Floyd-Warshall 算法扩展研究——一种获取路径信息及处理路径更新的方法

北京市陈经纶中学 杨熙坤

摘要

Floyd-Warshall 是处理全源最短路径问题最有效的算法，该算法在计算所有城市之间的交通道路距离时将出现。但 Floyd-Warshall 最终得到的答案只是结点对之间的最短路径长度数值，并无法具体说明最短路径的结构。另外，如果给每一个结点、部分边设置“开放时间”的限制条件，即不同“时间点”图上可访问的结点、边集合不同，这时候的 Floyd-Warshall 算法需要改进。本文针对此对 Floyd-Warshall 算法进行了一定改进，合并解决了这两个问题——实现了输出在任何一个时间点任意一对结点的最短路径结构及长度的算法及程序。

关键词：最短路径，Floyd-Warshall，图论

目录

一、问题提出、文章约定、问题转化

1.1 问题提出

本文在于**合并解决**图论全源最短路径的三个问题：

1. 如何用 Floyd-Warshall 算法输出一个结点到另一个结点的最短路径结构（即这条路径经过了哪些中间结点）？这在需要获得城市之间最短路径结构时将出现。
2. 如果每个结点设置了“开放时间”的限制条件，即不同时间点时图上可访问的结点集合不同，如何计算在某一时间点一个结点到另一个结点的最短路径？
3. 如果部分边也设置了“开放时间”的限制条件，即不同时间点时图上可访问的边集合不同，如何计算在某一时间点一个结点到另一个结点的最短路径？

本文提出了合并解决上述三个问题的算法——该算法的目标是获得任何一个“时间点”图上任意一对结点的最短路径结构及长度。

1.2 文章表述的约定

本文将采用伪代码，值得注意的一点是，针对一个量的属性，本文将采用点标注法（例如量 n 具有属性 m，则表述 n 的属性 m 时将使用 $n.m$）。为保证伪代码的普遍性，伪代码将不使用大括号 ({}) 来区分代码级别，而是使用缩进格式。另外，本文的 C++代码使用控制台程序 (console application)，在任何支持 STL 的 C++ IDE 上都可以正常运行。

在本文的图论分析中，表示从结点 i 到 j 的无向边记为 $i—j$，有向边、从 i 到 j 的路径记为 $i→j$。

1.3 对问题的简单分析、转化、重要概念说明

经过简单分析，发现问题 3 可以转化为问题 2。假设存在未来开放的边为 e，连接两个结点 $i—j$，其开放时间为 t，其权重为 w，那么将其处理为添加一个开放时间为 t 的结点 k，设置两条新边 $i—k$ 和 $k—j$，使得两条新边权重均不小于 0、且权重之和为 w 即可。那么实际上我们只需要设计算法解决上述 3 个问题中的前两个问题。为了模型的简洁性，下文的表述、模型、算法将不再考虑问题 3 的存在及转化，假定只有结点有开放时间 t 的概念。

开放时间 t 是一个数值，越早开放的结点，其 t 越小；越晚开放的结点，其 t 越大。如果最开始此结点就是开放的，则该结点的 t=0。t 可以是结点间开放时间顺序相对值也可以是由具体时间数值转化而来的一个实数。为方便起见，C++程序定义的 t 是整数。

二、最短路径和 Floyd-Warshall 算法

2.1 最短路径的概念

最短路径的定义非常简单——在图上确定 u, v 两个结点，找到一条路径从 u 出发，到 v 结束的路径 p，使得 p 所经过的边的权重之和最短，这样的路径 p 称为 $u→v$ 的最短路径。在很多时候，人们讨论边的权重能否为负数。但在问题情境中，边的权重对应的是长度信息，所以不可能是负数。

