}

暨南大学研究生课程设计报告

学生姓名：邵同　　　　　　　**学号：202134261058**

学院：网络空间安全学院　　　**专业：电子信息（网络空间安全）**

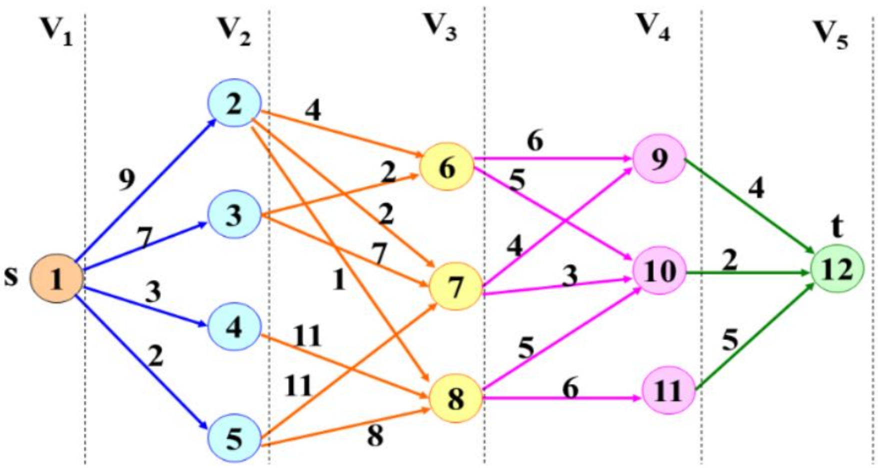
课程名称：《算法分析与设计》　　　　　**课程设计名称：多段图问题**

类型：验证-设计-综合　　　　　　　　**时间：2021 年 12 月 20 日**

课程设计内容：

多段图是一个带权有向图并且无环，有且仅有一个起始点（原点 source）和一个终止节点（汇点 target），它有 n 个阶段，每个阶段由特定的几个结点构成，每个结点的所有结点都只能指向下一个相邻的阶段，阶段之间不能越界。

多段图问题：一个带权有向图并且无环，有且仅有一个起始点(原点 source)和一个终止节点(汇点 target), 求 s 到 t 的最小成本路径。



算法描述：

（1）数据结构：

对于图的存储有邻接表与邻接矩阵。

邻接矩阵直接使用二维数组 g[N][N]。

邻接表使用数组模拟——链式前向星。

```
int h[N], e[N], ne[N], w[N], idx;
void add(int a, int b, int c)    //加边操作
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}
```

