

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & KỸ THUẬT MÁY TÍNH



TIỂU LUẬN ĐỀ TÀI MỞ RỘNG KSTN CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

KD-TREE & HASH TRONG BÀI TOÁN NEAREST NEIGHBOR SEARCH

Giảng viên hướng dẫn: *TS. Nguyễn Đức Dũng*
Sinh viên thực hiện: *Nguyễn Đức An - 2010102*
Email: an.nguyenduc1406@hcmut.edu.vn

Mục lục

1	Mục đích nghiên cứu	2
2	Mục tiêu đề tài	2
3	Mô tả dữ liệu	2
3.1	Cấu trúc dữ liệu Point3D	2
3.2	Giải thuật Random dữ liệu đầu vào	3
3.2.1	Giải thuật	3
3.2.2	Kết quả	4
4	Cơ sở lý thuyết và ý tưởng giải thuật	5
4.1	KD-Tree	5
4.1.1	Ý tưởng KD-Tree	5
4.1.2	KD-Tree Construction	5
4.1.3	KD-Tree Nearest Neighbor Search	6
4.2	Hash	6
4.2.1	Ý tưởng Hash	6
4.2.2	Hash Function	7
4.2.3	HashTable Construction	8
4.2.4	HashTable Nearest Neighbor Search	9
5	Giải thuật hiện thực Kd-Tree	9
5.1	Cấu trúc dữ liệu KdNode	9
5.2	Cấu trúc dữ liệu KdTree	10
5.3	Build Tree	10
5.3.1	Build Kd-Tree bằng cách thêm tuần tự	10
5.3.2	Build Balanced Kd-Tree bằng cách tìm trung vị	10
5.4	Search Nearest Neighbor Point KD-Tree	12
5.5	Giải phóng Tree	13
6	Giải thuật hiện thực Hash	14
6.1	Hash Function	14
6.2	Cấu trúc dữ liệu HashTable	14
6.3	Build HashTable	15
6.4	Search Nearest Neighbor Point HashTable	15
6.5	Giải phóng HashTable	17
7	Kiểm thử chương trình và so sánh hiệu năng	17
7.1	Tương tác người dùng	18
7.2	Kiểm thử chương trình và so sánh hiệu năng khi search với một điểm	18
7.2.1	Dataset với size = 100	18
7.2.2	Dataset với size = 1000	20
7.2.3	Dataset với size = 10000	22
7.2.4	Dataset với size = 100000	23
7.2.5	Dataset với size = 500000	25
7.2.6	Dataset với size = 1000000	26
8	So sánh hiệu năng giữa KdTree và Hash ở các tập dữ liệu tăng dần	28
8.1	So sánh hiệu năng tạo cấu trúc dữ liệu	28
8.2	So sánh hiệu năng search Nearest Neighbor Point	30
9	Kết luận	31
9.1	Xét về Build Data Structure	31
9.2	Xét về Search Nearest Neighbor Point	31
10	Tài liệu tham khảo	31

1 Mục đích nghiên cứu

Nearest Neighbor Search (NNS) là một dạng tìm kiếm lân cận trong bài toán tối ưu hóa việc tìm điểm gần nhất (hoặc tương tự nhất) trong một tập hợp nhất định với một điểm đã cho. Bài toán có nhiều ứng dụng trong các lĩnh vực khác nhau như :

- Statistical classification – see **k-Nearest Neighbor Algorithm (KNN)**
- Pattern recognition
- Computer vision
- Recommendation systems and Internet marketing

Trong bài tiểu luận này, em sẽ trình bày hai cách hiện thực giải thuật NNS bằng hai cấu trúc dữ liệu **KD-Tree** và **Hashing** và so sánh hiệu năng của hai giải thuật trên đối với các tập dữ liệu khác nhau.

2 Mục tiêu đề tài

Trong đề tài mở rộng này, chúng ta sẽ tự tạo tập dữ liệu đầu vào bằng cách sinh ngẫu nhiên các điểm trong không gian 3D, tầm giá trị trong khoảng $[0,100]$ trên mỗi chiều, độ chính xác của các điểm là 0.001.

Với việc tìm kiếm **Nearest Neighbor Point (NNP)** trong không gian này, chúng ta sẽ hiện thực **KD-Tree**, một cấu trúc dữ liệu nâng cao hỗ trợ việc tìm kiếm dữ liệu đa chiều và một giải thuật **Hash** để băm tọa độ vào các bin (bucket). Sau đó chúng ta sẽ đi tìm điểm gần nhất với **Query Point** cho trước bằng hai cấu trúc dữ liệu trên, cũng như đánh giá hiệu năng giữa hai giải thuật.

3 Mô tả dữ liệu

Về source code implement, sử dụng ngôn ngữ C++ và được chia thành các phần như sau:

- **randomDataSet.h** : source code tự sinh tập dữ liệu đầu vào theo yêu cầu đề tài, tạo các dataSet với các size tùy ý, mỗi phần tử của dataSet là một điểm Point3D (x, y, z), accuracy = 0.001.
- **KdTree.cpp** : source code implement Kd-Tree và searchNNP theo cấu trúc dữ liệu Kd-Tree.
- **Hashing.cpp** : source code implement Hash và searchNNP theo cấu trúc dữ liệu Hash.
- **main.cpp** : source code chạy kết quả từ **KdTree.cpp** và **Hashing.cpp** và tương tác người dùng.

3.1 Cấu trúc dữ liệu Point3D

Class **Point3D** bao gồm :

- Dữ liệu : 3 tọa độ x,y,z có kiểu **float**, giá trị trong khoảng $[0, 100]$ và độ chính xác 0.001.
- Phương thức : **print()** dùng để in tọa độ object Point3D ra theo định dạng (x, y, z).

```
class Point3D {
public:
    float x, y, z;
    map<int, float> myMap;
```

```
Point3D() {
    x = 0;
    y = 0;
    z = 0;
    myMap.insert({ 0, x });
    myMap.insert({ 1, y });
    myMap.insert({ 2, z });
}
Point3D(float x, float y, float z) {
    this->x = x;
    this->y = y;
    this->z = z;
    myMap.insert({ 0, x });
    myMap.insert({ 1, y });
    myMap.insert({ 2, z });
}
void print() {
    cout << "(" << x << ", " << y << ", " << z << ")";
}
};
```

3.2 Giải thuật Random dữ liệu đầu vào

3.2.1 Giải thuật

```
void ranDomData(Point3D*& data, int size){
    vector<float> randomVect;
    int curr_index = 0;
    srand(time(nullptr));
    for(int i = 0; i < 3 * size && curr_index < size; i++) {
        float temp = ((float)rand() / RAND_MAX) * 99.999;
        randomVect.push_back(temp);
        if(randomVect.size() == 3){
            float x_rand = round(randomVect[0] * 1000) / 1000;
            float y_rand = round(randomVect[1] * 1000) / 1000;
            float z_rand = round(randomVect[2] * 1000) / 1000;
            Point3D randomPoint(x_rand, y_rand, z_rand);
            data[curr_index] = randomPoint;
            randomVect.clear();
            curr_index++;
        }
    }
}

Point3D* createDataSet(int size) {
    Point3D* data = new Point3D[size];
    ranDomData(data, size);
    return data;
}

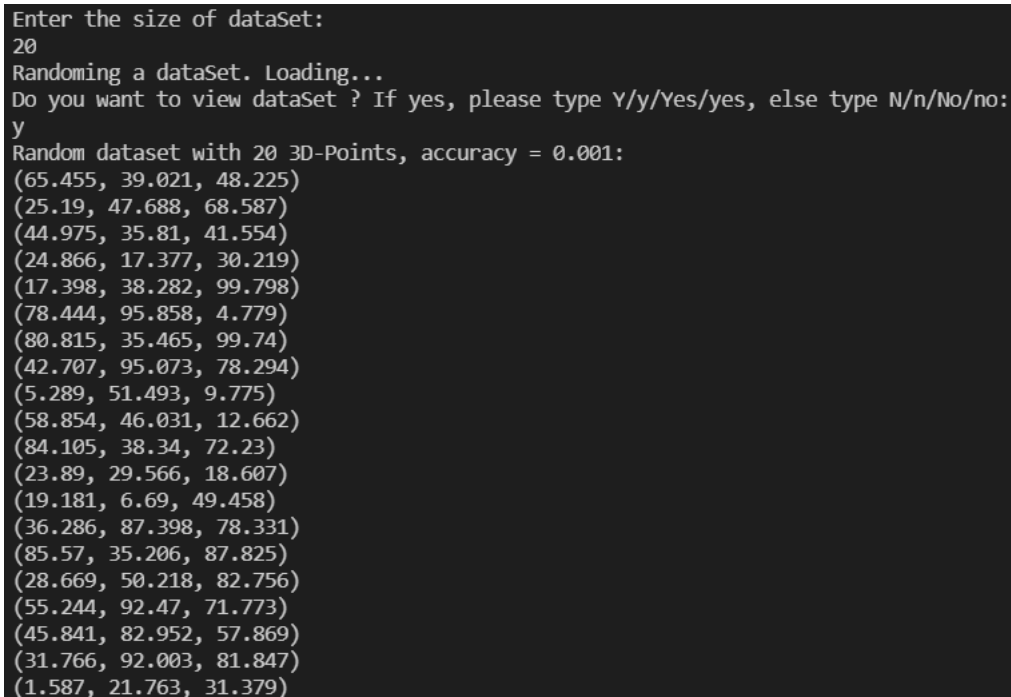
void printDataSet(Point3D* dataSet, int size) {
```

```
cout << "Random dataset with " << size << " 3D-Points, accuracy = 0.001: " << endl;
for (int i = 0; i < size; i++) {
    dataSet[i].print();
    cout << endl;
}
}
```

