第九章-图论算法

注意各种算法的时间复杂度的分析

分为两个部分

第一部分以书为主总结的知识点概述，根据BUPT数据结构课程与自己的理解进行一定的补充

第二部分为leetcode基本数据结构图-难度简单的部分题解

9.1 若干定义

图的组件

1. 顶点集： 非空
2. 边集：可为空

   -----> 与树的区别：图至少有一个点
3. 弧 ： 点对（v,u）
   - a. v 和 u邻接
   - b. 点对的有序与否决定图的有向/无向
4. 权或值
5. 路径
   - a. 顶点序列，至少包含一个顶点（环）
   - b. 长是 **边数**
   - c. 可以不包含边 路径长度为0

d. 简单路径：所有顶点互异 **但**第一个顶点和最后一个顶点可以相同（路径**中间**不含圈）

6. 圈
    a. 长至少为一的路径，路径首尾同点
    b. loop是一个特殊的圈
    c. 简单路径 ---- 简单圈
    d. 无向图中的圈 ：边是互异的 ----> **u v u 不是一个圈**
    e. 有向图中，u v u是一个圈
    f. 有向无圈图 directed acyclic graph (DAG)
7. 连通
    a. 无向图：任意两个顶点都存在路径
    b. 有向图
        i. 强连通：任意两个顶点都存在路径
        ii. 弱连通：基础图（对应的无向图）是连通的
8. 完全图：任意两个顶点都存在边
    a. 有向图边数：n(n-1) ----> 每一个顶点都与其余的n-1个顶点有边，这样的顶点有n个
    b. 无向图边数：n(n-1)/2---> 对无向图来说，上述方法数了两边，故除2

图的表示

邻接矩阵

对于边(u,v):

1. 无权图
    a. A[u][v] = 1 表示存在边
    b. A[u][v] = 0 表示边不存在
2. 有权图
    a. A[u][v] = weight/cost 表示该边存在
    b. 该边不存在

i. 正无穷（寻找最便宜的路线时）

ii. 负无穷/0（寻找最贵的路线时）

空间 = $O(|V^2|)$

若边很多，即 **稠密** 邻接矩阵合适

若边不多，即 **稀疏** 邻接表合适

邻接表

空间 = $O(|E|+|V|)$

头单元数组：存放所有顶点

+

链表：存放该点邻接的所有顶点 （有向图中，邻接 = →）

*逆邻接表

只适用于有向图，在求一个节点的入度时使用

链表存放邻接到该点的所有顶点

9.2拓扑排序（Topological sort）

有向无圈图（DAG）

若有圈，无拓扑排序

拓扑排序不唯一

一个顶点的入度可以存放在头单元中，便于Indegree数组的创建

拓扑排序算法

思路

1. 找出任意一个没有入度的顶点 ： 这些顶点是当前图中最先的节点，开始位置
2. 显示该节点，并删除该节点的相关弧：更新图，使这张图为除了该节点的其余节点
3. 重复1，2

伪代码

```
void topSort( Grapg G){
    int Counter;//即充当计数器，也作为顶点拓扑排序的位次
    Vertex V,W;
    //循环VertexNum次，遍历整个图的所有顶点
    for(Counter = 0 ; Counter < VertexNum ; Counter++){
        //寻找入度为零的顶点即step1
        V = FindVertexOfIndegreeZero();
        if(!V)//查找失败 每个节点都有入度说明含有圈（若无圈，一定有开始位置，而
开始位置是无入度的）
        {
            Error("graph has a circle");
            break;
        }
        //查找成功则设置该节点的位次并更新图
        TopNum[V] = Counter;
        for each W adjacent to V // 表示（V，W） V adjacent W，W
adjacent to V
            Indegree[W]--;
    }
}
```

