桂林电子科技大学

实验 4 图的基本操作、算法和应用 实验报告

实验名称	图的基本操作、算法和应用						
院　系	计算机与信息安全学院		专业	信息安全			
学　号	2000301708		姓名	蔡响			
实验日期	2021	年	11	月	12	日	

辅导员意见：

成绩　　辅导员签　名

一、实验目的

1. 掌握图的基本概念-路径和回路
2. 掌握图的遍历
3. 掌握最小生成树prim算法
4. 掌握拓扑排序
5. 掌握最短路径及其应用

二、实验内容

1. 判断两点之间是否存在路径
2. 判断一个有向图是否存在回路
3. 计算最小生成树的权值
4. 求一个顶点到其他顶点的最短路径
5. 判断一个有向图是否能够完成拓扑排序
6. 选取医院建立的位置
7. 迷宫变种-最短路径

三、实验环境

在PTA平台进行实验

四、实验要求

根据每个实训的每个关卡要求完成代码提交和测评

五、实验步骤

1. 描述算法的原理或实现流程
2. 记录每个实训的代码和运行截图（截图中能够包含自己的学号和姓名）

六、问题记录和实验总结

（一）图的含义

图是计算机中常用的一类数据结构，也是最复杂的数据结构，其实现涉及到数组、链表、栈、队列、树等。

一个图就是说是一堆*顶点*的集合，这些顶点通过一系列*边*结对（连接）。顶点用圆圈表示，边就是这些圆圈之间的连线。顶点之间通过边连接。

就我现在的经历来看，图在现实中的应用广泛，一个图可以表示一个社交网络，每一个人就是一个顶点，互相认识的人之间通过边联系。图有各种形状和大小。边可以有*权重（weight）*，即每一条边会被分配一个正数或者负数值。考虑一个代表航线的图。各个城市就是顶点，航线就是边。那么边的权重可以是飞行时间，或者机票价格。

机器学习中存在图神经网络，可以训练图神经网络进行现实问题的分配决策。

（二）链表的插入算法

读题：

输入邻接矩阵，判断是否存在闭环。

思路：

储存邻接矩阵：

由于是邻接矩阵，所以我就想到了二维数组存储数据。定义一个二维数组weight[][]，若节点$V_i \rightarrow V_j$存在路径，即将$weight[V_i][V_j]$的值设置为1。

判断是否存在闭环：

第一反应就是走迷宫，根据邻解矩阵将图画出，就是一个走迷宫的问题，所以可选的算法就有DFS与BFS。

```
1 1 0 0 1 0 0
2 0 1 1 1 0 1
3 0 0 1 0 0 0
4 1 0 1 0 0 1
5 0 0 1 1 0 0
```

实现：

下面是采用DFS的简略代码：

```
Qstack.push(Vi)                                  #起始点入栈
while(Qstack.size())
   V = Qstack.top();                             #出栈将中心点赋值给V
   for(i;i<length;i++)
      if(weight[Vi][i])
        {Qstack.push(i);              #顺序遍历节点，将可行的节点入栈
        if(i==V) { cout<<1; return; }}        #到达终点，结束循环
cout<<0;
```

其中Vi是起始点，length是节点的个数。只要遍历到了终点，就打印1，结束程序。当程序找遍了所有的路径（栈为空），却不能到达终点，就打印0。

注意点就是在走过某条路径后，需要将权重置为0。

（三）判断一个有向图是否存在回路

读题：

输入是有向图的邻接矩阵表示中的1的相邻顶点。判断图是否存在回路。这题不同于上一题，上一题只需要判断两点之间的路径，无需经过所有的点，而该题还需判断能否经过所有的点。

思路：

采用DFS算法，不同的是添加了一个label[]数组，用于防止重复经过节点。

注意点：

遍历完所有的点后，还需判断最后一个点与起点是否存在路径。

实现：

由于需要判断是否存在回路，故此自定义一个Judge函数用于判断是否存在回路。

具体的代码基于第一题为baseline进行改进。

为降低时间复杂度，我将，判断是否已经经过节点的判断代码放在循环最前面，只要已经经过该节点，直接continue。

（四）计算最小生成树权值

读题：

本题要求采用Prim算法求最小生成树，输出其权值之和。

思路：

题目要求用Prim算法实现，故按照Prim算法的思想进行。

实现：

如同前面的题目一样，定义一个weight[][]的二维数组，用于存放节点之间的权重。需要注意的是，如果两个节点之间不存在路径，则二者之间的权重应该被置为某个特定的数（例如：65535）。