Các hàm của giải thuật:

- **randomData** : nhận vào hai tham số là mảng dataSet rỗng và size. Hàm random ngẫu nhiên các số thực theo yêu cầu đề tài, ở mỗi tọa độ sau đó làm tròn 3 chữ số thập phân để được độ chính xác là 0.001, cứ mỗi bộ 3 số thì sẽ tạo được một điểm Point3D.
- **createDataSet**: nhận vào một tham số là size đầu vào của dataSet, trả về một mảng dataSet có kích thước là size sau khi random.
- **printDataSet**: nhận vào hai tham số là mảng dataSet sau khi random và size. Hàm in ra các điểm trong dataSet để giúp người dùng kiểm tra các điểm đã random.

3.2.2 Kết quả



```
Enter the size of dataSet:
20
Randoming a dataSet. Loading...
Do you want to view dataSet ? If yes, please type Y/y/Yes/yes, else type N/n/No/no:
y
Random dataset with 20 3D-Points, accuracy = 0.001:
(65.455, 39.021, 48.225)
(25.19, 47.688, 68.587)
(44.975, 35.81, 41.554)
(24.866, 17.377, 30.219)
(17.398, 38.282, 99.798)
(78.444, 95.858, 4.779)
(80.815, 35.465, 99.74)
(42.707, 95.073, 78.294)
(5.289, 51.493, 9.775)
(58.854, 46.031, 12.662)
(84.105, 38.34, 72.23)
(23.89, 29.566, 18.607)
(19.181, 6.69, 49.458)
(36.286, 87.398, 78.331)
(85.57, 35.206, 87.825)
(28.669, 50.218, 82.756)
(55.244, 92.47, 71.773)
(45.841, 82.952, 57.869)
(31.766, 92.003, 81.847)
(1.587, 21.763, 31.379)
```

Hình 1: Example random dataSet with size = 20

4 Cơ sở lý thuyết và ý tưởng giải thuật

4.1 KD-Tree

4.1.1 Ý tưởng KD-Tree

KD-Tree là một cây nhị phân chia các điểm giữa các trục xen kẽ, trong đó mọi nút là một điểm thứ k . Mọi nút không phải là nút lá đều có thể ngầm tạo ra một siêu phẳng phân tách chia không gian thành hai phần, được gọi là nửa không gian. Các điểm ở bên trái của siêu phẳng này được biểu thị bằng cây con bên trái của nút đó và các điểm ở bên phải của siêu phẳng được biểu thị bằng cây con bên phải.

4.1.2 KD-Tree Construction

1. Xây dựng Kd-Tree bằng cách thêm tuần tự

Phương pháp thêm tuần tự bao gồm các bước sau:

- Nếu cây rỗng, điểm được chèn đầu tiên trở thành gốc của cây.
- Dựa trên độ sâu của cây để chọn trục đối xứng cho các điểm. Tính giá trị $axis = depth \% 3$, trong đó độ sâu của gốc là 0. Nếu $axis = 0$ thì chọn theo trục x , $axis = 1$ thì chọn theo trục y , $axis = 2$ thì theo trục z .
- Lần lượt insert các phần tử tuần tự vào cây như insert cây **BST**, nếu $axis = 0$ thì so sánh độ lớn giá trị theo x , $axis = 1$ theo y , $axis = 2$ theo z .
- Traverse cây cho đến khi nút trống, sau đó gán điểm cho nút.
- Lặp lại đệ quy cho đến khi tất cả các điểm được thêm vào cây.

Ưu nhược điểm của phương pháp này:

- Ưu điểm:** Mỗi lần insert một phần tử vào cây thì độ phức tạp trung bình là $O(h)$ (h là chiều cao của cây), tương tự như cây **BST**, time complexity trung bình để build một cây với N phần tử là $O(N \log N)$. Thời gian insert nhanh hơn nhiều so với balanced Kd-Tree.
- Nhược điểm:** Cây dễ bị mất cân bằng, có thể trở thành **skew tree**. Hiệu quả search giảm khi tập dữ liệu tăng dần và tìm kiếm điểm ở những trường hợp xấu.

2. Xây dựng Balanced Kd-Tree bằng phương pháp "Median of list"

Phương pháp "Median of list" bao gồm các bước sau:

- Dựa trên độ sâu của cây để chọn trục đối xứng cho các điểm. Tính giá trị $axis = depth \% 3$, trong đó độ sâu của gốc là 0. Nếu $axis = 0$ thì chọn theo trục x , $axis = 1$ thì chọn theo trục y , $axis = 2$ thì theo trục z .
- Sắp xếp danh sách điểm theo trục dựa vào $axis$ và chọn điểm trung vị làm phần tử pivot. Các phần tử nằm trước trung vị được thêm vào cây con bên trái, các phần tử sau trung vị thêm vào cây con bên phải, tương tự như **Merge Sort**.
- Chọn node trung tâm của mỗi lần đệ quy là phần tử trung vị.
- Lặp lại đệ quy tương tự cho các nhánh left và right của phần tử trung vị cho đến khi tất cả các điểm được thêm vào cây.

Ưu nhược điểm của phương pháp này:

- (a) **Ưu điểm:** Cây được tạo thành trong đa số trường hợp đều là cây cân bằng (**Balanced Tree**), thời gian search ổn định với mọi trường hợp, thường là $O(\log N)$.
- (b) **Nhược điểm:** Chi phí để tìm phần tử median trong mỗi lần insert cao, mỗi lần gọi đệ quy phải dùng hàm **sort()** để sort lại các mảng con, độ phức tạp của thuật toán sort trung bình là $O(N \log N)$. Mỗi lần insert ta sort theo độ sâu của các cây con nên time complexity trung bình của thuật toán build một cây Balanced KD-Tree là $O(N(\log N)^2)$

4.1.3 KD-Tree Nearest Neighbor Search

Thuật toán search Nearest Neighbor Point bằng KD-Tree bao gồm các bước sau:

1. Bắt đầu với nút gốc, traverse down như cây BST, nếu Query Point nhỏ hơn nút hiện tại thì traverse sang trái, ngược lại traverse sang phải (dựa vào axis để chọn trục so sánh).
2. Khi thuật toán đạt đến một nút lá, điểm nút đó được lưu là "điểm tốt nhất hiện tại" (**best**)
3. Traverse upward, và kiểm tra các node đã đi qua:
 - Nếu nút đó gần với Query Point hơn, gán best cho nút đó.
 - Thuật toán cũng kiểm tra xem có thể có bất kỳ điểm nào ở phía bên kia của mặt phẳng tách gần với điểm tìm kiếm hơn điểm tốt nhất hiện tại hay không. Điều này được thực hiện bằng cách giao siêu phẳng phân tách với một siêu cầu xung quanh điểm tìm kiếm có bán kính bằng khoảng cách gần nhất hiện tại.
 - Nếu siêu cầu vượt qua mặt phẳng, có thể có các điểm gần hơn ở phía bên kia của mặt phẳng, do đó, thuật toán phải di chuyển xuống nhánh khác của cây từ nút hiện tại để tìm kiếm các điểm gần hơn, theo cùng một quy trình đệ quy như toàn bộ tìm kiếm
 - Nếu siêu cầu không giao với mặt phẳng phân tách, thì thuật toán tiếp tục đi lên cây và toàn bộ nhánh ở phía bên kia của nút đó sẽ bị loại bỏ.
4. Khi traverse đến nút gốc, thì quá trình tìm kiếm đã hoàn tất.

