**华中科技大学计算机科学与技术学院**

**算法分析与设计报告**



专 业： 计算机科学与技术

班 级： ACM1701

学 号： U201714850

姓 名： 丁文隆

成 绩：

指导教师： 何琨

**完成日期： 2019年11月17日**

# 实验一

## 实验题目：贪心算法——编程实现最小生成树MST算法

## 二、实验目的与内容

1、实验目的：

深入了解最小生成树的有关性质与经典算法。

学习贪心算法的设计方法与实现方法。

在具体实现时，学习如何使用的具体的数据结构完成一些算法操作（例如本问题中的union和find）

分析不同的最小生成树算法的性能差异。

2、实验内容：

设计并编程实现最小生成树算法

输入：第一行有两个数，给出图的点数量n和边数量m；接着有m行，每一行有三个数字，分别是边的两个端点u和v，以及其对应的边权wi。

输出：输出边权和的最小值。

自行设计测试用例测试数据，测试程序的正确性。

分析算法的正确性与复杂度，并分析kruskal和prim算法的性能差异。

## 三、算法设计

1、算法描述

实验中我们采用kruskal算法，每次加入最小边权的边，并保证不成环，直到已经加入的边的数量已经到了n-1为止（n为点的数量）。为了表示每个点的加入情况与连通情况，我们利用并查集标识每一个节点的从属关系，并用路径压缩技术减少并查集的查找时间。

下面用分步骤解释算法的具体实现：

STEP 1: 对输入的边权从小到大排序。

在这里我们可以调用c++的algorithm库中的sort函数，这个函数采用的是快排的方法，复杂度为O(nlogn)。

STEP 2: 初始化并查集，每个元素先各自为一个独立的连通块。

这里我们将我们所设置的并查集全部设置为root[v]=v，即每个点的根节点都是自己，也就是各自为一个块。（并查集中把root一样的点称为一个块）

STEP 3: 按边权从小到大枚举所有边，直到加入n-1个边或者m条边循环完毕，跳转到STEP 6。

我们开始循环，初次循环的时候，初始化edgenum=0记录已经加入的边。执行下列步骤。

STEP 4: 判断当前所判断的边的两个端点是否在一个测试块中，若是则合并。

这一步要实现并查集的findroot函数，我们可以采用递归的方式实现这个函数，具体实现如下：

然而这种方法效率是比较低的，显然，我们访问过的递归链路上的root都应该是点v的root，而不幸运的是，我们只将v节点的root进行了更新，没有将其他节点进行更新。所以，在这里提出路径压缩，就是保存下根节点的数值，将链路上的所有节点的更节点在访问过程中进行更新。我们只需要在递归过程中的时候插入以下逻辑即可：

要合并则要实现并查集的union函数，对于一条边的两个端点u和v，我么先调用findroot函数寻找到了两个点的root，分别记为uroot和vroot，则有合并规则如下：

可以看出，若两点root不同，将uroot合到vroot上，这里说明，由于并查集经过了路径压缩，所以将vroot合到uroot上也是一样的复杂度。我们在程序中统一向vroot合并。

STEP 5: 增加边权sum，循环STEP 3到STEP 5。

若上一步合并成功（即uroot=vroot），将sum加上该边的权值。（sum初始化为0），edgenum加1。循环STEP 3到STEP 5，一直到edgenum等于n-1或者m条边循环完毕，进行STEP 6。

STEP 6: 判断edgenum是否等于n-1，若不等，证明图不连通，显示错误；若相等，输出sum，即最小权值和。

总结上面所给出的步骤，算法伪代码如下：

krukal( ) {  
 sum=最小生成树的边权之和 (init sum=0);

edgenum=当前加入最小生成树的边的数量 (init edgenum=0);

root[ ]=所有点的并查集 (init root[v]=v (v=0 to n-1 )

for(从小到大枚举输入的边){

if(当前所枚举的边的两个端点的root不同){

该边加入最小生成树(合并并查集);

sum=sum+边权;

edgenum=edgenum+1;

