MỤC LỤC

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-flord 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 Konigsberg. 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 liền kề. Như vậy, K_n có n*(n-1)/2 cạnh và mỗi đỉnh của Kn 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 Đỉnh liền kề và cạnh liên thuộc đỉnh:

Hai đỉnh u và v trong đồ thị (vô hướng) G=(V,E) được gọi là liền 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) e1, e2, ..., en của đồ thị sao cho e1=(x0,x1),e2=(x1,x2), ...,en=(xn-1,xn), với x0=u và xn=v. Khi đồ thị không có cạnh (hoặc cung) bội, ta ký hiệu đường đi này bằng dãy các đỉnh x0, x1, ..., xn.

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.

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 ∈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, nhưng các thuật toán sẽ trình bày đều có thể áp dụng cho các đồ thị vô hướng có trọng 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 lượng nhỏ nhất.

Đối với các khuyên có trọng lượng 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:

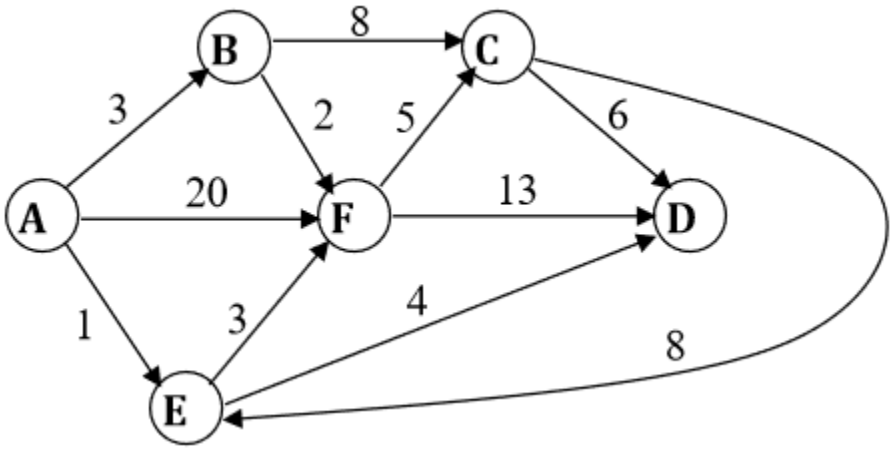
Ma trận trọng số Arr_{ij} là ma trận trọng số của đồ thị G(V,E) có các quy tắc sau:

Nếu giữa 2 đỉnh (i,j) có cạnh nối thì $Arr_{ij}= \#v_{ij}$

Nếu giữa 2 đỉnh (i,j) có nhiều cạnh nối thì $Arr_{ij}= \min(\#v_{ij1}, \#v_{ij2},\ldots)$

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

Ví dụ:



Ma trận trọng số:

	A	**B**	**C**	**D**	**E**	**F**
A	∞	3	∞	∞	1	20
B	∞	∞	8	∞	∞	2
C	∞	∞	∞	6	8	∞
D	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	4	∞	3
F	∞	∞	5	13	∞	∞

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={v1, v2, ..., vn} 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 ($0 \le k \le n$).

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 vi với đỉnh vj đi qua các đỉnh trung gian trong {v1, v2, ..., vk}.

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 vi với vj và đi qua các đỉnh trung gian trong {v1, v2, ..., vk}, có một đường đi γ sao cho vk không có trong γ. Khi đó γ cũng là đường đi ngắn nhất nối vi với vj đi qua các đỉnh trung gian trong {v1, v2, ..., vk-1}, nên theo giả thiết quy nạp:

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

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 vi với vj và đi qua các đỉnh trung gian trong {v1, v2, ..., vk}, đều chứa vk. Gọi γ = vi ... vk ... vj là một đường đi ngắn nhất như thế thì v1 ... vk và vk ... vj cũng là những đường đi ngắn nhất đi qua các đỉnh trung gian trong {v1, v2, ..., vk-1} và

$$W_{k-1}[i,k]+W_{k-1}[k,j] = \text{chiều dài(v1 ... vk)} + \text{chiều dài(vk ... vj)}$$
$$= \text{chiều dài } \gamma < W_{k-1}[i,j].$$

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≤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 $k \le n$ 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 phưc 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 arr 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 dầu 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.

