全源最短路径问题的Johnson算法研究

姓 名： 感谢组长及组员的协同配合

组内人员分工或贡献：33.3%、33.3%、33.3%

课 程 名 称：算法设计与分析

指 导 教 师： 王

2021年11月

基于负边更新创新的全源最短路径Johnson算法

（大学 理学院）

1. 作为图论的经典问题，最短路径在诸多领域有着广泛应用，为拓宽适用范围、提升算法的计算效率，针对含负边图全源最短路径问题，在单源最短路径算法基础上，研究以重新赋权的创新理念。本文依据Dijkstra、Bellman-Ford做基础，研究Johnson算法，分析其的时间复杂度。

# 问题的定义与研究现状综述

## 问题的定义

最短路径问题是图论研究中的一个经典算法问题，旨在寻找图（由结点和路径组成的）中两结点之间的最短路径。即从某顶点出发，沿图的边到达另一顶点所经过的路径中，各边上权值之和最小的一条路径。按照解的精度，最短路径算法可分为精确和近似算法，按照求解目标的数量，最短路径问题又可分为单源和全源（多源）最短路径。

对于全源最短路径问题（All-Pairs Shortest Paths Problem），可以认为是单源最短路径问题的推广，即分别以每个顶点作为源顶点并求其至其它顶点的最短距离。

解决多源最短路径问题，因为能够求解单源点 到所有终点 的最短路径，所以就可以用同样的方式求解其他源点到所有终点的最短路径问题，只需要循环轮换源点即可。[1]

## 问题的历史与研究现状

1959 年荷兰计算机科学家狄克斯特拉提出迪杰斯特拉算法 (Dijkstra)。算法解决从一个顶点到其余各顶点的最短路径问题，解决的是有权图中最短路径问题。

1958 年美国应用数学家Richard Bellman (理查德.贝尔曼, 动态规划的提出者)发表了Bellman算法。此外Lester Ford在1956年也发表了该算法，Bellman-Ford算法由此得名。1957年EdwardF. Moore发表同样的算法。Bellman-ford 算法比Dijkstra算法更具普遍性，因为它对边没有要求，可以处理负权边与负权回路。缺点是时间复杂度过高，高达O(VE), V为顶点数，E为边数。

1977 年Donald B. Johnson 提出Johnson 算法，Johnson和 Floyd 一样，是一种能求出无负环图上任意两点间最短路径的算法。Johnson算法结合Dijkstra算法和Bellman-Ford算法整合，依照更新负边权值的创新思想改进而成。

求解单源最短路径有多种不同算法，最经典的算法是Dijkstra、Floyd、A\*算法，这些算法均得到了广泛的应用，如较好地解决交通网络，通信网络、厂局布局、旅游路线推荐等实际问题，同时，针对不同的应用领域涌现出多种经典的最短路径算法以及相应的改进算法，这些算法的设计均各具特色，其中最具有代表性的最短路径算法是Dijkstra算法，其最大的不足，在于不能解决负权图的最短路径问题。[2] 基于单源最短路径求解的算法，进行全源最短路径求解可能性及复杂性的分析与研究。

目前，求解全源最大路径问题的典型算法为Floyd算法。由此可见，全源最短路径算法的时间复杂度依然很高，难以满足现有大规模网络或实时计算的要求，为进一步提升算法的计算效率，除设计较低复杂度的高效算法外，还有学者采用并行计算技术来加速全源最短路径问题的求解，但并行计算不能减少走的计算量。[3]

作为图论的经典问题，最短路径在诸多领域都有着广泛的应用，如智能交通系统，地理信息系统计算机网络与通讯等，为提升算法的计算效率或满足特定的应用需求，已有大量优秀算法被提出，本文的研究对象是精确全源最短路径问题及在求解所有点之间的精确最短路径。提出基于重新赋权的创新理念，依据Dijkstra算法和Bellman-Ford算法进行整合，研究分析Johnson算法。

