

最小生成树

M20W0236 胡稼伟

M20W0120 曹菁仪

M20W0238 徐梓迪

1 定义

一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。最小生成树可以用 kruskal（克鲁斯卡尔）算法或 prim（普里姆）算法求出。在一个给定的无向图 $G = (V, E)$ 中， (u, v) 代表连接顶点 u 与顶点 v 的边（即），而 $w(u, v)$ 代表此边的权重，若存在 T 为 E 的子集（即）且为无循环图，使得的 $w(T)$ 最小，则此 T 为 G 的最小生成树。最小生成树其实是最小权重生成树的简称。

$$\omega(t) = \sum_{(u,v) \in t} \omega(u, v)$$

举一个实际的例子：我们需要在 n 个城市中建立一个通信网络，连通 n 个城市需要 $n-1$ 条路线，这时候如何在成本最低的情况下建立通信网络？

我们就可以引入连通图来解决这个问题， n 个城市就是 n 个顶点，然后，边表示两个城市的通信线路，每条边上的权重就是我们搭建这条线路所需要的成本，所以现在我们有 n 个顶点的连通网可以建立不同的生成树，每一颗生成树都可以作为一个通信网，当我们构造这个连通网所花的成本最小时，搭建该连通网的生成树，就称为最小生成树。

2 相关性质

2.1 存在个数

最小生成树在一些情况下可能会有多个。例如，当图的每一条边的权值都相同时，该图的所有生成树都是最小生成树。

如果图的每一条边的权值都互不相同，那么最小生成树将只有一个[1]。这一定理同样适用于最小生成森林。

2.2 边的权值之和最低的子图

如果图的边的权值都为正数，那么最小生成树就是该图的所有包含所有顶点的子图中权值最低的子图。

2.3 切分定理

在一幅连通加权无向图中，给定任意的切分，如有一条横切边的权值严格小于所有其他横切边，则这条边必然属于图的最小生成树。

证明：

令 e 为权重最小的横切边，假设 T 为图的最小生成树，且 T 不包含 e 。那么如果将 e 加入 T ，得到的图必然含有一条经过 e 的环，且这个环也含有另一条横切边——设为 e' ， e' 的权重必然大于 e ，那么用 e 替换 e' 可以形成一个权值小于 T 的生成树，与 T 为最小生成树矛盾。所以假设不成立[2]。

2.4 最小权值边

如果图的具有最小权值的边只有一条，那么这条边包含在任意一个最小生成树中。

证明：假设该边 e 没有在最小生成树 T 中，那么将 e 加入 T 中会形成环，用 e 替换环中的任意一条权值更大的边，将会形成权值更小的生成树，与题设矛盾。

3 相关算法

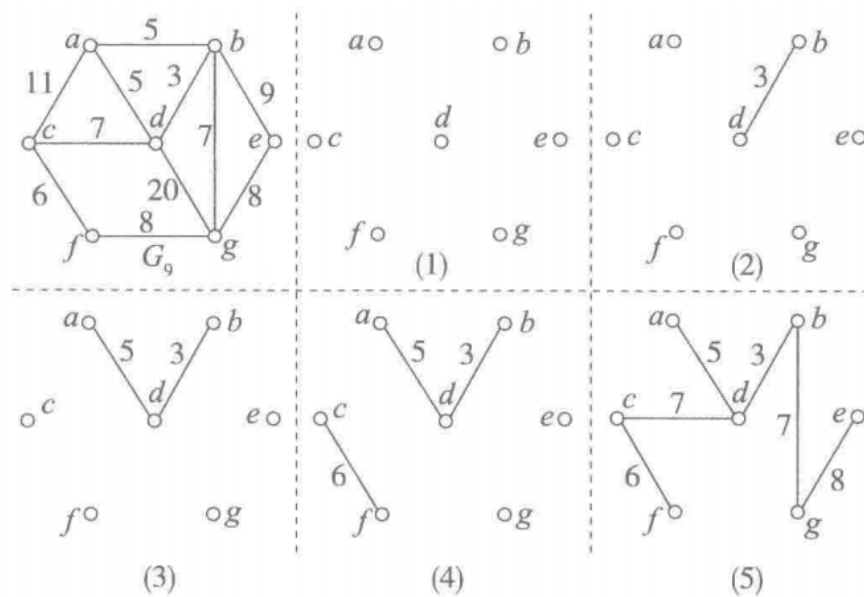
3.1 历史简介

计算稠密图的最小生成树最早是由罗伯特·普里姆在 1957 年发明的[3]，即 Prim 算法。之后艾兹赫尔·戴克斯特拉也独自发明了它[4]。但该算法的基本思想是由沃伊捷赫·亚尔尼克于 1930 年发明的[5]。所以该算法有时候也被称为 Jarník 算法或者 Prim-Jarník 算法。20 世纪 70 年代，优先队列发明之后很快被用在了寻找稀疏图中的最小生成树上。1984 年，迈克尔·弗里德曼和罗伯特·塔扬发明了斐波那契堆，Prim 算法所需要的运行时间在理论上由 $\log E$ 提升到了 $E+V$ 。约瑟夫·克鲁斯卡尔在 1956 年发表了他的算法，在他的论文中提到了 Prim 算法的一个变种，而奥塔卡尔·布卢瓦卡在 20 世纪 20 年代的论文中就已经提到了该变种。M. Sollin 在 1961 年重新发现了该算法，该算法后成为实现较好渐进性能的最小生成树算法和并行最小生成树算法的基础[6]。

以下各算法介绍中， E 表示图的边数， V 表示图的顶点数。

3.2 Kruskal 算法

Kruskal 算法基于简单连通分量的最小代价互联。将初始图 G 中各边按权值从小到大排列成列表 $edges$ ，存储方式为 $(weight, v_i, v_j)$ ，每次取出一条边，检查其连接的两端是否已连通，若尚未连通则将该边加入生成树，并修改该边所连接的两个连通分量的状态，否则删除该边。