Giải thuật search Nearest Neighbor Point của KD-Tree có time complexity trung bình là $O(N \log N)$.

4.2 Hash

4.2.1 Ý tưởng Hash

Xây dựng một hàm **Hash Function** sử dụng các tham số là giá trị các tọa độ x, y, z để hash mỗi điểm trong không gian 3D thành một địa chỉ của một bucket trong Hash Table. Các điểm có **phần nguyên các tọa độ giống nhau** thì sẽ được hash vào cùng chung một bucket. Đối với các điểm trong dataset, ta sẽ thêm điểm đó vào các bin (bucket) tương ứng trong Hash Table.

Đối với Hash Table, dựa vào size đầu vào của dataset mà ta sẽ lựa chọn bin-width cho từng bucket tương ứng để đảm bảo độ chính xác cao, tối ưu số lần so sánh.

Khi truy xuất một điểm Query Point, ta sẽ tìm địa chỉ của Query Point bằng hàm Hash Function, dựa vào địa chỉ Query Point chúng ta sẽ biết được vị trí của các bin (bucket) chứa những điểm có khoảng cách gần nhất với Query Point trong không gian 3D.

4.2.2 Hash Function

Để làm tăng độ chính xác và tận dụng space complexity hợp lý, vị trí của mỗi bucket sẽ được tính toán bằng chính phần nguyên của các tọa độ x,y,z của mỗi điểm đó.

Trong không gian 3D, các điểm có tọa độ x, y, z có giá trị trong khoảng [0, 100] nên chúng ta cần **tối đa** $100 \times 100 \times 100 = 1000000$ buckets để lưu trữ các điểm trong trường hợp bin-width = 1 (Chiều rộng của một bucket là 1).

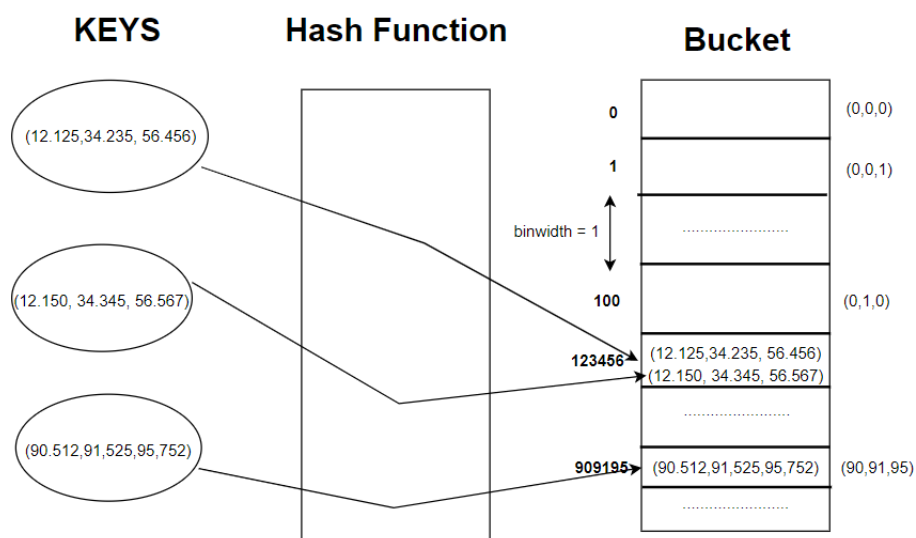
Nhưng trong trường hợp tập dữ liệu nhỏ, nếu sử dụng HashTable với 1.000.000 buckets thì sẽ có hai vấn đề xảy ra. Thứ nhất, việc lưu trữ dữ liệu thế này sẽ làm lãng phí data, space complexity cao. Thứ hai, khoảng cách giữa các điểm trong tập dữ liệu lớn, việc tìm kiếm theo bucket sẽ gặp nhiều khó khăn (nhiều bucket không có dữ liệu nhưng vẫn phải duyệt), ảnh hưởng rất lớn đến performance bao gồm time complexity.

Vì vậy số bucket trong HashTable nên được thay đổi dựa trên size của input dataSet, ta có thể làm giảm số bucket bằng cách tăng bin-width(chiều rộng của một bucket) đối với các tập dữ liệu nhỏ. Khi làm tăng bin-width thì vô tình cũng sẽ làm tăng độ gộp giữa các điểm trong một bucket, số điểm trong một bucket tăng lên, performance giảm, nên việc tăng bin-width không nên áp dụng với các tập dữ liệu lớn.

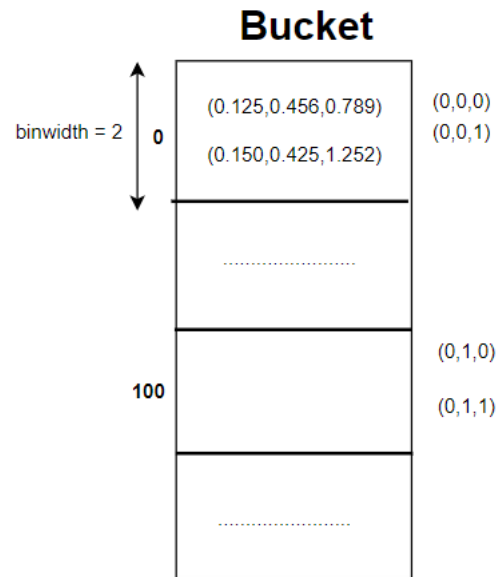
Sau khi tính toán, em sử dụng công thức của hàm **Hash Function** như sau:

$$f(x, y, z) = \lfloor x \rfloor \cdot \frac{10000}{binwidth} + \lfloor y \rfloor \cdot \frac{100}{binwidth} + \lfloor z \rfloor \cdot \frac{1}{binwidth}$$

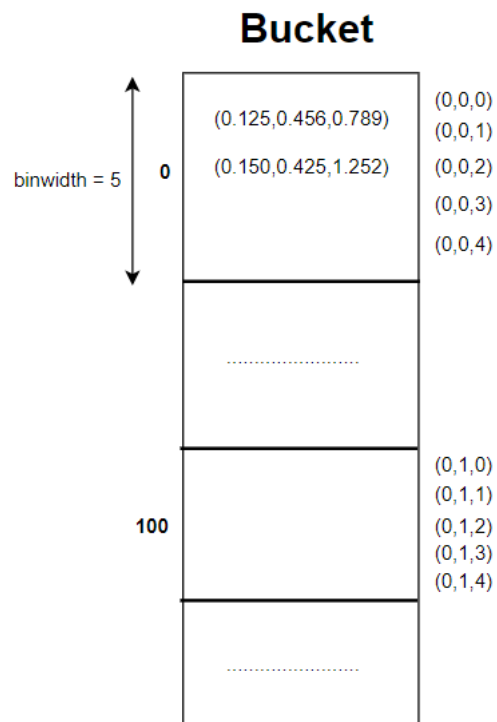
- $f(x, y, z)$ được làm tròn về phần nguyên, là index của bucket trong HashTable.
- Với dataSet có size trong khoảng [0,1000], **binwidth = 5**,
- Với dataSet có size trong khoảng [1000,100000], **binwidth = 2**.
- Với dataSet có size trong khoảng [100000, +∞], **binwidth = 1**.



Hình 2: HashTable with binwidth = 1



Hình 3: HashTable with binwidth = 2



Hình 4: HashTable with binwidth = 5

4.2.3 HashTable Construction

Phương pháp construct Hash Table gồm các bước sau:

1. Dựa vào binwidth mà ta sẽ **resize()** cho HashTable, size của HashTable = **1000000 / binwidth**. Với binwidth = 5 ta khởi tạo size của HashTable = 200000, binwidth = 2 size của HashTable = 500000, binwidth = 1 size của HashTable = 1000000.
2. Lần lượt thêm các điểm trong dataSet vào Hash Table theo thứ tự. Sử dụng hàm **Hash Function** trên để tìm được index của bucket tương ứng với mỗi điểm đó, các điểm có **phần nguyên các tọa độ** giống nhau, phù hợp với binwidth của bucket thì sẽ được thêm vào chung một bucket.
3. Tiếp tục quá trình cho đến khi tất cả các điểm của dataset đều được thêm thành công vào HashTable.