2.2 Floyd-Warshall 算法描述

Floyd-Warshall 算法（下文简称 Floyd 算法）是解决全源最短路径最有效的方法，该算法在计算所有城市之间的交通道路距离时将出现[1]。Floyd 算法依赖于动态规划 (dynamic programming) 思想——考虑一条最短路径上的中间结点。通过不断考虑新中间结点来更新不同结点对之间的最短路径，最终获得准确值。Floyd 的算法的时间复杂度是 $O(|V|^3)$，其中$|V|$是图的结点数。

Floyd 算法基于下列观察[1]：

对于任意的一对结点 i,j，假设 $i\to j$ 的路径上的点均取自集合$\{1,2,...,k\}$，在这样的情况下的最短路径为 p。现在考虑 p 和 $i\to j$ 的路径上的点均取自集合$\{1,2,...,k\text{-}1\}$的最短路径的关系。

i. 如果 k 不是 p 上的中间结点，则 p 上的所有中间结点属于$\{1,2,...,k\text{-}1\}$。因此，$i\to j$ 的中间结点取自$\{1,2,...,k\text{-}1\}$的一条最短路径也是 $i\to j$ 的 中间结点取自$\{1,2,...,k\}$的一条最短路径。

ii. 如果 k 是 p 上的中间结点，那么 $i\to j$ 的最短路径 p 可以分解为 $i\to k$ 和 $k\to j$。$i\to k$ 的所有中间结点都取自$\{1,2,...,k\text{-}1\}$，$k\to j$ 的所有结点也遵守相同的规则。参考文献[1]给出了一张图（*fig I*）来解释此问题。

根据上述观察，可以得到递推公式。假设 $d^s[i][j]$表示最新考虑的一个中间结点编号为 s 的情况下 $i\to j$ 的最短路径长度，那么考虑新中间结点的过程中，路径更新过程如下：

$$d^k[i][j]=\min(d^{k-1}[i][j],\ d^{k-1}[i][k]+d^{k-1}[k][j]) \tag{1}$$

如果最终 $d^k[i][j]= d^{k-1}[i][k]+ d^{k-1}[k][j]$，这个将称为一次松弛 (relax) 操作。松弛是不断更新最短路径的关键。

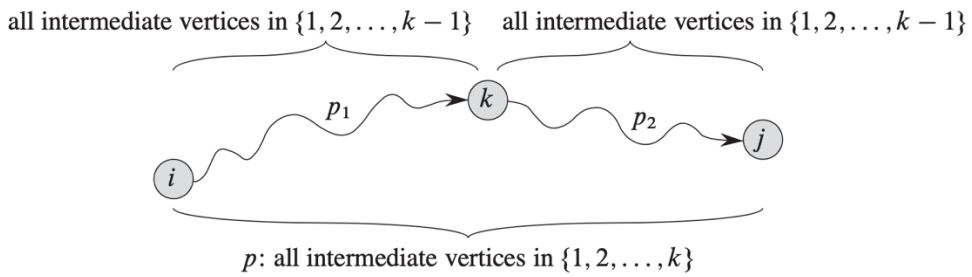

fig I

在实际程序编写中，往往不需要复杂的递推公式[1]，只需要三层循环即可实现。假设 $d[i][j]$表示 $i\to j$ 的最短路径（会发生变化），k 为中间结点，$|V|$为图结点数。伪代码可写为：

```
for k=1 to |V| // Add new nodes into consideration
    for i=1 to |V| // Starting point
        for j=1 to |V| // Destination
            d[i][j]=min(d[i][j], d[i][k]+d[k][j])
```

代码的最后一行即为松弛操作，我所设计的算法在松弛这一步骤上有改进，实现了更多功能。

三、构建算法和程序求解

3.1 数据结构定义

定义一张无向图 $G=(V,E)$（V 为结点集合，E 为边集合）。随着时间的推移，图中的结点会被逐一开放；每条边拥有一个长度信息（权重）w。一个合乎情理的约定：在结点未开放之前，这个结点不可访问，连接这个未开放结点的边也全部不可访问。

