|  |  |
| --- | --- |
| **学 号：** | 0121801100913 |



**《算法设计与分析B》课程报告**

|  |  |
| --- | --- |
| **题 目** | **基于贪心法的最小生成树问题及其基本应用** |
| **学 院** | **计算机科学与技术** |
| **专 业** | **软件卓越工程师** |
| **班 级** | **软件zy1802** |
| **姓 名** | **张淳慷** |
| **指导教师** | **李晓红** |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 2020 | 年 | 5 | 月 | 19 | 日 |

# 摘 要

最小生成树是图论中的一个既经典又重要的问题，它体现在图这种数据结构上的应用，涉及到我们的日常，工程，学术以及科研等方方面面，最小生成树广泛的应用价值主要取决于其通俗简单的的理论算法和严密成熟的数学基础。

本文几种介绍最小生成树的特征规律；其次给出几种最小生成树的经典算法，

在此基础上对这些算法的优缺点进行分析比较，得到一些结论；最后介绍与最小生成树经典算法以及最小生成树有关的实例并进行分析总结。

**关键词**：最小生成树；Prim算法；Kruskal算法；应用

## 1选题背景

现如今，随着现代社会与科技的不断发展，许多实际上的应用问题都是一个求无向连通图的最小生成树问题。例如：要在n个城市之间建立高速公路，主要目标是要使这n个城市的任意两个之间都可以来往，但搭建高速公路的费用昂贵，而且各个城市之间建立高速公路的费用不同；另一个目标则是要是建立高速公路的总费用最低，不难看出，这就需要找到带权的最小生成树。又或者是城市内部光缆的铺设等等，许多研究工作表明，最小生成树结果是通信网络设计的最优拓扑结构。哪怕在如今各以最小生成树算法为基础衍生出来的变体如广义最小生成树或者支撑树搜索方法等依然被高度应用在世界的各个领域当中，它们也正在逐渐显现出其不可被替代的优越性。

本文的基本撰写思路是：根据上文的介绍，对最小生成树的经典算法进行分析与研究，以此，对算法的优缺点以及适用的应用进行更深入的分析，具体的安排如下：

1：主要介绍最小生成树的形成的方式以及相关的定理以及推论。

2：就最小生成树的经典算法进行讨论，首先介绍两种算法的描述以及实现，

包括Prim算法和Kruskal算法。其次，对这两种算法的复杂性和优缺点进行分析比较，并以此得出其使用范围

3：通过应用实例将以上所介绍的算法进行实际的应用避免大谈理论所导致的空洞乏味，其中重点解析Prim算法解决实际问题的过程以及相关代码

4：对全文进行小结，并给出自己在本次课设中的收获与体会

5：展示两种算法的具体代码以及与实例相关联的Prim方法的代码。

# 2最小生成树的形成

本章将大致给出最小生成树产生的基本方法，首先将会给出最小生成树的基本定义和表现形式；然后，将会介绍最小生成树的性质及图论的基本理论。最后将给出最小生成树产生的伪代码及核心思想。

## 2.1 最小生成树定义

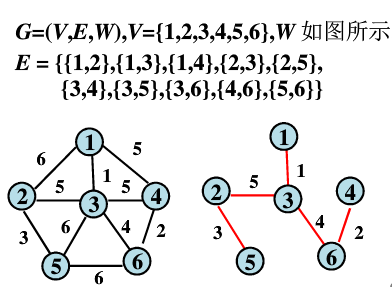
**定义2.1：**设图=<>,=<>,如果，,则称为的子图，当时，称为的生成子图。

**定义2.2：**如果图T是无向图G的生成子图且T是树，则T是G的生成树.

**定义2.3：**当给图G赋予映射,或，为任意集合，则此时成G为赋权图，称为顶点的权，称为边的权。

**定义2.4**：设G为连通的边赋权图，T为G的生成树，则T的各边权之和

称为生成树T的权，权最小的生成树称为最小生成树。

最小生成树实例

## 2.2 最小生成树产生的基本算法策略

本文所讨论的两种算法都是使用贪心策略来解决最小生成树的问题，但是他们使用贪心策略的方式却是有所不同。

**2.2.1贪心策略的基本思想**：其实该思想可以由一个通用方法来描述，该通用方法在每个时刻生长最小生成树的一条边，并在整个策略的实施过程中，管理一个遵守下述循环不变式的边集合A：**在每遍循环之前，A是某颗最小生成树的一个子集**。伪码如下：