4.2.4 HashTable Nearest Neighbor Search

Thuật toán search Nearest Neighbor Point bằng Hash bao gồm các bước sau:

- Sử dụng hàm **Hash Function** trên để tìm được index của bucket tương ứng với Query Point. Có hai trường hợp xảy ra như sau.
- Nếu bucket của index được hash ra từ Query Point không chứa điểm nào (nghĩa là trong dataset không có điểm nào có cùng **phần nguyên các tọa độ** với Query Point), ta sẽ tìm theo chiều rộng (**Breadth First Search**) trong các khối không gian xung quanh Query Point đó. (Ban đầu là khối $3 \times 3 \times 3$ ($x \pm 1, y \pm 1, z \pm 1$), tiếp theo là $5 \times 5 \times 5$, rồi khối $7 \times 7 \times 7, \dots$) cho đến khi tìm được bucket có chứa ít nhất một điểm thì duyệt tất cả các điểm trong không gian chứa bucket đó rồi dừng lại.
- Nếu bucket của index được hash ra từ Query Point có chứa điểm, ta ưu tiên duyệt các điểm trong bucket chứa Query Point đó trước, sau đó duyệt lan ra khối $3 \times 3 \times 3$ xung quanh Query Point để đảm bảo kết quả tìm được là chính xác.
- Trong quá trình duyệt, ta dùng **Euclidean Distance** để tính khoảng cách giữa Query Point và các điểm trong các bucket, Nearest Neighbor Point là điểm có khoảng cách nhỏ nhất tới Query Point . Mỗi lần tìm được khoảng cách nhỏ hơn khoảng cách trước đó thì ra sẽ cập nhật lại min_distance và Nearest Neighbor Point cho các điểm trong bucket đến khi duyệt hết những trường hợp có thể xảy ra.

5 Giải thuật hiện thực Kd-Tree

5.1 Cấu trúc dữ liệu KdNode

```
class KdNode {
public:
    Point3D val;
    KdNode* left;
    KdNode* right;
    KdNode() : val(Point3D()), left(nullptr), right(nullptr){}
    KdNode(Point3D val) {
        this->val = val;
        this->left = this->right = nullptr;
    }
};
```

5.2 Cấu trúc dữ liệu KdTree

```
class KdTree {
public:
    KdNode* root;
    KdTree() {
        this->root = nullptr;
    }
};
```

5.3 Build Tree

5.3.1 Build Kd-Tree bằng cách thêm tuần tự

```
KdNode* insertRec(KdNode* root, Point3D val, int depth) {
    if (!root) {
        return new KdNode(val);
    }
    int axis = depth % 3;
    if (val.myMap[axis] < root->val.myMap[axis]) {
        root->left = insertRec(root->left, val, depth + 1);
    }
    else {
        root->right = insertRec(root->right, val, depth + 1);
    }
    return root;
}

void insert(Point3D val) {
    this->root = insertRec(root, val, 0);
}

void insertDataSet(Point3D* val, int size) {
    for (int i = 0; i < size; i++) {
        insert(val[i]);
    }
}
```

Các hàm của giải thuật:

- **insertRec**: thêm một điểm vào cây bằng cách gọi đệ quy, ở đây tính **axis = depth % 3** để chọn trục so sánh, nếu axis = 0 chọn theo x, axis = 1 chọn theo y, axis = 2 chọn theo z. Còn lại tương tự như insert vào cây BST.
- **insert**: thêm một điểm vào cây KdTree.
- **insertDataSet**: insert dataSet vào để build KdTree.

5.3.2 Build Balanced Kd-Tree bằng cách tìm trung vị

```
struct ComparatorX {
```

```
bool operator() (const Point3D point1, const Point3D point2) const {
    return point1.x < point2.x;
}
};
struct ComparatorY {
    bool operator() (const Point3D point1, const Point3D point2) const {
        return point1.y < point2.y;
    }
};
struct ComparatorZ {
    bool operator() (const Point3D point1, const Point3D point2) const {
        return point1.z < point2.z;
    }
};
KdNode* insertBalancedTreeRec(vector<Point3D> dataSet, int depth) {
    if (dataSet.size() == 0) return nullptr;
    int axis = depth % 3;
    if (axis == 0) {
        sort(dataSet.begin(), dataSet.end(), ComparatorX());
    }
    else if (axis == 1) {
        sort(dataSet.begin(), dataSet.end(), ComparatorY());
    }
    else {
        sort(dataSet.begin(), dataSet.end(), ComparatorZ());
    }
    int median = dataSet.size() / 2;
    KdNode* node = new KdNode(dataSet[median]);
    vector<Point3D>left;
    vector<Point3D>right;
    for (int i = 0; i < dataSet.size(); i++) {
        if (i == median) continue;
        if (i < median) left.push_back(dataSet[i]);
        else if (i > median) right.push_back(dataSet[i]);
    }
    node->left = insertBalancedTreeRec(left, depth + 1);
    node->right = insertBalancedTreeRec(right, depth + 1);
    return node;
}
void insertDataSetToBalancedTree(vector<Point3D> dataSet) {
    this->root = insertBalancedTreeRec(dataSet, 0);
}
```

Các hàm của giải thuật:

- **insertBalancedTreeRec**: insert điểm vào cây Kd-Tree theo phương pháp "**Median of list**" để đảm bảo cây **balanced**, đầu tiên tính axis để chọn trục. Tiếp theo là **sort()** dataSet đầu vào theo trục đã chọn, nếu axis = 0 thì sort dataSet **tăng dần** theo x, axis = 1 sort dataSet theo y, axis = 2 sort dataSet theo z. Tiếp theo lấy vị trí trung vị làm pivot, các vị trí bên trái pivot (nhỏ hơn trung vị **tính theo trục đã chọn từ axis** thì lưu vào mảng left), các vị trí bên phải pivot(lớn hơn trung vị

tính theo trục đã chọn từ axis thì lưu vào mảng right. Left và right lần lượt là cây con trái và phải của điểm trung vị vừa tìm được, tiếp tục gọi đệ quy với các cây con trái và phải tương tự.

- `insertDataSetToBalancedTree`: insert dataSet to build a Balanced Tree.

5.4 Search Nearest Neighbor Point KD-Tree

```
bool isEqual(Point3D point1, Point3D point2) {
    for (int i = 0; i < 3; i++) {
        if (point1.myMap[i] != point2.myMap[i]) {
            return false;
        }
    }
    return true;
}

double distance(Point3D point1, Point3D point2) {
    return sqrt(pow(point1.x - point2.x, 2) + pow(point1.y - point2.y, 2) + pow(point1.z -
        point2.z, 2));
}

KdNode* closer(KdNode* n0, KdNode* n1, KdNode* query){
    if(!n0) return n1;
    if(!n1) return n0;
    double d1 = distance(n0->val, query->val);
    double d2 = distance(n1->val, query->val);
    if(d1 < d2) {
        return n0;
    }
    return n1;
}

KdNode* nearestNeighborRec(KdNode* root, KdNode* query, int depth, int& numberOfComparison) {
    if(!root) return nullptr;
    KdNode* nextBranch = nullptr;
    KdNode* otherBranch = nullptr;
    int axis = depth % 3;
    if (query->val.myMap[axis] < root->val.myMap[axis]) {
        nextBranch = root->left;
        otherBranch = root->right;
    }
    else {
        nextBranch = root->right;
        otherBranch = root->left;
    }
    numberOfComparison += 1;
    KdNode* temp = nearestNeighborRec(nextBranch, query, depth + 1, numberOfComparison);
    KdNode* best = closer(temp, root, query);
    double radius = distance(query->val, best->val);
    double dist = abs(query->val.myMap[axis] - root->val.myMap[axis]);
    if(dist <= radius) {
        temp = nearestNeighborRec(otherBranch, query, depth + 1, numberOfComparison);
    }
}
```

```
        best = closer(temp, best, query);
    }
    return best;
}

KdNode* nearestNeighbor(KdNode* query, int& numberOfComparison) {
    return nearestNeighborRec(root, query, 0, numberOfComparison);
}

void searchNearestNeighborPoint(Point3D queryVal, bool flag = true) {
    KdNode* query = new KdNode(queryVal);
    int numberOfComparison = 0;
    auto startTreeSearch = high_resolution_clock::now();
    KdNode* nearestNeighborPoint = nearestNeighbor(query, numberOfComparison);
    auto endTreeSearch = high_resolution_clock::now();
    double searchTree_time = duration_cast<microseconds>(endTreeSearch -
        startTreeSearch).count() * pow(10, -6);
    if(!nearestNeighborPoint) {
        if(flag) cout << "-KdTree is empty !" << endl;
        return;
    }
    bool checkEqual = isEqual(nearestNeighborPoint->val, query->val);
    if(!flag) return;
    if(checkEqual) {
        cout << "-Output: Found exacly Query Point ";
        query->val.print();
        cout << " in dataSet." << endl;
    }
    else {
        cout << "-Output: Nearest Neighbor Point: ";
        nearestNeighborPoint->val.print();
        cout << " - Distance: " << distance(nearestNeighborPoint->val, query->val) << endl;
    }
    cout << "-SearchNNP Time: " << searchTree_time << " seconds." << endl;
    cout << "-Number of comparisons: " << numberOfComparison << endl;
}
```