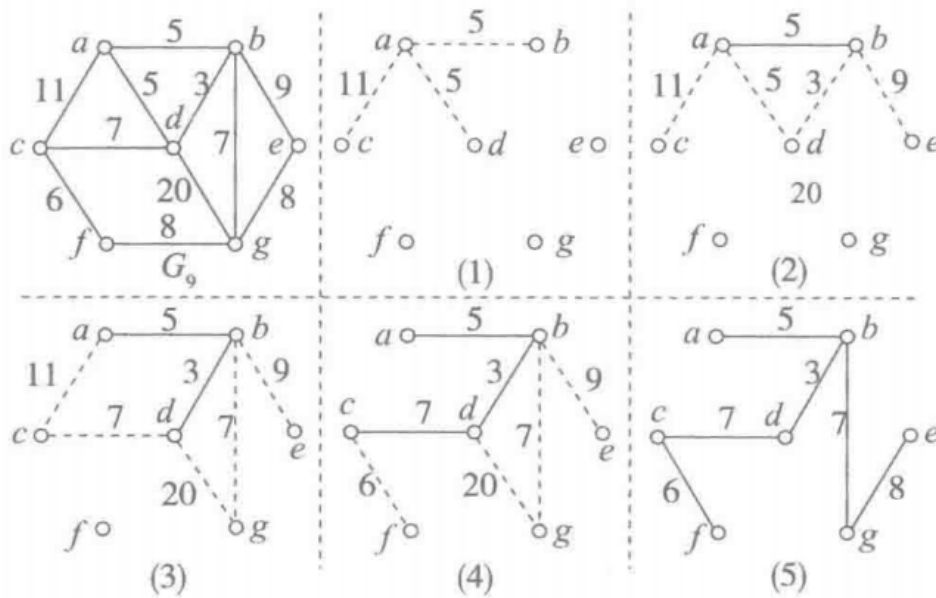


相应的 Python 代码实现如下：

```
def Kruskal(graph):  
    vnum = graph.vertex_num()    #得到图中点的个数  
    mst, edges = [], []  
    reps = [i for i in range(vnum)]    #初始化代表元  
    for vi in range(vnum):            #收集各边  
        for vj, weight in graph.out_edges(vi):  
            edges.append((weight, vi, vj))  
    edges.sort()  
    #将边按权值 weight 从小到大排序  
    for weight, vi, vj in edges:  
        #逐个遍历边，将其加入到 mst 中  
        if reps[vi] != reps[vj]:  
            mst.append((vi, vj, weight))  
            repi, repj = reps[vi], reps[vj]  
            for v in range(vnum):  
                #更新代表元  
                if reps[v] == repj:  
                    reps[v] = repi  
    return mst
```

3.3 Prim 算法

Prim 算法是基于所谓的 MST 准则，将图的点集分为两部分，mst 和 V，依次将边顶点分属于两个点集的最小权值边加入到生成树中，同时将 V 中连接的点加入到 mst 中，相比于 Kruskal 不断将最小权值边加入生成树，Prim 则是连续扩大最小生成树中的点集。



相应的 Python 代码实现如下：

```
def Prim(graph):  
    vnum = graph.vertex_num()  
    edges = PrioQueue((0,0,0))  
    #每次将新边加入到一个优先队列中  
    mst = [None] * vnum  
    #用于判断边所连接的点是否已经遍历过  
    edge_count = 0  
    while edge_count < vnum and not edges.is_empty():  
        weight, vi, vj = edges.dequeue()  
        if mst[vj] == None:  
            edge_count += 1  
            mst[vj] = (vi, weight)  
            for i, w in graph.out_edges(vj):  
                #将新点的出边加入优先队列  
                if not mst[i]:  
                    edges.enqueue((w, vj, i))  
    return mst
```

Kruskal 算法和 Prim 算法两种算法的对比

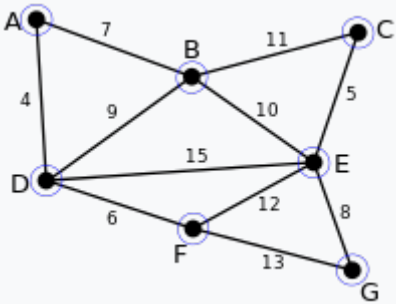
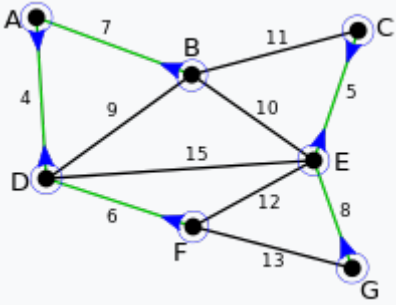
因为 Kruskal 算法需要对所有的边根据权值进行排序并储存，所以在图比较复杂的时候可能会占用大量空间。

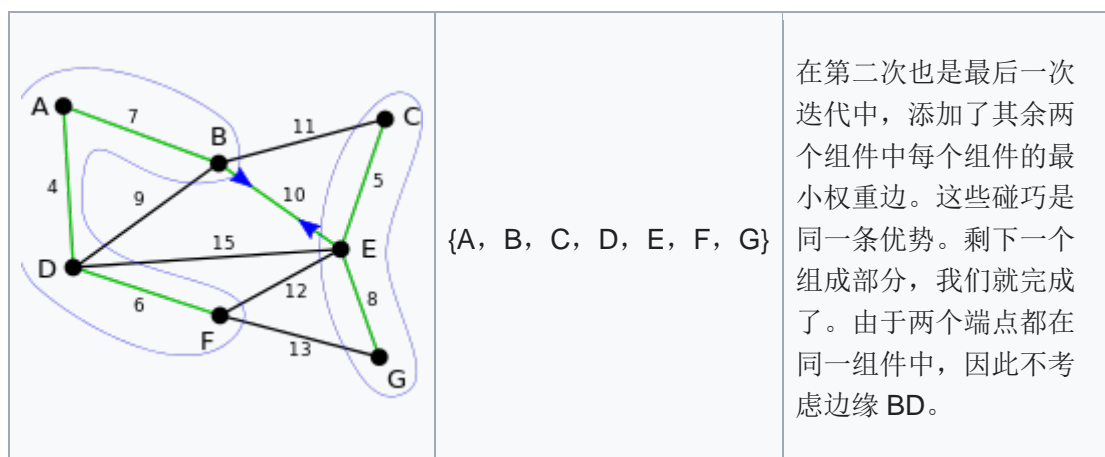
所以一般来说：

prim 算法适合复杂模型

Kruskal 算法适合简单模型

3.4 Borůvka 算法

图片	组件	描述
	<div><div>{A}</div><div>{B}</div><div>{C}</div><div>{D}</div><div>{E}</div><div>{F}</div><div>{G}</div></div>	这是我们原始的加权图。边缘附近的数字表示其重量。最初，每个顶点本身就是一个组成部分（蓝色圆圈）。
	<div><div>{A, B, D, F}</div><div>{C, E, G}</div></div>	在外循环的第一次迭代中，添加了每个组件的最小权重边。一些边缘被选择两次（AD，CE）。剩下两个部分。



Borůvka 算法是一种贪婪算法，用于在图中找到最小生成树，或者在未连接图的情况下找到最小生成林。

它最早由 Otakar Borůvka 于 1926 年出版，是为摩拉维亚建立有效的电力网络的一种方法。[7] [8] [915] 该算法在 1938 年由 Choquet 重新发现；[10] 再次由 Florek, Łukasiewicz, Perkal, Steinhaus 和 Zubrzycki 于 1951 年提出；[11] 和 1965 年由 Georges Sollin 再次提出。[12] 该算法经常被称为 Sollin 算法，尤其是在并行计算文献中。

该算法首先找到入射到图的每个顶点的最小权重边，然后将所有这些边添加到森林中。然后，它重复进行类似的过程，即从到目前为止构造的每棵树到另一棵树中找到最小权重的边，然后将所有这些边添加到森林中。此过程的每次重复都会将图的每个连接组件内的树数减少到最多此先前值的一半，因此在对数次重复之后，该过程结束。当这样做时，它添加的一组边形成了最小的生成林。

