最短路问题

最短路是指，给定两个顶点，在以这两个点为起点和终点的路径中，边的权值之和最小的路径。其中，固定起点的也叫做单源最短路问题，起点终点都固定的叫两点之间最短路问题，任意两点间的叫做任意两点间的最短路问题。

Bellman-Ford算法

贝尔曼-福特是一个单源最短路算法，还可以处理边是负数的情况。

先规定如下含义：

```
struct edge // 从顶点from指向顶点to的权值为cost的边
{
    int from, to, cost;
};

edge es[MAX_E]; // 边
int d[MAX_V];   // 从起点s出发到顶点i的最短距离d[i]
int V, E;       // V是顶点数，E是边数
```

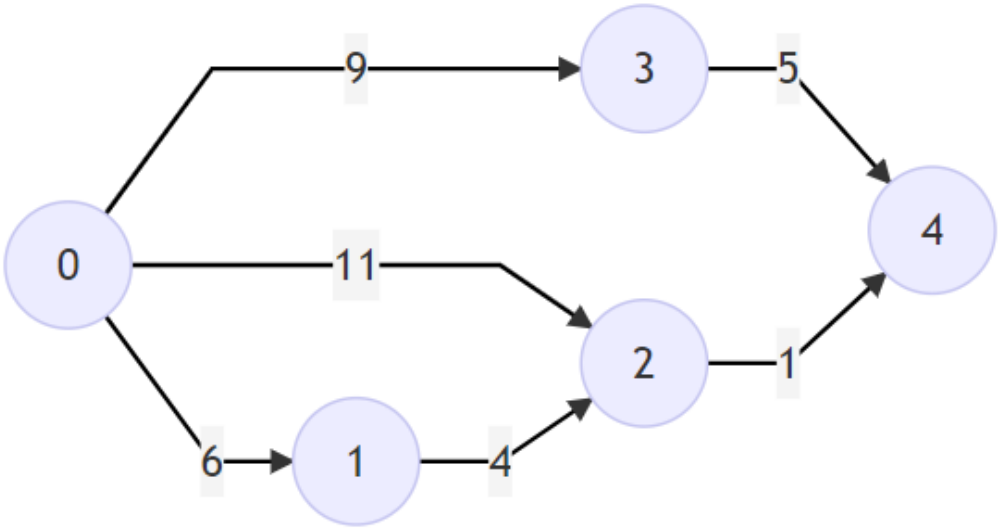
初始化时将除了起点 s 以外的 d[i] 设为足够大的常数，并将起点到起点的最短距离 d[s] 设为 0 。

假如一条边是从顶点 i 指向顶点 j ，权值为 w ，则 `d[j] = min(d[j], d[i] + w)`

然后枚举每一条存在的边并进行如上行所示的更新，显然 d[i] 和 d[j] 都在不断地被更新，虽然中间可能会产生许多没有用的更新，但最终一定会达到最佳状态，即可求得最短路。

```
bool bellman_ford(int s) // s为起点
{
    for (int i = 0; i < V; ++i)
        d[i] = INF;
    d[s] = 0;
    // int n = 0;
    while (1)
    {
        bool update = false;
        for (int i = 0; i < E; ++i) // 枚举每条边并更新d的值
        {
            edge e = es[i];
            if (d[e.from] != INF && d[e.to] > d[e.from] + e.cost)
            {
                d[e.to] = d[e.from] + e.cost;
                update = true;
            }
        }
        if (!update)
            break;
        // if (n == V - 1) // 存在负圈
            // return true;
        // n++;
    }
    return false;
}
```