Các hàm của giải thuật:

- **isEqual** : kiểm tra hai điểm có trùng nhau hay không.
- **distance**: tính **Euclidean Distance** giữa hai điểm trong không gian
- **closer**: so sánh giữa hai điểm point1 và point2 điểm nào gần với query hơn.
- **nearestNeighborRec**: tìm Nearest Neighbor Point bằng phương pháp đệ quy, ý tưởng tương tự những phần đã trình bày ở **mục 4.1.3**.
- **searchNearestNeighborPoint**: in ra kết quả sau khi tìm NNP bằng Kd-Tree, số lần so sánh và execution time để tìm NNP (đơn vị là giây), đo thời gian bằng thư viện **chrono**. Biến **flag** chỉ dùng để in kết quả, không có vai trò trong giải thuật.

5.5 Giải phóng Tree

```
void clear(KdNode*& root) {  
    if (root) {  
        clear(root->left);  
        clear(root->right);  
        delete root;  
        root = nullptr;  
    }  
}
```

Các hàm của giải thuật:

- **clear** : Giải phóng cây sau khi sử dụng bằng phương pháp duyệt hậu thứ tự.

6 Giải thuật hiện thực Hash

6.1 Hash Function

```
int hashFunct(Point3D point) {  
    int x_hash = (int)point.x;  
    int y_hash = (int)point.y;  
    int z_hash = (int)point.z;  
    return (int)(x_hash * 10000 / binwidth) + (int)(y_hash * 100 / binwidth) + (int)(z_hash * 1  
        / binwidth);  
}
```

Các hàm của giải thuật:

- **hashFunct** : xây dựng hàm **Hash Function** tương tự như phần đã trình bày ở mục 4.2.2

6.2 Cấu trúc dữ liệu HashTable

```
class Hashing {  
public:  
    vector <vector<Point3D>> hashTable;  
    int binwidth;  
    int sizeOfTable;  
    Hashing(int sizeOfDataSet) {  
        if (sizeOfDataSet >= 0 && sizeOfDataSet < 1000) {  
            this->binwidth = 5;  
            this->sizeOfTable = 200000;  
            hashTable.resize(sizeOfTable);  
        }  
        else if (sizeOfDataSet >= 1000 && sizeOfDataSet < 100000) {  
            this->binwidth = 2;  
            this->sizeOfTable = 500000;  
            hashTable.resize(sizeOfTable);  
        }  
        else {
```

```
        this->binwidth = 1;
        this->sizeOfTable = 1000000;
        hashTable.resize(sizeOfTable);
    }
};
```

Dựa vào binwidth mà ta sẽ **resize()** cho HashTable, size của HashTable = **1000000 / binwidth**. Với binwidth = 5 ta khởi tạo size của HashTable = 200000, binwidth = 2 size của HashTable = 500000, binwidth = 1 size của HashTable = 1000000.

6.3 Build HashTable

```
void insertPoint(Point3D point) {
    int index = hashFunct(point);
    hashTable[index].push_back(point);
}

void insertDataSet(Point3D* dataSet, int size) {
    for (int i = 0; i < size; i++) {
        insertPoint(dataSet[i]);
    }
}

void insertDataSetForVector(vector<Point3D>dataSet, int size){
    for (int i = 0; i < size; i++) {
        insertPoint(dataSet[i]);
    }
}
```

Các hàm của giải thuật:

- **insertPoint**: sử dụng hàm **hashFunct** để tìm ra index của bucket tương ứng với point trong Hash Table, sau đó thêm điểm đó vào bucket với index vừa tìm được.
- **insertDataSet**: insert dataSet để build Hash Table.

6.4 Search Nearest Neighbor Point HashTable

```
bool isValidPoint(Point3D point){
    return point.x < 0 || point.y < 0 || point.z < 0 || point.x > 100 || point.y > 100 ||
        point.z > 100;
}

void localitySearch(Point3D query, double& min_dist, Point3D& nearestNeighborPoint, int&
    numberOfComparison) {
    int i = binwidth;
    bool flag = false;
    while (true) {
        for (int dx = -i; dx <= i; dx++) {
            for (int dy = -i; dy <= i; dy++) {
                for (int dz = -i; dz <= i; dz++) {
```



```
        Point3D temp = Point3D(query.x + dx, query.y + dy, query.z + dz);
        if(isInvalidPoint(temp)) continue;
        int temp_pos = hashFunct(temp);
        if (hashTable[temp_pos].size() == 0) continue;
        flag = true;
        for (auto it = hashTable[temp_pos].begin(); it != hashTable[temp_pos].end();
             it++) {
            numberOfComparison++;
            double dist = distance(query, *it);
            if (dist < min_dist) {
                min_dist = dist;
                nearestNeighborPoint = *it;
            }
        }
    }
}

if (flag) return;
i += binwidth;
}
}

void searchNearestNeighborPoint(Point3D query, bool flag = true) {
    double min_dist = 99999;
    Point3D nearestNeighborPoint;
    auto startHashSearch = high_resolution_clock::now();
    int query_index = hashFunct(query);
    int numberOfComparison = 0;
    if (hashTable[query_index].size() == 0) {
        localitySearch(query, min_dist, nearestNeighborPoint, numberOfComparison);
    }
    else {
        for (auto it = hashTable[query_index].begin(); it != hashTable[query_index].end(); it++) {
            double dist = distance(query, *it);
            numberOfComparison++;
            if (dist < min_dist) {
                min_dist = dist;
                nearestNeighborPoint = *it;
            }
        }
        localitySearch(query, min_dist, nearestNeighborPoint, numberOfComparison);
    }
    auto endHashSearch = high_resolution_clock::now();
    double searchHash_time = duration_cast<microseconds>(endHashSearch -
        startHashSearch).count() * pow(10, -6);
    if(!flag) return;
    bool checkEqual = isEqual(nearestNeighborPoint, query);
    if (checkEqual) {
        cout << "-Output: Found exacly Query Point ";
        query.print();
    }
}
```

```
    cout << " in dataSet." << endl;
}
else {
    cout << "-Output: Nearest Neighbor Point: ";
    nearestNeighborPoint.print();
    cout << " - Distance: " << distance(nearestNeighborPoint, query) << endl;
}
cout << "-SearchNNP Time: " << searchHash_time << " seconds." << endl;
cout << "-Number of comparisons: " << numberOfComparison << endl;
}
```

Các hàm của giải thuật:

- **isInvalidPoint**: Kiểm tra xem điểm Point3D vào có ngoài khoảng [0, 100] hay không, vì do **size** của HashTable là cố định, nên nếu x,y,z tồn tại số âm hoặc lớn hơn 100 thì sẽ xảy ra lỗi **Overflow**. Hàm dùng để xét khi **localitySearch** ở những khối gần biên có thể xảy ra tràn, trong trường hợp này ta bỏ qua các điểm trên.
- **localitySearch**: Dùng để search những khối lân cận với **Query Point** trong trường hợp bucket tương ứng với index của Query Point đang rỗng (không tồn tại các điểm có cùng **phần nguyên các tọa độ** trùng với Query Point), ta sẽ tìm lan ra các khối xung quanh (**BFS**), ý tưởng giải thuật đã được trình bày ở **mục 4.2.4**.
- **searchNearestNeighborPoint**: in ra kết quả sau khi tìm NNP bằng phương pháp Hash, số lần so sánh và execution time để tìm NNP (đơn vị là giây), đo thời gian bằng thư viện **chrono**. Biến **flag** chỉ dùng để in kết quả, không có vai trò trong giải thuật.