这里定义了label[]用于标记每个节点是否经过遍历。lowcost[]用于存储到达每个节点的最小消耗，lowcost[Vi]=weight，即代表从已有的点集中到达Vi的最小带权为weight。

主程序主要分为两个部分：

第一个部分用于循环判断读取lowcost[]中最小权重对应的节点i，然后将i加入已遍历的点集。

由于已遍历点集已更新，所以第二步就需要更新lowcost[]

我认为，第二步的更新lowcost[]是整个Prim代码算法实现中最新颖的部分，因为其涉及到了一种**动态更新**的思想，也是以前写题几乎没有遇到的。感觉有点类似于动态规划，根据当前的情况计算接下来的解。

（五）求一个顶点到其他顶点的最短路径

读题：

本题要求计算一个顶点v0到其他顶点的最短路径，并输出该路径。

读到题目第一个反应是去修改最小生产树的代码，最短路径的计算不就是最小生成树的权值和吗。

发现需要以特定的格式进行打印，从0到每个节点的最小路径以及对应权重。

思路：

在已有的算法中，只了解Dijkstra可以用于解决相关问题。故采用Dijkstra进行解决。感觉Dijkstra就是升级版的Prim。

实现：

主要定义了一个二维数组weight[][]，以及三个一维数组path[]，distance[]，found[]。

- 二维数组用于存放weight
- path[j]=k代表j节点指向的下一个节点为k
- distance[j]=value代表到达节点j的消耗为value
- label[j]用于标记该节点是否被遍历

主程序主要分为三个部分：循环读取找最小权重，更新最小消耗distance[]，迭代打印结果。

这里主要讲一下打印结果的部分，个人采用递归实现，以前比较少写递归的代码。

打印的主函数：

```
void PrintWeight(int *distance, int *path)
{
   for (int i = 1; i < count; i++)
   {
      cout << 0 << "->" << i << ':' << distance[i] << endl;
      PrintRoute(path, i);
      cout<<i << endl;
   }
}
```

递归函数：

```
void PrintRoute(int *path, int i)
{
   if (i)
   {
      PrintRoute(path, path[i]);
      cout << path[i]<<"->";
   }
}
```

我们知道打印的起点与终点，由于起点只有一个，终点有很多，所以我们只能通过终点索引回起点。故此采用了先调用PrintRoute()的打印函数，再打印。这样就可以做到从终点递归会起点打印的效果。

（六）判断一个有向图是否能够完成拓扑排序

读题：

求输出一个有向图是否能够完成拓扑排序，输入的图采用邻接表表示。

思路：

拓扑排序的实现方法：

1. 从有向图中选择一个没有前驱(即入度为0)的顶点，将其加入点集，并且输出它。
2. 从网中删去该顶点，并且删去从该顶点发出的全部有向边。
3. 重复上述两步,直到剩余的图中不再存在没有前趋的顶点为止。

感觉这个还是类似于走迷宫，可以通过队列和栈的方式进行DFS与BFS边的删除与点集添加。

实现：

为了利于拓扑排序的实现，我在定义节点的时候添加了入度entrynode以及边该节点的指向的节点点集Adjacency[]。

```
typedef struct node
{
    int number;
    int entrynode;
    int *Adjacency;
    struct node *next;
} * Node
```

实现选择的是DFS:

```
1. Qstack_push();
2. while(!Qstack.empty()){
    V = Qstack.top();
    adjacency();
    Qstack_push();
    }
```

Qstack_push():将度为空的节点入栈。

adjacency():将V.Adjacency中节点的入度entrynode减1。

（七）选取医院建立的位置

读题：

设a、b、c、d、e、f表示一个乡的6个村庄,弧上的权值表 示两村之间的距离。现要在这6个村庄中选择一个村庄建一所医院,问医院建在哪个村庄 才能使离医院最远的村庄到医院的距离最短?