# 算法原理及计算复杂性阐述

## 算法原理阐述

### Dijkstra

**算法描述**

设G=(V,E)是一个带权有向图，把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合（用S表示，初始时S中只有一个源点，以后每求得一条最短路径 , 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了），第二组为其余未确定最短路径的顶点集合（用U表示），按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

**算法步骤**

（1） 创建一个动态数组，它记录最短路径树中包含的顶点，即计算并最终确定其与源点的最小距离。 初始化时这个数组是空的。

（2） 为输入图中的所有顶点分配一个距离值。 将所有距离值初始化为 INFINITE。 将源顶点的距离值指定为 0，以便它首先被检索。

（3） 动态数组不包括所有顶点

a) 选择一个不存于动态数组中的顶点，它具有与已在动态数组中的顶点的最短距离值。

b) 将该顶点划入到动态数组中。

c) 更新与动态数组所有相邻顶点的距离值。要更新距离值，需要遍历所有相邻顶点。 对于每一个相邻的顶点v，如果u（距源点）的距离值和边uv的权重之和小于v的距离值，则更新顶点v的距离值。

**算法示意**

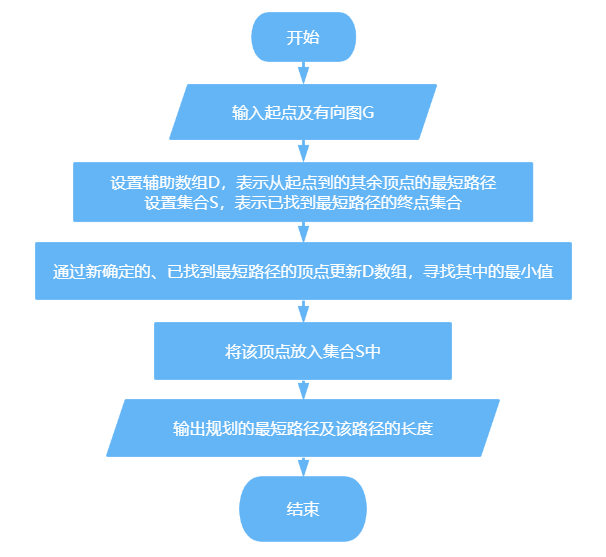
|  |
| --- |
|  |
| 根据算法初始化距离 |

|  |
| --- |
|  |
| 选择第一个节点并计算到相邻节点的距离 |

|  |
| --- |
|  |
| 选择距离最小的下一个节点； 重复相邻节点距离计算 |

|  |
| --- |
|  |
| 最短路径树的最终结果 |

**算法流程**



### Bellman-Ford

为了能够求解边上带有负值的单源最短路径问题，Bellman(贝尔曼，动态规划提出者)和Ford(福特)提出了从源点逐次绕过其他顶点，以缩短到达终点的最短路径长度的方法。 Bellman-ford算法是求含负权图的单源最短路径算法。

**算法描述**

对所有的边进行n-1轮松弛操作，因为在一个含有n个顶点的图中，任意两点之间的最短路径最多包含n-1边。换句话说，第1轮在对所有的边进行松弛后，得到的是源点最多经过一条边到达其他顶点的最短距离；第2轮在对所有的边进行松弛后，得到的是源点最多经过两条边到达其他顶点的最短距离；第3轮在对所有的边进行松弛后，得到的是源点最多经过一条边到达其他顶点的最短距离......