例如对于下面那副图，求解的过程如下：



边	边上的权值
0 -> 1	6
2 -> 4	1
0 -> 3	9
1 -> 2	4
0 -> 2	11
3 -> 4	5

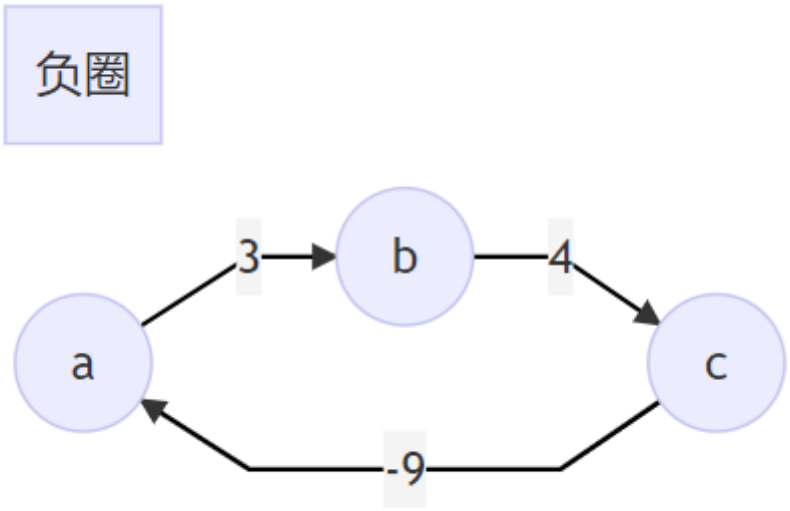
以 0 为起点的最短路，当枚举每条边并更新 d[i] 的值时：

初始化	es[0]	es[1]	es[2]	es[3]	es[4]	es[5]
d[0] = 0	0	0	0	0	0	0
d[1] = INF	d[0] + 6 = 6	6	6	6	6	6
d[2] = INF	INF	INF	INF	d[1] + 4 = 10	10(d[0] + 11 > 10)	10
d[3] = INF	INF	INF	d[0] + 9 = 9	9	9	9
d[4] = INF	INF	INF	INF	INF	INF	d[3] + 4 = 13

update = true ， 继续进行下一轮更新：

第二轮枚举边	es[0]	es[1]	es[2]	es[3]	es[4]	es[5]
d[0] = 0	0	0	0	0	0	0
d[1] = 6	6	6	6	6	6	6
d[2] = 10	10	10	10	10	10	10
d[3] = 9	9	9	9	9	9	9
d[4] = 13	13	11(d[2] + 1 < 13)	11	11	11	11

update还是等于true，继续进行下一轮更新，但由于已经是最简了，下一轮更新的update = false，跳出循环。



如果图中不存在 s 可以到达的负圈，那么最短路最多通过 |V| - 1条边，这样的更新操作是有限的，循环最多执行 |V| - 1 次，因此时间复杂度为 O(|V| x |E|)，若第 |V| 次循环仍然 更新，那么也就说明图中含有负圈，假如将上面代码中注释的代码取消注释，那么它可以在发现负圈时及时退出，但仅能说明图中存在 s 可以到达的负圈，如果一开始对所有的顶点 i，都把 d[i] 初始化为 0，那么可以检查出所有的负圈。

```
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 = es[j];
            if (d[e.to] > d[e.from] + e.cost)
            {
                d[e.to] = d[e.from] + e.cost;

                // 如果第V次仍然更新了，则存在负圈
                if (i == V - 1)
                    return true;
```

```
            }
        }
    }
    return false;
}
```

有的问题不止需要输出最短距离，需要求解最短路的路径。如果在求解的过程中增加一个数组 prev[j] 来记录最短路上顶点 j 的前趋节点，当满足 `d[j] = d[k] + cost[k][j]` 时，顶点 k 就是顶点 j 的前趋节点，当 d[j] 被 `d[j] = d[k2] + cost[k2][j]` 更新时，便修改 prev[j] = k2 即可。不断地像这样寻找前驱节点就可以恢复出最短路径，将 prev 数组初始化为-1，当路径寻找到起点时，起点的前驱节点 prev[s] = -1，便可停止。

```
int prev[MAX_V]; // 多加一个全局数组记录前趋节点
void Bellman_Ford(int s)
{
    for (int i = 0; i < V; ++i)
    {
        d[i] = INF;
        prev[i] = -1; // 初始化为-1
    }
    d[s] = 0;
    while (1)
    {
        bool update = false;
        for (int i = 0; i < E; ++i)
        {
            edge e = es[i];
            if (d[e.from] != INF && d[e.to] > d[e.from] + e.cost)
            {
                d[e.to] = d[e.from] + e.cost;
                prev[e.to] = e.from; // 记录前趋节点
                update = true;
            }
        }
        if (!update)
            break;
    }
}

// 到顶点t的最短路
vector<int> get_path(int t)
{
    vector<int> path;
    for (; t != -1; t = prev[t]) // 不断沿着prev[t]走直到起点s
        path.push_back(t);
    reverse(path.begin(), path.end()); // 这样得到的是按照t~s的路，翻转后即为最短路径
    return path;
}
```