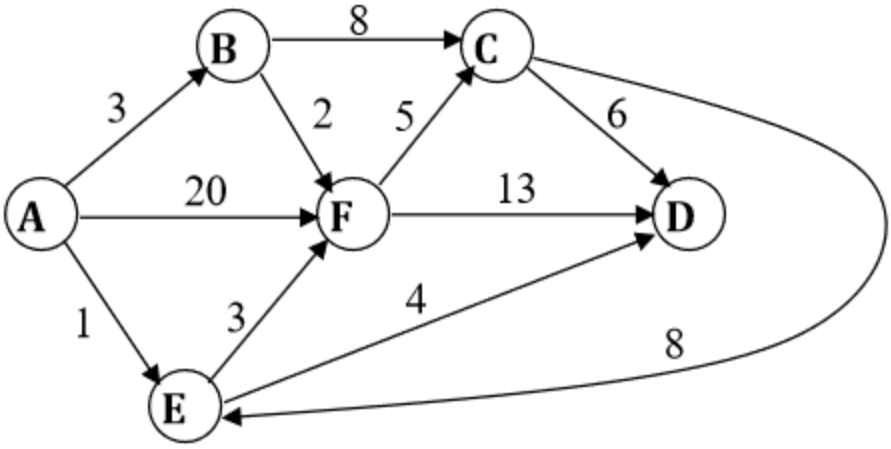
Chương trình cụ thể được viết trên c++:

```
#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	∞	3	∞	∞	1	20
B	∞	∞	8	∞	∞	2
C	∞	∞	∞	6	8	∞
D	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	4	∞	3
F	∞	∞	5	13	∞	∞

W_1:

	A	**B**	**C**	**D**	**E**	**F**
A	∞	3	∞	∞	1	20
B	∞	∞	8	∞	∞	2
C	∞	∞	∞	6	8	∞
D	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	4	∞	3
F	∞	∞	5	13	∞	∞

W_2:

	A	**B**	**C**	**D**	**E**	**F**
A	∞	3	11	∞	1	5
B	∞	∞	8	∞	∞	2
C	∞	∞	∞	6	8	∞
D	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	4	∞	3
F	∞	∞	5	13	∞	∞

W_3:

	A	B	C	D	E	F
A	∞	3	11	17	1	5
B	∞	∞	8	14	16	2
C	∞	∞	∞	6	8	∞
D	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	4	∞	3
F	∞	∞	5	11	13	∞

W_4:

	A	B	C	D	E	F
A	∞	3	11	17	1	5
B	∞	∞	8	14	16	2
C	∞	∞	∞	6	8	∞
D	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	4	∞	3
F	∞	∞	5	11	13	∞

W_5:

	A	B	C	D	E	F
A	∞	3	11	5	1	4
B	∞	∞	8	14	16	2
C	∞	∞	∞	6	8	11
D	∞	∞	∞	∞	∞	∞
E	∞	∞	∞	4	∞	3
F	∞	∞	5	11	13	16

W_6:

	A	B	C	D	E	F
A	∞	3	9	5	1	4
B	∞	∞	7	13	15	2
C	∞	∞	16	6	8	11
D	∞	∞	∞	∞	∞	∞
E	∞	∞	8	4	16	3
F	∞	∞	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[1], 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 điền 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 u0 từ nhỏ đến lớn.

- Khoảng cách d(u0,u0)=0.
- Trong các đỉnh v khác u0, tìm đỉnh có khoảng cách k1 đến u0 là nhỏ nhất.

Đỉnh này phải là một trong các đỉnh kề với u0. Giả sử đó là u1. Ta có: d(u0,u1) = k1.

- Trong các đỉnh v khác u0 và v khác u1, tìm đỉnh có khoảng cách k2 đến u0 là nhỏ nhất.

Đỉnh này phải là một trong các đỉnh kề với u0 hoặc với u1. Giả sử đó là u2. Ta có: d(u0,u2) = k2.

Lặp lại quá trình trên cho đến khi tìm được khoảng cách từ u0 đến mọi đỉnh v của G. Nếu V={u0, u1, ..., un} thì: 0 = d(u0,u0) < d(u0,u1) < d(u0,u2) < ... < d(u0,un).

Cụ thể chúng ta đi theo từng bước sau:

Bước 1: xác định các đình 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 tạm 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 tap 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;
```

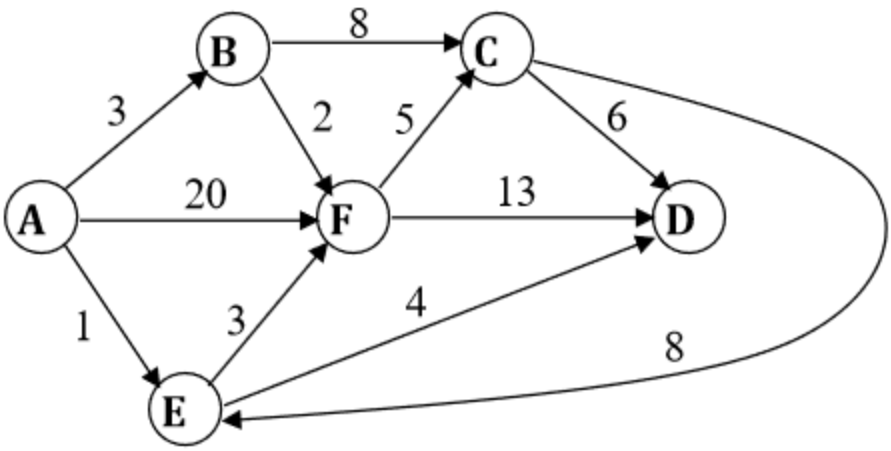
```
                if(arr[numberx][numbery]!=0)arr[numberx][numbery]=min(arr[num
                berx][numbery],values);
                else arr[numberx][numbery]=values;
        }

}
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];

}
```

Hiển nhiên nếu muốn tăng tốc độ có thể tìm thầy 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
Ban đầu	∞	∞	∞	∞	∞	∞
Lần lặp 1	∞	3	∞	∞	1	20
Lần lặp 2	∞	3	∞	5	1	4
Lần lặp 3	∞	3	11	5	1	4
Lần lặp 4	∞	3	9	5	1	4
Lần lặp 5	∞	3	9	5	1	4
Lần lặp 6	∞	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 sếp 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 tiền 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][n
            umbery],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-flord 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 gia trij 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 còng 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_Flord(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 tạm thời gian và không gian

Độ phức tạp về mặt thời gian: Vì chương trình lặp 2 vong 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*E).