1 A =

2 **While** A不形成一个生成树

3 遍历目标图找到可以正常纳入A的边（u，v）

4

5  **return** A

这条伪码便是在说，进入循环后，A通过只添加正常边的方式来不断重复，而直到A真正形成一颗最小生成树的时候，便将其输出。

通过算法我们不难发现，满足该条件的边是必然存在的，因为在执行第3行时，该循环式会通过判断表现出来一定存在一颗生成树T使得A是其真子集（否则跳出循环）那么循环之中该生成树T的一条边必然满足其不在A之中而且可以加入A。

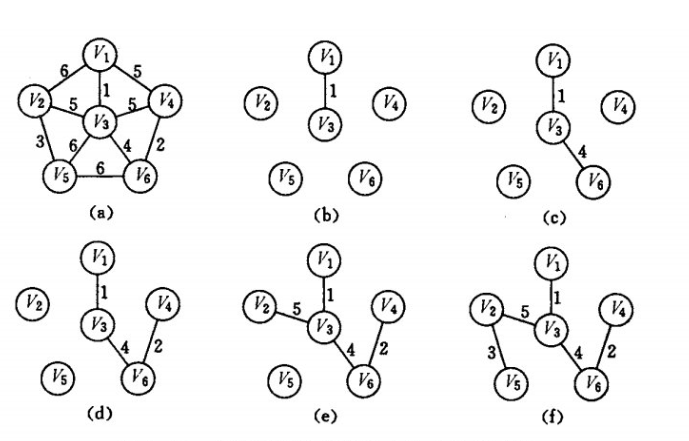
所以说整个最小生成树算法的根本策略就是要得到一个方法使得当一条边加入到A时，其仍为某颗最小生成树的子集，于是就有了接下来的算法。

# 3最小生成树具体算法实现

最小生成树主要有两种算法：Prim算法和Kruskal算法，本章将重点探讨这两种算法。

## 3.1 Prim算法

**3.1.1算法思路分析：**Prim算法所具有的一个性质是集合A中的边总是构成一棵树，这棵树从一个任意的节点开始，已知长大覆盖到V中的所有节点为止，其每一步在连接集合A与A之外的节点的所有边中选择一条权重最小的边加入其中，这便是Prim利用贪心策略的地方，找寻已连接顶点中的最小权边，之后不断重复以上步骤直到其形成最小生成树，图例如下：



不难发现，找生成树的过程中可能出现两条最小权值一样的边，不同的选择可能会导致最终生成树的不同，但是最终结果的树的权值是一样的。

**3.1.2 代码实现及分析：**

得知了基本的运行流程思路后，接下来便要进行伪码的实现（**具体代码见附录**）在下面的伪代码中，连通图G和最小生成树的树节点r将作为算法的输入。在算法的执行过程中，所有不在树A（边集合）中的节点都存放在一个最小优先队列Q中，对于每个节点v，属性v.key保存的是连接v和树中节点的所有边的最小边的权重，若这样的边不存在，则v.key=。属性v.给出的是v的父节点。当最小优先队列为空时，算法终止，测试将得到G的最小生成树A=

Prim（G,,r）

1 for each u∈G,V

2 u:key =

3 u: = NIL

4 r:key = 0

5 Q=G.V

6 While Q

7 u = EXTRACT-MIN(Q)

8 for each v∈G.Adj[u]

9 if v∈Q and () < v.key

10 v. = u

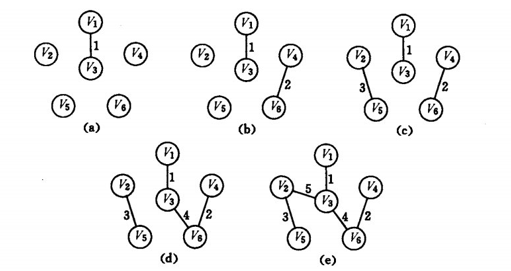
11 v.key =()

