G^+Tree 算法两点最短路查询说明

一、输入文件说明

输入文件为纯文本格式，第一行两个整数 n,m 表示点数和边数，接下来 m 行每行三个整数 U,V,C 表示 U->V 有一条长度为 C 的边。路网文件可以为有向图，也可为无向图。具体设置参数在后面详述。

```
举例：
1089933 2545844   //表示路网图有 1089933 个点，2545844 条边。
0 1 655           // 0-> 1 路网距离为 655，  共 m 行
0 80 1393
0 81 1426
1 0 655
```

二、索引树生成与两点路查询函数

与 GTree 相同，G^+Tree需要对路网文件进行处理，并建立索引结构，以便后面查询使用。在本程序文件中，对单独的树结构构建时采取每次重新构建的方法。实际应用中可以将 save()和 load()函数分开，就不需要每次重新建立索引树。

2.1 路网图的G^+Tree索引构建

G^+Tree建立过程，主要使用了如下函数：

- void init() //初始化
  其中，int Additional_Memory //表示全连接矩阵规模，设置值越大两点路速度越快，按理论复杂度不超限，设置为$2\times V\times \log_2 V$，如果内存大，可以调大该参数。
- void read() //读取 const char Edge_File[] 路网图边权值文件
- void tree.build() //建立 gptree

2.2 G^+Tree的保存于装载

G^+Tree可以不需要每次都单独构建树结构索引，使用的时候，先用 save（）生成索引结构，可以在之后的最短路查询中直接 load（）。

- void save() //将 gptree 存储如文件"GP_Tree.data"
- void load() //从"GP_Tree.data"文件读取数据，使用时可替代上面的 tree.build()

2.3 G^+Tree的图上两点最短路查询

根据不同情况设计了两点最短路不同调用方法：

- int search(int S, int T) //查询路网点 S、T 之间的最短路，返回值为距离
- int search_catch(int S,int T) //带 cache 的两点最短路，速度慢于 search，频繁查询速度会提高
- int find_path(int S,int T,vector<int> &order)//返回 S-T 最短路长度，并将沿途经过的结点存储到 order 数组中

三、相关参数说明

为了保证性能及适应不同路网结构，制定了较灵活的参数，大部分在源文件中有说明，以下是比较重要的参数。

- const bool Optimization_G_tree_Search 是否使用全连接矩阵。使用全连接矩阵能够提高两点路查询速度，但建树时间会增加，该矩阵大小为 Additional_Memory 个 int。
- Edge_File 为边权文件，Node_File 为节点位置文件。（Node_File 为欧几里得剪裁等提供支持，但在两点路中不需要此文件）
- Partition_Part 表示 gptree 分叉数，对应论文中 fallout
- Naive_Split_Limit 表示叶子节点最大规模+1，对应论文中 τ+1
- RevE false 代表有向图，true 代表无向图读入边复制反向一条边。目前测试均使用有向图。

四、关于路网结构的补充

使用时，不同路网起始点编号不同（如有的是 0，有的是 1），如对应路网图 NW.edge 时，起点为 0，应该将 2383 行：

```
if(RevE==false)G.add_D(j-1,k-1,l);//单向边
```

改为

```
if(RevE==false)G.add_D(j,k,l);//单向边
```