Kruskal 算法与最小生成树

摘要

作为图论中的一个重要领域，最小生成树在工程领域有许多重要的应用，如如何建设公路、管道可以使得我们的造价最低同时能够基本要求。最小生成树可以通过两个为人熟知的 Prim 算法和 Kruskal 算法进行构建，两者都是基于贪心思想。区别在于 Prim 算法通过点集合来构造，Kruskal 通过边集合来构造。本文将会介绍 Kruskal 算法求解最小生成树的过程，以及介绍相关算法并查集。

关键字： Kruskal 广度优先搜索 深度优先搜索 并查集

Kruskal 算法简介

Kruskal 算法是用于求解无向图的最小生成森林，如果无向图是一个连通图，那么将会求出最小生成树，如果该无向图并非是一个连通图，将会求解出最小生成森林。以下都以连通无向图作为例子。

对于一个无向图连通图 G（V，E）（后称为 G），通过 Kruskal 算法求解最小生成森林，步骤如下：

1、 首先按照边的权重进行非递减排序

2、 以此遍历排序后的边，如果将边加入已经选择的边集中，不会形成环路，我们就选择这条边，否则就不选择这条边。

经过以上两条步骤就可以求得最小生成树，可以确定的是最小生成树的权值和是一定的，同时一定是选择出了|V|-1 条边（树的定义，是一个连通图并且具有节点个数-1 条边）但是最小生成树的边集是不确定的。

Kruskal 算法与环路的检测

对于第一步，我们可以在 O(|E|*log(|E|))的时间复杂度下完成，不进行探讨。

对于第二步，我们没加入一条边，检测是否成环，可以通过简单的深度优先搜索进行暴力检测，对于每次深度优先搜索，时间复杂度为 O(|V|)，总体时间复杂度为 O(|E|*|M|)。

还有一个方法检测是否有环，通过广度优先搜索，在广度搜索过程中检测节点 a，b 是否属于同一个连通子集，时间复杂度为 O(|V|)，总体的时间负责度也是 O(|E|*|M|)。

深度优先搜索和广度优先搜索是在书上初识 kruskal 自己能想到的检测环的算法，算法也很朴素，之后在学习中，逐渐了解到一般情况下都是通过并查集（Union-Find）算法来实现环的检测。

并查集与 Kruskal 算法

实际上并查集不仅仅是应用在环的检测上，通俗来描述并查集可以检测两个节点（姑且这样描述）是否属于同一个集合（连通子集），如果在我们要将连接边<a,b>加入到已有的边集中，他们可以形成环的条件是，他们属于同一个集合，而检测两个节点是否属于同一个集合真是并查集要解决的事情。

并查集的实现：

假设我们的节点是由 1 到 n 这 n 个数字表示的，我们需要一个数组 fa[n]，fa[i]表示节点 i 的父亲节点，当一个节点 i==fa[i]就表明节点 i 是节点 i 所在连通子集的代表节点。

并查集主要有两个主要函数代码及介绍如下

```
namespace unionFind{
    int find(int a){
        if(fa[a]==a) return a;
        else return find(fa[a]);
    }
    bool merge(int a,int b){
        a=find(a);
        b=find(b);
        if(a==b) return false;
        fa[a]=b;
        return true;
    }
};
```

1、 merge(a，b)，合并 a，b 两个节点，将 a 所在的连通子集与 b 所在的联通子集进行合并。

2、 find(a)，查找节点 a 所属于的连通子集的代表节点

通过 find(a)!=find(b)来确定 a，b 不属于同一个连通子集，因此将边<a,b>加入已有的边集中不会形成环。

我们遍历到一条边时，会进行一次 merge 操作，merge 操作的时间复杂度为 O(1)，find 的时间复杂度经过查询复杂度取决于节点 a 所在连通子集树的高度，在一定程度上查询复杂度会退化为 O(|V|)，因此一般我们需要在查询过程中进行一些处理，以此来降低时间复杂度。因为 find(a)是进行查找节点 a 所在连通子集的代表节点，在 find 过程中，我们可以将路径进行压缩，经过径压缩的 find 函数代码如下：

```
int find(int a){
    if(fa[a]==a) return a;
    else return fa[a]=find(fa[a]);
    //路径压缩
}
```