**代码解释**：该代码的第1-5行将每个节点的key值设为（根节点r设为0，以便使其处理），将每个节点的父节点设置为NIL，并对最小优先队列进行初始化，使其包含图中所有的结点。而代码的6-11行，即循环体则有如下的作用：在每一遍循环之前，A=,已经加入到最小生成树的结点为集合V-Q，且对于所有的节点v∈Q，如果v.NIL.则v.key是连接结点v和最小生成树中某个节点的最小权边（,.）的权重。第7行中的**EXTRACT**找出结点∈Q，它是某条A与外部结点的最小权边上的一个端点，紧接着便将u从队列Q中删除并加入到集合V-Q中，（主要的时间全耗在这部分上面）即将（,.）加入到集合A中，8-11行的for循环将每个与邻接但不在树中的节点的key与属性更新，从而使其已知循环下去，知道得到结果。

**算法分析：**如果最小优先队列是以二叉最小优先队列实现而且采用邻接表的方法的话，不难看出while循环的语句一共要执行V次（入队V次），而其中每个EXTRACT的操作时间成本为O()（排序）,所以其总时间为O（），根据时间复杂度不难发现该方法更加适用于稠密图。

## 3.2 Kruskal算法

**3.2.1算法思路分析：**先构造一个只含 n 个顶点，而边集为空的子图，若将该子图中各个顶点看成是各棵树上的根结点，则它是一个含有 n 棵树的一个森林。之后，从网的边集 E 中选取一条权值最小的边（贪心策略），若该条边的两个顶点分属不同的树，则将其加入子图，也就是说，将这两个顶点分别所在的两棵树合成一棵树；反之，若该条边的两个顶点已落在同一棵树上，则不可取，而应该取下一条权值最小的边再试之。依次类推，直至森林中只有一棵树，也即子图中含有 n-1条边为止。



Kruskal算法图例展示

**3.2.2代码实现及分析：**得知了基本的运行流程思路后，接下来便要进行伪码的实现（**具体代码见附录**）其中A为最小生成树集合，输入为带求的图的结点V和边E

Kruskal (V,E)

1{ A = { } ;

2 while ( A 中不到 |V| −1 条边 && E 中还有边 ) {

3 从 E 中取一条权重最小的边 ;

4 将 从 E 中删除;

5 if ( 不在 MST 中构成回路)

6 将 加入 MST;

7 else

8 彻底无视 ;

9 }

10 if ( MST 中不到 |V| −1 条边 )

11 Error ( “生成树不存在” );

}

**代码解释：**第三行可以采用最小堆的方式将边按照权重的大小存入对中进行循环获取，在寻找到最小的边之后便会将其取出并从E中删除，而5-8行则是对取出的边进行判断其是否在最小生成树中构成回路，在第5行判断的时候可以采用并查集的方法，当把两条树收入的时候，及时将两个集合合并成了一个，判断时如果v与w本属于不同的树的话则可以正常加入，否则将构成回路不符合算法的基本要求

**算法分析：主要时间耗费在3中，也即是利用最小堆根据权重大小进行排序，该时间复杂度根据堆排序可知为O（）,而while循环中的 重复以及判断并查集的与最终加入的时间复杂度综合为O（V+E），综上，该算法的时间复杂度为O（），所以说该算法更加适用于稀疏图。**

## ****3.3 两种算法的比较：****

从两种算法的实现步骤与设计思路可以看出：

1 Prim算法实现起来简洁清晰易懂，并且使用的判断语句较少，而Kruskal算法实现起来步骤多，判断语句多。

2 从结果来看，利用Prim算法从顶点出发求出用邻接矩阵G来表示图的最小生成树与利用Kruskal算法求结构体数组的最小生成树是一致的

3 从程序本身看，Kruskal算法的时间复杂度为**O（），与顶点数无关，适合求稀疏图的最小生成树，而Prim算法的时间复杂度为O()，与边数无关，所以适合求边稠密的图的最小生成树。**

**4 从贪心策略的角度来看，**Prim算法从顶点的角度出发，每次选择距离当前节点最近的节点加入，直到所有节点都加入。  
Kruskal算法从边的角度出发，每次总是选择权重最小的边加入，直到加入n-1条边为止。（如果加入一条边后出现回路，skip这条边）。

# 4实例展示

本章将重点与实际例子结合来讨论算法的使用，主要集中在Prim算法的讨论上。

## 4.1 实例描述

正如之前第一章所说到的，最小生成树可以用来解决实际生活中如电缆铺设，高速公路搭建等问题，那么不如看看其实际运用中的效果，以实例来作为佐证。一国有n个城市，它们互相之间没有公路相通，因此交通十分不便。为解决这一“行路难”的问题，政府决定修建公路。修建公路的任务由各城市共同完成。修建工程分若干轮完成。在每一轮中，每个城市选择一个与它最近的城市，申请修建通往该城市的公路。政府负责审批这些申请以决定是否同意修建。