改进

问题：若改图稀疏，则每次更新图导致入度数改变的顶点的数量很少，但在每次循环开始时都遍历每个顶点以找出入度为零的点

改进：

1. 将所有入度为零的顶点入队
2. 当队不为零时，pop -> a
    a. 遍历a的所有邻接的顶点，入度减一
    b. 遍历a的所有邻接顶点，若存在入度为0的顶点，入队
    c. 继续pop
3. 当队为零时，要么排序结束，要么出现圈

```
void TopSort(Graph G){
    Queue Q;//入度为零的顶点 队列
    int Counter = 0;//计数，标识位次
    Vertex V, W;

    //初始化队列
    Q = createQueue(VertexNum);
    makeEmpty(Q);

    //将所有入度为零的顶点入队
    for each V{
        if(Indegree[V] == 0)
            Enqueue(V,Q);
    }

    //队列不为零，就出队，出队顺序为拓扑排序
    while( !isEmpty( Q ) ){
        V = Dequeue(Q);
        TopNum[V] = ++Counter;//从一开始数，即标识位次，有标识已排序的顶点个
数

        for each W adjacent to V{//遍历该顶点的邻接顶点
            if(--Indegree[V] == 0){
                Enqueue(V,Q);
            }
        }
    }

    if(Counter != VertexNum){//还未遍历所有结点队列就空了
        Error("graph has a cycle")
    }
```

```
    DisposeQueue(Q);//free memory
}
```

9.3最短路径算法

单源最短路径问题?

若图中存在负值圈，则最短路径问题就变得不确定（可以重复走负值圈以减小花费）

无负值圈时，s到s的最短路径为0

dv：从s到该节点的距离

pv：帮助显示实际路径

9.3.1无权最短路径问题

BFS

将路径都赋值为1

广度优先搜索（BFS）类似于树的层序遍历

当顶点标为已知后，确信不会有更短的路径产生：因为在BFS中，越往后遍历，所经历的路径肯定越长

伪代码：

```
void Unweighted(Table T){//Table T 就是记录dv，pv这些数据的表，如图9-15
    int CurrDist;//当前对于开始点的位置，即BFS的层次
    Vertex V,W;
    for(CurrDist = 0;CurrDist < VertexNum ; CurrDist++){//循环
VertexNum次，确保遍历所有层次
        for each Vertex V{
            if(!T[V].Known && T[V].Dist == CurrDist)//据顶点的距离等于
当前遍历所处的层次 且 未访问过
            {
                T[V].Known = 1;
                for each W adjacent to V{
                    if(T[W].Dist == Infinity){//该节点还未访问过
                        T[W].Dist = CurrDist+1;
                        T[W].Path = V;//可以通过追踪Path来确定路径
                    }
                }
            }

        }
    }

}
```

缺点：

即使所有的点都被置为Known，最外层循环仍然执行，可以增加测试语句来break以提高效率，也可以：

改进

思路：

在任意时刻，除了标记为正无穷的顶点，只有两种状态：

- dv = CurrDist
- dv = CurrDist +1
- 将查找条件‘if(!T[V].Known && T[V].Dist == CurrDist)‘转换为限定在这两者之间

- 使用队列的思想实现
    - 将开始节点放入队列以启动程序
    - 顺序取出CurrDist的顶点，然后设置CurrDist +1 的顶点，入队
- Know域是不需要的，因为处理过或已入队的值不会是Infinity，故不会进行入队操作
- 队列过早为空，意味着有些顶点是不可达的，这时候保持dist为Infinity是合理的

```
void unweighted (Table T){
    Queue Q;
    Vertex v,w;

    EnQueue(S,Q);//将开始节点放入队列以启动程序

    while(!isEmpty(Q)){
        //出队处理该节点---设置该节点的邻接节点
        V = DeQueue(Q);

        T[V].Known = 1;//not really needed anymore

        if(T[W].dist == Infinity){//确保了处理过的不会再处理，因为处理过的
节点的dist不会为Infinity，不会入队
            T[w].dist = T[V].dist +1;
            T[w].path = V;
            EnQueue(W,Q);
        }
    }

    DisposeQueue( Q );

}
```