**算法步骤**

输入： 有向图和源顶点。

输出： 从源顶点到所有顶点的最短距离值。 如果存在负权重边，则不计算最短距离，返回存在负权重边。

（1）将源顶点到所有顶点的距离初始化为无穷大，源顶点本身的距离初始化为0。创建一个大小为V的动态数组 ，除了 其中源顶点值是0之外，其他所有值都为无穷大。

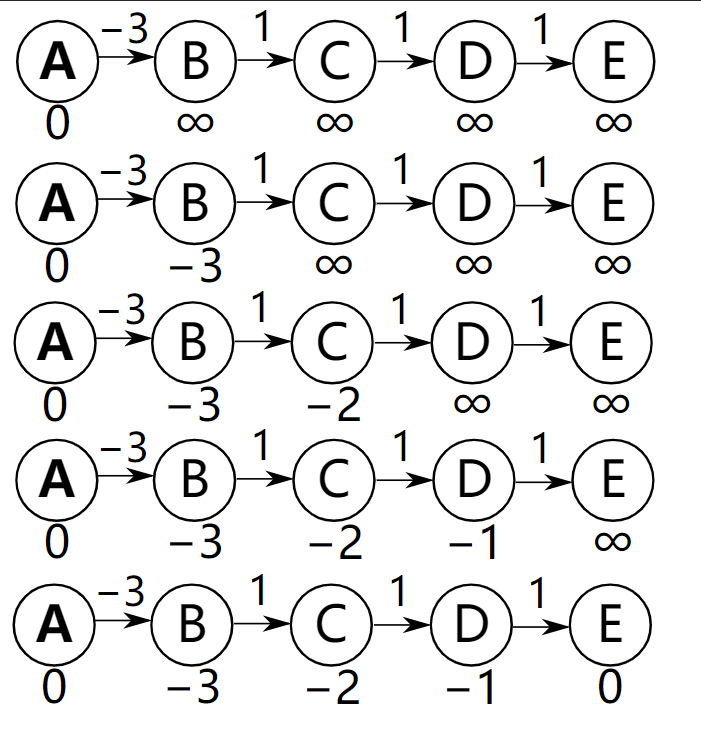
（2）计算最短距离。 循环V-1次，其中V是给定图中的顶点数。

a) 对每条边uv，如果dist[v] > dist[u] +边uv的权重，则更新dist[v]，dist[v] = dist[u] +边 uv 的权重

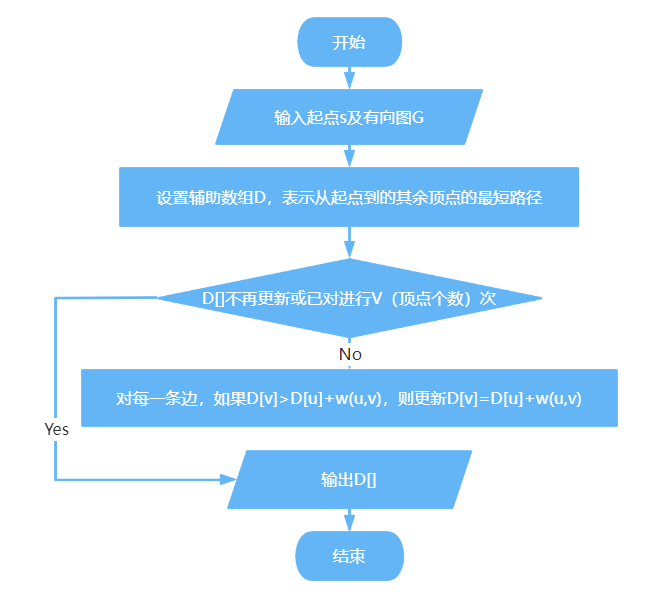
（3）循环中需要返回图中是否存在负权重边。对每条边 uv ，如果 dist[v] > dist[u] +边 uv 的权重，则返回“图包含负权重边”

第 3 步，如果图不包含负权重循环，则第 2 步保证得到了最短距离值。与其他动态规划问题一样，该算法以自底向上的方式计算最短路径。首先计算路径中最多有一条边的最短距离，然后，再计算最多有 2 条边的最短路径，依此类推。在外循环的第 i 次迭代之后，计算最多具有 i 条边的最短路径。假设没有负权重边的循环，如果我们计算了最多有i条边的最短路径，那么对所有边的迭代保证能得到最多有 (i+1) 条边的最短路径。