政府审批的规则如下：

（1）如果两个或以上城市申请修建同一条公路，则让它们共同修建；

（2）如果三个或以上的城市申请修建的公路成环，如A申请修建公路AB，B申请修建公路BC，C申请修建公路CA。则政府将否决其中最短的一条公路的修建申请；

一轮修建结束后，可能会有若干城市可以通过公路直接或间接相连。这些可以互相：连通的城市即组成“城市联盟”。在下一轮修建中，每个“城市联盟”将被看作一个城市，发挥一个城市的作用。当所有城市被组合成一个“城市联盟”时，修建工程也就完成了。

输入：第一行一个整数n，表示城市的数量。(n≤5000)以下n行，每行两个整数x和y，表示一个城市的坐标。(-1000000≤x，y≤1000000)

输出：一个实数，四舍五入保留两位小数，表示公路总长。（保证有惟一解）

## 4.2实例分析

在给出代码之前，不妨先思考一下题目的两条规则，第一条规则显然适用于最小生成树的产生，而第二条规则不妨思考一下，看似是十分复杂的情况，而实际上是说**“存在三个或三个以上的城市，他们两两间的最近城市连起来成环”，不难看出，这一条件对于最小生成树显然是不成立的，因为最小生成树不能成环，所以说在经过翻译后可以发现，该题就是一个经典的最小生成树问题，而其中城市的输入数量达到了5000,所以本题所形成的图是一个稠密图，所以显然采取Prim算法,**因为在Kurscal算法中, 必须事先求出所有边的长度才能对之排序. 但是一个有5000节点的完全图, 这样做的空间开销是巨大的. Prim算法只在更新点到树的距离时需要用到边长, 所以更为适用。

## 4.3设计思路

我们可以把一棵树理解成一个有智能的生命, 可以感知它附近的点到它的距离. 每次生长枝条, 它都选择离它最近的那个点.点到树的距离, 是指树外一个点到树上的任意点的最小距离.所以,在代码实现的时候, 需要维护这样一个数组: 树外的点到树的距离. 所以, 还需要区分一下点究竟在树上还在树外.维护数组就是要做两件事: 更改数组和调用数组.

何时更改: 树外的点到树的距离发生变化. 这种事只能在树生长了新的枝条的时候发生, 因为新加入的那个点可能更新了树到树外的某个点的距离. 同时, 新加入的那个点变成了树上的点.

何时调用: 想扩张的时候, 找最近的树外点.有了这些思路就能写代码了。

## 4.4代码展示以及分析

#include<iostream>

#include<cstdio>

#include<algorithm>

#include<cstring>

#include<cmath>

1 using namespace std;

2 const long long maxn=1e4+5;

3 const long long inf=1e9+5;

4 long long n,x[maxn],y[maxn],vis[maxn];

5 double d[maxn],ans;

6 double get(long long i,long long j){

7 return sqrt((x[i]-x[j])\*(x[i]-x[j])+(y[i]-y[j])\*(y[i]-y[j]));

8 }//各种初始化声明

9 void prim(){

10 for(long long i=1;i<=n;i++) d[i]=get(1,i);//在prim中，d数组表示的是第i个点到图中已经有的所有的点的最短距离，因为用二维存距离的话会导致空间不足，所以只有需要用到时再去处理。

11 d[0]=inf;

12 for(long long i=1;i<n;i++){

13 long long xi=0;

14 for(long long j=1;j<=n;j++){

15 if(!vis[j]&&(xi==0||d[j]<d[xi])) xi=j;//找到距离最近的那个点

16 }

17 vis[xi]=1;//往树上加一个点

18 ans+=d[xi];

19 for(long long j=1;j<=n;j++){

20 if(!vis[j]) d[j]=min(d[j],get(xi,j));//更新点距离部分

21 }

22 }

23 }

24 int main(){

25 scanf("%lld",&n);

26 for(long long i=1;i<=n;i++){

27 scanf("%lld%lld",&x[i],&y[i]);

28 }

29 vis[1]=1;

30 prim();