}

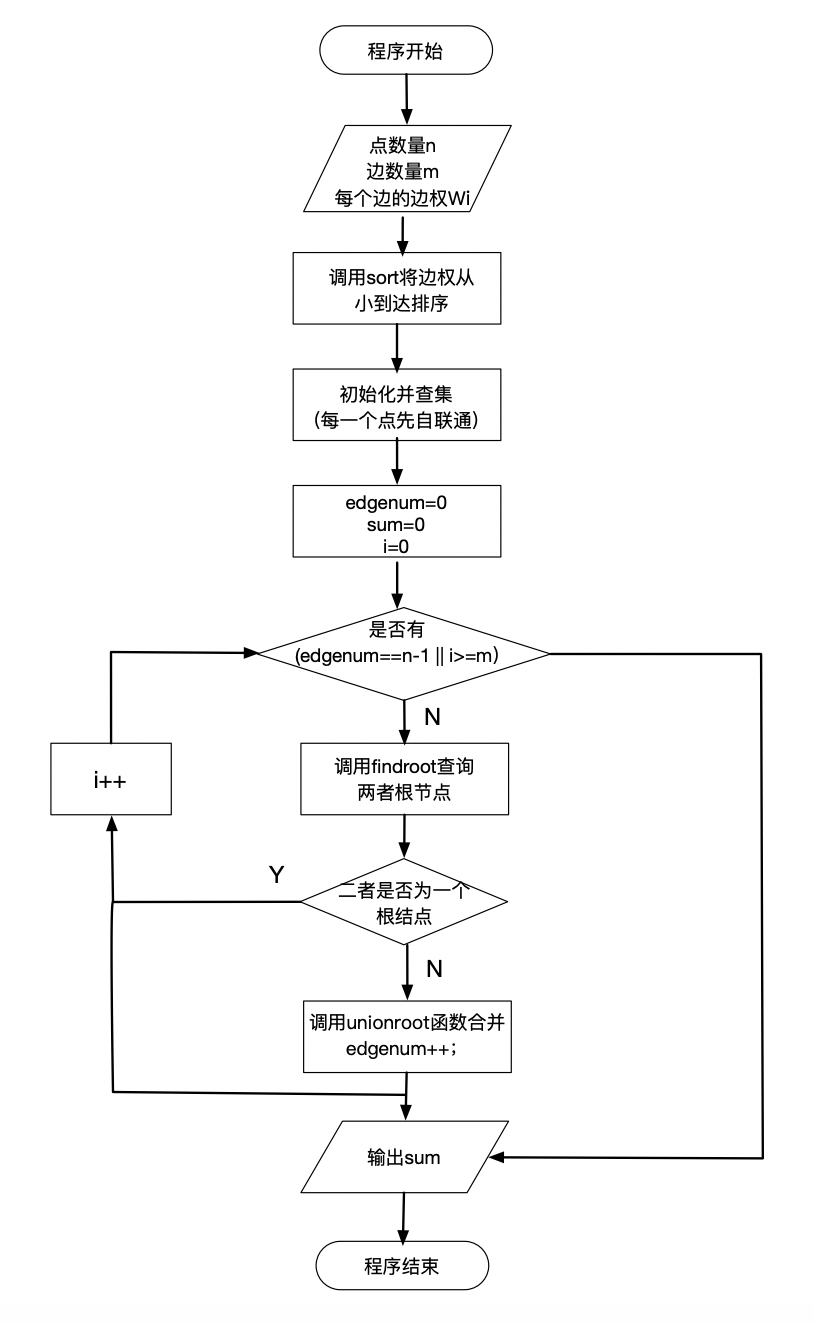
}

return sum;