（2）算法设计及算法思路

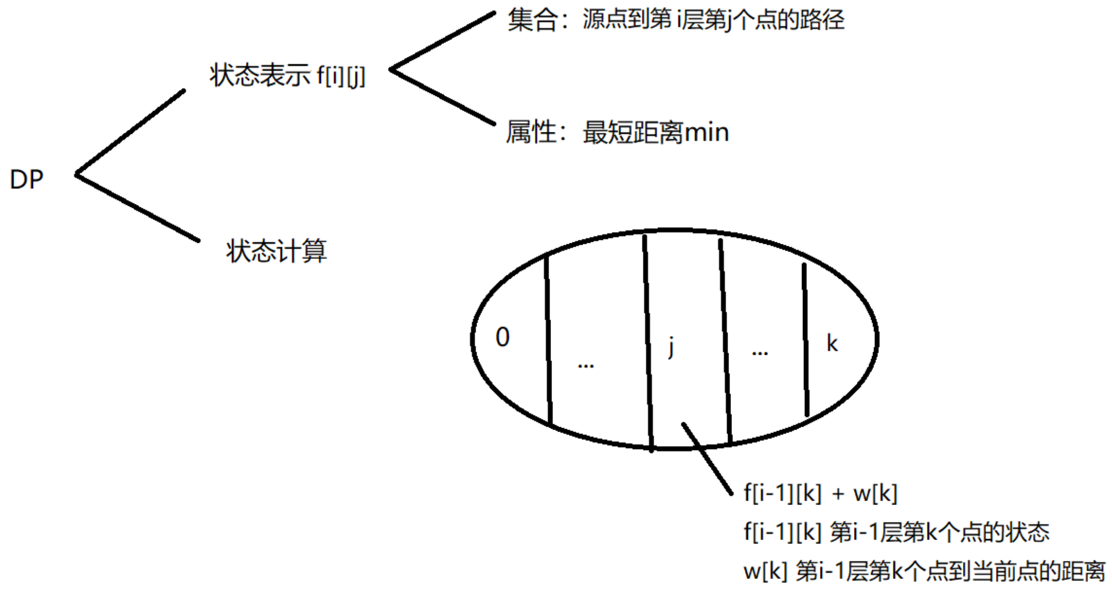
方案 1：递归与蛮力法

递归与蛮力法使用回溯法进行深度优先遍历 DFS，主要思想从初始结点进行扩展，扩展

顺序为每次扩展最新产生的结点，即为沿着一条路径走到底，当当前结点不能扩展出新结点时，回溯到上一个结点继续扩展下一个结点。

方案 2：递归与动态规划算法

(1)朴素动态规划



状态转移方程为：f[i][j] = min(f[i - 1][0 ... k] + w[0 ... k])。

递归与动态规划采用记忆化搜索（备忘录算法），记忆化搜索在求解时按照自顶向下的顺序，但是每求解一个状态就将它的解保存下来，当以后再次求解到该状态时，就不用再次求解该状态。

（2）记忆化搜索 dfs

将方案 1 的深度优先遍历算法使用记忆化搜索（备忘录算法）进行优化。

方案 3：贪心算法

贪心算法使用 Dijkstra 算法思想，进行优化。Dijkstra 算法思想为：每次选择距离集合最近的点加入集合，然后使用新加入集合的点对其他结点的距离进行更新。

而本题中图为多段图，当前层的结点只会去更新它所连接的下一层的结点，因此可以进行优化，每次只更新此点连接的点。

源程序：

- **方案 1：递归与蛮力法**

```
#include<bits/stdc++.h>
using namespace std;

const int N = 110, INF = 0x3f3f3f3f;

int h[N], e[N], ne[N], w[N], idx; //链式前向星
int n, m;
bool st[N];
int pre[N];
vector<int> tmp;
```

```
vector<int> path;
int f[N];
int res = INF;

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

void dfs(int x, int cost)
{
    if(cost > res)
        return;
    if(x == n)
    {
        if(cost < res)
        {
            res = cost;
            path.assign(tmp.begin(), tmp.end()); //记录路径
        }
        return;
    }

    st[x] = true;
    for(int i = h[x]; i != -1; i = ne[i])
    {
        int j = e[i];
        if(!st[j])
        {
            st[j] = true;
            tmp.push_back(j);
            dfs(j, cost + w[i]);
            tmp.pop_back();
            st[j] = false;
        }
    }
}

int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
    while(m --)
    {
```

```
        int x, y, z;
        cin >> x >> y >> z;
        add(x, y, z);
    }
    st[1] = true;
    tmp.push_back(1);
    dfs(1, 0);

    if(res > INF / 2) //没有最短路径，不连通
    {
        cout << "The minimum cost is: -1" <<endl;
        return 0;
    }
    cout << "The minimum cost is: " << res << endl;
    cout << "The path is: ";
    for(auto x : path)
        cout << x << ' ';

    return 0;
}
```