**算法示意**



**算法流程**

****

### Johnson

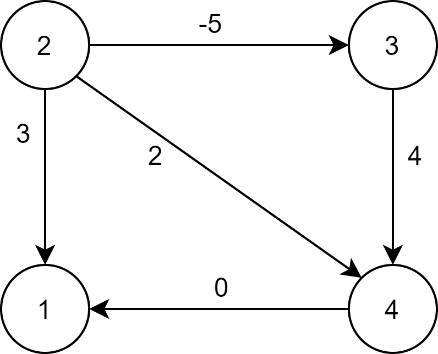
**算法描述**

新建一个虚拟节点（在这里我们就设它的编号为 ）0。从这个点向其他所有点连一条边权为 0 的边。接下来用 Bellman-Ford 算法求出从 0 号点到其他所有点的最短路，记为 hi。假如存在一条从 u 点到v 点，边权为w 的边，则我们将该边的边权重新设置为w+hu+hv 。接下来以每个点为起点，跑 n 轮 Dijkstra 算法即可求出任意两点间的最短路了。一开始的 Bellman-Ford 算法并不是时间上的瓶颈，若实现 Dijkstra 算法，该算法的时间复杂度是 O（V2logV + VE）。

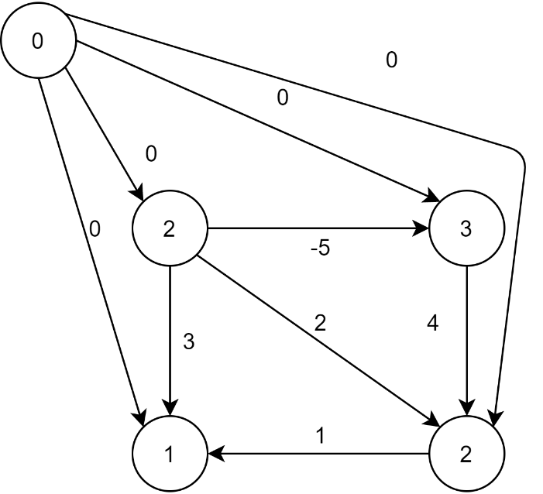
**算法步骤**

下面解释一下全源最短路径求解中权重调整的原理。

首先做一个含负权重边的图。



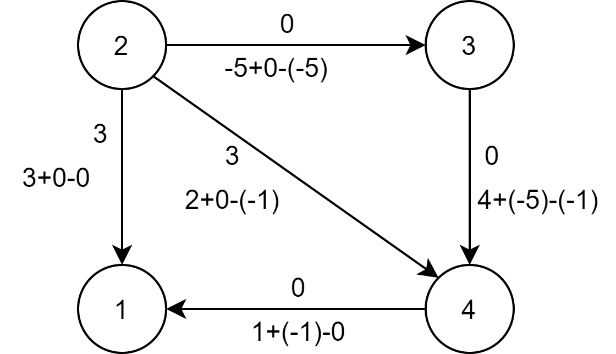
然后，在图中含有的源点之外新增一个源点(编号为0)，从它指向所有其他顶点，这些新的边先赋权值为0，设修改后的图为 。



然后，对新图使用Bellman-Ford 算法，以0号节点为源节点，求到其它节点的最短路径距离 。如果发现负的权重，则返回。对原始图的边重新加权。 对于每条边 (u, v)，将新权重分配为：