本模型将主要采用 Floyd 算法求解。Floyd 基于一张以邻接矩阵形式存储的图，在真正实践中，将其设置为一个二维数组 adj 即可（adj 代表 ***adj****acency matrix*）。其中，$adj[i][j]$表示当前情况下从 i 到 j 的最短路径（此数组是随时更新的）。adj 的初始化参见 3.3.3。

此外，因为求解需要，还需要一个空间来存储结点间的最短路径结构（通过描述路径经过了哪些结点（包括起点和终点））。由于路径新加结点需要经常从尾部“推入”元素，因此使用队列 (queue) 类数据结构来处理最为合适。在 C++程序中，考虑到 STL::vector 的灵活性大于 STL::queue，因此使用 vector 实现此功能。

Floyd 的每一步依赖于之前的答案，因此每一步的答案都需要存储，使用二维数组嵌套队列最为合适——设数组 path，$path[i][j]$表示当前情况下 $i\to j$ 的最短路径结构。根据描述，path 数组的初始化应该是这样的（意义是一短路径必将从自身开始）：

$$path[i][j]=\{i\},\ i,j=1,2,...,|V|\text{-}1,|V| \tag{2}$$

3.2 样本输入方法

输入应当保证有：

i. 每个结点的开放时间 *t*

ii. 每条边的数据，包括两端连接的结点、边的权重 *i,j,w*（下文将以(*i,j,w*)描述一条边）

3.3 样本处理方法

3.3.1 结点的处理方法

由于 Floyd 的算法包括三层循环，最外一层循环是新考虑的中间结点，按照结点编号从 1~|*V*|循环。因为 Floyd 的性质，要保证每个结点的开放时间 *t* 满足如下约束：越早开放的结点越排在前面（参见 3.5）

$$t_1 \le t_2 \le ... \le t_{|V|-1} \le t_{|V|} \tag{3}$$

但是输入并不见得遵照如此完美的顺序，因此需要对输入进行一些处理——映射(cast)。下文将称呼此处理方法为映射方法 (casting method)。

映射的意思是把每一个输入的结点编号（原始值）唯一对应一个新编号（映射值）（就像函数的 *x* 对应 *f*(*x*)一样）。把输入的结点原始值按照开放时间 *t* 从小到大排序，然后把它们分别对应成新编号 1, 2, ..., |*V*|-1, |*V*|（这样可以保证唯一性）。这样就完成了映射。在未来的算法运行中，使用映射值而非原始值，因为映射值是按照结点开放时间排序的。

因为实际需求，需要既可以从原始值获取映射值，也可以从映射值获取原始值。

i. 映射值获得原始值

在实践中，使用二元组(*node*, *t*)来描述每一个输入的结点开放时间。其中 *node* 表示结点编号，*t* 表示这个结点的开放时间。这些二元组将存储在数组 original 中。在存储完之后，使用排序算法按照 *t* 的顺序对 original 进行排序。排序之后的 original 将满足条件——original[*i*]表示第 *i* 小开放时间的原始结点的相关情况。只需要调取 original[*i*].*node* 就可以知道一个结点的映射值编号对应的原始值编号是什么。

ii. 原始值获得映射值

开设数组 cast，cast[*i*]表示原始值 *i* 对应的映射值。用以下方法求得 cast 中的元素：

$$\text{cast}[\text{original}[i].node]=i,\ i=1,2,...,|V|-1,|V| \tag{4}$$

3.3.2 C++实现结点的处理方法

i. 全局部分代码（定义结点存储结构）：

```
struct Node
{
    int node;
    int t;
    // There can be more information

    bool operator < (const Node &a) const { return this->t<a.t; } // For sorting
};
Node original[max_node_num];
int cast[max_node_num];
```

ii. 主函数内部代码（表达结点 original 和 cast 的生成方法）：

```
// Process nodes data
for (int i=1; i<=n; i++)
{
    int time;
    cin>>time;
    original[i].node=i; original[i].t=time;
}
```

```
// Casting
sort(original+1,original+n+1); // Sorting
for (int i=1; i<=n; i++) cast[original[i].node]=i;
```

