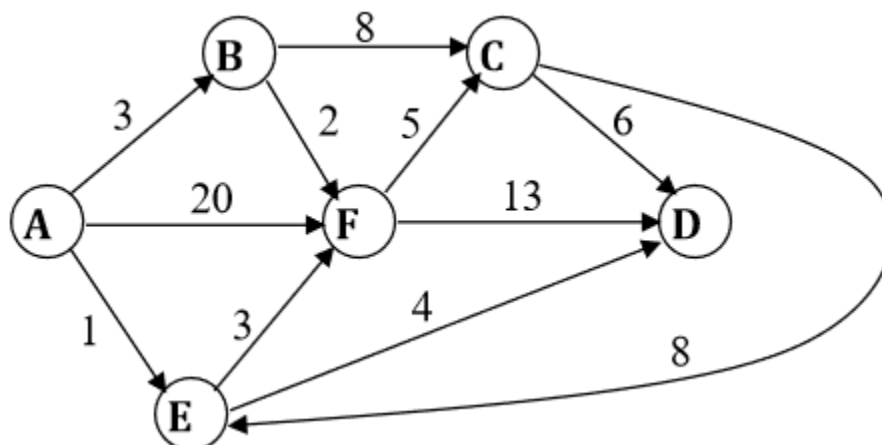
```
for(int i=1; i<vertical; i++)
{
    check=1;
    for(it=arr.begin();it!=arr.end();it++)
    {
        if(values_arr[it->first.first]+it->second<values_arr[it->first.second])
        {
            values_arr[it->first.second]=values_arr[it->first.first]+it->second;
            check=0;
        }
    }
    if(check)return;
}
for(it=arr.begin(); it!=arr.end(); it++)
{
    if(values_arr[it->first.first]+it->second<values_arr[it->first.second])
    {
        values_arr[it->first.second]=values_arr[it->first.first]+it->second;
        bool_arr[it->first.second]=1;
    }
}
}
main()
{

    int vertical,edge,verticalx,verticaly;
    input_file(vertical, verticalx,verticaly);
    Bellman_Flrod(vertical);
    printf_arr(vertical);
    if(values_arr[vertical]==10e17)cout<<"khong ton tai duong di tu dinh "<<verticalx<<" sang dinh "<<verticaly;
    else if(bool_arr[vertical])cout<<"có chu trình âm\t";
    else cout<<"duong di ngan nhât tu dinh "<<verticalx<<" sang dinh "<<verticaly<<" la: "<<values_arr[vertical];
```

}

#### 2.4.4 Minh họa với dữ liệu mẫu

Ứng với bộ dữ liệu mẫu:



Giả sử cần bắt đầu từ B:

	A	B	C	D	E	F
<b>Ban đầu</b>	$\infty$	0	$\infty$	$\infty$	$\infty$	$\infty$
<b>Lần lặp 1</b>	$\infty$	0	7	14	16	2
<b>Lần lặp 2</b>	$\infty$	0	7	13	15	2
<b>Lần lặp 3</b>	$\infty$	0	7	13	15	2

Vì lần lặp 2 và 3 hoàn toàn giống nhau nên vòng lặp được kết thúc trả về bảng kết quả từ B sang tất cả các đỉnh còn lại.

## CHƯƠNG 3: KẾT QUẢ VÀ ĐÁNH GIÁ

### 3.1 Chương trình sinh test

Để đánh giá và so sánh kết quả của các thuật toán ta dựa vào các bộ test mẫu với các dữ liệu được sinh hoàn toàn ngẫu nhiên. Bộ test mẫu gồm:

Dòng đầu là 4 số vertical edge vertical, verticaly biểu thị số đỉnh đồ thị số cạnh và mục tiêu cần tìm đường đi ngắn nhất từ x đến y.

Edge dòng tiếp theo chứa bộ ba numberx, numbery và values biểu thị có đường nối giữa 2 đỉnh u và v và có trọng số là values.