3.5 更快的算法

一些研究者希望可以找出更为高效的算法，在这方面也有了一定的成果。Karger, Klein & Tarjan (1995) 针对边的权值可以成对比较的特殊模型提出了一个基于 Borůvka 算法和翻转删除算法的可以在线性时间内解决最小生成树的算法 [13] [14]。

最快的非随机比较算法是由伯纳德·沙泽勒提出的。该算法依赖于 soft heap 这样一个类似于优先级队列的数据结构 [15] [16]。该算法的时间复杂度为 $O(E \alpha(E, V))$ 。 α 就是阿克曼函数反函数， α 的增长速度非常慢，对于一般的数值来说，其值很难超过 5，所以该算法的复杂度可以近似看成是线性时间。

4 实际应用

4.1 最优灌溉问题

A 承包了很多片麦田，为了灌溉这些麦田，A 在第一个麦田挖了一口很深的水井，所有的麦田都从这口井来引水灌溉。为了灌溉，A 需要建立一些水渠，以连接水井和麦田，A 也可以利用部分麦田作为“中转站”，利用水渠连接不同的麦田，这样只要一片麦田能被灌溉，则与其连接的麦田也能被灌溉。现在 A 知道哪些麦田之间可以建设水渠和建设每个水渠所需要的费用（注意不是所有麦田之间都可以建立水渠）。请问灌溉所有麦田最少需要多少费用来修建水渠。

输入形式：

输入的第一行包含两个正整数 n , m ，分别表示麦田的片数和雷雷可以建立的水渠的数量。麦田使用 1, 2, 3, ……依次标号。

接下来 m 行，每行包含三个整数 a_i , b_i , c_i ，表示第 a_i 片麦田与第 b_i 片麦田之间可以建立一条水渠，所需要的费用为 c_i 。

输出形式：

输出一行，包含一个整数，表示灌溉所有麦田所需要的最小费用。

样例输入：

```
4 4
1 2 1
2 3 4
2 4 2
3 4 3
```

样例输出：

```
6
```

相应的 Python 代码实现如下：

```
def prim(graph): #最小生成树

    n = len(graph) #n 就是田的数量

    cost = [99999 for _ in range(n)] # 父结点到该结点的边权值

    cost[0] = 0 #第一块田有井水
```

```

visited = [False for _ in range(n)]

t = []          #不断往 t 中添加田

while len(t) < n:    # 在 costs 找最短边，把该最短边的结点加入
t, 标记为已访问

    minCost = 99999

    minNode = None

    for i in range(n):

        if not visited[i] and cost[i] < minCost:

            minCost = cost[i]

            minNode = i

    t.append(minNode)

    visited[minNode] = True

    for edge in graph[minNode]:# 遍历该结点的边，更新最短边

        if not visited[edge[0]] and edge[1] < cost[edge[0]]:

            cost[edge[0]] = edge[1]

    return cost

```

tian, qv = map(int, input().split())# 构建带权邻接表

```
graph = [[] for _ in range(tian)]
```

```
for i in range(qv):
```

```
    qi, zhong, quan = map(int, input().split())
```

```
    graph[qi - 1].append([zhong - 1, quan])
```

```
    graph[zhong - 1].append([qi - 1, quan])
```

```
cost = prim(graph)
```



```

total = 0

for c in cost:

    total += c

print(total)

```

4.2 最小树形图（朱刘算法）

给定包含 nnn 个结点， mmm 条有向边的一个图。试求一棵以结点 rrr 为根的最小树形图，并输出最小树形图每条边的权值之和，如果没有以 rrr 为根的最小树形图，输出 $-1-1-1$ 。第一行包含三个整数 n,m,rn,m,rn,m,r ，意义同题目所述。

接下来 mmm 行，每行包含三个整数 u,v,wu,v,wu,v,w ，表示图中存在一条从 uuu 指向 vvv 的权值为 www 的有向边。

输出格式：如果原图中存在以 rrr 为根的最小树形图，就输出最小树形图每条边的权值之和，否则输出 $-1-1-1$ 。

输入输出样例：

```

输入：
4 6 1
1 2 3
1 3 1
4 1 2
4 2 2
3 2 1
3 4 1
输出：
3

```

相应的 Python 代码实现如下：

```

class Edge:
    def __init__(self, u, v, w):

```

```

        self.u = u
        self.v = v
        self.w = w

    def __str__(self):
        return str(self.u) + str(self.v) + str(self.w)

def zhuliu(edges, n, m, root):
    res = 0
    while True:
        pre = [-1]*n
        visited = [-1] * n
        circle = []
        inderee = [INF] * n
        # 寻找最小入边
        inderee[root] = 0
        for i in range(m):
            if edges[i].u != edges[i].v and edges[i].w <
inderee[edges[i].v]:
                pre[edges[i].v] = edges[i].u
                inderee[edges[i].v] = edges[i].w
        # 有孤立点，不存在最小树形图
        for i in range(n):
            if i != root and inderee[i] == INF:
                return -1
        # 找有向 h 环
        tn = 0 # 记录环的个数
        circle = [-1] * n
        for i in range(n):
            res += inderee[i]
            v = i
            # 向前遍历找环，中止情况有：
            # 1. 出现带有相同标记的点，成环
            # 2. 节点属于其他环，说明进了其他环
            # 3. 遍历到 root 了
            while visited[v] != i and circle[v] == -1 and v != root:
                visited[v] = i
                v = pre[v]
            # 如果成环了才会进下面的循环，把环内的点的 circle 进行标记
            if v != root and circle[v] == -1:
                while circle[v] != tn:
                    circle[v] = tn
                    v = pre[v]

```

```

        tn += 1
    # 如果没有环了，说明一定已经找到了
    if tn == 0:
        break
    # 否则把孤立点都看作自环看待
    for i in range(n):
        if circle[i] == -1:
            circle[i] = tn
            tn += 1
    # 进行缩点，把点号用环号替代
    for i in range(m):
        v = edges[i].v
        edges[i].u = circle[edges[i].u]
        edges[i].v = circle[edges[i].v]
        # 如果边不属于同一个环
        if edges[i].u != edges[i].v:
            edges[i].w -= inderee[v]

    n = tn
    root = circle[root]
return res

INF = 9999999999
if __name__ == '__main__':
    n, m, root = list(map(int, input().split()))
    edges = []
    for i in range(m):
        u, v, w = list(map(int, input().split()))
        # 输入的点是1开始的，-1改为0开始的
        edges.append(Edge(u-1, v-1, w))
    print(zhuliu(edges, n, m, root-1), end = "")

```

4.3 最小生成树（贪婪算法）

最小生成树 prim 算法（贪婪算法）python 实现

相应的 Python 代码实现如下：

```

#!/usr/bin/env python
# -*- coding: utf-8 -*-

# prim 算法