3.3.3 边的处理方法

对于边的存储方法需要做出如下处理：

i. 先将 adj 数组初始化为 INF（即无穷大，在实践中，只需要设为 1e7 规模的大数即可）

$$adj[i][j]=INF,\ i,j=1,2,...,|V|-1,|V| \tag{5}$$

ii. 因为自身到自身的最短路径永远是 0，所以设置

$$adj[i][i]=0,\ i=1,2,...,|V|-1,|V| \tag{6}$$

iii. 然后按照输入的每条边(i,j,w)赋值（注意图是无向边）

$$adj[cast[i]][cast[j]]=adj[cast[j]][cast[i]]=w \tag{7}$$

3.3.4 C++实现边的处理方法、path 数组的处理方法

以下代码放置于主函数中，且在结点处理结束之后运行。关于 path 数组的意义，参见 3.1；关于 path 数组在之后是如何工作的，参见 3.4。

```
// Initialize the adj[][] and path[][]
for (int i=1; i<=n; i++) for (int j=1; j<=n; j++) adj[i][j]=INF;
for (int i=1; i<=n; i++)
{
    adj[i][i]=0;
    for (int j=1; j<=n; j++) path[i][j].push_back(i);
}
// input edges
for (int i=1; i<=m; i++)
{
    int from,to,weight;
    cin>>from>>to>>weight;
    from=cast[from], to=cast[to]; // Cast the original value into cast value
    adj[from][to]=adj[to][from]=weight;
    path[from][to].push_back(to); path[to][from].push_back(from);
}
```

3.4 求解最短路径结构

注：本节提到的所有结点均指的是映射值。

3.1 提到，可以使用队列类数据结构（C++程序使用 STL::vector）来维护一段路径。

观察到 Floyd 算法应用了动态规划的思想——即每次选取一个中间结点，然后进行松弛 (relax) 操作。动态规划的进行依赖于之前的答案，然后根据规律推出本次的答案。根据这样的思想，可以做出一个大致规划——当考虑一个新中间结点的时候，根据之前循环记录的最短路径结构，再加上本次循环的松弛信息，得到本次循环的最短路径结构。

给出如下策略：在进行松弛操作的时候，假设当前的中间结点为 k，正在考虑 $i\sim j$ 的路径，如果 $adj[i][k]+adj[k][j]<adj[i][j]$，需要对 $path[i][j]$进行更新，伪代码如下：

```
clear path[i][j]
push(path[i][k]) to the back of path[i][j]
pop the last element of path[i][j] (i)
push(path[k][j]) to the back of path[i][j]
```

注释：
(*i*) 下次推入 $k \rightarrow j$ 的路线还会包括元素{k}，所以先弹出尾部

这样就完成了对 path[i][j]的更新。现在 path[i][j]就是当前中间结点下的最短路径结构了。

伪代码的操作在 C++中使用 vector 很容易实现：

```
// Update the path
path[i][j].clear();
for (int cur=0; cur<=path[i][k].size()-1; cur++)
       path[i][j].push_back(path[i][k][cur]);
path[i][j].pop_back();
for (int cur=0; cur<=path[k][j].size()-1; cur++)
       path[i][j].push_back(path[k][j][cur]);
```

3.5 处理新开放的结点及需要映射方法的理由

新结点被开放在程序中体现为考虑新的中间结点，因为新结点被开放就等于可以通过本结点了，也就是说更新最短路径数据就要考虑这个新结点了。考虑新的中间结点后，最短路径可能发生变化——一个重要结论是，后考虑的结点对应的数据一定是基于前考虑的结点对应的数据。