- 方案 2：递归与动态规划法

2.1 朴素动态规划

```
#include<bits/stdc++.h>
using namespace std;

const int N = 510, INF = 0x3f3f3f3f, K = 10;

int n, m;
int g[N][N];    //邻接矩阵
int k; //k 表示一共几段
vector<vector<int>> t(K);//顶点在第几层
vector<int> path;
int pre[N];

/* 朴素写法
int f[K][K]; //f[i][j] 表示起点到 i 层的第 j 个点的最短距离
void dp()
{
    for(int i = 1; i <= k; i ++)
    {
        int ki = t[i].size(), kj = t[i - 1].size();     //第 i 和 i - 1 层有几个点
        for(int j = 0; j < ki; j ++)
        {
```

```
            for(int _ = 0; _ < kj; _ ++)
            {
                int v_ = t[i - 1][_], vi = t[i][j];
                if(f[i - 1][_] + g[v_][vi] < f[i][j])
                {
                    f[i][j] = f[i - 1][_] + g[v_][vi];
                    pre[vi] = v_;
                }
            }
        }
    }
}
*/
// 记忆化搜索递归写法
int dp(int u, int v)
{
    int &tmp = f[u][v];
    if(tmp != INF)
        return tmp;
    tmp = 1e9; //一个较大的数但不能是 INF
    int ki = t[u - 1].size(), vi = t[u][v];
    for(int i = 0; i < ki; i ++)
    {
        int c = dp(u - 1, i), vj = t[u - 1][i];      //递归求解状态
        if(c + g[vi][vj] < tmp)
        {
            tmp = c + g[vi][vj];
            pre[vi] = vj;
        }
    }
    return tmp;
}

int main()
{
    cin >> n >> m;
    memset(g, 0x3f, sizeof g);
    while(m --)
    {
        int x, y, z;
        cin >> x >> y >> z;
        g[y][x] = min(g[y][x], z);
    }
    cout << "Please enter the level of the graph: ";
```

```
    cin >> k;
    for(int i = 1; i <= k; i ++)
    {
        int tmp;
        cout << "Please enter the vertices of level " << i << " (enter 0 to end): ";
        while(cin >> tmp && tmp)        //输入 0 结束
        {
            t[i].push_back(tmp);
        }
    }

    memset(f, 0x3f, sizeof f);
    f[1][0] = 0;
    dp(k, 0);
    cout << "The minimum cost is: " << f[k][0] << endl;

    for(int i = n; pre[i]; i = pre[i]) //倒序路径
        path.push_back(pre[i]);
    reverse(path.begin(), path.end());
    path.push_back(n);
    cout << "The path is: ";
    for(auto x : path)
        cout << x << ' ';

    return 0;
}
```

2.2 记忆化搜索 DFS

```
#include<bits/stdc++.h>
using namespace std;

const int N = 510, INF = 0x3f3f3f3f;

int h[N], e[N], ne[N], w[N], idx; //链式前向星
int n, m;
int pre[N];
vector<int> path;
int f[N];

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}
```

```
int dp(int u)
{
	int &v = f[u];
	if(v != INF)	//已被搜索过
		return v;
	v = 1e9; //一个较大的数但不能是 INF
	for(int i = h[u]; i != -1; i = ne[i])
	{
		int j = e[i];
		int t = dp(j) + w[i];
		if(t < v)
		{
			v = t;
			pre[u] = j;
		}
	}
	return v;
}

int main()
{
	cin >> n >> m;
	memset(h, -1, sizeof h);
	while(m --)
	{
		int x, y, z;
		cin >> x >> y >> z;
		add(y, x, z); //建立反向边
	}
	memset(f, 0x3f, sizeof f);
	f[1] = 0;
	dp(n);

	if(f[n] > INF / 2) //没有最短路径，不连通
	{
		cout << "The minimum cost is: -1" <<endl;
		return 0;
	}
	cout << "The minimum cost is: " << f[n] << endl;

	for(int i = n; pre[i]; i = pre[i]) //倒序路径
		path.push_back(pre[i]);
	reverse(path.begin(), path.end());
	path.push_back(n);
```

```
	cout << "The path is: ";
	for(auto x : path)
		cout << x << ' ';

	return 0;
}
```