}

2、算法流程图

算法描述的流程图如图3.1：



图X.1

## 四、实验环境

操作系统：Windows 10

编译环境：Dev C++ 5.11

编译器 ：TDM-GCC 4.9.2 64-bit Release

## 五、实验过程

运用kruskal算法实现MST的源代码如下，每段程序代表的释义、每个变量的意义已经用注释详细标明，可以对照第三章算法设计进行对照查看：

|  |
| --- |
| kruskal实现MST |
| #include <iostream>  #include <algorithm>  #define Vmax 1000  #define Emax 500000  using namespace std;  struct edge{  int u,v,w; //u:一个端点；v:另一个端点；w:边权  }E[Emax];  bool cmp(edge a,edge b){  return a.w<b.w; //边按照从小到大排序  }  int m,n; //m:边的条数，n:点的个数  int sum,edgenum; //sum:边权和，edgenum:已经遍历的边的数量  //利用并查集实现findroot函数  int root[Vmax];  int findroot(int v){  if(v==root[v]){ //找到了根节点  return v;  }  //路径压缩  else{  int Root=findroot(root[v]); //寻找上一级的根节点  root[v]=Root; //更新路上所有节点接入根节点  return Root;  }  }  //实现unionroot函数(对于编号为i的边)  int unionroot(int i){  int uroot=findroot(E[i].u);  int vroot=findroot(E[i].v);  if(uroot!=vroot){  root[uroot]=vroot;  sum=sum+E[i].w;  edgenum=edgenum+1;  }  }  int main(){  //读入边数、点数、边的信息  cin>>m>>n;  for(int i=0;i<m;i++){  cin>>E[i].u>>E[i].v>>E[i].w;  }    for(int i=0;i<n;i++){ //初始化并查集  root[i]=i;  }    sort(E,E+m,cmp); //边从小到大排序    for(int i=0;i<m;i++){ //遍历边，进行合并操作  unionroot(i);  if(edgenum==n-1){ //边的数量已经达到n-1  break;  }  }  if(edgenum!=n-1){ //若图不连通  cout<<"MST does not exist!"<<endl;  }  else{  cout<<"The minimal sum is: "<<sum<<endl;  }  return 0;  } |

## 六、算法测试

为了保证算法的正确性，我们设置三组测试样例，设置意义如下（我们的输入输出已经在第二节实验内容中作出了约定）：

第一组，设置普通算例，6个顶点、10条边，没有重复的边权和为0的边权。

第二组，测试特殊情况，4个顶点、6条边，有重复的边权。

第三组，测试更边界的情况，4个顶点、6条边，有重复边权以及为0的边权。

测试样例1

|  |  |  |
| --- | --- | --- |
| 样例输入 | 理论输出 | 样例输出 |
| 6 10  0 1 4  0 4 1  0 5 2  1 2 1  1 5 3  2 3 6  2 5 5  3 4 5  3 5 4  4 5 3 | 11 |  |

由于理论输出与样例输出相符，所以测试样例1验证成功。

测试样例2

|  |  |  |
| --- | --- | --- |
| 样例输入 | 理论输出 | 样例输出 |
| 4 6  0 1 1  0 2 3  0 3 1  1 2 2  1 3 2  2 3 2 | 4 |  |

由于理论输出与样例输出相符，所以测试样例2验证成功。

测试样例3

|  |  |  |
| --- | --- | --- |
| 样例输入 | 理论输出 | 样例输出 |
| 4 6  0 1 0  0 2 3  0 3 3  1 2 2  1 3 2  2 3 0 | 2 |  |

由于理论输出与样例输出相符，所以测试样例3验证成功。

综上，算法通过所有样例的测试。

## 七、结果分析

（分析算法的正确性、时间复杂度、空间复杂度、实验中遇到的问题等）

算法正确性证明：

证明该算法之前，先证明下面这个引理：

对于任何点集合和，这两个点集合中权值最小的割边一定属于最小生成树。

证明如下：假设我们现有的生成树为，对于任意点集合V和V-S，设其割边为。假设有另一割边,使得 。我们可以构造一棵新的生成树。这棵树的权值和小于原生成树的权值和，即：。也就是说，新树的总权值更优。现在证明新树能从点路由到：在和中，必然各自有路径满足从到和从到。所以，我们分析出到会有新路径：，满足到连通，故我们的新树是一棵生成树且权值更小，更接近最小生成树。综上，在最小生成树中，割边中权值最小的一定包含在最小生成树中。

