三维点云最近点算法

二维平面最近点算法

在考虑三维点云最近点算法之前，先考虑在平面当中如何找出距离最近的两个点，结合分治算法的思想，我们使用$P=\{p_1,\ldots,p_n\}$来表示这组点，将P的左半部分 Q 和右半部分 R 分别寻找最近点对，之后利用从左右两半部分得到的最近点对信息，在两部分的分界线 L 附近，利用线性的时间里面得到整体的最近点解。

假设我们已经从左右两部分的递归当中得到了两个最近点的距离为σ，则如果存在 $q\in Q$ 和 $r\in R$, 且 q,r 两点之间的距离 $d(q,r)<\sigma$，则两点可以被限制在一个距离L距离不超过 σ 的窄带范围S内。

将区域 S 的所有点按照点在纵轴的位置排序，则任意两个 S 中的点 s_1,s_2，如果满足 $d(s_1,s_2)<\sigma$，则两个点相距不超过11个位置，因为如果超过了11个位置，则两点必定相距至少 σ——产生矛盾

得到该性质之后，便可以得到求解二维平面最近点的算法，即当一个点集中点的数目不小于3，就将点集分为左右两部分，并在左右两部分递归地调用算法，最终得到左右两部分的两点最近距离σ之后，根据找到的距离，将分界点L周围的窄带区域进行排序，最后在线性的时间内找到在窄带范围内的最近点对距离σ_L，与 σ 进行比较，返回$min(\sigma,\sigma_L)$。如果一个点集之中点的数目小于3，就直接返回所有两点距离的最小值。

设在点的数目为 n 时，该算法的运行时间为 $T(n)$，则对于某个常数 c，$T(n)$满足下列递归关系

$$T(n)\leq 2T(n/2)+cn$$
$$T(2)\leq c$$

因此， 二维平面内最近点对算法最终的运行时间为$O(nlogn)$。

问题描述

下一步考虑解决三维点云的最近点问题，即给定三维空间中的n个点，找到最近点对

定义三维空间中两点 i, j 距离$d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2}$

显然这个问题存在着一个时间复杂度为$O(N^2)$的解法，即计算每两个点之间的距离并取最小值，而这样的时间复杂度在实际应用当中如果需要处理的点对非常多，那么该种算法的效率还存在不小的优化空间，下面探讨的就是三维点云最近点算法的设计以及逐步优化过程以及实现与分析

设计算法

建立递归

对于三维点云的最近点算法，用符号 $P = \{p_1, p_2, .., p_n\}$来表示这组点，首先假设$P$中任意两个点都没有相同的x坐标或者相同的y坐标或者相同的z坐标，这样可以使得下面的讨论更加清晰，并且总是可以通过坐标系的变换使得该假设成立。

首先如果传入的点集大小 $|P| \leq 3$，则直接计算所有点之间的距离，得到最小距离，直接返回。当传入点集的大小大于3时，按照接下来的方法进行递归。

借鉴在二维平面最近点中的分治思想，为了将三维中的点分为左右两个部分，首先在任何递归开始之前，我们将分别按照 x 轴和按照 z 轴对P中的所有点进行排序，得到 P_x, P_z，这里所需要的时间复杂度为$O(nlogn)$。

第一层递归将如下工作，所有其它层以完全类似的方式工作：在完成排序之后，我们寻找位于按照x排序后点集数组位置为$floor(n/2)$的点 x^*，定义Q为集合 P 中 x 坐标小于等于 x^*的点的集合（左半边），R为集合P中x坐标大于 x^*的点的集合（右半边）。这样整个集合P便可以被划分为左右两部分。通过在$O(n)$的时间内一次遍历 Q以及 R，我们可以创建关于 x 轴以及关于 z 轴排序后的两个列表：Q_x，由P_x中按照 x轴坐标递增排序的点组成，R_x的生成也同理；而Q_z以及R_z的生成就要从下往上遍历P_z再根据其在 x 轴上面的分区决定。对于列表的每一项，我们都记录它所属的点在点集当中的坐标位置。

下面我们以递归的方式确定Q中的最近点对（通过访问列表Q_x和Q_y），假设点对(q_0^*, q_1^*)作为Q中最近点对返回。同样地，假设点对(r_0^*, r_1^*)作为R中最近点对返回。