Để đáp ứng các bộ test mẫu ta viết một chương trình sinh test mẫu như sau:

```
#include<iostream>
#include<fstream>
#include <cstdlib>
#include <ctime>
using namespace
std; main()
{
    srand(time(NULL));
    ofstream fo("test2.inp");
    int vertical=10;
    int edge=10;
    int mod=vertical;
    int verticalx=rand()%mod+1;
    int verticaly=verticalx;
    while(verticalx==verticaly)
    {
        verticaly=rand()%mod+1;
    }
    fo<<vertical<<" "<<edge<<" "<<verticalx<<"
"<<verticaly<<endl;
    int numberx, numbery, values;
    for(int i=1; i<=edge; i++)
    {
        numberx=rand()%mod+1;
        numbery=numberx;
        while(numberx==numbery)
        {
```

```

        numbery=rand()%mod+1;
    }
    values=rand()%mod+1;
    fo<<numberx<<" "<<numbery<<" "<<values<<endl;
}
}

```

### 3.2 Kết quả, đánh giá và ứng dụng

**Bảng kết quả thời gian chạy** của các thuật toán kết quả tính bằng mili giây chạy trên máy tính I7-7700HQ Ram 16gb:

Bộ test(V là số đỉnh, E là số cạnh)	Floyd	Dijkstra	Dijkstra cải tiến	Bellman- Flord cải tiến
V=10,E=10	2ms	5ms	1ms	17ms
V=100,E=50	4ms	3ms	2ms	20ms
V=100,E=500	5ms	7ms	4ms	14ms
V=1000,E=5000	2530ms	14ms	9ms	16ms
V=1000,E=50000	2578ms	64ms	89ms	156ms
V=10000,E=50000	2512657ms	399ms	92ms	186ms
V=10000,E=500000	2676657ms	1065ms	977ms	1978ms
V=100000,E=500000	Quá lớn để cấp phát bộ nhớ lên đến vài Gb	6934ms	988ms	2059ms

Qua kết quả bảng so sánh thời gian chạy của các thuật toán ta có thể thấy được để tìm đường đi ngắn nhất từ 1 điểm đến một điểm thì thuật toán Dijkstra cải tiến có khả năng tìm kiếm được nhanh nhất về mặt thời gian.

Qua các kết quả ta thấy với dữ liệu lớn dựa vào độ phức tạp về mặt thời gian ta có:  
 $Floyd = V^3 = 10^{15}$  nên thời gian chạy khá lớn.

$Dijkstra = V^2 + E = 10^{10}$  thời gian chạy nằm ở tầm hơn 7s kết quả chạy.

$Dijkstra \text{ cải tiến} = (V+E) \log n$  = sấp xỉ  $2 * 10^8$  cũng chính vì thế thời gian chạy thực là thấp nhất. Sau khi cải tiến từ Dijkstra thì thời gian chạy của Dijkstra cải tiến nhanh hơn gấp 7 lần đối với dữ liệu lớn so với Dijkstra chưa cải tiến.

$Bellman-Flord \text{ cải tiến} = V * E = 5 * 10^{10}$  lớn hơn Dijkstra thông thường theo nguyên lý thì thời gian chạy phải gấp 5 lần Dijkstra nhưng Bellman-Flord đã được cải tiến làm số V giảm nhiều lần so với thực tế làm kết quả thời gian chạy thực của Bellman-Flord thấp hơn 2.5 lần so với Dijkstra bình thường.

**Bảng kết quả cấp phát bộ nhớ** của các thuật toán kết quả tính bằng Mb chạy trên máy tính I7-7700HQ Ram 16gb:



Bộ test(V là số đỉnh, E là số cạnh)	Floyd	Dijkstra	Dijkstra cải tiến	Bellman- Ford cải tiến
V=10,E=10	3Mb	3Mb	3Mb	3Mb
V=100,E=50	3Mb	3Mb	3Mb	3Mb
V=100,E=500	3Mb	3Mb	3Mb	3Mb
V=1000,E=5000	11Mb	3Mb	3Mb	3Mb
V=1000,E=50000	11Mb	6Mb	6Mb	6Mb
V=10000,E=50000	766Mb	7Mb	7Mb	6Mb
V=10000,E=500000	766Mb	34Mb	35Mb	33Mb
V=100000,E=500000	Quá lớn để cấp phát bộ nhớ lên đến vài Gb	39Mb	45Mb	33Mb

Thuật toán Floyd cấp phát bộ nhớ theo  $O(V^2)=10^{10}$  dung lượng bộ nhớ này lên đến vài GB nên không thể cấp phát bộ nhớ cho thuật toán này. Đây cũng chính là hạn chế của thuật toán này khi sử dụng áp dụng với các đồ thị có số lượng đỉnh là quá lớn.