- 方案 3：贪心算法

3.1 Dijkstra 算法

```
#include<bits/stdc++.h>
using namespace std;

typedef pair<int, int> PII;
const int N = 510;

int h[N], e[N], ne[N], w[N], idx; //链式前向星
int n, m;
bool st[N];
int d[N];
int pre[N];
vector<int> path;

void add(int a, int b, int c)
{
	e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

int dijkstra()
{
	memset(d, 0x3f, sizeof d);
	d[1] = 0;
	priority_queue<PII, vector<PII>, greater<PII>> heap; //小根堆
	heap.push({0, 1});
	while(heap.size())
	{
		auto x = heap.top();
		heap.pop();
		auto t = x.second, dist = x.first;
		if(st[t])
			continue;
		st[t] = true;
		for(int i = h[t]; i != -1; i = ne[i])
		{
			int j = e[i];
```

```
                if(d[j] > dist + w[i])
                {
                    d[j] = dist + w[i];
                    pre[j] = t;
                    heap.push({d[j], j});
                }
            }
        }
        return d[n] == 0x3f3f3f3f ? -1 : d[n];
}

int main()
{
        cin >> n >> m;
        memset(h, -1, sizeof h);
        while(m --)
        {
            int x, y, z;
            cin >> x >> y >> z;
            add(x, y, z);
        }
        cout << "The minimum cost is: " << dijkstra() << endl;

        for(int i = n; pre[i]; i = pre[i]) //倒序路径
            path.push_back(pre[i]);
        reverse(path.begin(), path.end());
        path.push_back(n);
        cout << "The path is: ";
        for(auto x : path)
            cout << x << ' ';

        return 0;
}
```

3.2 优化后算法

```
#include<bits/stdc++.h>
using namespace std;

const int N = 510;

int h[N], e[N], ne[N], w[N], idx; //链式前向星
int n, m;
int d[N];
int pre[N];
```

```
vector<int> path;

void add(int a, int b, int c)
{
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

int greedy()
{
    memset(d, 0x3f, sizeof d);
    d[1] = 0;
    queue<int> q;
    q.push(1);
    while(q.size())
    {
        auto t = q.front();
        q.pop();
        for(int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if(d[j] > d[t] + w[i])
            {
                d[j] = d[t] + w[i];
                pre[j] = t;
                q.push(j);
            }
        }
    }
    return d[n] == 0x3f3f3f3f ? -1 : d[n];
}

int main()
{
    cin >> n >> m;
    memset(h, -1, sizeof h);
    while(m --)
    {
        int x, y, z;
        cin >> x >> y >> z;
        add(x, y, z);
    }
    cout << "The minimum cost is: " << greedy() << endl;

    for(int i = n; pre[i]; i = pre[i]) //倒序路径
```

```
            path.push_back(pre[i]);
        reverse(path.begin(), path.end());
        path.push_back(n);
        cout << "The path is: ";
        for(auto x : path)
            cout << x << ' ';

        return 0;
}
```

时间复杂度与空间复杂度的详细分析：

（1）方案 1：递归与蛮力法

时间复杂度：

对于图采用邻接表存储，每次递归遍历当前结点邻接表内所有边连接的结点，递归次数为 m 次，因此时间复杂度为 O(nm)。

空间复杂度：

算法过程中使用的额外空间均为一维数组，因此空间复杂度为 O(n)。

（2）方案 2：递归与动态规划

时间复杂度：

2.1 虽然图的存储使用了邻接矩阵，但是因为存储了每个顶点的层级，即每次寻找边时不需要遍历邻接矩阵，且使用了记忆化搜索，因此对于图中每条边只会遍历一次，因此时间复杂度为 O(m)。

2.2 因为采用了记忆化搜索，并且对于图使用邻接表存储，因此同样对于每个边只会遍历一次，因此时间复杂度为 O(m)。

空间复杂度：

2.1 算法额外使用了二维变长数组 vector 存储顶点的层级，总存储顶点数为 n 因此为 O(n)，一维数组存储路径 O(n)，并且递归层数为 k 层 O(k)，因此空间复杂度为 O(n)。

2.2 算法过程中使用的额外空间均为一维数组 O(n)，且递归过程对于每条边递归计算一次，即递归层数为 m 层 O(m)，因此空间复杂度为 O(m)。

（3）方案 3：贪心算法

时间复杂度：

3.1 堆优化的 Dijkstra 算法：算法每次选择需要更新的点通过小根堆进行选择为 O(1)，对于极限状态每次更新所有点加入小根堆，故最多 n ^ 2 个点加入小根堆，因此对于每一次更新小根堆时间复杂度为 O(2logn)，最多会进行 m 次，因此时间复杂度为 O(mlogn)。