组合解

正如在二维平面最近点算法中提到的那样，假设我们已经找到了Q中的最近点距离σ_q以及R中的最近点距离σ_r，则此时返回的最近点距离$\sigma = min(\sigma_1, \sigma_2)$，区域 S为 x^*左右宽为 σ的区域，而S_1为S左边的区域，S_2为S右边的区域。

$$(1)\text{如果存在}q \in Q\text{和}r \in R, \text{其中}d(q, r) < \sigma, \text{则}q\text{和}r\text{中的每一个都位于区域}S\text{内}$$

证：考虑使用反证法，不失一般性，如果点 q 不在区域 S内，则该点到 Q, R分界面的距离就至少为σ了，此时$d(q, r)$之间的距离不会小于 σ

此时，我们并不能像在二维平面最近点算法中，简单地将每一个点与其按照 z 轴排序周围的常数个点进行比较，进而得出关于该点的最小距离，因为此时还需要考虑 y 轴对该最近距离的影响，即使两个点 p_1, p_2 的 z坐标相距较近，但是两点的 y轴坐标可能会相距较远，此时考虑离其中一个点p_1的常数个最近点的时候便不能优先考虑 p_2.

基于这个推理，我们在考虑位于 S 中一个点的常数个最近点的时候还需要对 y 坐标进行考虑。

错位分区

为了解决这个问题，我们可以按照每个点的 y 坐标对每个点进行分区，并且为了限制在y轴上查找的点的范围，对位于 S_1和 S_2 中的点进行错位分区并编号，即对于 S_1 而言，y 轴分区的起始区域和结束区域长度均为 σ ，其他区域的长度均为 2σ ；而对于 S_2而言，y 轴分区的长度均为 2σ这样，当需要比较的点s位于S_1 的 y 轴区域 $S_1[i]$的时候，在y轴方向仅仅需要考虑 S_2中 $S_2[i-1]$以及 $S_2[i]$区域即可，反过来也有这样的关系。这样，通过在 $O(n)$的时间内，对 S_1以及 S_2中的点按照 其在y轴上的位置进行错位分区就可以限制在 y轴上查找点的范围。

$$(2)\text{如果存在}q \in Q[i]\text{和}r \in R,\text{其中}d(q,r) < \sigma,\text{则}r\text{位于区域}S[i-1]\text{或者}S[i]\text{内}$$

证： 使用反证法，设点 r 不位于两个区域范围内，则对该点来说，其与 q点在y轴上面的距离至少为 σ，与假设相矛盾。

此时利用集合 S 再次描述(1)

$$(3)\text{如果存在}q \in Q\text{和}r \in R,\text{其中}d(q,r) < \sigma,\text{当且仅当存在}s, s^{'} \in S,\text{使得}d(s,s^{'}) < \sigma$$

进一步地，不失一般性，如果需要检测最近距离的点位于区域 S_1[i]，则以它的 z 轴高度为起始高度，在 S_2以 z 轴向上向下，各取出一个对应子区间$S_2[i-1], S_2[i]$内高度为σ, 长度为 2σ，宽度为 σ的长方体，并将该长方体等划分为16个边长为 $\sigma/2$的正方体。在每个正方体当中，对角线的长度为 $\frac{\sqrt{3}}{2}\sigma$，又由于在S_2 中每两个点之间的距离至少为 σ，所以每个正方体至多容纳一个点。

(4)对于位于$S_1[i]$中的点s来说，只需要考虑z轴位置大于它，且位于区域$S_2[i-1]$和$S_2[i]$的16个点，以及z轴位置小于它，且位于区域$S_2[i-1]$和$S_2[i]$的16个点，总共32个点与它之间的距离即可。

证： 根据(3)，我们已经证明过只需要考虑位于 $S_2[i]$ 以及 $S_2[i-1]$的点即可，不失一般性，考虑在 z 轴位置大于点 s 的点 q, 它离点 s 之间相隔了 16个点，则即使它与 点 s 的 y 轴坐标完全一致，根据鸽巢原理，其与 点 s 在 z轴上的距离也一定至少为 σ，因为间隔的16个点一定将长方体当中所有的正方体都占了——产生矛盾。

此时，仅仅需要计算 S_1 区域的所有点能够形成的最小距离即可，因为在计算 S_1区域点的最小距离过程中也同样将 S_2 中所有点能够与 S_1形成的最短距离计算了一遍。