```

```

def update(a,b, nodepair, G, n):
    # 函数实现代码开始
    # 请在其中写入你的代码
    if b == 0:
        return nodepair
    nodepair[b][0] = a
    nodepair[b][1] = G[a][b]
    # 函数实现代码结束
    return nodepair

def prim(G, n):
    INF = 100000
    visit = [-1] * n # 遍历情况记录数组
    nodepair = [[-1, INF] for i in range(n)] # 节点伴随二元数组 第一个表示前一个索引 preIndex, 第二个表示距离
    # 先取第一个节点
    visit[0] = 1
    print(0, nodepair[0]) # 打印第一个节点
    # 更新节点伴随二元数组
    nodepair = update(0,0, nodepair, G, n)
    # print(nodepair)
    # 函数实现代码开始
    # 请在其中写入你的代码
    #对 visit 进行遍历

    for key in range(1,n):
        tmp = INF
        for key1 in range(n):
            for key2 in range(n):
                if visit[key1]==1 and visit[key2]==-1:
                    if G[key1][key2] != 0:
                        if G[key1][key2] < tmp:
                            a = key1
                            b = key2
                            tmp = G[key1][key2]

            visit[b] = 1
            nodepair = update(a,b, nodepair, G, n)
            print(b, nodepair[b])

    # 函数实现代码结束

    return nodepair

```

```

if __name__ == '__main__':
    Glist = input().split('-')
    G = []
    for item in Glist:
        row = item.split(',')
        g = [int(i) for i in row]
        G.append(g)
    n = len(G)
    prim(G, n)

```

4.4 旅行家的预算问题

问题描述：

一个旅行家想驾驶汽车以最少的费用从一个城市到另一个城市（假设出发时油箱是空的）。给定两个城市之间的距离 $D1$ 、汽车油箱的容量 C （以升为单位）、每升汽油能行驶的距离 $D2$ 、出发点每升汽油价格 P 和沿途油站数 N （ N 可以为零），油站 i 离出发点的距离 D_i 、每升汽油价格 P_i （ $i=1, 2, \dots, N$ ）。计算结果四舍五入至小数点后两位。如果无法到达目的地，则输出“No Solution”。

输入格式：

第一行为 4 个实数 $D1$ 、 C 、 $D2$ 、 P 与一个非负整数 N ；
接下来 N 行，每行两个实数 D_i 、 P_i 。

输出格式：

如果可以到达目的地，输出一个实数（四舍五入至小数点后两位），表示最小费用；否则输出“No Solution”（不含引号）。

样例输入：

```

275.6 11.9 27.4 2.8 2
102.0 2.9
220.0 2.2

```

样例输出：

```

26.95

```

分析问题：

设第 i 个加油站的加油量为 a_i （ $i=0, 1, 2, \dots, N$ ），设第 i 个加油站到起点的距离为 D_i （ $i=0, 1, 2, \dots, N+1$ ），设第 i 个加油站的油价为 P_i （ $i=0, 1, 2, \dots, N+1$ ）， D_i 和 P_i 是已知量。旅行家目的是花最少的钱到达终点，也就求 a_i 的组合，一共 $N+1$ 个变量，显然，这是一个多元线性规划问题。

$$\sum_{i=0}^N a_i * p_i$$

运用最小生成树（贪婪算法）：

核心思想就是每次都尽量去更便宜的油站加油，除非在能到达的油站内没有更便宜的油站了。

一共有 N 个油站，约定 $D_i[0]=0, P_i[N+1]=0$ ，即起点的距离为 0，终点的油价为 0

假设旅行家当前处于第 i ($0 \leq i \leq N$) 个油站，车上剩余油量为 remain，已经花费 cost，

先判断满油下能否到达下一站，假如不能达到，得出结论：此问题无解，输出 No Solution。如果能到达，则从当前油站开始，寻找下一个比当前站更便宜的油站，记为 k 站 ($k \geq i$)。i 依次加 1，只要油价比当前站便宜即找到 k 站，因为终点的油价为 0，所以一定会找到 k 站，有可能还没到达终点就找到 k，也可能 k 就是终点。找到 k 站后有两种情况

1、能到达 k 站

(1) 剩油量能去 k 站>>>直接去 k 站

(2) 剩油量不能去 k 站>>>加油到刚好去 k 站

2、不能到达 k 站（也就是说在能到达的所有站点中，当前站油价最便宜）

(1) 加满油去下一站

python 实现代码如下：

```
D1,C,D2,P,N = input().split()
```

```
D1 = float(D1)    #两城市间距离
```

```
C = float(C)      # 汽车油箱容量
```

```
D2 = float(D2)    #每升汽油能行驶的距离
```

```
P= float(P)       #出发点每升汽油价格
```

```
N = int(N)        #沿途油站数
```

```
Di = [0,]         #每个加油站到起点的距离
```

```
Pi = [P,]        #每个加油站的油价
```

```
a = [ ]          #用来存储在 n 个加油站的加油量
```

```

remain = 0 #剩余油量
cost = 0    #花费油钱
Solution = 0 #问题是否有解的标志，0 有解，1 无解
full = C * D2
for i in range(N):
    d,p= input().split()
    d = float(d)
    p = float(p)
    Di.append(d)
    Pi.append(p)
Di.append(D1)
Pi.append(0)
# 初始化每个加油站的加油量
for k in range(N + 1) :
    a.append(0)
i = 0
while i <= N :
    To_next = Di[i + 1] - Di[i]    #当前站点到下一站的距离
    if To_next > full :            #不能够到达下一站
        Solution = 1
        break
    else :                        #能去下一站
        k = i + 1
        while k <= N + 1 :
            if Pi[i] >= Pi[k] :    #搜索油价小于或等于 i 站下一个油站 k
                break
            k += 1
        To_cheap = Di[k] - Di[i]    #当前站点到 k 站点的距离
        if full >= To_cheap :      #能到达 k 站
            use_cheap = To_cheap/D2
            a[i] = use_cheap - remain
            if a[i] > 0 :          #剩下的油不能直接到达 k 站
                cost += a[i] * Pi[i]
                remain = 0
            else :                 #剩下的油能直接到达 k 站
                a[i] = 0
                remain -= use_cheap
            i = k
        else :#不能达到 k 站  #加满油去下一个站点'''
            a[i] = C - remain
            cost += a[i] * Pi[i]
            remain = C - To_next/D2
            i += 1

```

```

if Solution == 0:
    print('%.2f'% cost)
else :
    print('No Solution')
##    print(a)  #打印每个油站的加油量

```

结果如下：

```

=====
275.6 11.9 27.4 2.8 2
102.0 2.9
220.0 2.2
26.95
>>>

```

```

=====
275.6 11.9 27.4 2.8 2
102.0 2.4
220.0 2.2
25.22
>>> |

```

```

=====
275.6 11.9 27.4 2.8 0
28.16
>>> |

```

```

=====
1000 10 31.5 3.0 2
500 5.0
800 6.2
No Solution
>>>

```

4.5 破圈法实现最小生成树

在 kruskal 算法基础上，我国著名数学家管梅谷教授于 1975 年提出了最小生成树的破圈法。

算法过程：

- 1) 先将所有的边按权值进行降序排列.
- 2) 之后对于取出的每一个边来说, 判断其连接的两个结点是否具有圈. (即先删除次边, 然后判断这两个结点是否连接, 之后对删除的边进行恢复)

- 3) 对于有圈的, 将这条边删除, 否则, 往下查找.
- 4) 算法结束条件: 剩下的边=结点数-1.

