最短路径问题

> 以下的图均为没有圈的有向图(DAG)，无向图处理一下输入就可以。

任意两点间的最短路径

Floyd-Warshall算法

使用dp进行求解，由0点中转到n点中转。可以处理负权边，判断是否有dp[i][i]为负数的顶点

```
dp[k][i][j]:从i到j的最短路径。//k为中转点
dp[0][i][j] = cost[i][j];
//选取min经过k点和不经过k点 0<=k<V(V为顶点数) 当k=0，k-1实质上是不经过任何一个点故为dp[0][i][j]
dp[k][i][j] = min(dp[k - 1 < 0 ? 0 : k - 1][i][j], dp[k - 1 < 0 ? 0 : k - 1][i][k] + dp[k - 1 < 0 ? 0 : k - 1][k][j]);
//优化若dp[k-1][i][j](即不经过k点较小)那么dp[k][i][j]=dp[k-1][i][j]
//否则dp[k][i][j]=dp[k-1][i][k]+dp[k-1][k][j]与dp[k-1][i][j]无关
//因此去掉多余拷贝的dp[k]数组
dpi][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
```

三维数组：

```
//INF不能使用Integer.MAX_VALUE或numeric_limits<int>::max();两个int极限相加会超过int极限导致错误
constexpr int maxsize = 100, INF = 999999;
int V, M, dp[maxsize][maxsize][maxsize];

void init() {
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (i != j)
                dp[0][i][j] = INF;
}

void floyd() {
    for (int k = 0; k < V; k++)
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                dp[k][i][j] = min(dp[k - 1 < 0 ? 0 : k - 1][i][j], dp[k - 1 < 0 ? 0 : k - 1][i][k] + dp[k - 1 < 0 ? 0 : k - 1][k][j]);
}
```

二维数组：

```
void init() {
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (i != j)
                dp[i][j] = INF;
}
void floyd() {
    for (int k = 0; k < V; k++)
        for (int i = 0; i < V; i++)
            for (int j = 0; j < V; j++)
                dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);
}
```

> 在Floyd循环中,当k=0时，经过0中转；当k=1时，经过1进行中转，此时已经包含了k=0时中转的最优解
>
> 以此类推，进行dp求解

测试数据：

> 输入：V M(V为顶点数，M为边数)
>
> 4 8
> 1 2 2
> 1 3 6
> 1 4 4
> 2 3 3
> 3 1 7
> 3 4 1
> 4 1 5
> 4 3 12
>
> 输出：
>
> 0 2 5 4
> 9 0 3 4
> 6 8 0 1
> 5 7 10 0

单源最短路径

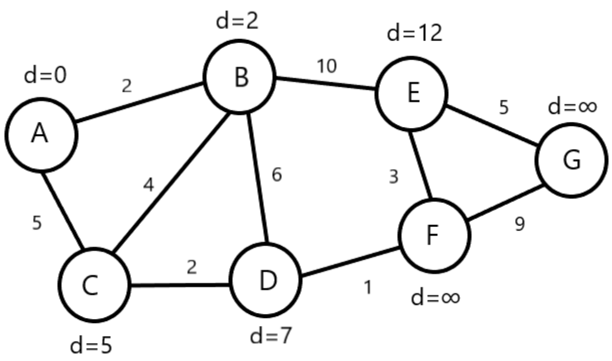
单源最短路径：固定起点，求该点到任意一点的最短路径。

设s为源点，i为目标点，则：d[i] = min(d[i], d[j] + cost[j][i]);通过点j进行中转，j为已知最短距离顶点。d为存储源点到各个顶点的最短距离的数组。

Dijkstra算法：

找到最短距离已经确定的顶点，从该顶点出发对该顶点能到达的顶点进行松弛，从而求解最短路径。因为每次查询到的顶点都已经是最短路径了，所以不用再对已经求过的顶点进行求解。

不能处理负权边的情况，每次求最小，负数相加更小。时间复杂度O(v^2)。




```
constexpr int maxsize = 100, INF = 999999;
int V, M, d[maxsize], cost[maxsize][maxsize];
bool used[maxsize];
void init() {
    for (int i = 0; i < V; i++)
        for (int j = 0; j < V; j++)
            if (i != j)
                cost[i][j] = INF;
}
void Dijkstra(int s) {
    fill(d, d + V, INF);
    fill(used, used + V, false);
    d[s] = 0;
    while (true) {
        int v = -1;
        for (int u = 0; u < V; u++)
            if (!used[u] && (v == -1 || d[u] < d[v])) v = u;
        if (v == -1) break;
        used[v] = true;
        for (int u = 0; u < V; u++)
            d[u] = min(d[u], d[v] + cost[v][u]);
    }
}
```

测试数据：

> 输入：V M s (s为源点)
>
> 6 9 1
> 1 2 1
> 1 3 12
> 2 3 9
> 2 4 3
> 3 5 5
> 4 3 4
> 4 5 13
> 4 6 15
> 5 6 4
>
> 输出：
>
> 0 1 8 4 13 17