6.5 Giải phóng HashTable

```
void clear(){
    for(int i = 0; i < sizeOfTable; i++) {
        if(hashTable[i].size() == 0) continue;
        hashTable[i].clear();
    }
    hashTable.clear();
    this->sizeOfTable = 0;
    this->binwidth = 0;
}
```

Các hàm của giải thuật:

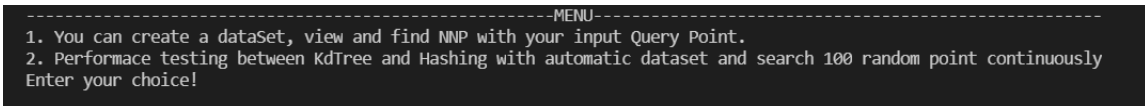
- **clear**: Giải phóng HashTable sau khi sử dụng

7 Kiểm thử chương trình và so sánh hiệu năng

***Note:** Các số liệu được đo trên máy tính của tác giả với thông số như sau, time execution có thể sẽ khác nhau (tăng hoặc giảm) trên các máy khác nhau.

Processor: Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz
Installed memory (RAM): 8.00 GB (7.78 GB usable)
System type: 64-bit Operating System, x64-based processor
Pen and Touch: No Pen or Touch Input is available for this Display

7.1 Tương tác người dùng



Hình 5: Bảng menu tương tác người dùng

Chương trình cho phép người dùng chọn 2 option

- **Option 1** : Người dùng sẽ được lựa chọn **size** của dataSet muốn random, có thể lựa chọn view dataSet hay không ? Nếu size của dataSet nhỏ thì có thể view dataSet để kiểm tra kết quả random có đúng yêu cầu đề tài hay không. Quan trọng nhất, ở option 1 người dùng có thể test tay bằng cách nhập **Query Point** và xem kết quả tìm kiếm **NNP** cũng như so sánh hiệu năng giữa các giải thuật trên với các tập dữ liệu tùy ý.
- **Option 2**: Chương trình sẽ tự động random dữ liệu với các dataSet với size theo thứ tự sau : **100,500,1000,5000,10000,50000,100000, 500000, 1000000** và search liên tiếp 100 điểm random để đo hiệu năng giữa các giải thuật trên.

7.2 Kiểm thử chương trình và so sánh hiệu năng khi search với một điểm

7.2.1 Dataset với size = 100

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 2.56
Y_0: 14.56
Z_0: 25.89
Query Point is: (2.56, 14.56, 25.89)
Performace with dataSet size = 100:
*Linear search O(N):
-Output: Nearest Neighbor Point: (11.707, 7.977, 28.574) - Distance: 11.5848
-SearchNNP Time: 0 seconds.
-Number of comparisons: 100

*Kd-Tree:
-Build Tree Time: 0.001002 seconds.
-Output: Nearest Neighbor Point: (11.707, 7.977, 28.574) - Distance: 11.5848
-SearchNNP Time: 0 seconds.
-Number of comparisons: 35

*Balanced Kd-Tree:
-Build Balanced Tree Time: 0.005982 seconds.
-Output: Nearest Neighbor Point: (11.707, 7.977, 28.574) - Distance: 11.5848
-SearchNNP Time: 0 seconds.
-Number of comparisons: 8

*Hashing:
-Build Hash Time: 0 seconds.
-Output: Nearest Neighbor Point: (11.707, 7.977, 28.574) - Distance: 11.5848
-SearchNNP Time: 0.011961 seconds.
-Number of comparisons: 5
```

Hình 6: Example 1 of dataSet size = 100

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 1.23
Y_0: 2.34
Z_0: 4.56
Query Point is: (1.23, 2.34, 4.56)
Performace with dataSet size = 100:
*Linear search O(N):
-Output: Nearest Neighbor Point: (2.603, 11.56, 17.545) - Distance: 15.9845
-SearchNNP Time: 0 seconds.
-Number of comparisons: 100

*Kd-Tree:
-Build Tree Time: 0.002031 seconds.
-Output: Nearest Neighbor Point: (2.603, 11.56, 17.545) - Distance: 15.9845
-SearchNNP Time: 0.000996 seconds.
-Number of comparisons: 31

*Balanced Kd-Tree:
-Build Balanced Tree Time: 0.006037 seconds.
-Output: Nearest Neighbor Point: (2.603, 11.56, 17.545) - Distance: 15.9845
-SearchNNP Time: 0 seconds.
-Number of comparisons: 9

*Hashing:
-Build Hash Time: 0 seconds.
-Output: Nearest Neighbor Point: (2.603, 11.56, 17.545) - Distance: 15.9845
-SearchNNP Time: 0.038199 seconds.
-Number of comparisons: 5
```

Hình 7: Example 2 of dataSet size = 100

Với size = 100:

- Xét về Build Time: **Balanced Kd-Tree** có time construction lâu nhất so với các cấu trúc dữ liệu còn lại. **Balanced Kd-Tree** có thời gian build **gấp 3 lần** so với **Normal Kd-Tree**. **Hash** có thời gian build nhanh nhất.

⇒ Với size = 100, **Hash** > **Normal KdTree** > **Balanced Kd-Tree** về build Data Structure.

- Xét về số lần so sánh : Số lần so sánh của **Normal Kd-Tree** cao hơn so với **Balanced Kd-Tree** bởi vì lúc này **Normal Kd-Tree** chưa cân bằng, với $N = 100$ thì **Normal Kd-Tree** đã phải so sánh đến hơn 30 lần, trong khi đó **Balanced Kd-Tree** chỉ cần so sánh 8 - 9 lần. Số lần so sánh của **Hash** chỉ dao động 5 lần nhưng chưa tính số **traveled bucket**.
- Xét về SearchNNP Time: **Hash** có số lần so sánh ít nhất nhưng searchNNP Time cao nhất vì **Hash** phải thực hiện các giai đoạn trung gian như **traverse sang các bin lân cận** để tìm được bin chứa phần tử gần với Query Point nhất, trong thuật toán thì nếu bucket rỗng thì chúng ta sẽ bỏ qua (không đếm vào số lần so sánh), lúc này do size nhỏ nên khoảng cách giữa các bucket tương đối lớn (tồn tại nhiều bucket rỗng). Số lần so sánh ở đây chỉ tính là số lần so sánh với các phần tử trong bucket, chưa tính số traveled bucket.

⇒ Với size = 100, **Balanced KdTree** > **Normal KdTree** > **Hash** về searchNNP.

7.2.2 Dataset với size = 1000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 12.34
Y_0: 23.45
Z_0: 34.56
Query Point is: (12.34, 23.45, 34.56)
Performace with dataSet size = 1000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (8.853, 19.242, 32.88) - Distance: 5.71742
-SearchNNP Time: 0.001 seconds.
-Number of comparisons: 1000

*Kd-Tree:
-Build Tree Time: 0.010008 seconds.
-Output: Nearest Neighbor Point: (8.853, 19.242, 32.88) - Distance: 5.71742
-SearchNNP Time: 0 seconds.
-Number of comparisons: 81

*Balanced Kd-Tree:
-Build Balanced Tree Time: 0.054979 seconds.
-Output: Nearest Neighbor Point: (8.853, 19.242, 32.88) - Distance: 5.71742
-SearchNNP Time: 0 seconds.
-Number of comparisons: 37

*Hashing:
-Build Hash Time: 0.002033 seconds.
-Output: Nearest Neighbor Point: (8.853, 19.242, 32.88) - Distance: 5.71742
-SearchNNP Time: 0.000976 seconds.
-Number of comparisons: 2
```

Hình 8: Example 1 of dataSet size = 1000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 12.56
Y_0: 54.45
Z_0: 14.6
Query Point is: (12.56, 54.45, 14.6)
Performace with dataSet size = 1000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (8.798, 59.263, 13.901) - Distance: 6.14868
-SearchNNP Time: 0.001 seconds.
-Number of comparisons: 1000

*Kd-Tree:
-Build Tree Time: 0.00913 seconds.
-Output: Nearest Neighbor Point: (8.798, 59.263, 13.901) - Distance: 6.14868
-SearchNNP Time: 0 seconds.
-Number of comparisons: 125

*Balanced Kd-Tree:
-Build Balanced Tree Time: 0.055891 seconds.
-Output: Nearest Neighbor Point: (8.798, 59.263, 13.901) - Distance: 6.14868
-SearchNNP Time: 0 seconds.
-Number of comparisons: 33

*Hashing:
-Build Hash Time: 0.001993 seconds.
-Output: Nearest Neighbor Point: (8.798, 59.263, 13.901) - Distance: 6.14868
-SearchNNP Time: 0.003495 seconds.
-Number of comparisons: 4
```