更直观地说，第二个新开放的结点对整张图的改变一定是基于第一个新开放的结点的。如果在第一个结点开放之前就开放第二个结点，那么得到的答案不正确。此即为需要映射方法的理由：如果不映射，可能考虑的中间结点先是第二个开放的结点，然后才是第一个开放的结点。这就是为什么如果新结点开放顺序和中间结点考虑顺序不一样，那么答案将不正确。

但注意 Floyd 有三层循环，第二层、第三层循环的变量 i,j 代表起点和终点。在考虑此中间结点之前，也就是这个结点还未开放之前，第二层、第三层循环也必须包含这个未开放结点。因为在这个结点开放之后，所有计算都需要用到这个结点和其他结点之间的路径数据，如果之前就忽略掉了，此处将得不到所需数据。

3.6 算法实现、程序实现

3.6.1 伪代码实现

下面给出主函数程序实现的伪代码。

```
// Process nodes data
input original[] // Input the original version
sort original[]
get data of cast[] from original[]

// Process edges data
initialize adj[][]
input and store the casted numbers into adj[][]

// Floyd program
for k=1 to |V| // Add new nodes into consideration
       for i=1 to |V|
              for j=1 to |V|
                     if adj[i][k]+adj[k][j]<adj[i][j]
                            adj[i][j]=adj[i][k]+adj[k][j]
                            update the path of i→j
       if (k<n and original[k+1].t > original[k].t) or k==n (i)
              output or store the target information
              continue
```

注释：
(*i*) 本代码指定了什么时候输出答案——在本时间点开放的所有中间结点都计算完成之后。

3.6.2 C++程序输出指定信息

由于本算法可以获得大量信息，因此给出其中一个版本的 C++代码——下列代码的目标是求出从 1 号结点到|*V*|号结点（原始值）在不同时间的最短路径结构以及最短路径长度。如果存在其他需求，进行一定更改即可实现。注意下文 C++代码中的量 *n* 指的是图结点数（等价于|*V*|）。

在这个版本的 C++代码中，伪代码的 **`output`** 具体为输出“从 1 号结点到|*V*|号结点（原始值）在不同时间的最短路径结构以及最短路径长度”。因此在最终处理的时候，需要反复用到映射方法的两个数组 original 和 cast。

本段代码嵌入 Floyd 算法的在第一层循环内，在第二、三循环运行结束后开始运行。C++具体实现为：

```
if ((k<n && original[k+1].t>original[k].t) || k==n) (i)
{
    cout<<"In the time "<<original[k].t<<", the shortest path from 1 to "<<n<<" is: "; (ii)
    if (path[cast[1]][cast[n]].back()!=cast[n] || original[cast[n]].t>original[k].t) (iii)
    {
        cout<<"Unreachable."<<endl;
        continue;
    }
    cout<<original[path[cast[1]][cast[n]][0]].node; (iv)
    if (path[cast[1]][cast[n]].size()>1) (v)
        for (int cur=1; cur<=path[cast[1]][cast[n]].size()-1; cur++)
            cout<<"→"<<original[path[cast[1]][cast[n]][cur]].node;
    cout<<", and the length of this path is: "<<adj[cast[1]][cast[n]]<<endl; (vi)
}
```

注释：
(*i*) 本代码指定了什么时候输出答案——在本时间点开放的所有中间结点都计算完成之后。
(*ii*) 本代码输出提示信息，表示在× ×时间点时结点 1→|*V*|（实际值）。
(*iii*) 本代码 `if` 语句的目标是判断能否到达目标结点，其内部代码指明：如果不能到达目标结点，程序将要做什么。如果路径图的最后一个元素不是目标结点，或者目标结点在当前之间点尚未开放，将被判断为不可到达。在判断为不可到达之后，首先输出不可到达，然后开始计算下一个中间结点(`continue`)。
(*iv*) 本代码目标是输出最短路径的第一个位置（实际值）
(*v*) 本代码 `if` 语句及其内部语句的目标是输出最短路径的其他位置
(*vi*) 本代码的目标是输出最短路径长度

