实验报告

一、BBF 算法设计

原有的 KDTree 最近邻查找算法中，是沿着 KDTree 搜索至叶子节点后依次回溯，回溯的路程就是之前查找时的逆序。而 BBF 算法则在此基础上，利用了这些点的信息，在回溯时为各个需要回溯的结点区分优先级，优先查找离查询点更近、更可能包含最近邻点的空间。

具体过程如下：通过对回溯可能需要的路过的结点加入优先队列，并按照查找点到该结点确定的 Bin 的距离进行排序，然后每次取优先级最高的节点（即距离最短的结点），计算该节点到查找点的距离是否比最近邻距离小，如果是则更新最近邻距离。如果查找点在切分维度上在该节点左部，则把他的右孩子加入到队列中，同时检索它的左孩子，否则就把他的左孩子加入到队列中，同时检索它的右孩子。这样一直重复检索，并加入队列，直到检索到叶子节点。上述过程又不断重复直到队列为空、算法结束。

此外，BBF 算法设立了一个限制，如果访问的 Bin 的个数超过该限制，不管队列是不是为空，都会停止运行，返回当前的最近邻点作为结果。并且也用 E_max 限制原有的 KDTree 最近邻查找算法中最多访问 E_max 个叶节点，变为 Restricted NN 方法。

二、代码实现及说明

BBF 算法的代码详见“KDTree_BBF.cpp”文件，Restricted NN 方法则详见“KD_RestrictedNN.cpp”文件。由于其余部分都与实验 5 相同，所以仅展示实现 BBF 算法的函数以及更改部分。

实现 BBF 算法的函数，流程与算法设计中的描述相同：

```
data nearNeighbor_BBF(KDNode root, Point q){
    if(root.location == NIL) return 0;
    KDNode p = root;
    data bestDist = 1e30;   //初始化最近邻距离
    priority_queue<KD_pri_info> queue; //优先级队列，查询点q到队列中现有Bin的距离越小优先级越大
    int t = 0;   //记录访问的Bin的个数
    if(!root.dim){  //按照维度区分当前是x还是y，然后插入队列
        queue.push(KD_pri_info(p, q.distanceTo(P[root.location]),
            (data)(q.x-P[root.location].x)*(data)(q.x-P[root.location].x)));
    }else{
        queue.push(KD_pri_info(p, q.distanceTo(P[root.location]),
            (data)(q.y-P[root.location].y)*(data)(q.y-P[root.location].y)));
    }
    while(!queue.empty()){
        t++;
        KD_pri_info tmp = queue.top();
        queue.pop();
        int dimemsion = tmp.node.dim;
        data cmp_dis;
        if(!dimemsion){
            cmp_dis = (data)(q.x-P[tmp.node.location].x)*(data)(q.x-P[tmp.node.location].x);
        }else{
            cmp_dis = (data)(q.y-P[tmp.node.location].y)*(data)(q.y-P[tmp.node.location].y);
        }
        //如果最近邻距离小于查找点到当前点确定的Bin的距离则不访问该分支
        if(bestDist < cmp_dis){
            continue;
        }
```

```
        //记录当前点到查找点的距离并尝试更新
        data tmp_dis = q.distanceTo(P[tmp.node.location]);
        bestDist = min(tmp_dis, bestDist);

        KDNode qq = tmp.node;
        //遍历以当前点为根的子树，直到叶子节点
        while(qq.r != NIL || qq.l != NIL){
            t++;
            int newdim = qq.dim;
            if(!newdim){  //在x维度上
                if(q.x <= P[qq.location].x){  //如果查找点在当前点的左部
                    if(qq.l != NIL){ //进入左子树前先判断是否为空
                        if(qq.r != NIL){   //将右结点添加到队列前先判断其是否为空
                            data point_dis = q.distanceTo(P[T[qq.r].location]);
                            data cell_dis;
                            if(!T[qq.r].dim)
                                cell_dis = (data)(q.x-P[T[qq.r].location].x)*(data)(q.x-P[T[qq.r].location].x);
                            else
                                cell_dis = (data)(q.y-P[T[qq.r].location].y)*(data)(q.y-P[T[qq.r].location].y);
                            queue.push(KD_pri_info(T[qq.r], point_dis, cell_dis));
                        }
                        qq = T[qq.l];
                    }else{
                        break;
                    }
```

```
                }else{   //如果查找点在当前点的右部，同理
                    if(qq.r != NIL){
                        if(qq.l != NIL){
                            data point_dis = q.distanceTo(P[T[qq.l].location]);
                            data cell_dis;
                            if(!T[qq.l].dim)
                                cell_dis = (data)(q.x-P[T[qq.l].location].x)*(data)(q.x-P[T[qq.l].location].x);
                            else
                                cell_dis = (data)(q.y-P[T[qq.l].location].y)*(data)(q.y-P[T[qq.l].location].y);
                            queue.push(KD_pri_info(T[qq.l], point_dis, cell_dis));

                        }
                        qq = T[qq.r];
                    }else{
                        break;
                    }
                }
            }else{  //在y维度，与x同理
                if(q.y <= P[qq.location].y){
                    if(qq.l != NIL){
                        if(qq.r != NIL){
                            data point_dis = q.distanceTo(P[T[qq.r].location]);
                            data cell_dis;
                            if(!T[qq.r].dim)
                                cell_dis = (data)(q.x-P[T[qq.r].location].x)*(data)(q.x-P[T[qq.r].location].x);
                            else
                                cell_dis = (data)(q.y-P[T[qq.r].location].y)*(data)(q.y-P[T[qq.r].location].y);
                            queue.push(KD_pri_info(T[qq.r], point_dis, cell_dis));
                        }
                        qq = T[qq.l];
                    }else{
                        break;
```

```
                    }
                }else{
                    if(qq.r != NIL){
                        if(qq.l != NIL){
                            data point_dis = q.distanceTo(P[T[qq.l].location]);
                            data cell_dis;
                            if(!T[qq.l].dim)
                                cell_dis = (data)(q.x-P[T[qq.l].location].x)*(data)(q.x-P[T[qq.l].location].x);
                            else
                                cell_dis = (data)(q.y-P[T[qq.l].location].y)*(data)(q.y-P[T[qq.l].location].y);
                            queue.push(KD_pri_info(T[qq.l], point_dis, cell_dis));

                        }
                        qq = T[qq.r];
                    }else{
                        break;
                    }
                }
            }
            //尝试更新最近邻距离
            bestDist = min(q.distanceTo(P[qq.location]), bestDist);
        }
        //如果更新次数超过E_max则跳出循环，即返回现有的最近邻距离
        if(t > E_max){
            break;
        }
    }
    return bestDist;
}
```