相应的 Python 代码实现如下:

```
def break_circle(self):
    parents = []

    def init(parents):
        for n in range(self.numVertices):
            parents.append(n)

    def find(i):
        while (i != parents[i]):
            i = parents[i]
        return i

    def union(i, j):
        if (find(i) == find(j)):
            return
        else:
            parents[find(i)] = find(j)

    def isconnected(i, j):
        return find(i) == find(j)

    def reunion(aGraph):
        parents.clear()
        init(parents)
        for edge in self.edgeList:
            union(edge[0], edge[1])
    reunion(self)
    self.edgeList.sort(key=lambda x: x[2], reverse=True)
    n = self.numEdges
    for x in range(n):
        edge = self.edgeList.pop(0)
        reunion(self)
        bool = isconnected(edge[0], edge[1])
        if bool:
            self.numEdges = self.numEdges - 1
        else:
            self.edgeList.append(edge)
            reunion(self)
    print(self.edgeList)
```

4.6 欧几里得最小生成树

欧几里得最小生成树或 EMST 是一个最小生成树一组的 \tilde{N} 在点平面（或更一般地在 \mathbb{R}^d ），其中，每对点之间的边缘的权重是欧几里得距离这两个点之间。用更简单的术语来说，EMST 使用线连接一组点，以使所有线的总长度最小，并且通过跟随这些线，可以从任何其他点到达任何点。

在飞机上，可以在计算的代数决策树模型中使用 $O(n)$ 空间在 $\Theta(n \log n)$ 时间内找到给定点集的 EMST。更快的复杂度 $O(n \log \log n)$ 的随机算法在更强大的计算模型中广为人知，该模型可以更准确地对真实计算机的能力进行建模。[17]

在更高的尺寸（ $d \geq 3$ ），找到最优算法保持一个打开的问题。

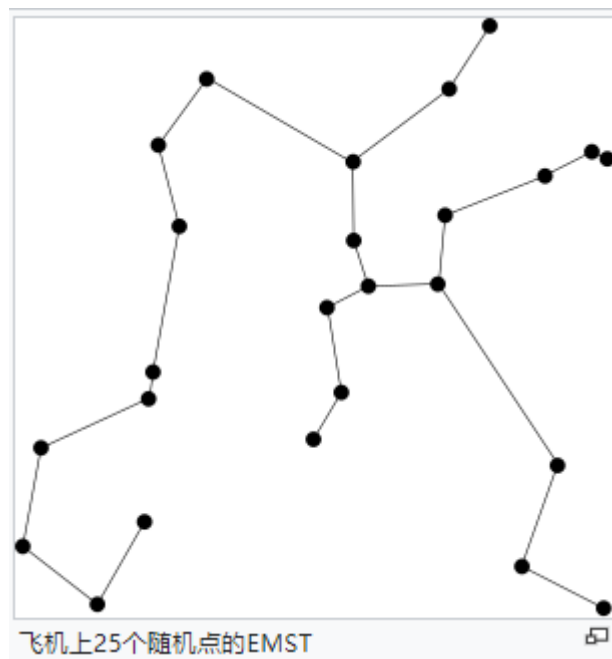
在给定 n 个点的情况下，二维查找 EMST 的最简单算法是在 n 个顶点上实际构建完整图形，该顶点具有 $n(n-1)/2$ 条边，通过找到每对边之间的距离来计算每个边权重，然后在其上运行标准的最小生成树算法（例如 Prim 算法或 Kruskal 算法的版本）。由于该图在 n 个不同的点上具有 $\Theta(n^2)$ 边，因此构造它已经需要 $\Omega(n^2)$ 时间。此解决方案还需要 $\Omega(n^2)$ 存储所有边缘的空间。在一个平面上寻找 EMST 更好的方法是要注意，这是每一个子 Delaunay 三角中的 \tilde{n} 点，大大减少的边集：

1. 计算 $O(n \log n)$ 时间和 $O(n)$ 空间中的 Delaunay 三角剖分。因为 Delaunay 三角剖分是一个平面图，并且在任何平面图中的边都不超过顶点的三倍，所以仅生成 $O(n)$ 边。

2. 用长度标记每个边缘。

3. 在该图上运行图最小生成树算法以查找最小生成树。由于存在 $O(n)$ 条边，因此使用任何标准最小生成树算法（例如 Borůvka 算法，Prim 算法或 Kruskal 算法）都需要 $O(n \log n)$ 时间。

这个问题也可以推广到 \tilde{n} 在点 d 维空间 \mathbb{R}^d 。在更高的维度上，由 Delaunay 三角剖分（同样将凸包划分为 d 维简化）确定的连通性包含最小生成树。但是，三角剖分可能包含完整的图形。[18]因此，找到欧几里得最小生成树作为完整图的生成树还是 Delaunay 三角剖分的生成树都需要 $O(dn^2)$ 时间。对于三个维度，可以找到时间为 $O((n \log n)^{4/3})$ ，并且在任何大于 3 的维度上，都可以在比完整图形和 Delaunay 三角剖分算法的二次时间界更快的时间内求解。[18]对于均匀随机的点集，可以像排序一样快速地计算最小生成树。[19]使用分离良好的对分解，可以在 $O(n \log n)$ 时间产生 $(1 + \varepsilon)$ 逼近。[20]

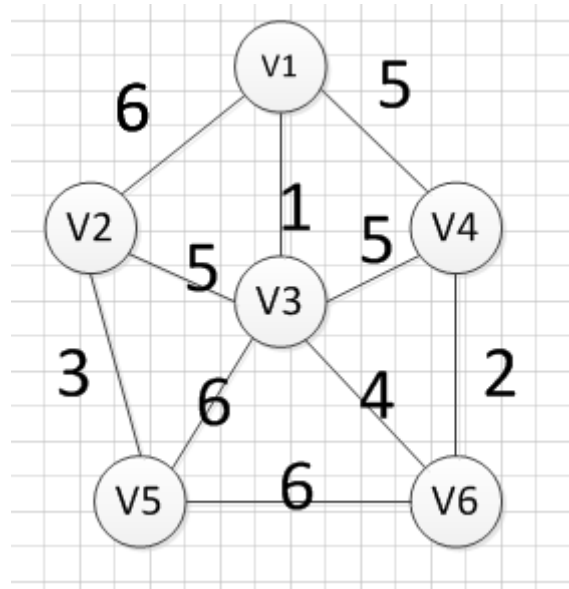


4.7 Prim 算法的实现

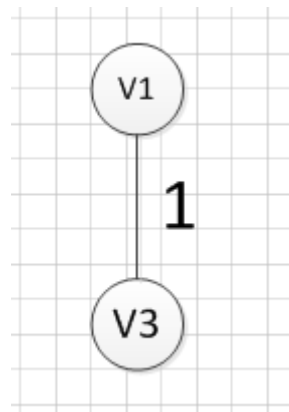
算法思路：

首先就是从图中的一个起点 a 开始，把 a 加入 U 集合，然后，寻找从与 a 有关的边中，权重最小的那条边并且该边的终点 b 在顶点集合： $(V-U)$ 中，我们也把 b 加入到集合 U 中，并且输出边 (a, b) 的信息，这样我们的集合 U 就有： $\{a, b\}$ ，然后，我们寻找与 a 关联和 b 关联的边中，权重最小的那条边并且该边的终点在集合： $(V-U)$ 中，我们把 c 加入到集合 U 中，并且输出对应的那条边的信息，这样我们的集合 U 就有： $\{a, b, c\}$ 这三个元素了，一次类推，直到所有顶点都加入到了集合 U 。