由于对 S_1中的每个点来说，只需要考虑其周边常数个点与它之间的距离，因此计算最短距离的时间复杂度为 $O(n)$。

寻找参考起始高度

接下来遇到的问题就是如何根据位于 $S_1[i]$的点s去找到该点分别在区域 $S_2[i-1]$ 和 $S_2[i]$中开始寻找上下各16个点的起始高度。

一种方法是针对每一个位于 $S_1[i]$中的点，都遍历一遍位于区域 $S_2[i-1]$以及 $S_2[i]$中的所有点，找到最接近它的点的 z 轴坐标。这样的时间复杂度为 $O(n^2)$

进一步地，可以想到使用二分查找寻找最接近 点 s 的点的 z 轴坐标，此时的时间复杂度为 $O(nlogn)$

思考到这一步，我们进一步地考虑将该过程优化到 $O(n)$ 的时间复杂度。为了达成这个目标，开始，我们首先从小到大地依次遍历 P_z，如果遍历到的点属于 $S_1[i]$区域，则记录参考位置为此时 $max(len(S_2[i-1]), len(S_2[i]))$，如果遍历到的点属于 S_2 区域，则根据其位于 y 分区的位置j将其放入$S_2[j]$ 当中。通过这种方法，便可以在 $O(n)$ 的时间内寻找到位于 S_1区域的所有点在 S_2 中的参考距离。

实现算法(C++)