Thuật toán Dijkstra được cấp bộ nhớ theo  $O(V+E)=6*10^5$  bộ nhớ còn khá nhỏ để có thể cấp và thực hiện thuật toán. Chính vì vậy thuật toán Dijkstra thường được áp dụng đối với các đồ thị có trọng số dương, đặc biệt là ứng dụng vào thực tế trong GPS và Google map.

Thuật toán Dijkstra cải tiến bộ nhớ lớn hơn thuật toán Dijkstra ban đầu bởi vì tốn thêm chi phí lưu trữ để tăng tốc độ tính toán như nói từ ban đầu. Sự chênh lệch về chi phí bộ nhớ của Dijkstra cải tiến so với Dijkstra thường là không nhiều cũng chính vì vậy khi sử dụng nên cải tiến Dijkstra để tăng tốc độ tính toán thấy rõ.

Thuật toán Bellman-Ford có độ phức tạp về không gian là  $O(E+V)=6*10^5$  bằng với thuật toán Dijkstra nhưng số lượng cấp phát bộ nhớ cho các biến bị lượt bỏ trong quá trình tính độ phức tạp theo quy tắc tính độ phức tạp nên vì vậy bộ nhớ có ít hơn Dijkstra.

### Ứng dụng vào bài toán thực tế

Một khách du lịch đang muốn đi mua sắm quà tặng tại một thành phố, người du lịch này là một người cực kỳ khó tính chỉ muốn đi mua sắm tại trung tâm mua sắm nằm ở chốt giao thông thứ j. Thành phố hiện tại có rất nhiều chốt giao thông khác nhau. Khách du lịch này tính toán được thời gian di chuyển giữa các chốt giao thông hiện có đường đi trực tiếp trên bản đồ nhưng không thể tính được thời gian di chuyển giữa 2 chốt giao thông bất kỳ. Hiện tại khách du lịch này ở chốt giao thông thứ i và muốn biết

rằng liệu sau thời gian ngắn nhất là bao nhiêu khách du lịch ấy sẽ đến được trung tâm mua sắm nằm ở chốt giao thông thứ  $j$ .

**Đầu vào:**

Dòng đầu là bộ 4 số: vertical, edge, vertical, verticaly với

Vertical: Số lượng nút giao thông hiện có,  $0 < \text{vertical} \leq 10^5$ .

Edge: Số lượng đường đi từ nút giao thông này sang nút giao thông kia, đặc biệt các đường này đều là đường một chiều,  $0 < \text{edge} \leq 5 * 10^5$ .

Verticalx: Nút giao thông thứ  $i$  mà khách du lịch đang đứng,  $0 < \text{verticalx} \leq \text{vertical}$ .

Verticaly: Nút giao thông  $j$  mà khách du lịch muốn tới,  $0 < \text{verticaly} \leq \text{vertical}$ ,  $\text{verticalx} \neq \text{verticaly}$ .

Edge dòng tiếp theo mỗi dòng chứa bộ 3 số numberx, numbery và values với numberx, numbery thể hiện có đường đi ở nút giao thông thứ numberx, numbery với số phút để di chuyển là values biết rằng số phút di chuyển là một số nguyên dương nằm ở đoạn  $[1, 10^5]$ .

**Đầu ra:**

Một số duy nhất là thời gian ngắn nhất để khách du lịch đó đi đến trung tâm mua sắm. Nếu không tồn tại đường đi hãy trả lời “No answer” .

**Nhận xét:**

Bài toán này chúng ta hoàn toàn có thể quy về đồ thị để giải với các nút giao thông là các đỉnh của đồ thị, số phút là trọng số cạnh nối giữa 2 đồ thị bất kì, số phút ít nhất để khách du lịch di chuyển chính là đường đi ngắn nhất giữa 2 đỉnh bất kỳ trên đồ thị.

Chính vì vậy chúng ta sẽ giải bài này thông qua 4 thuật toán đã trình bày ở trên.