Hình 9: Example 2 of dataSet size = 1000

Với size = 1000:

- Xét về Build Time: **Balanced Kd-Tree** có time construction lâu nhất so với các cấu trúc dữ liệu còn lại. **Balanced Kd-Tree** có thời gian build **gấp 5 lần** so với **Normal Kd-Tree**. **Hash** có thời gian build nhanh nhất.
⇒ Với size = 1000, **Hash** > **Normal KdTree** > **Balanced Kd-Tree** về build Data Structure.
- Xét về số lần so sánh : Số lần so sánh của **Normal Kd-Tree** cao hơn nhiều so với **Balanced Kd-Tree** bởi vì lúc này **Normal Kd-Tree** chưa cân bằng. Tiếp tục qua ví dụ này, ta thấy **Balanced Kd-Tree** có hiệu quả cao hơn so với **Kd-Tree** trong về số phép so sánh khi searchNNP. Số lần so sánh của **Hash** chỉ dao động 2 - 4 lần nhưng chưa tính số **traveled bucket**
- Xét về SearchNNP Time: **Hash** có số lần so sánh ít nhất nhưng searchNNP Time cao nhất vì **Hash** phải thực hiện các giai đoạn trung gian như **traverse sang các bin lân cận** để tìm được bin chứa phần tử gần với Query Point nhất, trong thuật toán thì nếu bucket rỗng thì chúng ta sẽ bỏ qua (không đếm vào số lần so sánh), lúc này do size nhỏ nên khoảng cách giữa các bucket tương đối lớn (tồn tại nhiều bucket rỗng). Số lần so sánh ở đây chỉ tính là số lần so sánh với các phần tử trong bucket, chưa tính số traveled bucket
⇒ Với size = 1000, **Balanced KdTree** > **Normal KdTree** > **Hash** về searchNNP.

7.2.3 Dataset với size = 10000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 86.51
Y_0: 99.89
Z_0: 25.75
Query Point is: (86.51, 99.89, 25.75)
Performace with dataSet size = 10000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0.008 seconds.
-Number of comparisons: 10000

*Kd-Tree:
-Build Tree Time: 0.09386 seconds.
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0 seconds.
-Number of comparisons: 33

*Balanced Kd-Tree:
-Build Balanced Tree Time: 0.925439 seconds.
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0 seconds.
-Number of comparisons: 53

*Hashing:
-Build Hash Time: 0.01394 seconds.
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0.000775 seconds.
-Number of comparisons: 2
```

Hình 10: Example 1 of dataSet size = 10000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 86.51
Y_0: 99.89
Z_0: 25.75
Query Point is: (86.51, 99.89, 25.75)
Performace with dataSet size = 10000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0.008 seconds.
-Number of comparisons: 10000

*Kd-Tree:
-Build Tree Time: 0.09386 seconds.
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0 seconds.
-Number of comparisons: 33

*Balanced Kd-Tree:
-Build Balanced Tree Time: 0.925439 seconds.
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0 seconds.
-Number of comparisons: 53

*Hashing:
-Build Hash Time: 0.01394 seconds.
-Output: Nearest Neighbor Point: (84.654, 97.457, 25.29) - Distance: 3.09448
-SearchNNP Time: 0.000775 seconds.
-Number of comparisons: 2
```

Hình 11: Example 2 of dataSet size = 10000

Với size = 10000:

- Xét về Build Time: **Balanced Kd-Tree** có time construction lâu nhất so với các cấu trúc dữ liệu còn lại. **Balanced Kd-Tree** có thời gian build **gấp 9 lần** so với **Normal Kd-Tree**. Qua những trường hợp trên, ta nhận thấy với tập dữ liệu càng lớn thì thời gian **Balanced Kd-Tree** build càng có sự chênh lệch lớn. **Hash** có thời gian build nhanh nhất.
⇒ Với size = 10000, **Hash** > **Normal KdTree** > **Balanced Kd-Tree** về build Data Structure.
- Xét về số lần so sánh : Số lần so sánh của **Normal Kd-Tree** cao hơn so với **Balanced Kd-Tree** bởi vì lúc này **Normal Kd-Tree** chưa cân bằng. Tiếp tục qua ví dụ này, ta thấy **Balanced Kd-Tree** có hiệu quả cao hơn so với **Kd-Tree** trong về số phép so sánh khi searchNNP. Số lần so sánh của **Hash** chỉ dao động 2 lần nhưng chưa tính số **traveled bucket**. Tuy nhiên dựa vào searchNNP Time ta có thể dự đoán số **traveled bucket** là không đáng kể.
- Xét về SearchNNP Time: Với tập dữ liệu cỡ vừa, ta thấy seachNNP Time của cả **Hash**, **Balanced KdTree**, **KdTree** đều **xấp xỉ 0**. Lý do vì với tập dữ liệu cỡ vừa thì các điểm phân phối vào các bucket đều hơn, số bucket rỗng giảm, số **traveled bucket** giảm, performance của **Hash** tăng.
⇒ Với size = 10000, **Balanced KdTree** > **Normal KdTree** > **Hash** về searchNNP.

7.2.4 Dataset với size = 100000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 9.999
Y_0: 24.25
Z_0: 17.12
Query Point is: (9.999, 24.25, 17.12)
Performace with dataSet size = 100000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (10.575, 24.314, 17.267) - Distance: 0.597897
-SearchNNP Time: 0.085 seconds.
-Number of comparisons: 100000

*Kd-Tree:
-Build Tree Time: 1.24421 seconds.
-Output: Nearest Neighbor Point: (10.575, 24.314, 17.267) - Distance: 0.597897
-SearchNNP Time: 0 seconds.
-Number of comparisons: 40

*Balanced Kd-Tree:
-Build Balanced Tree Time: 14.7558 seconds.
-Output: Nearest Neighbor Point: (10.575, 24.314, 17.267) - Distance: 0.597897
-SearchNNP Time: 0 seconds.
-Number of comparisons: 19

*Hashing:
-Build Hash Time: 0.141983 seconds.
-Output: Nearest Neighbor Point: (10.575, 24.314, 17.267) - Distance: 0.597897
-SearchNNP Time: 0 seconds.
-Number of comparisons: 5
```

Hình 12: Example 1 of dataSet size = 100000


```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 22.5
Y_0: 33.3
Z_0: 37.5
Query Point is: (22.5, 33.3, 37.5)
Performace with dataSet size = 100000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (22.937, 34.229, 36.9) - Distance: 1.18912
-SearchNNP Time: 0.121 seconds.
-Number of comparisons: 100000

*Kd-Tree:
-Build Tree Time: 1.75523 seconds.
-Output: Nearest Neighbor Point: (22.937, 34.229, 36.9) - Distance: 1.18912
-SearchNNP Time: 0 seconds.
-Number of comparisons: 84

*Balanced Kd-Tree:
-Build Balanced Tree Time: 20.9714 seconds.
-Output: Nearest Neighbor Point: (22.937, 34.229, 36.9) - Distance: 1.18912
-SearchNNP Time: 0 seconds.
-Number of comparisons: 63