31 printf("%.2lf",ans);//输出答案

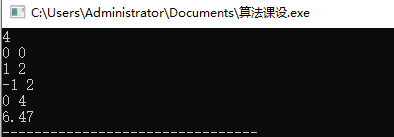
32 return 0;

33 }

**算法分析：**直接看向12到18行的循环体，其为本次代码的核心，其目的可以这么理解我们图中的第一个点是1，已经求得了所有和1相连的点的距离，于是可以求出这个点具体是哪个点，和这条最短的边是多少，将当前这个点加入最小生成树的点集中去，此时树上就已经有2个点了，下一步，要求的是与这两个点相邻的所有的点到这两个点距离最短的一个点，因为d[i]之前表示的是到1的距离，此时在与到这个新的点的距离作比较以决定更不更新这个点，更新完之后，d[i]表示的是i这个点到目前已经构成的最小生成树的最短距离，也就是prim所需要的，之后便可以依次找点。

**不足：**因为两层循环的存在，所以这段代码的时间复杂度为O（）。与之前分析所用的最小堆为数据结构不同，本次实验代码并未采取该方法，于是导致时间复杂度相对较大如果只是面对5000的输入数据尚且可以勉强应对，如果输入数据过大则会导致无结果输出，所以我们可以采用**堆优化**之后的Prim算法作为求解。（具体代码将在附录展示）。

**测试结果：**



# 5 课设小结及展望

老实讲这可能是我第一次查阅如此多的资料去完成一篇论文，虽然说可能这篇论文在许多层面上还达不到身为一篇论文的要求，但是不得不说在写这篇论文的过程中我的收获还是很大的。先从最小生成树这个选题开始讲起，其实最开始知道最小生成树这个概念就是在学离散结构的时候，之后又在学习数据结构的时候了解了它的算法，但是无一例外自己都没有什么机会进行实操，，一直都停留在自己认为自己掌握的基础上，既不能确定自己是否真正掌握也不确定自己能否把肚子里的货成功讲述出来给别人听，而最小生成树作为图论中比较有发散性和延展性的一类问题我觉得是有必要进行详细了解的，于是便有了这篇论文。

当时在到处查阅资料的时候就发现了我对于两种算法一直以来的误区，当时听数据结构的自己始终相信我只需要掌握一种算法便足矣，毕竟我当时认为只要能够把问题解决就行，也正是这种思想导致我一直到写这篇论文之前我都只熟悉Kruskal算法，不过也正是在查阅了资料以及其他人写的论文之后才知道了二者真正的差别，更别说是算法内部数据存储结构的不同所带来的时间复杂度的不同，就拿Prim算法来说，原来我从来没有想过可以用堆的方式对其进行优化，只觉得可以解决当下的问题就足矣，其实这种思想真的是特别坏事的，容易让我错过许多机会，努力改正这种思想也是我在做完这次课设之后最大的收获。

还有一点收获便是我真真正正感觉到了算法本身的魅力，不同的思考，不同的效率，有的时候经过深思熟虑的几行代码便可以轻松替代之前的几百行代码，并且可能还会有更简单的代码去替代这寥寥数行代码，只不过是算法思想上的差距，算法实在是迷人，一方面它让我们陷入思维桎梏，一方面又让我们跳出它。其实仔细想想如果可以活用这种想法对于自己的人生成长都是特别有帮助的，有的时候自己总是会陷到一种思想里无法跳出来，最终会导致许多时间乃至是机会的浪费，而时刻去寻找一种更好的解决方法，去提升自己的思维灵敏度正是改变这种情况的解药，也是算法本身最大的要求，学会去以一种不同的角度看待问题，跳出自己思维的限制，当然了，想做到这个程度不仅仅是需要过硬的基础知识，可能也需要有数十万的编码量，虽然看似遥远，但是也给了我予以前行的目标，正想《三体》里的那句话所说的，“当人类看向天空思考的时候，他便能冲出蓝天了。”希望自己这颗幼苗有一天也可以冲出那边蓝天吧。