3.6.3 C++完整程序实现

下面给出 C++解决本问题并输出 3.6.2 指定信息的实际代码。一些内容需要使用者输入，包括图结点数 (*n*)，图边数 (*m*)，每个结点的开放时间 (*t*)，以及每条边的情况 (*i*,*j*,*w*)。

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <cstring>
using namespace std;
const int INF=1e7;
const int max_node_num=5005; // Max size of nodes
```

```
struct Node
{
    int node;
    int t;
    // There can be more information
    bool operator < (const Node &a) const { return this->t<a.t; } // For sorting
};
Node original[max_node_num];
int cast[max_node_num];
int adj[max_node_num][max_node_num];
vector <int> path[max_node_num][max_node_num];

// These variations are for input
int n; // Size of nodes
int m; // Size of edges

int main()
{
    cin>>n>>m;

    // Input nodes & Process nodes
    for (int i=1; i<=n; i++)
    {
        int time;
        cin>>time;
        original[i].node=i; original[i].t=time;
    }

    // Casting
    sort(original+1,original+n+1);
    for (int i=1; i<=n; i++) cast[original[i].node]=i;

    // Process edges
    // Initialize the adj[][] and path[][]
    for (int i=1; i<=n; i++) for (int j=1; j<=n; j++) adj[i][j]=INF;

    for (int i=1; i<=n; i++)
    {
        adj[i][i]=0;
        for (int j=1; j<=n; j++) path[i][j].push_back(i);
    }
    // Input edges
    for (int i=1; i<=m; i++)
    {
        int from,to,weight;
        cin>>from>>to>>weight;
        from=cast[from], to=cast[to]; // Cast the original value into cast value
        adj[from][to]=adj[to][from]=weight;
        path[from][to].push_back(to); path[to][from].push_back(from);
    }
```

```
    // Floyd
    for (int k=1; k<=n; k++)
    {
        for (int i=1; i<=n; i++)
            for (int j=1; j<=n; j++)
                if (k!=i && k!=j && i!=j)
                    if (adj[i][k]+adj[k][j]<adj[i][j]) // Satisfies the relaxation conditions
                    {
                        adj[i][j]=adj[i][k]+adj[k][j];
                        // Update the length of the shortest path

                        // Update the path
                        path[i][j].clear();
                        for (int cur=0; cur<=path[i][k].size()-1; cur++)
                            path[i][j].push_back(path[i][k][cur]);
                        path[i][j].pop_back();
                        for (int cur=0; cur<=path[k][j].size()-1; cur++)
                            path[i][j].push_back(path[k][j][cur]);
                    }
         // Get answer
        if ((k<n && original[k+1].t>original[k].t) || k==n)
        {
            cout<<"In the time "<<original[k].t<<", the shortest path from 1 to "<<n<<" is: ";
            if (path[cast[1]][cast[n]].back()!=cast[n] || original[cast[n]].t>original[k].t)
            {
                cout<<"Unreachable."<<endl;
                continue;
            }
            cout<<original[path[cast[1]][cast[n]][0]].node;
            if (path[cast[1]][cast[n]].size()>1)
                for (int cur=1; cur<=path[cast[1]][cast[n]].size()-1; cur++)
                    cout<<"→"<<original[path[cast[1]][cast[n]][cur]].node;
            cout<<", and the length of this path is: "<<adj[cast[1]][cast[n]]<<endl;
        }
    }

    return 0;
}
```

3.7 测试正确性

注：本处对正确性的测试指的是测试 C++程序的正确性（伪代码正确性的理论分析已经完成）。

3.7.1 测试映射方法的正确性

为了测试映射方法（包括 original 数组和 cast 数组）能否正常工作，只需要在结点数据输入完成后查看 original 数组和 cast 数组各元素是否正确即可。

假设输入五个结点，其对应的 *t* 值分别为 41, 32, 23, 55, 9，可以推测出 original 数组的构造应该是 original={(5,9), (3,23), (2,32), (1,41), (4,55)}，如果只显示每个二元组的 *node* 信息，那应该是{5,3,2,1,4}。针对 cast 数组，它应该是 cast={4,3,2,5,1}。

使用如下 C++代码验证：