以下是使用C++ 对三维点云最近点距离算法的具体实现

```
#include<iostream>
#include<vector>
#include<algorithm>
#include<limits>
#include<map>

using namespace std;

// define a Point structure
struct Point{
    size_t x;
    size_t y;
    size_t z;
    Point(): x(0), y(0), z(0){};
};

// comparator to compare x
bool cmp_x(Point p1, Point p2){
    return p1.x < p2.x;
}

// comparator to compare y
bool cmp_z(Point p1, Point p2){
    return p1.z < p2.z;
}

// get distance of two points
double get_distance(Point p1, Point p2);
```

```
double Recursively_Get_Closest_Distance(vector<Point>& p_x,
vector<Point>& p_z);

double Find_Closest_Pair_distance(vector<Point>& P){
    size_t P_size = P.size();
    vector<Point> P_z(P_size);
    for(size_t i = 0; i < P_size; i++){
        P_z[i] = P[i];
    }

    sort(P.begin(), P.end(), cmp_x); // sort according to the x axis,
then get P_x
    sort(P_z.begin(), P_z.end(), cmp_z); // sort according to the z axis,
then get P_z
    return Recursively_Get_Closest_Distance(P, P_z);

}

double get_distance(Point p1, Point p2){
    int delta_x = p1.x - p2.x;
    int delta_y = p1.y - p2.y;
    int delta_z = p1.z - p2.z;
    double distance = sqrt(pow(delta_x, 2) + pow(delta_y, 2) +
pow(delta_z, 2));
    return distance;
}


double Recursively_Get_Closest_Distance(vector<Point>& p_x,
vector<Point>& p_z){
    // if the size of P is no larger than 3, then return the minimum
distance directly
    size_t p_size = p_x.size();
    if(p_size <= 3){
        if(p_size == 1){
            // inform the user about the wrong size
            cerr << "The input point set's size should be larger than 1"
<< endl;
            return -1;
        }else if(p_size == 2){
            return get_distance(p_x[0], p_x[1]);
        }else{
            double d1 = get_distance(p_x[0], p_x[1]);
            double d2 = get_distance(p_x[1], p_x[2]);
            double d3 = get_distance(p_x[0], p_x[2]);
            return min(d1, min(d2, d3));
        }
    }

    // the middle point of P
    Point mid_point_x = p_x[p_size / 2];
```

```
    size_t mid_x = mid_point_x.x;

    // construct the four arrays according to their position
    vector<Point> Q_x = vector<Point>(p_x.begin(), p_x.begin() + p_size /
2);
    vector<Point> R_x = vector<Point>(p_x.begin() + p_size / 2,
p_x.end());
    vector<Point> Q_z;
    vector<Point> R_z;

    // iterate the P_z from small to large, and put the points into Q_z
and R_z according to the position in x axis
    for(size_t i = 0; i < p_size; i++){
        Point p = p_z[i];
        if(p.x <= mid_x){
            Q_z.push_back(p);
        }else{
            R_z.push_back(p);
        }
    }
    // call the method recursively
    double dist1 = Recursively_Get_Closest_Distance(Q_x, Q_z);
    double dist2 = Recursively_Get_Closest_Distance(R_x, R_z);
    double dist = min(dist1, dist2); // assign the dist to the minimum of
dist1, dist2.

    vector<Point> S1, S2;
    size_t max_y = 0;
    for(size_t i = 0; i < p_size; i++){
        Point p = p_z[i];
        if(abs(int(p.x) - int(mid_x)) < dist && p.x <= mid_x){
            S1.push_back(p);
            if(p.y > max_y){
                max_y = p.y;
            }
        }else if(abs(int(p.x) - int(mid_x)) < dist && p.x > mid_x){
            S2.push_back(p);
            if(p.y > max_y){
                max_y = p.y;
            }
        }
    }

    // declare and initialize the map
    map<size_t, vector<Point>> map;
    size_t block_num = max_y / (2*dist);
    for(size_t i = 0; i < block_num; i++){
        map[i] = vector<Point>(0);
    }
```

```
    size_t ptr1 = 0, ptr2 = 0;
    size_t s1_size = S1.size();
    size_t s2_size = S2.size();
    // record the reference location of the points of S2
    // so that if I know the point idx of S1, I can find the reference
location immediately
    vector<size_t> refer1(s1_size, 0l);
    vector<size_t> refer2(s1_size, 0l);
    while(ptr1 < s1_size && ptr2 < s2_size){
        // if the fetched element is from S1
        if(S1[ptr1].z < S2[ptr2].z){
            Point p = S1[ptr1];
            size_t y = p.y;
            // find the index of field
            if(y / dist == 0){
                refer1[ptr1] = -1;
                refer2[ptr1] = map[0].size();
            }else{
                size_t idx = (y - dist) / (2*dist) + 1;
                refer1[ptr1] = map[idx - 1].size();
                refer2[ptr2] = map[idx].size();
            }
        }else{ //if the fetched element is from S2
            Point p = S2[ptr2];
            size_t y = p.y;
            // calculate the index of S2
            size_t idx = y / (2 * dist);
            map[idx].push_back(p);
        }
    }

    double min_dist = numeric_limits<double>::max();
    for(size_t i = 0; i < s1_size; i++){
        Point p = S1[i];
        size_t y = p.y;
        size_t idx = 0;
        if(y >= dist){
            idx = y / (2 * dist) + 1;
        }
        size_t ref1 = refer1[idx - 1];
        size_t ref2 = refer2[idx];
        for(size_t i = 0; i < 8; i++){
            double d1 = -1, d2 = -1, d3 = -1, d4 = -1;
            // update the distance if the access is legal
            if(ref1 - i >= 0){
                d1 = get_distance(map[idx - 1][ref1 - i], p);
                min_dist = min(min_dist, d1);
            }
            if(ref2 - i >= 0){
                d2 = get_distance(map[idx][ref2 - i], p);
                min_dist = min(min_dist, d2);
```

```
            }
            if(ref1 + i < map[idx - 1].size()){
                d3 = get_distance(map[idx - 1][ref1 + i], p);
                min_dist = min(min_dist, d3);
            }
            if(ref2 + i < map[idx - 1].size()){
                d4 = get_distance(map[idx][ref2 + i], p);
                min_dist = min(min_dist, d4);
            }
        }
        return min(min_dist, dist);
    }
}
```

分析算法

根据之前对算法的设计，一开始对所有点进行排序所需要的时间为 $O(nlogn)$，再通过把问题分成两部分分别进行递归，在得到递归的结果之后，在 $O(n)$的时间内用对 y 轴进行错位分区，并且在$O(n)$的时间内完成对所有位于 S_1区域的点在 S_2处参考位置的标记，下一步就可以在 $O(n)$的时间里面，通过对 S_1 里面所有点的遍历，完成对跨越 Q, R 两部分区域产生的最小距离的计算。

对于三维点云中最近点对距离算法，设在点的数目为 n时，该算法的运行时间为 $T(n)$，则对于某个常数 c，$T(n)$满足下列递归关系

$$T(n) \leq 2T(n/2) + cn$$
$$T(2) \leq c$$

根据将该递归关系转化成递归树的形式进行求解，可以得到，三维点云中最近点对距离算法最终的运行时间复杂度为$O(nlogn)$。