下面利用这一引理来证明kruskal算法：

对于kruskal添加的任何边，令为在添加之前的点的集合，因为由kruskal算法性质有，加上边不会产生环，即有且∉。而且没有从到的边。那么，由kruskal算法添加，添加的边一定是割边中最小的。由上面引理可知，该边属于最小生成树。即我们可以通过kruskal算法生成边构造最小生成树。

算法复杂度分析

该算法的算法主要分为两个方面，一个是对于边权的排序，另一个是findroot和unionroot的并查集操作过程。

对于边权的排序，在函数实现的时候是调用的STL的algorithm库中的sort函数，采用的是快速排序的方法，算法复杂度为O(mlogm)。

对于并查集的find和union过程，每一次遍历的时候都会采取压缩策略，即压缩后，前序点的搜索都会变成O(1)，降低了一部分复杂度，复杂度会从本身的O(mlogm)（这里是一棵平衡树）变成了老师上课提到的O(mα(m))。

所以算法的总复杂度为：

附：可以用最小堆对于上面复杂度进一步优化

我们观察上面复杂度也可以发现，这里的sort算法拖慢了并查集操作好不容易节省下来的复杂度。

所以我们也可以采用建最小堆的方式，每一次并查集操作时弹出堆顶元素，这样不用排序就可以完成由小到大取遍历边，建堆的复杂度仅为O(m)，小于排序复杂度。这种情况下，复杂度为：

关于prim和krukal对于MST问题的的性能简析

对于prim问题，是以点出发，每次寻找最小的不成环边，并将最小的边加入，对应的点加入集合，直到找到了n个点。故我们需要遍历n个点，遍历n个点的时候，每一次都要遍历O(n)个点，故我们有，这样实现prim算法的复杂度为O(n2)。

对于kruskal，我们上面已经提到了一般而言（不用最小堆情况下），有复杂度O(mlogm)。

对于m和n，我们有以下关系：

如果是稠密图，比如我们取m=n(n-1)/2代入kruskal的复杂度，有：

可以看出，在该稠密图中，kruskal算法的复杂度是prim算法的logn倍。

如果是稀疏图，比如我们取m=n带入kruskal的复杂度，则有：

这个复杂度是要比prim算法的O(n2)要小的。

所以，我们可以看出kruskal适合解稠密图，prim适合解稀疏图。

## 八、总结

通过这次实验，

# 实验一（选做）

## 一、实验题目：分治算法——找第k小元素，基于二次取中算法

## 二、实验目的与内容

1、实验目的：

（请详细描述实验目的）

例：

掌握分治算法的基本思想与设计方法。

掌握基于分治的快速排序的实现方法。

了解并探究基准点的选择对于快速排序算法的性能影响。

2、实验内容：

利用分治算法和二次取中方法，设计并实现寻找第k小元素的算法。

输入：第一行有两个数字，分别是寻找的序号k和数组长度n；然后第二行有n个数字，是待查找数组中的n个数字。

输出：第k小的数。

设计数据测试程序正确性。

分析算法复杂度。

探索在二次取中时不同的分组数对于程序性能的影响。

## 三、算法设计

1、算法描述

本次实验主要运用快速排序的思想实现寻找第k小的元素，其中基准点的选取是采用了二次取中算法，降低最坏情况的算法复杂度。下述描述过程只针对分组n=5的情况进行说明，给出算法的思路，在结果分析的时候会具体说明分组数对于算法性能的影响。

STEP 1: 求出中位数的中位数，选取其作为快速排序的pivot

设计函数GetMid，将数组a下标从start到end的元素进行排序，取返回下标(start+end)/2。（即中位数）

设计函数GetMidPivot，取分组后所得的中位数数组中的中位数。先将数组a按照5个一组进行划分，用GetMid函数取每组的中位数，存入pivot数组。对pivot数组再次调用GetMid函数，得到中位数的中位数。