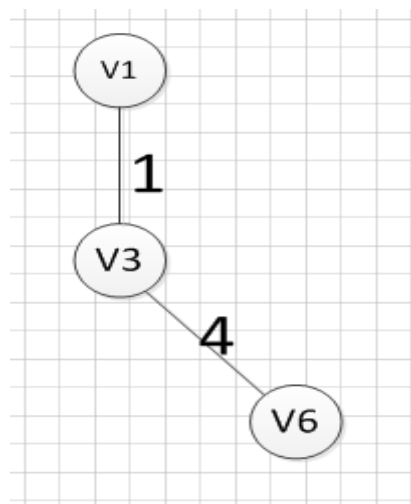
下面我们对下面这幅图求其最小生成树：



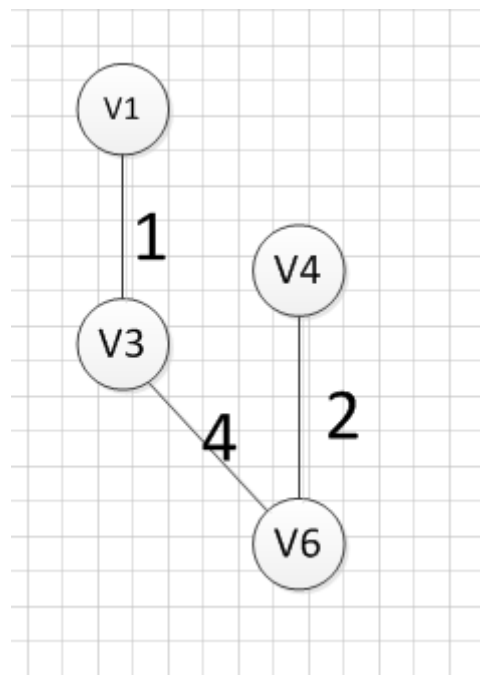
假设我们从顶点 v_1 开始，所以我们可以发现 (v_1, v_3) 边的权重最小，所以第一个输出的边就是： $v_1 - v_3 = 1$ ：



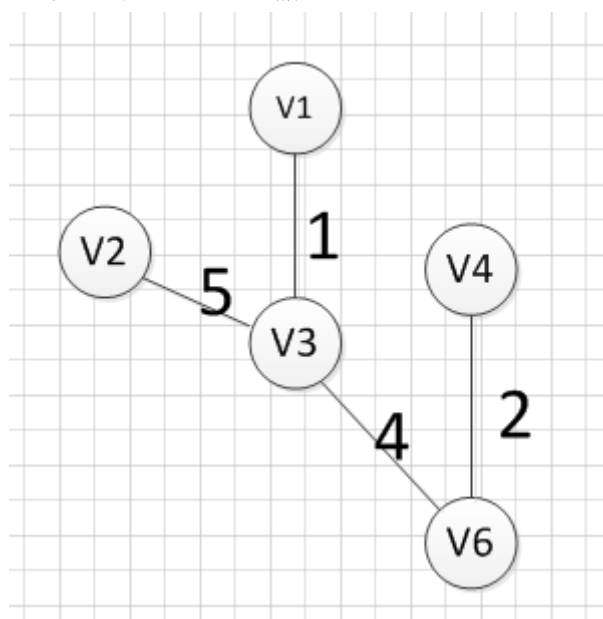
然后，我们要从 v_1 和 v_3 作为起点的边中寻找权重最小的边，首先了 (v_1, v_3) 已经访问过了，所以我们从其他边中寻找，发现 (v_3, v_6) 这条边最小，所以输出边就是： $v_3 - v_6 = 4$



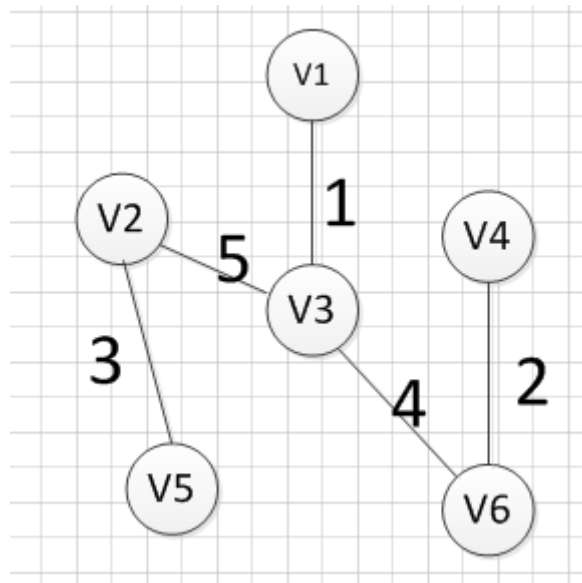
然后，我们要从 v_1 、 v_3 、 v_6 这三个点相关联的边中寻找一条权重最小的边，我们可以发现边 (v_6, v_4) 权重最小，所以输出边就是： $v_6 - v_4 = 2$ 。



然后，我们就从 v_1 、 v_3 、 v_6 、 v_4 这四个顶点相关联的边中寻找权重最小的边，发现边 (v_3, v_2) 的权重最小，所以输出边： $v_3 - v_2 = 5$



然后，我们就从 v_1 、 v_3 、 v_6 、 v_4 、 v_2 这五个顶点相关联的边中寻找权重最小的边，发现边 (v_2, v_5) 的权重最小，所以输出边： $v_2 - v_5 = 3$