3.2 优化的贪心算法，通过算法分析得，算法执行过程中对于每条边只会遍历一次，因此时间复杂度为 O(m)。

空间复杂度：

算法过程中使用的额外空间均为一维数组，因此空间复杂度为 O(n)

算法设计细节的具体分析、运行结果分析和截图 (不少于 400 字):

(1) 方案 1、递归与蛮力法

算法具体分析:

1、用一个状态数组 state 记录是否找到了源点到该节点的最短距离，初始时 state 数组全为 false。

2、首先以一个未被访问过的顶点作为起始顶点，沿当前顶点的边走到未访问过的顶点。

3、当没有未访问过的顶点时，则回到上一个顶点，继续试探别的顶点，直到所有的顶点都被访问过。

4、访问到最终汇点时，更新最小成本，当最小成本更新时记录当前路径。

算法运行结果:

```
PS E:\Compile\C\duoduantu> cd "e:\Compile\C\duoduantu\" ; if ($?) { g++ dfs.cpp -o dfs } ; if ($?) { .\dfs }
12 21
1 2 9
1 3 7
1 4 3
1 5 2
2 6 4
2 7 2
2 8 1
3 6 2
3 7 7
4 8 11
5 7 11
5 8 8
6 9 6
6 10 5
7 4 9
7 10 3
8 10 5
8 11 6
9 12 4
10 12 2
11 12 5
The minimum cost is: 16
The path is: 1 2 7 10 12
PS E:\Compile\C\duoduantu>
```

(2) 方案 2、递归与动态规划

算法具体分析:

2.1 由分析的算法的状态转移方程为：f[i][j] = min(f[i - 1][0 ... k] + w[0 ... k])。具体实现为：

1、使用记忆化数组 f[N][N]，记录每个点的最短距离，初始化为无穷大，在初始化图的过程中建立反向边，并且读入每个顶点的层级关系。

2、递归的去计算每一层的结点到起点的距离，取最小值为当前点的距离。

3、在递归时如果此点已经被算过了，即 f[i][j]被更新过，就直接返回记忆化数组的值，不需要再次向后计算。

2.2 即使用记忆化数组 f[N]将每次 dfs 的状态保存下来，并且同样在建图时建立反向边，自底向上进行 dfs。具体实现为：

1、使用记忆化数组 f[N]，记录每个点的最短距离，初始化为无穷大，在初始化图的过程中建立反向边。

2、递归的去计算连接到当前点的上一层结点的距离，取最小值为当前点的距离。

3、在递归时如果此点已经被算过了，即 f[i]被更新过，就直接返回记忆化数组的值，不需要再次向后计算。

算法运行结果：

```
PS E:\Compile\C\duoduantu> cd "e:\Compile\C\duoduantu\" ; if ($?) { g++ RecursiveDP.cpp -o RecursiveDP } ; if ($?) { .\RecursiveDP }
12 21
1 2 9
1 3 7
1 4 3
1 5 2
2 6 4
2 7 2
2 8 1
3 6 2
3 7 7
4 8 11
5 7 11
5 8 8
6 9 6
6 10 5
7 4 9
7 10 3
8 10 5
8 11 6
9 12 4
10 12 2
11 12 5
Please enter the level of the graph: 5
Please enter the vertices of level 1 (enter 0 to end): 1 0
Please enter the vertices of level 2 (enter 0 to end): 2 3 4 5 0
Please enter the vertices of level 3 (enter 0 to end): 6 7 8 0
Please enter the vertices of level 4 (enter 0 to end): 9 10 11 0
Please enter the vertices of level 5 (enter 0 to end): 12 0
The minimum cost is: 16
The path is: 1 3 6 10 12
PS E:\Compile\C\duoduantu>
```

```
PS E:\Compile\C\duoduantu> cd "e:\Compile\C\duoduantu\" ; if ($?) { g++ MemorySearch.cpp -o MemorySearch } ; if ($?) { .\MemorySearch }
12 21
1 2 9
1 3 7
1 4 3
1 5 2
2 6 4
2 7 2
2 8 1
3 6 2
3 7 7
4 8 11
5 7 11
5 8 8
6 9 6
6 10 5
7 4 9
7 10 3
8 10 5
8 11 6
9 12 4
10 12 2
11 12 5
The minimum cost is: 16
The path is: 1 2 7 10 12
PS E:\Compile\C\duoduantu>
```