STEP 2：设置left=0，right=n-1。（left和right为当前要处理的数据段在整个数组中的左下标和右下标）

STEP 3: 以pivot为界，在left得到right范围内，比pivot小的放pivot左边，比pivot大的放pivot右边，记排序后pivot的下标为i。

这里我们以挖坑法实现，主要步骤如下：

* + 选取pivot，这里我们选取最后一个数，首先把pivot记为坑。
  + 令i=left，j=right，即i和j最开始指向需要进行排序的序列两端。I一直向后移动，找到一个大于pivot的值，填入坑中，坑位变为a[i]。
  + J一直向前移动，找到一个小于pivot的值，填入坑，坑位变为a[j]。
  + 重复上面两个步骤到i=j，将pivot填入a[i]。

STEP 4: 若i=k-1，代表这个基准点是要求的第k小的数，返回a[i]的值。

STEP 5: 若i<k-1，则证明第k小的数在pivot右边，在右边递归。

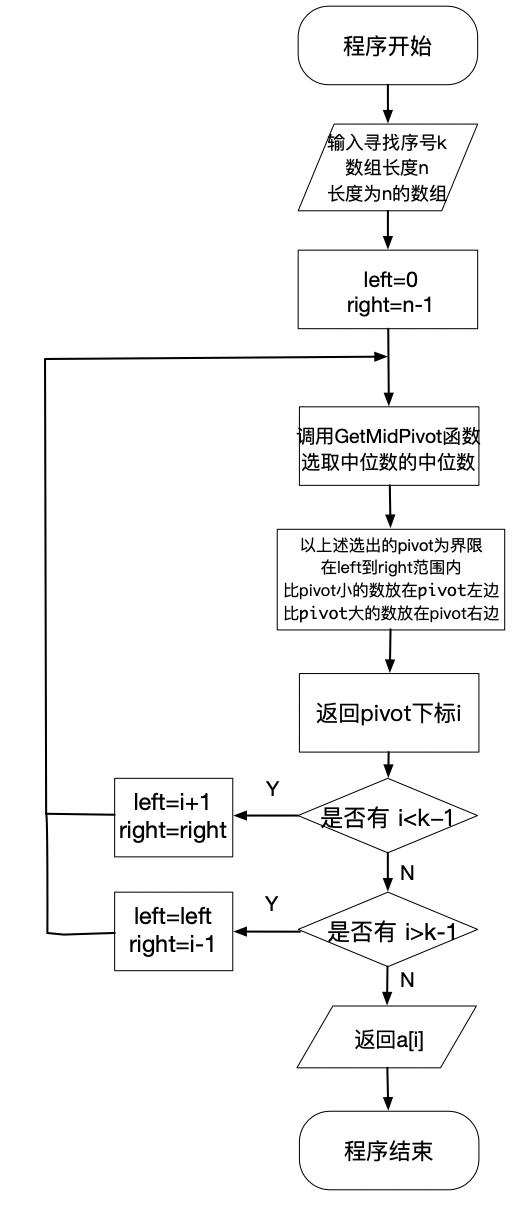
设left=i+1；right=right，回到STEP 1。

STEP 6: 若i>k-1，则证明第k小的数在pivot左边，在左边递归。

设left=left，right=i-1，回到STEP 1。

2、算法流程图

算法描述的流程图如图X.1：



图X.1

## 四、实验环境

操作系统：Windows 10

编译环境：Dev C++ 5.11

编译器 ：TDM-GCC 4.9.2 64-bit Release

## 五、实验过程

利用二次取中的选择算法实现K-top问题的源代码如下，代码每部分含义、变量名已经在注释中标出：

|  |
| --- |
| 二次取中的选择算法找出第k小元素 |
| #include <iostream>  #include <algorithm>  using namespace std;  int GetMid(int a[],int start,int end){ //求出数组a在start到end位置上的中位数的下标  sort(a+start,a+end+1); //start到end排序  int temp=start+end;  int mid=(temp/2)+(temp%2); //取中位数的下标  return mid;  }  int GetMidPivot(int a[],int start, int end){ //将数组a划分，求出中位数的中位数的下标  int len=end-start+1;  int group=len/5+(len%5==0?0:1); //求出组数，不足5个按照5个计算  int pivot[group]; //存储每一组中位数的下标  for(int k=0;k<group;k++){  int i=start+k\*5,w=i+4,j=min((i+4),end);  pivot[k]=GetMid(a,i,j); //求出每一组的中位数下标  }  int index=GetMid(pivot,0,group-1); //取中位数下标数组的中位数的下标  return pivot[index];  }  void bfqrt(int a[],int left,int right, int k){ //分治主体  //挖坑法实现  int i=left,j=right;  int pivot\_index=GetMidPivot(a,left,right); //求中位数的中位数的下标  