这是一个比较简单的实际应用，契合实际生活情况。在六个点中取一点，使该点遍历其他所有点的距离和最短。

思路：

求点到其他点的最短距离，这不就是前面的"求一个顶点到其他顶点的最短路径"。六个点中取一点，不就是挨个遍历所有的点，切换最小生成树的根节点进行吗？

实现：

实现就比较简单，有点缝合怪的操作，把最小生成树的代码copy过来，进行一番添加。通过一维的数组value[]按下标储存对应乡村的最小权重和。其中a,b,c,d,e,f分别对应0,1,2,3,4,5。村庄名字(char)name与对应value下标(int)index映射关系为：index=name-‘97’。

例如：value[0]=34 代表以a村庄建立医院时，最远的村庄到医院的最短距离。

本来想写一个打擂台的方式从value[]中得到最优距离解，但是想想我用的是C++啊，C++什么接口函数没有？？一百度，直接用*max_element(distance, distance + number)得到最大值，用min_element(value, value+ length) - value得到最小值对应的下标。

（八）迷宫变种-最短路径

读题：四周为-1表示围墙,内部为-1表示障碍,权 值1、2、5、9表示经过需要消耗的能量代价。请找出从入口(3,6)到出口(8,8),老鼠消耗能量最小的路径(注意本题是四个方向的迷宫)

```
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
-1  2  1  1  1  1  1  5  1 -1
-1  1  9  9  9  1  1 -1  1 -1
-1  1  1  1  1  1  1 -1  1 -1
-1  1 -1 -1 -1 -1 -1 -1  1 -1
-1  1  9  9  9  1  1  1  1 -1
-1  1  1  1  1  1  1  1  1 -1
-1  1  1  1  1  1  1  1  1 -1
-1  1  1  1  1  1  1  1  2 -1
-1 -1 -1 -1 -1 -1 -1 -1 -1 -1
```

该迷宫不同于往常的走迷宫，过去的迷宫只有路与墙，这次的迷宫通过路径还需要消耗能量。故此需要通过计算权值进行解题。

思路：

打算基于DFS进行解题。基础的DFS具有出栈得到中心点，入栈新的点，遇到障碍物回退的功能。但是变种迷宫不同，还需要根据消耗进行方向选择。

由于涉及到消耗的计算与累加，故打算采用A*算法进行。

实现：

通过比较中心点的四个范围中，某个点到终点的坐标绝对值距离加上经过该点所消耗的能量，作为是否将该点入栈的依据。即：abs(*x* - x2) + abs(*y* - y2) + map[*x*][*y*];

实现的过程中需要进行判断，防止出现越界的情况。

遇到的问题：

如果存在以下的路径消耗，此刻在权重为3的点,，要前往右下角的点1

```
1 1 7 9 9
2 7 1 1 9
3 4 1 3 9
4 1 4 9 1
```

通过算法，我们发现点（2,1）点（1,2）计算的权重相同，所以会随机选择一点入栈，最后的路径会从：

点（2,2）→点（1,2）变成　点（2,2）→ 点（2,1）→点（1,1）→点（1,2）

最后通过采用遍历判断的方式避免这种情况。

感觉自己的代码还是存在一些漏洞，无法处理一些特殊情况。 后面知道Floyd算法，故去了解了一番。

七、问题记录和实验总结

- 储存方式：图的储存方式有许多种，比较重要的是邻接矩阵和邻接表，具体的存储情况具体分析，结构修改少的图，用邻接矩阵合适，反之考虑邻接表。
- 图的判断：判断是否闭环,判断是否能够完成拓扑排序。

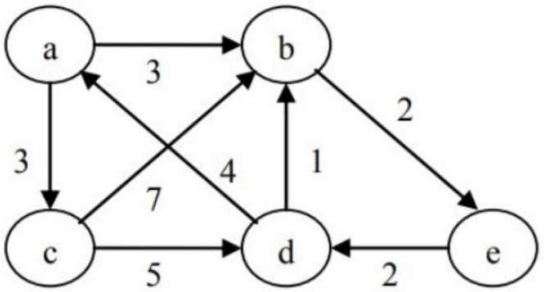
- 图的遍历：图的遍历分为深度（DFS）与广度（BFS）两种，各有千秋。
- 图的应用：
    - 最小生成树：普里姆(Prim)算法与克鲁斯卡尔(Kruskal)算法。
    - 最短路径：迪杰斯特拉（Dijkstra）算法与佛洛依德（Floyd）算法