包括像下面两个还没介绍的算法都可以用类似地办法还原最短路路径。

SPFA

SPFA 全称是 Shortest Path Faster Algorithm，这个名字一听就知道它肯定不慢。它是在 Bellman-Ford 的基础上进行了队列优化，平均时间复杂度为 O(|E|)，最坏的情况下与 Bellman-Ford 一样时间复杂度为 O(|V| x |E|)，所以也非常的好用，也能用于含有负权边的情况以及判断是否存在负环。

Bellman-Ford 要不停的枚举每一条边不断地更新 d[i] 直至达到最佳状态，其中无疑有许多没有变化的更新，还可能会向上面举的例子那样，第二轮枚举边的循环只更新了一个顶点的最短距离。而 SPFA 则对这一点进行了优化，利用队列将发生减小更新的 d[i] 的顶点 i 加入队列中。用一个 bool 数组来记录该顶点是否已经在队列中，若顶点 i 已存在于队列中，则不需要重复插入。然后只需要对队列中的顶点的相邻的边进行更新即可，不需要对那些没有变动的进行更新，节约了时间。

```
struct edge
{
    int from, to, cost, next;
};

edge es[MAX_V];
int d[MAX_V];
bool flag[MAX_V];
int head[MAX_V];
int V, E;

void spfa(int s)
{
    queue<int> que;
    fill(d, d + V, INF);
    fill(flag, flag + V, false);
    d[s] = 0;
    que.push(s);
    flag[s] = true;

    while (!que.empty())
    {
        int e = que.front();
        que.pop();
        flag[e] = false;

        for (int i = head[e]; i >= 0; i = es[i].next)
        {
            int u = es[i].to;
            if (d[u] > d[e] + es[i].cost)
            {
                d[u] = d[e] + es[i].cost;
                if (!flag[u])
                {
                    que.push(u);
                    flag[u] = true;
                }
            }
        }
    }
}

int main()
{
    cin >> V >> E;
```

```
    int s;
    cin >> s;
    for (int i = 0; i < E; ++i)
    {
        cin >> es[i].from >> es[i].to >> es[i].cost;
        if (i == 0)
            es[i].next = -1;
        else
            es[i].next = head[es[i].from];
        head[es[i].from] = i;
    }
    spfa(s);
    for (int i = 0; i < V; ++i)
    {
        if (d[i] == INF)
            cout << "-1 ";
        else
            cout << d[i] << " ";
    }
    cout << endl;
    return 0;
}
```

不过在实际的应用中 SPFA 的时间效率不是很稳定，为了避免最坏的情况，在不存在负边时，通常会使用时间效率更加稳定的 Dijkstra 算法。

Dijkstra算法

迪卡斯特拉也是一种单源最短路的算法，虽然它无法处理有负边的情况，但在没有负边的情况下，它可以找到最短距离已经确定的顶点，然后再从它出发更新相邻顶点的最短距离，并且不再关心最短距离已经确定的顶点，不用每一次循环都检查一遍从 i 出发的边，这显然节约了时间。

规定如下含义：

```
int cost[MAX_V][MAX_V]; // cost[u][v]表示边 e = (u, v)的权值（不存在这条边时设为INF）
int d[MAX_V];           // 从顶点s出发的最短距离
bool used[MAX_V];       // 标记该顶点是否已使用过
int V;                  // 顶点数
// int prev[MAX_V];        // 最短路上的前趋节点
```