再把话题拉回到最小生成树本身思考整个最小生成树提出的初衷，其实是为了节省最大的资源，或者说用于更加宏观的概念比如道路联通或者光缆铺设等等，我们不妨把目光转向微观，最小生成树的思想同样适用于电子元器件的部署和连接，如果让我自己来头脑风暴的话，我甚至觉得他可以作为理解人类神经元的一个基本思路，我们可以把人类的神经元看作是一个点集，彼此之间突触看做是边，那么这个问题是否就变成了提高我们自己的思维效率？虽然说早在高中我就读过与模拟人脑相关的书籍，有些是从纯数学角度进行模拟，有些则是结合了生物学的思想，但是无一突破神经元模拟的这项难点，哪怕是如今火爆的深度学习的卷积神经网络（CNN）也依然在客观层面上知识简单的模仿我们神经元的工作原理而已，说不定这种图论可以作为打破这些的契机？当然了，这不过只是个基础知识尚且缺乏的本科生提出的一点妄想而已，或许可以达成或许早已被否定，无论如何，希望以后的自己可以从事这方面的工作，让我在写这篇论文时所产生的萌芽已知成长下去。

# 参考文献

[1]Robert Sedgewick, Kelvin Wayne, Algorithms Fourth Edition , 人民电邮出版社， 2012

[2]陶午沙， 沈振康， 李吉成，一种基于模糊信息融合的Prim算法及应用[J]国防科技大学，2005,27

[3]Anany Levitin, Introduction to The Design and Analysis of Algorithms, 北京：清华大学出版社， 2008

[4]刘铎， 离散数学及应用，清华大学出版社，2016

[5]Mark Allen Weiss， Data Structures and Algorithm Analysis in C，机械工业出版社，2015

[6]王元元，张桂荟，计算机科学中的离散结构[M].北京：机械工业出版社，2004

# 源代码

1 **Prim算法具体实现**：

Vertex FindMinDist( MGraph Graph, WeightType dist[] )

{ /\* 返回未被收录顶点中dist最小者 \*/

Vertex MinV, V;

WeightType MinDist = INFINITY;

for (V=0; V<Graph->Nv; V++) {

if ( dist[V]!=0 && dist[V]<MinDist) {

/\* 若V未被收录，且dist[V]更小 \*/

MinDist = dist[V]; /\* 更新最小距离 \*/

MinV = V; /\* 更新对应顶点 \*/

}

}

if (MinDist < INFINITY) /\* 若找到最小dist \*/

return MinV; /\* 返回对应的顶点下标 \*/

else return ERROR; /\* 若这样的顶点不存在，返回-1作为标记 \*/

}

int Prim( MGraph Graph, LGraph MST )

{ /\* 将最小生成树保存为邻接表存储的图MST，返回最小权重和 \*/

WeightType dist[MaxVertexNum], TotalWeight;

Vertex parent[MaxVertexNum], V, W;

int VCount;

Edge E;

/\* 初始化。默认初始点下标是0 \*/

for (V=0; V<Graph->Nv; V++) {

/\* 这里假设若V到W没有直接的边，则Graph->G[V][W]定义为INFINITY \*/

dist[V] = Graph->G[0][V];

parent[V] = 0; /\* 暂且定义所有顶点的父结点都是初始点0 \*/

}

TotalWeight = 0; /\* 初始化权重和 \*/

VCount = 0; /\* 初始化收录的顶点数 \*/

/\* 创建包含所有顶点但没有边的图。注意用邻接表版本 \*/

MST = CreateGraph(Graph->Nv);

E = (Edge)malloc( sizeof(struct ENode) ); /\* 建立空的边结点 \*/

/\* 将初始点0收录进MST \*/

dist[0] = 0;

VCount ++;

parent[0] = -1; /\* 当前树根是0 \*/

while (1) {

V = FindMinDist( Graph, dist );

/\* V = 未被收录顶点中dist最小者 \*/

if ( V==ERROR ) /\* 若这样的V不存在 \*/

break; /\* 算法结束 \*/

/\* 将V及相应的边<parent[V], V>收录进MST \*/

E->V1 = parent[V];

E->V2 = V;

E->Weight = dist[V];

InsertEdge( MST, E );

TotalWeight += dist[V];

dist[V] = 0;