（一）储存方式

图的建立主要有两种方式：1. 邻接矩阵，2. 邻接表

1. 邻接矩阵

这个方法就是定义一个二位数组，用于存储边的关系。对于点u,v，若$e(u,v)$的权为w，则$a[u][v]=w$（只需要邻接关系的话w就置1）。十分直观，问题是如果点多边少，存在空间浪费的问题。另外就是这样没法存多重边。



顶点数组：

a	b	c	d	e

边数组：

	a	b	c	d	e
a	∞	3	3	∞	∞
b	∞	∞	∞	∞	2
c	∞	7	∞	5	∞
d	4	1	∞	∞	∞
e	∞	∞	∞	2	∞

邻接矩阵的优点：

1. 可以通过$M[u][v]$直接引用边(u, v)， 因此只需常数时间(O(1))即可确定顶点u 和顶点v 的关系
2. 只要更改$M[u][v]$就能完成边的添加和删除， 简单且高效O(1)

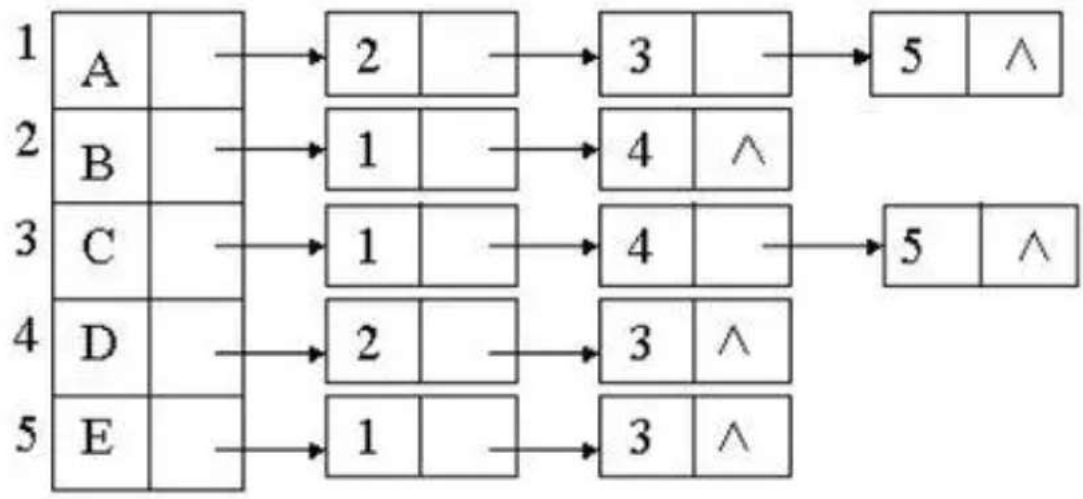
邻接矩阵的缺点：

1. 消耗的内存空间等于顶点的平方数。如果图的边数较少（稀疏图），则会浪费大量的内存空间
2. 一个邻接矩阵中，只能记录顶点u 到顶点v 的一个关系（一个基本型的二维数组中，无法在同一对顶点之间画出俩条边）

2. 邻接表法

邻接表一般用于来存每个点连了哪些边。

邻接表比较容易添加一起其他的信息，有利于进行一些复杂的操作以及算法实现的简化



邻接表的优点:

1. 节省空间，只存储实际存在的边。

邻接表的缺点：

1. 其缺点是关注顶点的度时，就可能需要遍历一个链表。
2. 对于无向图，如果需要删除一条边，就需要在两个链表上查找并删除。

3. 图的遍历

类似于走迷宫的深度搜索与广度搜索，这里就不进行过多阐述。

DFS：从一个节点找**任意**一条可以通向其他节点的路径。

BFS:从一个节点找**所有**可以通向其他节点的路径。

4. 图的最小生成树算法

Prim算法

基本思想：假设G＝(V，E)是连通的，TE是G上最小生成树中边的集合。算法从$U=u_0$（$u_0 \in V$）、$TE=$开始。重复执行下列操作：

在所有$u \in U$, $v \in V$ -U的边$(u, v) \in E$中找一条权值最小的边(u_0, v_0)并入集合TE中，同时v_0并入U，直到V =U为止。