最后，我们发现六个点都已经加入到集合 U 了，我们的最小生成树建立完成。

代码实现

(1) 采用的是邻接矩阵的方式存储图，代码如下：

```
#include<iostream>
#include<string>
#include<vector>
using namespace std;

//首先是使用邻接矩阵完成 Prim 算法
struct Graph {
    int vexnum; //顶点个数
    int edge; //边的条数
    int ** arc; //邻接矩阵
    string *information; //记录每个顶点名称
};

//创建图
void createGraph(Graph & g) {
    cout << "请输入顶点数： 输入边的条数" << endl;
    cin >> g.vexnum;
    cin >> g.edge; //输入边的条数

    g.information = new string[g.vexnum];
    g.arc = new int*[g.vexnum];
    int i = 0;

    //开辟空间的同时， 进行名称的初始化
```

```

    for (i = 0; i < g.vexnum; i++) {
        g.arc[i] = new int[g.vexnum];
        g.information[i] = "v" + std::to_string(i+1); //对每个顶点进行命名
        for (int k = 0; k < g.vexnum; k++) {
            g.arc[i][k] = INT_MAX; //初始化我们的邻接矩阵
        }
    }

    cout << "请输入每条边之间的顶点编号(顶点编号从 1 开始),以及该边的权重:" << endl;
    for (i = 0; i < g.edge; i++) {
        int start;
        int end;
        cin >> start; //输入每条边的起点
        cin >> end; //输入每条边的终点
        int weight;
        cin >> weight;
        g.arc[start-1][end-1] = weight; //无向图的边是相反的
        g.arc[end-1][start-1] = weight;
    }
}

//打印图
void print(Graph g) {
    int i;
    for (i = 0; i < g.vexnum; i++) {
        //cout << g.information[i] << " ";
        for (int j = 0; j < g.vexnum; j++) {
            if (g.arc[i][j] == INT_MAX)
                cout << "∞" << " ";
            else
                cout << g.arc[i][j] << " ";
        }
        cout << endl;
    }
}

//作为记录边的信息，这些边都是达到 end 的所有边中，权重最小的那个
struct Assis_array {
    int start; //边的终点
    int end; //边的起点
    int weight; //边的权重
};

//进行 prim 算法实现，使用的邻接矩阵的方法实现。
void Prim(Graph g, int begin) {

```

```
//close_edge 这个数组记录到达某个顶点的各个边中的权重最大的那个边
Assis_array *close_edge=new Assis_array[g.vexnum];
```

```
int j;
```

```
//进行 close_edge 的初始化，更加开始起点进行初始化
```

```
for (j = 0; j < g.vexnum; j++) {
    if (j != begin - 1) {
        close_edge[j].start = begin-1;
        close_edge[j].end = j;
        close_edge[j].weight = g.arc[begin - 1][j];
    }
}
```

```
//把起点的 close_edge 中的值设置为-1，代表已经加入到集合 U 了
```

```
close_edge[begin - 1].weight = -1;
```

```
//访问剩下的顶点，并加入依次加入到集合 U
```

```
for (j = 1; j < g.vexnum; j++) {
```

```
    int min = INT_MAX;
```

```
    int k;
```

```
    int index;
```

```
    //寻找数组 close_edge 中权重最小的那个边
```

```
    for (k = 0; k < g.vexnum; k++) {
        if (close_edge[k].weight != -1) {
            if (close_edge[k].weight < min) {
                min = close_edge[k].weight;
                index = k;
            }
        }
    }
```

```
}
```

```
//将权重最小的那条边的终点也加入到集合 U
```

```
close_edge[index].weight = -1;
```

```
//输出对应的边的信息
```

```
cout << g.information[close_edge[index].start]
    << "-----"
    << g.information[close_edge[index].end]
    << "="
    <<g.arc[close_edge[index].start][close_edge[index].end]
    <<endl;
```

```
//更新我们的 close_edge 数组。
```

```
for (k = 0; k < g.vexnum; k++) {
    if (g.arc[close_edge[index].end][k] <close_edge[k].weight) {
```



```

        close_edge[k].weight = g.arc[close_edge[index].end][k];
        close_edge[k].start = close_edge[index].end;
        close_edge[k].end = k;
    }
}
}
}
}

```

```

int main()
{
    Graph g;
    createGraph(g); //基本都是无向网图，所以我们只实现了无向网图
    print(g);
    Prim(g, 1);
    system("pause");
    return 0;
}

```

时间复杂度的分析：

其中我们建立邻接矩阵需要的时间复杂度为： $O(n^2)$ ，然后，我们 Prim 函数中生成最小生成树的时间复杂度为： $O(n^2)$ 。

（2）采用的是邻接表的方式存储图，代码如下：

```

#include<iostream>
#include<string>
using namespace std;
//表结点
struct ArcNode {
    int adjvex;        //某条边指向的那个顶点的位置（一般是数组的下标）。
    ArcNode * next;    //指向下一个表结点
    int weight;        //边的权重
};
//头结点
struct Vnode {
    ArcNode * firstarc; //第一个和该顶点依附的边 的信息
    string data;        //记录该顶点的信息。
};

struct Graph_List {
    int vexnum;        //顶点个数
    int edge;          //边的条数
    Vnode * node;      //顶点表
};

```

//创建图，是一个重载函数

```
void createGraph(Graph_List &g) {
    cout << "请输入顶点数： 输入顶点边的个数： " << endl;
    cin >> g.vexnum;
    cin >> g.edge;
    g.node = new Vnode[g.vexnum];
    int i;
    for (i = 0; i < g.vexnum; i++) {
        g.node[i].data = "v" + std::to_string(i + 1); //对每个顶点进行命名
        g.node[i].firstarc = NULL; //初始化每个顶点的依附表结点
    }

    cout << "请输入每条边之间的顶点编号(顶点编号从 1 开始),以及该边的权重:" << endl;
    for (i = 0; i < g.edge; i++) {
        int start;
        int end;
        cin >> start; //输入每条边的起点
        cin >> end; //输入每条边的终点
        int weight;
        cin >> weight;

        ArcNode * next = new ArcNode;
        next->adjvex = end - 1;
        next->next = NULL;
        next->weight = weight;
        //如果第一个依附的边为空
        if (g.node[start - 1].firstarc == NULL) {
            g.node[start - 1].firstarc = next;
        }
        else {
            ArcNode * temp; //临时表结点
            temp = g.node[start - 1].firstarc;
            while (temp->next) { //找到表结点中 start-1 这个结点的链表的最后一个顶点
                temp = temp->next;
            }
            temp->next = next; //在该链表的尾部插入一个结点
        }

        //因为无向图边是双向的
        ArcNode * next_2 = new ArcNode;
        next_2->adjvex = start - 1;
        next_2->weight = weight;
        next_2->next = NULL;
```

```

        //如果第一个依附的边为空
        if (g.node[end - 1].firstarc == NULL) {
            g.node[end - 1].firstarc = next_2;
        }
        else {
            ArcNode * temp; //临时表结点
            temp = g.node[end - 1].firstarc;
            while (temp->next) { //找到表结点中 start-1 这个结点的链表的最后一个顶点
                temp = temp->next;
            }
            temp->next = next_2; //在该链表的尾部插入一个结点
        }
    }
}

void print(Graph_List g) {
    cout<<"图的邻接表: "<<endl;
    for (int i = 0; i < g.vexnum; i++) {
        cout << g.node[i].data << " ";
        ArcNode * next;
        next = g.node[i].firstarc;
        while (next) {
            cout << "(" << g.node[i].data
            << ", "<< g.node[next->adjvex].data << ")=" << next->weight << " ";
            next = next->next;
        }
        cout << "^" << endl;
    }
}

```

////作为记录边的信息，这些边都是达到 end 的所有边中，权重最小的那个

```

struct Assis_array {
    int start; //边的终点
    int end; //边的起点
    int weight; //边的权重
};

```

```

void Prim(Graph_List g, int begin) {
    cout << "图的最小生成树: " << endl;
    //close_edge 这个数组记录到达某个顶点的各个边中的权重最大的那个边
    Assis_array *close_edge=new Assis_array[g.vexnum];
}