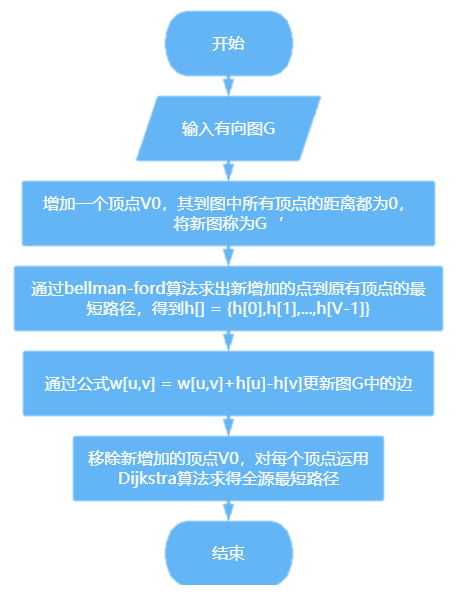


赋值结果为：



现在除新增结点外，其它结点的相关边权重值都已经为正数了，可以将新增结点删除，接下来以每个点为起点，跑 n轮 Dijkstra 算法即可求出任意两点间的最短路径了。

**Johnson算法整体流程图**



## 算法计算复杂性分析

### Johnson

算法的主要步骤是 Bellman Ford 算法调用一次和 Dijkstra 调用 V 次。 Bellman Ford 的时间复杂度为 O(VE)，Dijkstra 的时间复杂度为 O(VLogV)。 所以总的时间复杂度是 O(V2logV + VE)。

表1 最短路径算法比对

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 最短路径算法 | Floyd | Bellman-Ford | Dijkstra | Johnson |
| 最短路径类型 | 全源最短路径 | 单源最短路径 | 单源最短路径 | 全源最短路径 |
| 作用于 | 任意图 | 任意图 | 非负权图 | 任意图 |
| 是否检验负环 | 能 | 能 | 不能 | 能 |
| 时间复杂度 | O（V3） | O(VE) | O(VLogV) | O(V2logV + VE) |

# 算法实现与结果展示

#include<iostream>

#include<cstring>

#define INF INT\_MAX-100

#define MAX\_VERtEX\_NUM 20 //顶点的最大个数

#define N 100

using namespace std;

typedef enum{undiscovered, visited}VStatus;

typedef struct

{

VStatus vexstatus[MAX\_VERtEX\_NUM];

int arc[MAX\_VERtEX\_NUM][MAX\_VERtEX\_NUM];

int numV, numE;

}MGraph;

typedef struct Edge //边

{

int u, v; //两顶点

int cost; //权

}Edge;

class Graph

{

private:

MGraph G;

Edge e[N];

int pre[MAX\_VERtEX\_NUM];

int dis[MAX\_VERtEX\_NUM];

int mind[MAX\_VERtEX\_NUM];

public:

Graph()

{

int i, j;

int nv, ne, a, b, w;

cin >> nv >> ne;

G.numV = nv;

G.numE = ne;

for (i = 0; i != nv; ++i)

for (j = 0; j != nv; ++j)

G.arc[i][j] = INF;

for (i = 0; i != ne; ++i)

{

cin >> a >> b >> w;

G.arc[a-1][b-1] = w;

e[i].u = a;

e[i].v = b;

e[i].cost = w;

}

}

int numVex()

{

return G.numV;

}

int AddEdge()

{

int i, n;

n = G.numE;

for (i = G.numE-1; i >= 0; --i)

{

e[i+G.numV].u = e[i].u;

e[i+G.numV].v = e[i].v;

e[i+G.numV].cost = e[i].cost;

}

for (i = 0; i != G.numV; ++i)

{

e[i].u = 0;

e[i].v = i+1;

e[i].cost = 0;

++n;

}

return n;

}

void Bellman\_ford()

{

int i, j;

for(i = 1; i != G.numV+1; ++i) //初始化

dis[i] = INF;

dis[0] = 0;

bool p = 0;

int t = AddEdge();

for(i = 0; i != G.numV; ++i)

{

p = 1;

for(int j = 0; j != t; ++j)

if(dis[e[j].v] > dis[e[j].u] + e[j].cost) //松弛（顺序一定不能反）

{

dis[e[j].v] = dis[e[j].u] + e[j].cost;

p = 0;

}

if (p == 1)

break;

}

}

void UpdateGraph()

{

int i, j;

for(i = 0; i != G.numV; ++i)

for(j = 0; j != G.numV; ++j)

if (G.arc[i][j] != INF)