Restricted NN 方法的更改如下，添加了 t 变量记录访问的叶节点个数并进行限制：

```
int t = 0;  //记录已访问的叶节点数量
data nearNeighbor(Point q, KDNode p, Rectangle cell, data bestDist){
    if(p.l == NIL && p.r == NIL) t++;   //访问的如果是叶节点则计数加一
    if(t <= E_max){   //只有在访问叶节点个数小于E_max时才继续搜索
        if(p.location != NIL){
            data tmp = q.distanceTo(P[p.location]);
            bestDist = min(tmp, bestDist);  //更新最小距离

            int dimension = p.dim;
            Rectangle leftCell = cell.leftPart(dimension, P[p.location]);
            Rectangle rightCell = cell.rightPart(dimension, P[p.location]);
            if(!dimension){   //在x维度时
                if(q.x < P[p.location].x){//如果q离left部分更近
                    if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                    if(rightCell.distanceTo(q) < bestDist){  //先更新最近的部分，再以此判断是否需要遍历另一部分
                        if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                    }
                }else{  //否则离right部分更近，同理
                    if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                    if(leftCell.distanceTo(q) < bestDist){
                        if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                    }
                }
```

```
            }else{  //在y维度时，同理
                if(q.y < P[p.location].y){
                    if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                    if(rightCell.distanceTo(q) < bestDist){
                        if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                    }
                }else{
                    if(p.r != NIL)bestDist = nearNeighbor(q, T[p.r], rightCell, bestDist);
                    if(leftCell.distanceTo(q) < bestDist){
                        if(p.l != NIL)bestDist = nearNeighbor(q, T[p.l], leftCell, bestDist);
                    }
                }
            }

        }
    }

    return bestDist;
}
```

三、BBF 和 Restricted NN 方法结果对比

使用所给数据测试 E_max 取不同值时两种方法找到结果的差异，由于所给数据较多且较大，所以只选择数据量最小的“1.in”来展示，由于数据量很小，所以对应的 E_max 限制也很小。令 E_max 分别取 3，20 和 50。

E_max=3 时，BBF（左）与 Restricted NN（右）：

90578820469128065	90578820469128065
162647708103206597	137178975878053618
191739935679496493	150771606478260122
14477936575191029	14477936575191029
235256810385451258	235256810385451258
702936702330121170	364164671938693745
15529833846116330	15529833846116330
213771379926420922	213771379926420922
130188120081339017	47426609224478978
76648485808302437	76648485808302437

E_max=20 时，BBF（左）与 Restricted NN（右）：

90578820469128065	90578820469128065
162647708103206597	137178975878053618
191739935679496493	150771606478260122
14477936575191029	14477936575191029
235256810385451258	235256810385451258
364164671938693745	364164671938693745
15529833846116330	15529833846116330
213771379926420922	213771379926420922
130188120081339017	47426609224478978
76648485808302437	76648485808302437

E_max=50 时，BBF（左）与 Restricted NN（右）：

90578820469128065	90578820469128065
162647708103206597	137178975878053618
191739935679496493	150771606478260122
14477936575191029	14477936575191029
235256810385451258	235256810385451258
364164671938693745	364164671938693745
15529833846116330	15529833846116330
213771379926420922	213771379926420922
130188120081339017	47426609224478978
76648485808302437	76648485808302437

将以上输出与正确答案对比，发现 RestrictedNN 算法输出结果全部正确，BBF 算法却存在部分错误。猜测是因为 BBF 算法本就是近似算法，不能完全保证正确

性，并且其更适用于高维数据，因此在二维小数据量的测试样例上表现差于RestrictedNN。

自行实验了其他数据量较大的样例（“2.in”-“10.in”），并修改 E_max，也发现 BBF 算法的正确率还是比较高的，但相比之下仍然是 RestrictedNN 正确率更高。