（3）方案 3、贪心算法

算法具体分析：

3.1 贪心算法采用了 Dijkstra 算法，具体实现为：

1、用一个 dist 数组存储源点到其余各个节点的距离，初始时 dist 数组，源点距离为 0 即 dist[1] = 0，其余各个元素为无穷大。用一个状态数组 state 记录是否找到了源点到该节点的最短距离，初始时 state 数组全为 false。

2、遍历 dist 数组，找到一个节点，这个节点是：没有确定最短路径的节点中距离源点最近的点，将该结点 state 置为 true。在遍历结点时可以采用最小堆进行优化，具体实现为将每次更新选择的点加入堆中，不断循环直到堆空，每次循环操作为：弹出堆顶，用该点更新临界点的距离，如果更新成功就加入堆中。

3、通过找到的节点 i 对其他节点进行更新，遍历 i 所有可以到达的节点 j，如果 dist[j] 大于 dist[i] 加上 i 到 j 的距离，即 dist[j] > dist[i] + w[i][j]（w[i][j] 为 i 到 j 的距离），则更新 dist[j] = dist[i] + w[i][j]，使用一个 pre 数组记录此点从哪个点更新的距离。

4、重复 23 步骤，直到所有点都被选择（即 state 数组均为 true），此时 dist 数组中存储的即为源点到各个点的最短距离。pre 数组存储即为该点的前一个路径。

3.2 因为本题图为多段图，根据 dijkstra 算法进行更新时当前结点更新距离只会更新下一层的距离，因此我们可以进行优化，只需要顺序更新所有点，最终结果即为最优解。此时问题转换为广度优先搜索问题，不需要使用小根堆进行排序选择，只需使用一个队列 q 将所有扩展到的点记录，优化了排序的过程，每次从队列中弹出队头进行最短距离的更新，并将新扩展到的结点加入队列。

算法运行结果：

```
PS E:\Compile\C\duoduantu> cd "e:\Compile\C\duoduantu\" ; if ($?) { g++ Dijkstra.cpp -o Dijkstra } ; if ($?) { .\Dijkstra }
12 21
1 2 9
1 3 7
1 4 3
1 5 2
2 6 4
2 7 2
2 8 1
3 6 2
3 7 7
4 8 11
5 7 11
5 8 8
6 9 6
6 10 5
7 4 9
7 10 3
8 10 5
8 11 6
9 12 4
10 12 2
11 12 5
The minimum cost is: 16
The path is: 1 3 6 10 12
PS E:\Compile\C\duoduantu>
```

```
问题    输出    调试控制台    终端

PS E:\Compile\C\duoduantu> cd "e:\Compile\C\duoduantu\" ; if ($?) { g++ greedy.cpp -o greedy } ; if ($?) { .\greedy }
12 21
1 2 9
1 3 7
1 4 3
1 5 2
2 6 4
2 7 2
2 8 1
3 6 2
3 7 7
4 8 11
5 7 11
5 8 8
6 9 6
6 10 5
7 4 9
7 10 3
8 10 5
8 11 6
9 12 4
10 12 2
11 12 5
The minimum cost is: 16
The path is: 1 2 7 10 12
```

个人总结 (不少于 200 字)：

在解决多段图问题的算法编写过程中，我收获颇丰。首先在算法的编写中对递归、动态规划等算法思想有了更深的理解，并且通过课上的学习，我掌握了很多经典算法的算法思想以及实现方式，这对于未来的研究以及工作都有很大帮助。其次在程序编写过程中，对于各种数据结构的应用的理解也更近一步。最后对于一个问题，可以使用不同的算法进行解决，但通过分析各个算法的时间复杂度与空间复杂度，有助于选择合适的算法来更高效、更便捷的解决问题。作为计算机相关专业的学生，学习算法设计与分析是非常重要的，可以锻炼个人的逻辑思维能力，与解决实际问题的实践能力。