```
for (int i=1; i<=5; i++) cout<<original[i].node<<" ";
cout<<endl;
for (int i=1; i<=5; i++) cout<<cast[i]<<" ";
```

得到的答案是：

```
41 32 23 55 9
5 3 2 1 4
4 3 2 5 1
```

（绿色为输入内容，第 *i* 个绿色数字表示第 *i* 个结点的开放时间）

与预期结果相同。表明映射方法工作正常。

3.7.2 测试具体路线是否正常工作

3.7.1 已经测试了映射方法可以正确运行。因此在这里不涉及到映射方法的测试。假定输入了三个结点，其对应的 *t* 值分别为 1,2,3。存在这样三条边：(1,2,2), (2,3,2), (1,3,3)。也就是这样的图：

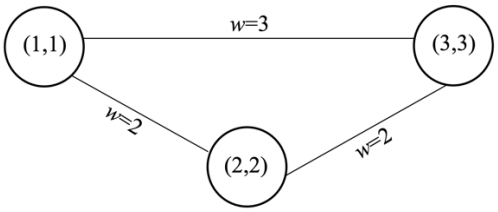

fig II

测试得到的结果是：

```
In the time 1, the shortest path from 1 to 3 is: Unreachable.
In the time 2, the shortest path from 1 to 3 is: Unreachable.
In the time 3, the shortest path from 1 to 3 is: 1→3, and the length of this path is: 3
```

与预期结果相同。

将 *fig II* 的图改为如下：

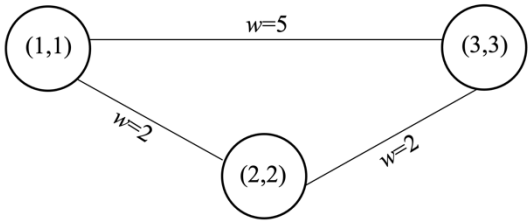

fig III

测试得到的结果是：

```
In the time 1, the shortest path from 1 to 3 is: Unreachable.
In the time 2, the shortest path from 1 to 3 is: Unreachable.
In the time 3, the shortest path from 1 to 3 is: 1→2→3, and the length of this path is: 4
```

与预期结果相同。

3.7.3 测试真实情况下工作是否正常

基于 *fig III* 测试，将图改为如下：

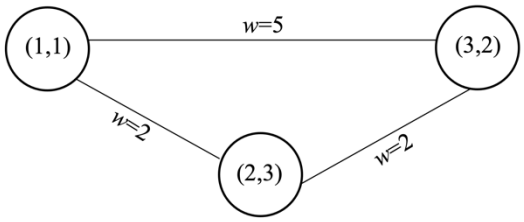

fig IV

可见，前往的目标结点 3 在时间点 2 就已经开放。这时候映射方法也需要工作。

测试得到的结果为：

```
In the time 1, the shortest path from 1 to 3 is: Unreachable.
In the time 2, the shortest path from 1 to 3 is: 1→3, and the length of this path is: 5
In the time 3, the shortest path from 1 to 3 is: 1→2→3, and the length of this path is: 4
```

和预期结果相同。

下面的 *fig V* 表达了一种更为复杂的情况。

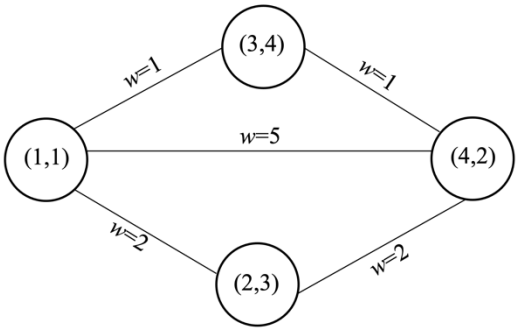

fig V

测试得到的结果为：