VCount++;

for( W=0; W<Graph->Nv; W++ ) /\* 对图中的每个顶点W \*/

if ( dist[W]!=0 && Graph->G[V][W]<INFINITY ) {

/\* 若W是V的邻接点并且未被收录 \*/

if ( Graph->G[V][W] < dist[W] ) {

/\* 若收录V使得dist[W]变小 \*/

dist[W] = Graph->G[V][W]; /\* 更新dist[W] \*/

parent[W] = V; /\* 更新树 \*/

}

}

} /\* while结束\*/

if ( VCount < Graph->Nv ) /\* MST中收的顶点不到|V|个 \*/

TotalWeight = ERROR;

return TotalWeight; /\* 算法执行完毕，返回最小权重和或错误标记 \*/

}

2 Kruskal算法具体实现：

/\*-------------------- 顶点并查集定义 --------------------\*/

typedef Vertex ElementType; /\* 默认元素可以用非负整数表示 \*/

typedef Vertex SetName; /\* 默认用根结点的下标作为集合名称 \*/

typedef ElementType SetType[MaxVertexNum]; /\* 假设集合元素下标从0开始 \*/

void InitializeVSet( SetType S, int N )

{ /\* 初始化并查集 \*/

ElementType X;

for ( X=0; X<N; X++ ) S[X] = -1;

}

void Union( SetType S, SetName Root1, SetName Root2 )

{ /\* 这里默认Root1和Root2是不同集合的根结点 \*/

/\* 保证小集合并入大集合 \*/

if ( S[Root2] < S[Root1] ) { /\* 如果集合2比较大 \*/

S[Root2] += S[Root1]; /\* 集合1并入集合2 \*/

S[Root1] = Root2;

}

else { /\* 如果集合1比较大 \*/

S[Root1] += S[Root2]; /\* 集合2并入集合1 \*/

S[Root2] = Root1;

}

}

SetName Find( SetType S, ElementType X )

{ /\* 默认集合元素全部初始化为-1 \*/

if ( S[X] < 0 ) /\* 找到集合的根 \*/

return X;

else

return S[X] = Find( S, S[X] ); /\* 路径压缩 \*/

}

bool CheckCycle( SetType VSet, Vertex V1, Vertex V2 )

{ /\* 检查连接V1和V2的边是否在现有的最小生成树子集中构成回路 \*/

Vertex Root1, Root2;

Root1 = Find( VSet, V1 ); /\* 得到V1所属的连通集名称 \*/

Root2 = Find( VSet, V2 ); /\* 得到V2所属的连通集名称 \*/

if( Root1==Root2 ) /\* 若V1和V2已经连通，则该边不能要 \*/

return false;

else { /\* 否则该边可以被收集，同时将V1和V2并入同一连通集 \*/

Union( VSet, Root1, Root2 );

return true;

}

}

/\*-------------------- 边的最小堆定义 --------------------\*/

void PercDown( Edge ESet, int p, int N )

{ /\* 将N个元素的边数组中以ESet[p]为根的子堆调整为关于Weight的最小堆 \*/

int Parent, Child;

struct ENode X;

X = ESet[p]; /\* 取出根结点存放的值 \*/

for( Parent=p; (Parent\*2+1)<N; Parent=Child ) {

Child = Parent \* 2 + 1;

if( (Child!=N-1) && (ESet[Child].Weight>ESet[Child+1].Weight) )

Child++; /\* Child指向左右子结点的较小者 \*/

if( X.Weight <= ESet[Child].Weight ) break; /\* 找到了合适位置 \*/

else /\* 下滤X \*/

ESet[Parent] = ESet[Child];

}

ESet[Parent] = X;

}

void InitializeESet( LGraph Graph, Edge ESet )

{ /\* 将图的边存入数组ESet，并且初始化为最小堆 \*/

Vertex V;

PtrToAdjVNode W;

int ECount;

/\* 将图的边存入数组ESet \*/

ECount = 0;

for ( V=0; V<Graph->Nv; V++ )

for ( W=Graph->G[V].FirstEdge; W; W=W->Next )

if ( V < W->AdjV ) { /\* 避免重复录入无向图的边，只收V1<V2的边 \*/

ESet[ECount].V1 = V;

ESet[ECount].V2 = W->AdjV;

ESet[ECount++].Weight = W->Weight;

}

/\* 初始化为最小堆 \*/

for ( ECount=Graph->Ne/2; ECount>=0; ECount-- )

PercDown( ESet, ECount, Graph->Ne );

}

int GetEdge( Edge ESet, int CurrentSize )

{ /\* 给定当前堆的大小CurrentSize，将当前最小边位置弹出并调整堆 \*/

/\* 将最小边与当前堆的最后一个位置的边交换 \*/

Swap( &ESet[0], &ESet[CurrentSize-1]);