G.arc[i][j] = G.arc[i][j]+dis[i+1]-dis[j+1];

}

void Dijkstra(int n)

{

int i, j, t = n, mi;

for (i = 0; i != G.numV; ++i)

{

mind[i] = INF;

G.vexstatus[i] = undiscovered;

pre[i] = -1;

}

mind[n] = 0;

G.vexstatus[n] = visited;

for (i = 1; i != G.numV; ++i)

{

mi = INF;

for (j = 0; j != G.numV; ++j)

{

if ((G.arc[t][j] + mind[t] < mind[j]) && (G.vexstatus[j] != visited))

{

mind[j] = G.arc[t][j] + mind[t];

pre[j] = t;

}

}

for (j = 0; j != G.numV; ++j)

if ((mind[j] < mi) && (G.vexstatus[j] != visited))

{

t = j;

mi = mind[j];

}

G.vexstatus[t] = visited;

}

cout << "以" << n+1 << "为源点至各点的最短路径为：";

for (i = 0; i != G.numV; ++i)

cout << mind[i] << ' ';

cout << "以" << n+1 << "为源点时各点的前驱为(若无法直达，则输出-1)：";

for (i = 0; i != G.numV; ++i)

if (pre[i] == -1)

cout << pre[i] << ' ';

else

cout << pre[i]+1 << ' ';

cout << endl;

}

};

int main()

{

Graph G;

int i;

G.Bellman\_ford();

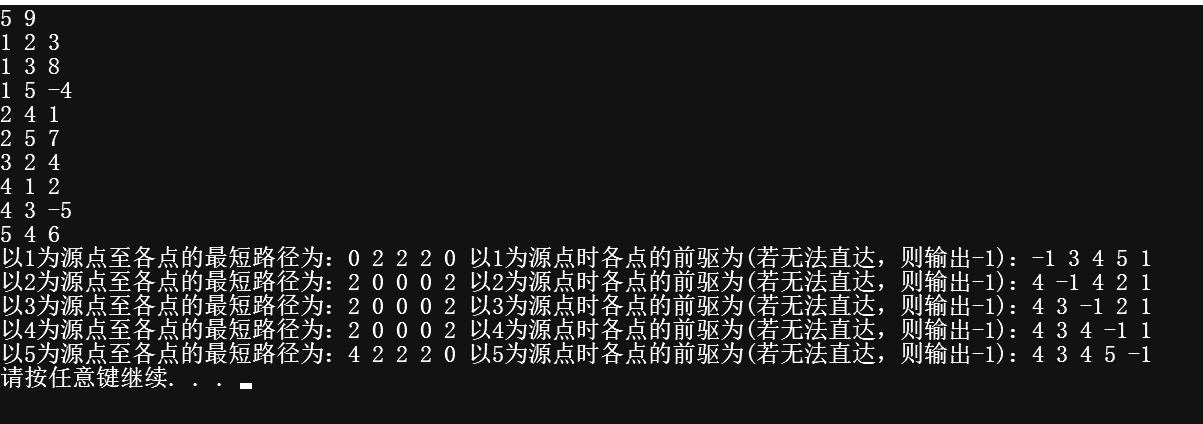
G.UpdateGraph();

for(i = 0; i != G.numVex(); ++i)

G.Dijkstra(i);

return 0;

}

**结果展示**

**参考文献：**

[1]祝国明.基于Dijkstra的多源点最短路径求解算法的设计与分析[J].电脑知识与技术,2021,17(16):177-178.

[2]杜鹃,盛新平,葛新同.“离散数学”创新实践教学的思考与探索——以最短路径选择为例[J].常州工学院学报,2020,33(02):75-80.

[3]江锦成,吴立新,张媛媛,刘善军.适用于大规模网络的全源最短路径重优化算法——RASP算法[J].东北大学学报(自然科学版),2017,38(07):1037-1042.