int pivot=a[pivot\_index];  swap(a[pivot\_index],a[right]); //将基准值放在数组末尾  while(i<j){  while(i<j&&a[i]<=pivot){ //左边开始寻找比pivot大的数进行交换  i++;  }  a[j]=a[i];  while(i<j&&a[j]>=pivot){ //右边开始寻找比pivot小的数进行交换  j--;  }  a[i]=a[j];  }  a[i]=pivot;  if(i>k-1){  bfqrt(a,left,i-1,k); //在左边继续搜索  }  else if(i<k-1){  bfqrt(a,i+1,right,k); //在右边继续搜索  }  else{  cout<<"The k-smallest number is: "<<a[i]<<endl;  return ;  }  }  int main(){  int k,n; //要寻找的序号k,数组长度n  cin>>k>>n; //读入序号k，长度n  int a[n]; //要查找的数组a  for(int i=0;i<n;i++){ //读入数组a[]  cin>>a[i];  }  if(k>n){ //若要寻找的需要大于数组大小，报错  cout<<"the k-smallest number does not exist!"<<endl;  return 0;  }  bfqrt(a,0,n-1,k); //调用函数寻找第k小的数  return 0;  } |

## 六、算法测试

为了保证算法的正确性，我们设置三组测试样例，设置意义如下（我们的输入输出已经在第二节实验内容中作出了约定）：

第一组：一般算例检验，7个数，求第4小的数字，7个数互相不相同。

第二组：边界情况检验，7个数，求第4小的数字，7个数全部一样。

第三组：边界情况检验，7个数，求第四小的数字，其中有相同的数

测试样例1

|  |  |  |
| --- | --- | --- |
| 样例输入 | 理论输出 | 样例输出 |
| 4 7  5 2 7 3 8 1 9 | 5 |  |

由于理论输出与样例输出相符，所以测试样例1验证成功。

测试样例2

|  |  |  |
| --- | --- | --- |
| 样例输入 | 理论输出 | 样例输出 |
| 4 7  5 5 5 5 5 5 5 | 5 |  |

由于理论输出与样例输出相符，所以测试样例2验证成功。

测试样例3

|  |  |  |
| --- | --- | --- |
| 样例输入 | 理论输出 | 样例输出 |
| 4 7  3 6 3 5 2 6 8 | 6 |  |

由于理论输出与样例输出相符，所以测试样例3验证成功。

综上，算法通过所有样例的测试。

## 七、结果分析

算法复杂度分析：

由于我们在算法实现的时候每组分成了五份，故算法复杂度为：

简要探究不同分组数对复杂度的影响：

通过分析，我们发现，并不是每种分组方案都能够获得线性时间，分组k要满足一定的条件才行，简析如下：

因为每一组分为k个元素，所以大于中位数的元素至少为：

故小于的元素至多为 3n/4+k个。

故得到递归式：

假设该递推式由线性解，即有，我们有：

即我们要有：

即最后我们要有：

通过计算时，复杂度为

即综上k>4的时候才有该算法的线性性质，即复杂度为O(n)

## 八、总结

(通过本次实验得到了什么结论，实验体会）