9.3.2赋非负权最短路径（Dijkstra）

BFS思想/贪婪算法

1. 选择所有节点中的小dv
2. 更新邻接到v的所有节点w的dw
    a. 如果dv＋cost < dw则更新dw（意味着对于w点来说，出现了一条更便宜的路径）
3. 最短路径可以跟踪pv给出

```
void Dijkstra(Table t){
    Vertex V,W;

    for ( ; ; )
    {
        V = smallestDistanceUnkownVertex;
        if(V == NoVertex){//遍历完了整张图
            break;
        }

        T[V].Known = 1;//V就是当前距离最小的节点，因为剩下的所有节点都比这个节点远，故可以断定后面的循环中生成的距离一定比当前距离大（因为经过了更远的节点），故该节点设为已知，不会有更小的距离出现

        for each w adjacent to V
            if(!T[w].Konw)//剪枝，避免无效的比较
                if(T[V].Dist + CvW < T[W].Dist){
                    //update
                    Decrease T[W].Dist to T[V].Dist + CvW;
                    T[V].Path = V;

                    //不设置w为Known因为走当前路径的距离不一定是最小的
                }
    }

}
```

```
//打印当前节点的最短路径
//递归思路:
//递归:先打印当前节点的前面的节点的最短路径(该节点最短路径的一部分),再打印该节点
//终止:当没有前驱节点时,只打印该节点
void PrintPath(Vertex V,Table T){
    if(T[V].path != NoVertex){
        pirntPath(T[V].path,T);
        printf("to");
    }
    printf(V);
}
```

9.3.3具有负边值的图

错误的方案:

- 将一个常数加到每个边上,以确保每个边都是正数,然后用Dijkstra算法
- 缺陷:
    - 起始节点到达目的节点的路径越长,所加的常数的次数就越多,可能导致更多边的路径比很少边的路径的权重大了,导致选择了错误的最短路径
    - 如:
        - 原来:选择第一条路径

      s --(2)-- a --(-1)-- b--(-2)-- c

      s --(1)-- c

        - 加常数3后:选择第二条,错误的最短路径

      s --(5)-- a --(2)-- b --(1)-- c

      s --(4)-- c

改进：

思路;

1. 忘记关于已知顶点的概念，因为不能确保处理过的顶点不会出现更短的路径
   - 每次没有选择最近节点
   - 即使选择了最近节点也会因为负值边的存在，可能会导致在经过更远顶点到达该顶点的距离更短
2. 让一个顶点出队
3. 对邻接到w所有点，当经过v到w的距离更优时，更新dw
4. 当w不在队列中时，将w入队（因为w的distance更新了，经过点w到达的点的距离也应该更新，若不更新w,则经过w的节点在该次循环中不需要更新）
5. 若出现负值圈，算法将 **无限循环**

```
void weigthtedNegative(Table T){
    Queue Q:
    Vertex V,W;

    //initialize Queue
    Q = CreateQueue(NumVertex);
    makeEmpty(Q);
    EnQueue(S,Q);

    while(!isEmpty){//队列为空表示无更短路径产生，遍历结束
        DeQueue(V,Q);
        for each w adjacent to v{
            if(T[W].Dist > T[V].Dist + CvW){
                //update W and enqueue
                T[W].Dist = T[V].Dist + CvW;
                T[W].path = V;
                if(W is not in queue){
                    Enqueue(W,Q);
                }
            }
        }
    }

    DisposeQueue(Q);
```

```
}
```

9.3.4 无圈图

利用拓扑排序改进Dijkstra算法：

1. 每次按照拓扑排序选择一个最近节点（因为该节点没有入度，故不存在更小的距离）
2. 选择性的更新邻接节点的最短路径并按照拓扑规则删除该节点（在更新完邻接节点的最短路径后该节点失去价值）
3. 重复执行1，2直至无节点可选

在拓扑排序的过程中实现节点的选择和更新，故算法一趟就可以完成。算法时间复杂度为**O(|E| + |V|)**

关键路径分析法

动作节点图

- 边表示优先关系，点表示任务及耗时
- 假设所有不相关动作都可以无限并行执行

动作节点图转化为事件节点图

- 模拟方案的构建
    - 方案最早完成时间
        - 计算**最长**路径（有正值圈时无法寻找最长路径，也无有效的方法寻找最长简单路径）
    - 哪些动作是可以延时的，延时多少，并且不影响最早完成时间
- 计算所有**结点**的最早完成时间：最短路径算法