此时，TE中必有n-1条边，$T = (V, TE)$为G的最小生成树。

Prim算法的核心:始终保持TE中的边集构成一棵生成树。

用简洁的话来讲，就是**寻找已走过节点通向其他节点的最小权边，然后将对应的点加入已遍历的点集，重复该过程，直至遍历所有点。**

Kruskal算法

算法简单简述：

1. 将每个顶点看成一个图
2. 在所有图中找权值最小的边．将这条边的两个图连成一个图
3. 重复上一步．直到只剩一个图

需要注意的是在变量的过程中不能形成闭环。

prim算法适合稠密图，其时间复杂度为$O(n^2)$，其时间复杂度与边得数目无关，而Kruskal算法的时间复杂度为$O(e \times loge)$跟边的数目有关，适合稀疏图。

5. 最短路径算法

Dijkstra算法

Dijkstra算法采用的是一种贪心的策略。

声明一个数组dis来保存源点到各个顶点的最短距离和一个保存已经找到了最短路径的顶点的集合：T，初始时，原点 s 的路径权重被赋为 0（dis[s] = 0）。若对于顶点 s 存在能直接到达的边（s,m），则把dis[m]设为w（s, m）,同时把所有其他（s不能直接到达的）顶点的路径长度设为无穷大。初始时，集合T只有顶点s。

然后，从dis数组选择最小值，则该值就是源点s到该值对应的顶点的最短路径，并且把该点加入到T中，OK，此时完成一个顶点，
然后，我们需要看看新加入的顶点是否可以到达其他顶点并且看看通过该顶点到达其他点的路径长度是否比源点直接到达短，如果是，那么就替换这些顶点在dis中的值。

然后，又从dis中找出最小值，重复上述动作，直到T中包含了图的所有顶点。

（二）A*算法

A*算法实际上是通过计算**代价函数f(n)**，来寻找终点的最优路径进行选取，代价函数公式表示为：

$$f(n) = g(n) + h(n) \tag{1}$$

- 其中f(n) 是节点n从初始点到目标点的估价函数。
- g(n)：起始节点到当前节点的实际代价。
- h(n)：当前节点到目标节点的估计代价。

  一般情况下，代价直接为曼哈顿距离$abs(x - ex) + abs(y - ey)$;

  abs为绝对值函数。

（三）Floyd算法

Floyd算法是一个经典的**动态规划**算法。是解决**任意两点间的最短路径**(称为多源最短路径问题)的一种算法，可以**正确处理有向图或负权的最短路径问题**。

PS: 动态规划算法是通过拆分问题规模，并定义问题状态与状态的关系，使得问题能够以递推（分治）的方式去解决，最终合并各个拆分的小问题的解为整个问题的解。最经典动划就是走台阶。

1. 算法思想

从任意节点i到任意节点j的最短路径不外乎2种可能：

1. 直接从节点i到节点j。
2. 从节点i经过若干个节点k到节点j

   所以，我们假设$arcs(i,j)$为节点i到节点j的最短路径的距离，对于每一个节点k，我们检查$arcs(i,k) + arcs(k,j) < arcs(i,j)$是否成立，如果成立，证明从节点i到节点k再到节点j的路径比节点i直接到节点j的路径短，我们便设置$arcs(i,j) = arcs(i,k) + arcs(k,j)$，这样一来，当我们遍历完所有节点k，$arcs(i,j)$中记录的便是节点i到节点j的最短路径的距离。

   由于动态规划算法在执行过程中，需要保存大量的临时状态（即小问题的解），因此它天生适用于用矩阵来作为其数据结构，因此在本算法中，我们将不使用Guava-Graph结构，而采用邻接矩阵来作为本例的数据结构。

（四）C++ 的学习

$<queue><stack>$ 可以省去定义队列与栈消耗的时间，并且极大的缩减了代码的长度，增强了代码的可读性。

$<vector>$容器，类似于数组与字符串，不过vector可以自行定义存放数据的数据类型，真的方便。

当然还有一些其他的函数，例如*max_element（）得到数组的最大值，加快了项目的实现与开发。

一开始用C语言写队列与栈，再到后面用C++写图。感觉是自己造轮子与使用轮子的过程。在现在已经将来的工作中，我想会时长体会到这种感觉：

调用API->迅速高效，提高开发效率。

用自己写的程序->费时费力，效果还不一定好。

我认为，需要权衡好造轮子问题。未来不是依靠现在，而是与时俱进的创新。