```

```

int j;
for (j = 0; j < g.vexnum; j++) {
    close_edge[j].weight = INT_MAX;
}
ArcNode * arc = g.node[begin - 1].firstarc;

while (arc) {
    close_edge[arc->adjvex].end = arc->adjvex;
    close_edge[arc->adjvex].start = begin - 1;
    close_edge[arc->adjvex].weight = arc->weight;
    arc = arc->next;
}
//把起点的 close_edge 中的值设置为-1，代表已经加入到集合 U 了
close_edge[begin - 1].weight = -1;
//访问剩下的顶点，并加入依次加入到集合 U
for (j = 1; j < g.vexnum; j++) {
    int min = INT_MAX;
    int k;
    int index;
    //寻找数组 close_edge 中权重最小的那个边
    for (k = 0; k < g.vexnum; k++) {
        if (close_edge[k].weight != -1) {
            if (close_edge[k].weight < min) {
                min = close_edge[k].weight;
                index = k;
            }
        }
    }
}

//输出对应的边的信息
cout << g.node[close_edge[index].start].data
    << "-----"
    << g.node[close_edge[index].end].data
    << "="
    << close_edge[index].weight
    << endl;
//将权重最小的那条边的终点也加入到集合 U
close_edge[index].weight = -1;
//更新我们的 close_edge 数组。
ArcNode * temp = g.node[close_edge[index].end].firstarc;
while (temp) {
    if (close_edge[temp->adjvex].weight > temp->weight) {
        close_edge[temp->adjvex].weight = temp->weight;
        close_edge[temp->adjvex].start = index;
    }
}

```

```

        close_edge[temp->adjvex].end = temp->adjvex;
    }
    temp = temp->next;
}
}

}

int main()
{
    Graph_List g;
    createGraph(g);
    print(g);
    Prim(g, 1);
    system("pause");
    return 0;
}

```

时间复杂分析：

在建立图的时候的时间复杂为： $O(n+e)$ ，在执行 Prim 算法的时间复杂还是： $O(n*n)$ ，总体来说还是邻接表的效率会比较高，因为虽然 Prim 算法的时间复杂度相同，但是邻接矩阵的那个常系数是比邻接表大的。

另外，Prim 算法的时间复杂度都是和边无关的，都是 $O(n*n)$ ，所以它适合用于边稠密的网建立最小生成树。但是了，我们即将介绍的克鲁斯卡算法恰恰相反，它的时间复杂度为： $O(e\log e)$ ，其中 e 为边的条数，因此它相对 Prim 算法而言，更适用于边稀疏的网。

5. 参考

1. Minimum Spanning Trees. princeton.edu. 2015-09-13 [2016-02-08] (英语).
2. Robert Sedgewick, Kevin Wayne. 算法. 北京: 人民邮电出版社. 2012 年 10 月. ISBN 978-7-115-29380-0. , p391--p392
3. Prim, R. C., Shortest connection networks And some generalizations, Bell System Technical Journal, November 1957, 36 (6): 1389 - 1401, doi:10.1002/j.1538-7305.1957.tb01515.x.
4. Dijkstra, E. W., A note on two problems in connexion with graphs (PDF), Numerische Mathematik, 1959, 1: 269 - 271, doi:10.1007/BF01386390.
5. Jarník, V., O jistém problému minimálním [About a certain minimal problem], Práce Moravské Přírodovědecké Společnosti, 1930, 6: 57 - 63 (捷克语).
6. Robert Sedgewick, Kevin Wayne. 算法. 北京: 人民邮电出版社. 2012 年 10 月. ISBN 978-7-115-29380-0. , p407--p408
7. Borůvka, Otakar (1926). “O jistém problému minimálním” [关于某个最小问题]. 普拉斯·莫 (Práce Mor). Přírodověd. Spol. V Brně III (捷克文和德文). 3: 37-58.
8. Borůvka, Otakar (1926). “Příspěvek k řešení otázky ekonomické stavby elektrovedních sítí (对解决电网经济建设问题的贡献)”. Elektronický Obzor (捷克语). 15: 153-154.
9. Nešetřil, 雅罗斯拉夫; 伊娃 (Eva) Milková; Nešetřilová, 海伦娜 (2001). “关于最小生成树问题的 Otakar Borůvka: 1926 年论文, 评论和历史的翻译”. 离散数学. 233 (1-3): 3 - 36. doi: 10.1016 / S0012-365X (00) 00224-7. hdl: 10338.dmlcz / 500413. MR 1825599.
10. Choquet, 古斯塔夫 (1938). “路线上的某些人”. 收集科学研究院的 Rendus de l'Académie (法语). 206: 310-313.
11. Florek, K .; Łukaszewicz, J. Perkal, J .; 斯坦豪斯, 雨果; Zubrzycki, S. (1951 年). “拉河畔联络等拉德司德点未合奏菲尼”. 数学专题讨论会 (法文). 2: 282 - 285. MR 0048832.
12. 乔治·索林 (1965). “运河改造”. 编程, 游戏和运输网络 (法语).
13. Karger, David R.; Klein, Philip N.; Tarjan, Robert E., A randomized linear-time algorithm to find minimum spanning trees, Journal of the Association for Computing Machinery, 1995, 42 (2): 321 - 328, MR 1409738, doi:10.1145/201019.201022
14. Pettie, Seth; Ramachandran, Vijaya, Minimizing randomness in minimum spanning tree, parallel connectivity, and set maxima algorithms, Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA '02), San Francisco, California: 713 - 722, 2002.

15. Chazelle, Bernard, A minimum spanning tree algorithm with inverse-Ackermann type complexity, Journal of the Association for Computing Machinery, 2000, 47 (6): 1028 - 1047, MR 1866456, doi:10.1145/355541.355562.
16. Chazelle, Bernard, The soft heap: an approximate priority queue with optimal error rate, Journal of the Association for Computing Machinery, 2000, 47 (6): 1012 - 1027, MR 1866455, doi:10.1145/355541.355554
17. Buchin, 凯文; 穆泽 (Mulzer), 沃尔夫冈 (Wolfgang) (2009)。O (sort (n)) 时间及更多时间中的 Delaunay 三角剖分 (PDF)。程序第 50 届 IEEE 计算机科学基础研讨会。第 139 - 148 页。doi: 10.1109 / FOCS.2009.53
18. Agarwal, PK; Edelsbrunner, H. O.Schwarzkopf; Welzl, E. (1991), “欧几里得最小生成树和双色最接近对”, 离散与计算几何, Springer, 6 (1): 407-422, doi: 10.1007 / BF02574698。
19. Chatterjee, S.; 康纳, M. Kumar, P. (2010), “带有 GeoFilterKruskal 的几何最小生成树”, 在 Festa, Paola (编), 实验算法专题讨论会, 计算机科学讲义, 6049, Springer-Verlag, 第 486 - 500 页, doi: 10.1007 / 978-3-642-13193-6_41。
20. Smid, Michiel (2005 年 8 月 16 日)。“分离良好的对分解及其应用” (PDF)。