- EC(i)表示i点最早完成时间
- EC(1) = 0
- EC(w) = **max**{EC(v) + Cost(v,w)}(v表示所有邻接w的结点)

- 计算**结点**的最晚完成时间：倒转拓扑排序计算最晚完成时间
    - 该节点尽可能地晚地到达但不影响整个过程地最早完成时间
    - LC(i)表示i点的最晚完成时间
    - LC(n) = EC(n)
    - LC(v) = **min** { EC(w) - Cost(v,w)}
    - 不管是不是关键路径上的结点，该观点都适用

- 计算**边**的松弛时间
    - 动作被推迟执行而不影响整个过程的最早完成时间
    - Slack(v,w) = LC(w) - (EC(v) + Cost(v,w))
    - 后一结点的最晚完成时间与前一结点到达后一结点的最早时间之差

9.4 网络流问题

- 发点source
- 收点sink
- 边的最大容量
- 即非发点又非收点的结点：流入＝流出
- 最大流问题确定从s到t可以通过的最大流量

简单的最大流算法

- 图
- 流图
- 残余图

- 增长通路：s到t的一条路径---这条路径上最小边值就是该路径的流量
- 随机选取增长通路，累加各路流量，直到无增长通路为止。
    - 选取的通路不对会导致算法结果不正确

改进算法

增加了**撤销流**的操作

- 在流图中增加一条值为f(v,w)的边（v,w）时，在残图中增加一条边（w,v）值为f(v,w),并删减原来的边大小，值为零时去除该方向的边
- 残图中的边代表：还能顺着变得方向再流多大的量，且不会超过边容量
    - 反向边的加入代表了撤销流操作，即改变设置的流过大，导致其他增长通路消失。撤销，或者说减少该部分的流的大小，使之出现尽可能多的增长路径，该操作只会增长s到t的最大流量
- 正反边的值加起来是该边的最大容量
- 仍然等到图中没有增长通路时终结算法

进一步改进

总选择时流增长最大的增长通路

9.5 最小生成树

- G应该是连通的
- 最小生成树不一定是唯一的
- 边数 = |V| - 1

Prim算法

- 使树连续地一步一步长成
- 已生成的树结点的集合
    - 每次选择一条最短的边（u，v）
    - u在树上
    - v不在树上

- 算法思想：
    - 与Dijsktra算法相似
    - dv表示结点d到**集合**的最短距离（与Dijsktra中对dv的定义不同）
    - pv，known与之一样
    - 当一个顶点v被选择后，其邻接点w
        - dw = min{dw,Cost(v,w)}

该算法整个的实现实际上和 Dijkstra 算法的实现是一样的，对于 Dijkstra 算法分析所做的每一件事都可以用到这里。不过要注意，Prim 算法是在无向图上运行的，因此当编写代码的时候要记住把每一条边都要放到两个邻接表中。不用堆时的运行时间为 $O(|V|^2)$，它对于稠密的图来说是最优的。使用二叉堆的运行时间是 $O(|E|\log|V|)$，对于稀疏的图它是一个好的界。

与课件上不一样？

Kruskal算法

- 连续选择最小的边
- 且不产生圈 -- 两个顶点不在同一个集合时，才能添加边
- Kruskal实际上处理的是森林，每次将两棵树合并为一棵树

Kruskal算法的伪代码？

9.6深度优先搜索的应用

深度优先搜索模板

- 对于每一个到达的点，深度优先搜索其没有被访问过的邻接点，直到没有临界点没有被访问过
- 伪代码：

```
void DFS(Vertex V){
    //到达结点V
    Visited[V] = ture;

    //深度优先搜索其邻接点
    for each w adjacent to v:
        if(!Visted[w])
            DFS(w)
    }
//使用了递归的方法，隐式的使用了栈结构。
//也可以显式使用栈结构，将不再使用递归调用

//BFS运用队结构
```