Độ phức tạp về mặt không gian: Cần n ô 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 đồ 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 toan này có độ phức tạp thời gian không đến O(V*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*E).

2.4.3 Cài đặt chương trình

Chương trình cài đặt được viết trên c++:

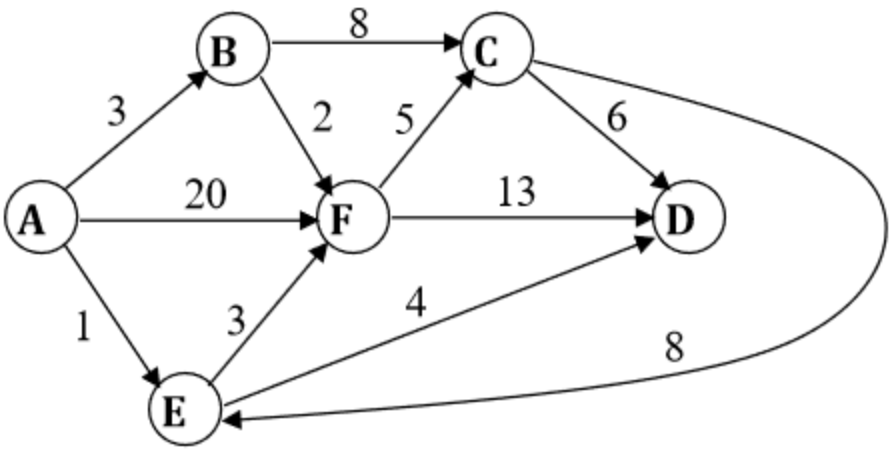
```
#include<iostream>
#include<vector>
#include<map>
#include<set>
#include<fstream>
using namespace std;
map<pair<int,int>,long long> arr;
long long* values_arr;
bool* bool_arr;
void init_arr(int vertical)
{
    values_arr=new long long [vertical+1];
    bool_arr=new bool[vertical+1];
    for(int i=1; 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,numbery)
                ]=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_Flord(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_Flord(vertical);
        printf_arr(vertical);
        if(values_arr[verticaly]==10e17)cout<<"khong ton tai duong di tu dinh
        "<<verticalx<<" sang dinh "<<verticaly;
        else if(bool_arr[verticaly])cout<<"có chu trình âm\t";
        else cout<<"duong di ngan nhat tu dinh "<<verticalx<<" sang dinh
        "<<verticaly<<" la: "<<values_arr[verticaly];

    }
```

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
Ban đầu	∞	0	∞	∞	∞	∞
Lần lặp 1	∞	0	7	14	16	2
Lần lặp 2	∞	0	7	13	15	2
Lần lặp 3	∞	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 dầu 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 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=10;
	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 và đánh giá:

Bảng sau là 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	13ms	14ms	13ms	14ms
V=100,E=50	19ms	14ms	16ms	13ms
V=100,E=500	21ms	18ms	16ms	19ms
V=1000,E=5000	3931ms	31ms	33ms	43ms
V=1000,E=50000	3810ms	176ms	173ms	215ms
V=10000,E=50000	3512657ms	723ms	188ms	313ms
V=10000,E=500000	3676657ms	1921ms	1471ms	2867ms
V=100000,E=500000	Quá lớn để cấp phát bộ nhớ lên đến vài Gb	11745ms	1526ms	2898ms

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 dự vào độ phức tạp về mặt thời gian ta có:

Floyd=$V^3=10^{15}$ nên thời gian chạy khá lớn.

Dijkstra=$V^2+E=10^{10}$ nên thòi gian chạy nằm ở tầm hơn 10s kết quả chạy gần đúng với thực tế đã dánh giá.

Dijkstra cải tiến = $(V+E)\log_n$= sấp sỉ $2*10^8$ cũng chính vì thế thời gian chạy thực là thấp nhất.

Bellman-Flord 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 5 lần so với Dijkstra bình thường.

3.3 Ứng dụng:

Cũng chính vì thuật toán Dijkstra có khả năng tính toán nhanh nên hiện tại nó được ứng dụng để làm các định vị GPS hay các công cụ tìm đường trên google map. Hiển nhiên

các thuật toán trên điều có ứng dụng của nó mỗi thuật toán điều có cái hay riêng và tùy vào nhu cầu của người sử dụng mà người ta cài đặt thuật toán cho thích hợp.

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 3 thuật toán đồng thời nắm bắt được cách thức cài đặt của 3 thuật toán.

Nắm bắt và cải tiến được các thuật toán trên.

Đánh giá được từng thuật toán.

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