除此之外，在 merge 过程中，通过按秩合并也能降低时间复杂度，即将小的连通子集合并到大的连通子集上。优化后的代码如下：

```
bool merge(int a,int b){
    a=find(a);
    b=find(b);
    if(a==b) return false;
```

```
    if(rank[a]>rank[b]) std::swap(a,b);
    //按秩合并
    fa[a]=b;
    rank[b]+=rank[a];
    return true;
}
```

经过优化，并查集的查询 find 操作的时间复杂度被证明为 Ackermann（α(x)）[1]。因此通过路径压缩和按秩合并优化的并查集实现的环路检测的总的时间复杂度为 O(|E|*α(x))，在这里 α(x)一般认为是常数级别 O（1），与深度优先搜索和广度优先搜索相比更为高效。

以下代码是 kruskal 算法的代码实现。

```
/*
This file please name as "xx.cpp"
Then you can follow the input standard
in the tail of this file to run kruskal algorithm
Author:
Long long Sang
Date:
2020-12-11
Email:
llsang@foxmail.com
*/
#include <stdio.h>
#include <stdlib.h>
#include <algorithm>
#define N 10000
int fa[N], ans[N], ranking[N];
namespace unionFind{
    int find(int a){
        if(fa[a] == a) return a;
        else return fa[a] = find(fa[a]);
        //路径压缩
    }
    bool merge(int a, int b){
        a = find(a);
        b = find(b);
        if(a == b) return false;
        if(ranking[a] > ranking[b]) std::swap(a, b);
        //按秩合并
        fa[a] = b;
        ranking[b] += ranking[a];
        return true;
    }
```

```
33. };
34. class kruskal{
35.     private:
36.         int n, cnt, ans_cnt,m, tot_weight;
37.         struct edge{
38.             int from, to, weight;
39.         }e[N];
40.         void initialize(){
41.             for(int i = 0; i <= n; i++) fa[i] = i,ranking[i] = 1;
42.             cnt = n, ans_cnt=0, tot_weight=0;
43.         }
44.         bool isFinish(){
45.             return cnt == 1;
46.         }
47.         void input(){
48.             scanf("%d %d", &n, &m);
49.             getchar();
50.             for(int i = 0; i < m; i++){
51.                 scanf("%d %d %d", &e[i].from, &e[i].to, &e[i].weight);
52.                 getchar();
53.             }
54.         }
55.         void print(){
56.             if(cnt == 1) printf("This is a Minimum Spanning Tree(MST)\n");
57.             else printf("This is a Minimum Spanning Forest\n");
58.             printf("Total weight is %d\n", tot_weight);
59.             printf("We will choose %d path below:\n", ans_cnt);
60.             for(int i = 0; i < ans_cnt; i++){
61.                 printf("City %d <-> City %d weight: %d\n", e[ans[i]].from, e
    [ans[i]].to, e[ans[i]].weight);
62.             }
63.         }
64.     public:
65.
66.         kruskal(){}
67.         void running(){
68.             input();
69.             initialize();
70.             std::sort(e, e + n, [&](edge& a, edge& b){
71.                 return a.weight < b.weight;
72.             });
73.             for(int i = 0; i < m && !isFinish(); i++){
74.                 if(unionFind::merge(e[i].from, e[i].to)){
75.                     cnt--;
```

```
                    ans[ans_cnt++] = i;
                    tot_weight += e[i].weight;
                }
            }
            print();
        }
};
/*
input standard:
first line tow decimal number:
n m
n meaning number of vertex and m menaing number of edge
Then m line follow, and each line include three num like:
city1 city2 weight
city1 and city2 range from 1 to n meaning the city index.
weight meaing the weight between this two city

Example:
4 4
0 1 12
3 2 12
0 2 1
3 1 2


  (0) ---12--- (1)
   |             |
   1             2
   |             |
  (2) ---12--- (3)
*/
int main(){
    kruskal console;//kruskal 的实例化
    console.running();//运行一个 kruskal 的运算
}
```

参考文献：

[1]. Tarjan R E, Van Leeuwen J. Worst-case analysis of set union algorithms[J]. Journal of the ACM (JACM), 1984, 31(2): 245-281.

[2]. kruskal 算法时间复杂度是如何推导的？ - 庄下秋竹的回答 - 知乎