由于这个算法就是在确定没有负边的情况下使用的，所以从起点开始的相邻顶点的最短距离是可以确定的，且由于不存在负边，在之后的更新中也不会变小，所以就不再需要关心已经确定的最短距离顶点。

每次从未使用过的顶点中找出一个距离最小的顶点 v ，然后以这个顶点为基础，对它所能到达的所有顶点 u 进行判断，若 u 还没有与起点连通，或者最短距离 d[u] 大于 d[v] 加上顶点 v 到 顶点 u 的距离，则将 d [u] 进行更新，即 `d[u] = min(d[u], d[v] + cost[v][u])`

初始化时，将起点的最短距离 d[s] 设为 0 ，其它 d[i] 设为足够大的常数，将 used 数组全部设为 false 。

```
void dijkstra(int s) // s为起点
{
    fill(d, d + V, INF);
    fill(used, used + V, false);
    // fill(prev, prev + V, -1);
```

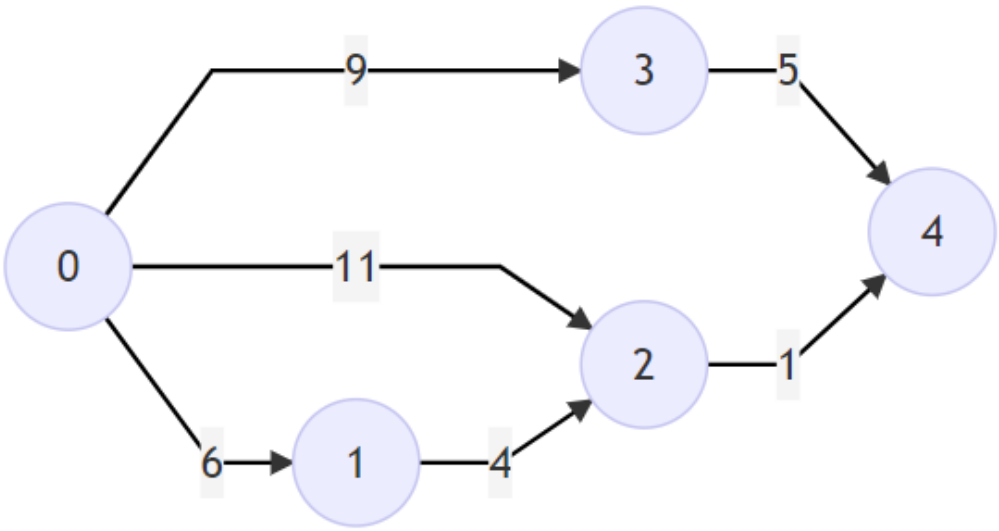
```
    d[s] = 0;

    while (1)
    {
        int v = -1;
        // 从尚未使用过的顶点中选择一个距离最小的顶点
        for (int u = 0; u < V; ++u)
        {
            if (!used[u] && (v == -1 || d[u] < d[v]))
                v = u;
        }

        if (v == -1)    // 当所有顶点都已使用过，则结束循环。
            break;
        used[v] = true; // 将该顶点标记为已使用

        for (int u = 0; u < V; ++u) // 更新最短距离
        {
            // if (d[u] > d[v] + cost[v][u]) //（这个if在不记录前趋节点时可以去掉）
            // {
                d[u] = min(d[u], d[v] + cost[v][u]);
                // prev[u] = v; // 记录前趋节点
            // }
        }
    }
}
```

例如对于下面那幅图，求解的过程如下：



i / j	0	1	2	3	4
0	0	6	11	9	INF
1	INF	0	4	INF	INF
2	INF	INF	0	INF	1
3	INF	INF	INF	0	5
4	INF	INF	INF	INF	0

求以 0 为起点的最短路

每一轮选择的顶点，及更新后的最短距离如下：

	顶点 0	顶点 1	顶点 3	顶点 2	顶点 4	结果
d[0] = 0	0	0	0	0	0	0
d[1] = INF	6	6	6	6	6	6
d[2] = INF	11	10(d[1] + 4 < 11)	10	10	10	10
d[3] = INF	9	9	9	9	9	9
d[4] = INF	INF	INF	d[3] + 5 = 14	11(d[2] + 1 < 14)	11	11