队列优化：时间复杂度O(ElogV)，每次取出最短距离的顶点，源点对该顶点能到达的点进行更新。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>

using namespace std;
constexpr int maxsize = 100, INF = 999999;
struct edge {
    int to, cost;
public:
    edge(int t, int c) { to = t, cost = c; }
};
typedef pair<int, int> P;//first为最短距离 second为最短距离对应顶点 pair默认排序先对
first
int V, M, d[maxsize];
vector<edge> edges[maxsize];

void Dijkstra(int s) {
    fill(d, d + V, INF);
    d[s] = 0;
    priority_queue<P, vector<P>, greater<P>> queue;
    queue.push(P(0, s));
    while (!queue.empty()) {
        P p = queue.top(); queue.pop();
        if (d[p.second] < p.first) continue;//当堆顶不是最小值
        for (int i = 0; i < edges[p.second].size(); i++) {
            edge e = edges[p.second][i];
            if (d[e.to] > d[p.second] + e.cost) {
                d[e.to] = d[p.second] + e.cost;
                queue.push(P(d[e.to], e.to));
            }
        }
    }
}

int main()
{
    int f, t, cost, s;
    cin >> V >> M >> s;
    for (int i = 0; i < M; i++) {
        cin >> f >> t >> cost;
        edges[f - 1].emplace_back(t - 1, cost);
    }
    Dijkstra(s - 1);
    for (int i = 0; i < V; i++)
        cout << d[i] << " ";
    return 0;
}
```

Bellman-Ford算法

V个顶点，从源点到任意一个点的最短距离最多经过V-1个边，每一条边可能经过最多V-1次更新。因此可以通过限制最大次数，在最大次数后判断还是否会更新值来判断是否存在负权圈。

```
constexpr int maxsize = 100, INF = 99999999;
struct edge { int from, to, cost; };
edge edges[maxsize];
int V, E, d[maxsize];
//注意下标均以0开始
bool ford(int s) {
    fill(d, d + V, INF);
    d[s] = 0;
    for (int i = 0; i < V - 1; i++) {
        for (int j = 0; j < E; j++) {
            edge e = edges[j];
            if (d[e.to] > d[e.from] + e.cost) {
                d[e.to] = d[e.from] + e.cost;
            }
        }
    }
    for (int j = 0; j < E; j++) {
        if (d[edges[j].to] > d[edges[j].from] + edges[j].cost) {
            return true;
        }
    }
    return false;
}
```

因为存在不到V-1次循环就求到最短路径的情况，为了节约时间，在题中表明**不存在负圈**时可以通过设置哨兵对循环进行优化。

```
constexpr int maxsize = 100, INF = 999999;
struct edge { int from, to, cost; };
edge edges[maxsize];
int V, E, d[maxsize];

void ford(int s) {
    fill(d, d + V, INF);
    d[s] = 0;
    while (true) {//若有负圈则会陷入死循环
        bool update = false;
        for (int i = 0; i < E; i++) {
            if (d[edges[i].from] != INF && d[edges[i].to] > d[edges[i].from] +
edges[i].cost) {
                d[edges[i].to] = d[edges[i].from] + edges[i].cost;
                update = true;
            }
        }
        if (!update) break;
    }
}
```

> 如果一个图如果没有负圈，那么最短路径所包含的边最多为n-1条，即进行n-1轮松弛之后最短路不会再发生变化。如果在n-1轮松弛之后最短路仍然会发生变化，则该图必然存在负权回路。

判断是否存在负圈：

```
bool find_negative_loop() {
    memset(d, 0, sizeof(d));
    for (int i = 0; i < V; i++) {
        for (int j = 0; j < E; j++) {
            edge e = edges[j];
            if (d[e.to] > d[e.from] + e.cost) {
                d[e.to] = d[e.from] + e.cost;
                if (i == V - 1)
                    return true;
            }
        }
    }
    return false;
}
```

测试数据：

> input:
>
> 5 5 1
> 2 3 2
> 1 2 -3
> 1 5 5
> 4 5 2
> 3 4 3
>
> output:
>
> flase //是否存在负圈
> 0 -3 -1 2 4 //与各点最短距离

> input:
>
> 5 5 1
> 1 2 -1
> 2 3 -2
> 3 4 -3
> 4 5 -4
> 5 1 -5
>
> output:
>
> true
>
> //陷入死循环

队列优化：**不存在负圈**可以使用。初始时，将源点加入队列，对该源点能到达的点A进行判断，如果可以A点对A可以到达的点B进行更新，则将B点加入队列(源点到B的距离被更新)，同时如果B点已经在队列中那么B点不需要再进行入队，通过book数组进行判断。因为如果有新的路径进行更新，那么涉及到的更新的顶点可能会重复入队，故在一次A判断过后将book[A]置为false;若有一个顶点入队的次数大于n则说明存在负圈。

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;
```

```java
import java.util.Vector;

class edge {
    int to, cost;

    public edge(int t, int c) {
        to = t;
        cost = c;
    }
}

class Solution {
    Vector<Vector<edge>> es;
    int V;
    int[] d, num;
    final int INF = 999999;
    boolean book[];

    public boolean init(Vector<Vector<edge>> edges, int v) {
        es = edges;
        V = v;
        d = new int[v];
        num = new int[v];
        book = new boolean[v];
        return ford();
    }

    public boolean ford() {
        Arrays.fill(d, INF);
        Arrays.fill(num, 0);
        Arrays.fill(book, false);
        d[0] = 0;
        Queue<Integer> queue = new LinkedList<Integer>();
        queue.offer(0);
        book[0] = true;
        ++num[0];
        while (!queue.isEmpty()) {
            int cur = queue.poll();
            for (edge e : es.get(cur)) {
                if (d[e.to] > d[cur] + e.cost) {
                    d[e.to] = d[cur] + e.cost;
                    if (!book[e.to]) {
                        book[e.to] = true;
                        queue.offer(e.to);
                        if (++num[e.to] > V)
                            return true;
                    }
                }
            }
            book[cur] = false;
        }
        return false;
    }
}
```

测试数据:

> input:

> 3 3 1
> 1 2 3
> 2 3 4
> 3 1 -8
>
> output:
>
> 负圈

	Floyd	Dijkstra	队列优化的Dijkstra	Bellman-Ford	队列优化的Frod
时间复杂度	$O(V^3)$	$O(V^2)$	O(ElogV)	O(EV)	最坏O(EV)
负权	可以	不可以	不可以	可以	可以