/\* 将剩下的边继续调整成最小堆 \*/

PercDown( ESet, 0, CurrentSize-1 );

return CurrentSize-1; /\* 返回最小边所在位置 \*/

}

int Kruskal( LGraph Graph, LGraph MST )

{ /\* 将最小生成树保存为邻接表存储的图MST，返回最小权重和 \*/

WeightType TotalWeight;

int ECount, NextEdge;

SetType VSet; /\* 顶点数组 \*/

Edge ESet; /\* 边数组 \*/

InitializeVSet( VSet, Graph->Nv ); /\* 初始化顶点并查集 \*/

ESet = (Edge)malloc( sizeof(struct ENode)\*Graph->Ne );

InitializeESet( Graph, ESet ); /\* 初始化边的最小堆 \*/

/\* 创建包含所有顶点但没有边的图。注意用邻接表版本 \*/

MST = CreateGraph(Graph->Nv);

TotalWeight = 0; /\* 初始化权重和 \*/

ECount = 0; /\* 初始化收录的边数 \*/

NextEdge = Graph->Ne; /\* 原始边集的规模 \*/

while ( ECount < Graph->Nv-1 ) { /\* 当收集的边不足以构成树时 \*/

NextEdge = GetEdge( ESet, NextEdge ); /\* 从边集中得到最小边的位置 \*/

if (NextEdge < 0) /\* 边集已空 \*/

break;

/\* 如果该边的加入不构成回路，即两端结点不属于同一连通集 \*/

if ( CheckCycle( VSet, ESet[NextEdge].V1, ESet[NextEdge].V2 )==true ) {

/\* 将该边插入MST \*/

InsertEdge( MST, ESet+NextEdge );

TotalWeight += ESet[NextEdge].Weight; /\* 累计权重 \*/

ECount++; /\* 生成树中边数加1 \*/

}

}

if ( ECount < Graph->Nv-1 )

TotalWeight = -1; /\* 设置错误标记，表示生成树不存在 \*/

return TotalWeight;

}

3.实例代码堆优化：

#include<iostream>

#include<cstring>

#include<cmath>

#include<iomanip>

#define f(i,l,r) for(i=(l);i<=(r);i++)

using namespace std;

const int MAXN=5005;

struct Node{

int x,y;

}a[MAXN];

int n,vis[MAXN],num;

double d[MAXN],ans=0,heap0[10000000];

int heap1[10000000],sz;

inline double cal(int u,int v)

{

return sqrt((double)(a[u].x-a[v].x)\*(a[u].x-a[v].x)+(double)(a[u].y-a[v].y)\*(a[u].y-a[v].y));

}

inline void pushup(int p)

{

int fa=p>>1,id=heap1[p];

double a=heap0[p];

while(fa&&a<heap0[fa]){

heap0[p]=heap0[fa];

heap1[p]=heap1[fa];

p=fa;

fa>>=1;

}

heap0[p]=a;

heap1[p]=id;

}

inline void pushdown(int p)

{

int son=p<<1,id=heap1[p];

double a=heap0[p];

while(son<=sz){

if(son<sz&&heap0[son]>heap0[son+1]) son++;

if(heap0[son]>=a) break;

heap0[p]=heap0[son];

heap1[p]=heap1[son];

p=son;

son<<=1;

}

heap0[p]=a;

heap1[p]=id;

}

inline void insert(double a,int id)

{

heap0[++sz]=a;

heap1[sz]=id;

pushup(sz);

}

inline void Pop()

{

heap0[1]=heap0[sz];

heap1[1]=heap1[sz--];

pushdown(1);

}

inline void Prime()

{

int i;

d[1]=0;

insert(0.0,1);

while(sz){

int u=heap1[1];

double w=heap0[1];

Pop();

if(vis[u]) continue;

vis[u]=1;

ans+=w;

num++;

if(num==n) break;

f(i,1,n){

if(vis[i]||i==u) continue;

double tmp=cal(i,u);

if(d[i]>tmp){

d[i]=tmp;

insert(d[i],i);

}

}

}

}

int main()

{

ios::sync\_with\_stdio(false);

memset(d,127,sizeof(d));

int i,j;

cin>>n;

f(i,1,n){

cin>>a[i].x>>a[i].y;

}

Prime();

cout<<fixed<<setprecision(2)<<ans<<endl;

return 0;

}