**Kết quả chạy thật của bài toán**

Thuật toán Floyd:

Partial Accepted						
Time: 20752ms Memory: 766MB Lang: C++ Author: admin						
ID	Status	Memory	Time	Score	Real Time	Singal
1	Accepted	3MB	0ms	10	2ms	0
2	Accepted	3MB	3ms	10	4ms	0
3	Accepted	3MB	4ms	10	5ms	0
4	Accepted	11MB	2527ms	10	2530ms	0
5	Accepted	11MB	2568ms	10	2578ms	0


*Chỉ có thể đạt đến 5/8 test vì thời gian chạy quá lớn.*

## Thuật toán Dijkstra:

<div>  Accepted </div> <div> Time: 6913ms   Memory: 39MB   Lang: C++   Author: admin </div>						
ID	Status	Memory	Time	Score	Real Time	Singal
1	Accepted	3MB	0ms	10	5ms	0
2	Accepted	3MB	2ms	10	3ms	0
3	Accepted	3MB	3ms	10	7ms	0
4	Accepted	3MB	5ms	10	14ms	0
5	Accepted	6MB	60ms	10	64ms	0
6	Accepted	7MB	391ms	10	399ms	0
7	Accepted	34MB	1044ms	10	1065ms	0
8	Accepted	39MB	6913ms	30	6934ms	0


*Đạt 8/8 test nhưng thời gian chạy khá lớn*

## Thuật toán Dijkstra cải tiến:

<div>  Accepted </div> <div> Time: 951ms   Memory: 45MB   Lang: C++   Author: admin </div>						
ID	Status	Memory	Time	Score	Real Time	Singal
1	Accepted	3MB	1ms	10	1ms	0
2	Accepted	3MB	1ms	10	2ms	0
3	Accepted	3MB	1ms	10	4ms	0
4	Accepted	3MB	6ms	10	9ms	0
5	Accepted	6MB	84ms	10	89ms	0
6	Accepted	7MB	86ms	10	92ms	0
7	Accepted	35MB	951ms	10	977ms	0
8	Accepted	45MB	950ms	30	998ms	0

*Đạt 8/8 test thời gian chạy nhanh dưới 1000ms*

## Thuật toán Bellman-Ford:

<div>  <b>Accepted</b> </div> <div> Time: 1993ms   Memory: 34MB   Lang: C++   Author: admin </div>						
ID	Status	Memory	Time	Score	Real Time	Singal
1	Accepted	3MB	2ms	10	17ms	0
2	Accepted	3MB	3ms	10	20ms	0
3	Accepted	3MB	0ms	10	14ms	0
4	Accepted	3MB	12ms	10	16ms	0
5	Accepted	6MB	142ms	10	156ms	0
6	Accepted	6MB	174ms	10	186ms	0
7	Accepted	33MB	1936ms	10	1978ms	0
8	Accepted	34MB	1993ms	30	2059ms	0

*Đạt 8/8 test thời gian chạy thấp*

## KẾT LUẬN

Các thuật toán tìm đường đi ngắn nhất là các thuật toán hay cần phổ cập phổ biến cho sinh viên. Hiện tại chúng ta có thể cải tiến các thuật toán đó để phù hợp với nhu cầu sử dụng từ đó đẩy nhanh được tốc độ tính toán của thuật toán.

Thông qua bài báo cáo chúng ta đã hiểu rõ được bản chất của các thuật toán đồng thời nắm bắt được cách thức cài đặt của các thuật toán.

Nắm bắt và cải tiến được các thuật toán.

Đánh giá so sánh được từng thuật toán trên lý thuyết và cả trên thực tiễn.

Thông qua ứng dụng cụ thể từ đó có được cái nhìn tổng qua hơn về các thuật toán tìm hiểu.

Ngoài ra chúng ta còn nắm bắt thêm được chương trình sinh test và test thử tự động các thuật toán nhằm mục đích kiểm thử và đánh giá lại thuật toán ở mức thực tế.

## **Tài liệu tham khảo**

Link code Floyd:

<https://github.com/nghiadh2016/doanthuattoan2019/blob/master/floyd-warshall.cpp>

Link code Dijkstra:

<https://github.com/nghiadh2016/doanthuattoan2019/blob/master/dijkstra.cpp>

Link code Dijkstra cải tiến:

<https://github.com/nghiadh2016/doanthuattoan2019/blob/master/dijkstracaitien.cpp>

Link code Bellman-Flord cải tiến:

<https://github.com/nghiadh2016/doanthuattoan2019/blob/master/bellman-fordcaitien.cpp>

Link code sinh test:

<https://github.com/nghiadh2016/doanthuattoan2019/blob/master/sinhtest.cpp>

Link test: <https://github.com/nghiadh2016/doanthuattoan2019>