使用邻接表实现也是类似的过程

使用邻接矩阵实现时的复杂度为 O(|V|2)

使用邻接表实现时，虽然更新最短距离只需要访问每条边一次复杂度为 O(|E|)，但是由于每次都要枚举所有顶点来查找下一个使用的顶点，最终复杂度还是为 O(|V|2)。在边的数量较少时，可以用堆进行优化，这样可以使得每次从堆中取出下一个使用的顶点的时间复杂度降为 O(log|V|)

使用 STL 的优先队列，通过指定 `greater<T>` 参数，堆按照 pair<int, int> first 从小到大的顺序取出值

```
/**************************************************************************
 *
 * 使用堆优化
 * 复杂度 O(|E|log|V|)
 *
 **************************************************************************/

struct edge
{
    int to, cost;          // 每条边指向的顶点和其权值
};
typedef pair<int, int> P; // first是最短距离, second是顶点的编号

int V;
vector<edge> G[MAX_V];
```

```
int d[MAX_V];
// int prev[MAX_V];

void dijkstra(int s) // s为起点
{
    priority_queue<P, vector<P>, greater<P> > que;
    fill(d, d + V, INF);
    d[s] = 0;
    que.push(P(0, s));

    while (!que.empty())
    {
        P p = que.top(); // 取出一个距离最小的顶点
        que.pop();
        int v = p.second;
        if (d[v] < p.first) // 当取出的最小值不是最短距离的话，就丢弃这个值
            continue;
        for (int i = 0; i < G[v].size(); ++i) // 更新最短距离
        {
            edge e = G[v][i];
            if (d[e.to] > d[v] + e.cost)
            {
                d[e.to] = d[v] + e.cost;
                // prev[e.to] = v;              // 记录前趋节点
                que.push(P(d[e.to], e.to)); // 将最短值更新后的节点重新插入队列
            }
        }
    }
}
```

Floyd-Warshall算法

弗洛伊德适用于求解任意两点间的最短路问题，可以在 O(|V|3) 时间里求得所有的两点间的最短路长度，也可以处理边是负数的情况。

规定如下含义：

```
int d[MAX_V][MAX_V]; // d[u][v]表示边e = (u, v)的权值（不存在时设为INF，不过d[i][i] = 0）
int V;               // 顶点数
```

初始化时将 `d[i][i]` 设为 0，其它设为足够大的常数，但不可太大，以免进行加法 `d[i][k] + d[k][j]` 时溢出。

弗洛伊德是用动态规划来求解的，记 `d[k+1][i][j]` 为只使用顶点 0~k 和 i, j 的情况下 i 到 j 的最短路长度，当 k = -1 时，认为只使用顶点 i 和 j，即 `d[0][i][j] = cost[i][j]`

只使用顶点 0~k 时分为两种情况，一种是完全不经过顶点 k，此时 `d[k][i][j] = d[k-1][i][j]`；另一种是正好经过一次顶点 k，此时 `d[k][i][j] = d[k-1][i][k] + d[k-1][k][j]`。两种情况合起来，就得到了 `d[k][i][j] = min(d[k-1][i][j], d[k-1][i][k] + d[k-1][k][j])`。可以使用滚动数组来实现，不断进行 `d[i][j] = min(d[i][j], d[i][k] + d[k][j])` 的更新即可。

```
void warshall_floyd()
{
    for (int k = 0; k < V; ++k)
        for (int i = 0; i < V; ++i)
            for (int j = 0; j < V; ++j)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}
```

判断图中是否有负圈，只需要检查是否存在 `d[i][i]` 是负数的顶点 i 就可以了。

如果只关心两个顶点间是否有通路，而不在乎路径的长度时，可以用 1 和 0 分别表示”连通“ 和 ”不连通“ ，然后将 `d[i][j] = min(d[i][j], d[i][k] + d[k][j])` 改为 `d[i][j] = (d[i][j] || (d[i][k] && d[k][j]))` 即可，预处理也需做出调整，这样的结果称为有向图的传递闭包 (Transitive Closure) 。