```
In the time 1, the shortest path from 1 to 4 is: Unreachable.
In the time 2, the shortest path from 1 to 4 is: 1→4, and the length of this path is: 5
In the time 3, the shortest path from 1 to 4 is: 1→2→4, and the length of this path is: 4
In the time 4, the shortest path from 1 to 4 is: 1→3→4, and the length of this path is: 2
```

和预期结果相同。

除本文列举之外，我还进行了更多测试，本程序均能够正确解决问题。

四、时间复杂度分析和测试情况

4.1 时间复杂度分析

i. 映射方法运算的时间复杂度为 $O(|V|\log|V|+|V|)$（假设使用的是快速排序算法进行排序）。
ii. 经过修改的 Floyd 算法的时间复杂度依然为 $O(|V|^3)$，但相对于原始版本的 Floyd 算法常数有增大。所以图的规模相同的情况下，本算法的运行时间大于原始版本的 Floyd 算法。

4.2 规模数据测试

4.2.1 Floyd 算法的规模问题

已知 Floyd 算法的时间复杂度为 $O(|V|^3)$，因为结构简单，所以复杂度常数比较小，对于中等规模的图也有较好效率[1]。通常情况下如果期望在 1s 内完成 Floyd 算法，图的规模应该保证$|V|\leq 500$。

4.2.2 本算法的规模测试

本文将对提出的算法进行时间复杂度测试。测试环境为 macOS Monterey 12.0.1，配置为 2.7 GHz Quad-Core Intel Core i7 6820HQ, 16GB 2133MHz LPDDR3，所使用的 IDE 为 CLion 2021.2.3, Built #CL-212.5457.51。

下表给出了不同规模的结点和边的情况下本文改进的 Floyd 算法的运行时间。

结点数 (n)	边数 (m)	本文算法平均运行时间 (s)
50	50	0.0035±0.0010
500	500	1.295
500	1 000	2.068
500	2 000	2.413
750	750	4.083
750	1 000	4.931
750	2 000	6.277
1 000	1 000	8.809
1 000	2 000	12.07
2 000	2 000	52.04
2 000	5 000	82.11
5 000	5 000	684.74

Table I. 随着图的大小逐渐加大，算法的运行时间增加逐渐明显。在 $n\geq 1\,000$, $m\geq 1\,000$ 之后，算法的性能有明显下降。另外，映射方法的速度极快，不会对程序运行造成影响（当 $n=5\,000$, $m=5\,000$ 时，映射方法也能够在 0.0013s 完成）。

五、反思

5.1 功能实现情况

通过使用 Floyd 算法并基于 Floyd 算法进行扩展，最终实现了所需要的功能——获得任何一个“时间点”图上任意一对结点的最短路径结构及长度。我们只需要知道整张图最终状态是什么样子的、哪个结点、那个边什么时候开放，就可以知道随着时间变化，路径结构将如何变化。

5.2 优势

本文提出的 Floyd 算法的改进方案并合并解决了 1.1 提出的三个图论问题。

映射方法和具体路径求解方法是本程序的创新之处。映射方法是对结点的有效处理，使 Floyd 算法不会得到错误答案；具体路径求解方法利用了 Floyd 算法的性质，有效地获取到了路径结构。

5.3 不足

随着图的规模逐渐增大，本算法运行效率下降明显，如果有明确的求解时间限制，那么本算法不能够较好满足大规模图的应用。

5.4 后记

通过此研究，我对离散数学的图论 Floyd 算法、动态规划思想有了更深入的认识。某些内容仍需要改进并进行更深入的研究，这将是我下一步前进的方向。

参考文献

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms (Third Edition)