*Hashing:
-Build Hash Time: 0.181261 seconds.
-Output: Nearest Neighbor Point: (22.937, 34.229, 36.9) - Distance: 1.18912
-SearchNNP Time: 0 seconds.
-Number of comparisons: 3
```

Hình 13: Example 2 of dataSet size = 100000

Với size = 100000:

- Xét về Build Time: **Balanced Kd-Tree** có time construction lâu nhất so với các cấu trúc dữ liệu còn lại. **Balanced Kd-Tree** có thời gian build **gấp 14 lần** so với **Normal Kd-Tree**. Qua những trường hợp trên, ta nhận thấy với tập dữ liệu càng lớn thì thời gian **Balanced Kd-Tree** build càng có sự chênh lệch lớn. **Hash** có thời gian build nhanh nhất.
⇒ Với size = 100000, **Hash** > **Normal KdTree** > **Balanced Kd-Tree** về build Data Structure.
- Xét về số lần so sánh : Số lần so sánh của **Normal Kd-Tree** cao hơn so với **Balanced Kd-Tree** bởi vì lúc này **Normal Kd-Tree** chưa cân bằng. Số lần so sánh của **Hash** chỉ dao động 3-5 lần nhưng chưa tính số **traveled bucket**. Tuy nhiên dựa vào searchNNP Time ta có thể dự đoán số **traveled bucket** là không đáng kể.
- Xét về SearchNNP Time: Với tập dữ liệu lớn, ta thấy seachNNP Time của cả **Hash**, **Balanced KdTree**, **KdTree** đều **xấp xỉ 0**. Lý do vì với tập dữ liệu lớn thì các điểm phân phối vào các bucket đều hơn, số bucket rộng giảm, số **traveled bucket** giảm, performance của **Hash** tăng.
⇒ Với size = 100000, **Hash** ≥ **Balanced Kd-Tree** > **Normal KdTree** về searchNNP. Với dataSet cỡ vừa và lớn, cả 3 cấu trúc dữ liệu đều đảm bảo được search **1** điểm NNP trong khoảng thời gian **xấp xỉ 0s**.

7.2.5 Dataset với size = 500000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 14.58
Y_0: 25.69
Z_0: 36.47
Query Point is: (14.58, 25.69, 36.47)
Performace with dataSet size = 500000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (14.337, 25.449, 36.213) - Distance: 0.427995
-SearchNNP Time: 0.389 seconds.
-Number of comparisons: 500000

*Kd-Tree:
-Build Tree Time: 7.9565 seconds.
-Output: Nearest Neighbor Point: (14.337, 25.449, 36.213) - Distance: 0.427995
-SearchNNP Time: 0.000999 seconds.
-Number of comparisons: 77

*Balanced Kd-Tree:
-Build Balanced Tree Time: 102.893 seconds.
-Output: Nearest Neighbor Point: (14.337, 25.449, 36.213) - Distance: 0.427995
-SearchNNP Time: 0 seconds.
-Number of comparisons: 20

*Hashing:
-Build Hash Time: 0.716453 seconds.
-Output: Nearest Neighbor Point: (14.337, 25.449, 36.213) - Distance: 0.427995
-SearchNNP Time: 0 seconds.
-Number of comparisons: 15
```

Hình 14: Example 1 of dataSet size = 500000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 97.79
Y_0: 14.46
Z_0: 32.23
Query Point is: (97.79, 14.46, 32.23)
Performace with dataSet size = 500000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (96.904, 14.633, 32.819) - Distance: 1.07789
-SearchNNP Time: 0.383 seconds.
-Number of comparisons: 500000

*Kd-Tree:
-Build Tree Time: 6.91994 seconds.
-Output: Nearest Neighbor Point: (96.904, 14.633, 32.819) - Distance: 1.07789
-SearchNNP Time: 0 seconds.
-Number of comparisons: 59

*Balanced Kd-Tree:
-Build Balanced Tree Time: 89.3426 seconds.
-Output: Nearest Neighbor Point: (96.904, 14.633, 32.819) - Distance: 1.07789
-SearchNNP Time: 0 seconds.
-Number of comparisons: 45

*Hashing:
-Build Hash Time: 0.695541 seconds.
-Output: Nearest Neighbor Point: (96.904, 14.633, 32.819) - Distance: 1.07789
-SearchNNP Time: 0 seconds.
-Number of comparisons: 8
```

Hình 15: Example 2 of dataSet size = 500000

Với size = 500000:

- Xét về Build Time: **Balanced Kd-Tree** có time construction lâu nhất so với các cấu trúc dữ liệu còn lại. **Balanced Kd-Tree** có thời gian build **gấp 12 - 14 lần** so với **Normal Kd-Tree**. **Hash** có thời gian build nhanh nhất.
⇒ Với size = 500000, **Hash > Normal KdTree > Balanced Kd-Tree** về build Data Structure.
- Xét về số lần so sánh : Số lần so sánh của **Normal Kd-Tree** cao hơn so với **Balanced Kd-Tree** bởi vì lúc này **Normal Kd-Tree** chưa cân bằng. Số lần so sánh của **Hash** tăng so với trước, dao động 8 -15 lần nhưng chưa tính số **traveled bucket**. Số lần so sánh tăng do lúc này độ giữa các điểm xảy ra nhiều hơn, trong một bucket đã tồn tại nhiều điểm. Dựa vào searchNNP Time ta có thể dự đoán số **traveled bucket** là không đáng kể.
- Xét về SearchNNP Time: Với tập dữ liệu lớn, ta thấy seachNNP Time của cả **Hash**, **Balanced KdTree**, **KdTree** đều **xấp xỉ 0s**. Lý do vì với tập dữ liệu lớn thì các điểm phân phối vào các bucket đều hơn, số bucket rỗng giảm, số **traveled bucket** giảm, performance của **Hash** tăng. Cả 3 cấu trúc dữ liệu đều đảm bảo được search 1 điểm NNP trong khoảng thời gian **xấp xỉ 0s**
⇒ Với size = 500000, **Hash ≥ Balanced Kd-Tree > Normal KdTree** về searchNNP. Với dataSet cỡ lớn, cả 3 cấu trúc dữ liệu đều đảm bảo được search 1 điểm NNP trong khoảng thời gian **xấp xỉ 0s**.

7.2.6 Dataset với size = 1000000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 23.456
Y_0: 45.678
Z_0: 78.912
Query Point is: (23.456, 45.678, 78.912)
Performace with dataSet size = 1000000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (23.234, 46.015, 79.393) - Distance: 0.62786
-SearchNNP Time: 0.799 seconds.
-Number of comparisons: 1000000

*Kd-Tree:
-Build Tree Time: 15.8814 seconds.
-Output: Nearest Neighbor Point: (23.234, 46.015, 79.393) - Distance: 0.62786
-SearchNNP Time: 0 seconds.
-Number of comparisons: 50

*Balanced Kd-Tree:
-Build Balanced Tree Time: 274.577 seconds.
-Output: Nearest Neighbor Point: (23.234, 46.015, 79.393) - Distance: 0.62786
-SearchNNP Time: 0 seconds.
-Number of comparisons: 26

*Hashing:
-Build Hash Time: 3.13979 seconds.
-Output: Nearest Neighbor Point: (23.234, 46.015, 79.393) - Distance: 0.62786
-SearchNNP Time: 0 seconds.
-Number of comparisons: 37
```

Hình 16: Example 1 of dataSet size = 1000000

```
Please input coordinate of Query Point that you want to find in dataSet:
X_0: 37.89
Y_0: 75.25
Z_0: 36.69
Query Point is: (37.89, 75.25, 36.69)
Performace with dataSet size = 1000000:
*Linear search O(N):
-Output: Nearest Neighbor Point: (37.928, 75.005, 36.942) - Distance: 0.353519
-SearchNNP Time: 0.806 seconds.
-Number of comparisons: 1000000

*Kd-Tree:
-Build Tree Time: 14.9502 seconds.
-Output: Nearest Neighbor Point: (37.928, 75.005, 36.942) - Distance: 0.353519
-SearchNNP Time: 0 seconds.
-Number of comparisons: 92

*Balanced Kd-Tree:
-Build Balanced Tree Time: 263.249 seconds.
-Output: Nearest Neighbor Point: (37.928, 75.005, 36.942) - Distance: 0.353519
-SearchNNP Time: 0.000918 seconds.
-Number of comparisons: 94

*Hashing:
-Build Hash Time: 2.74623 seconds.
-Output: Nearest Neighbor Point: (37.928, 75.005, 36.942) - Distance: 0.353519
-SearchNNP Time: 0 seconds.
-Number of comparisons: 42
```

Hình 17: Example 2 of dataSet size = 1000000

Với size = 1000000:

- Xét về Build Time: **Balanced Kd-Tree** có time construction lâu nhất so với các cấu trúc dữ liệu còn lại. **Balanced Kd-Tree** có thời gian build **gấp 18 - 20 lần** so với **Normal Kd-Tree**. Khi tập dữ liệu càng lớn, thời gian build **Balanced Kd-Tree** càng có sự chênh lệch rõ rệt với **Normal KdTree** và **Hash**. **Hash** có thời gian build nhanh nhất trong mọi trường hợp.
⇒ Với size = 1000000, **Hash** > **Normal KdTree** > **Balanced Kd-Tree** về build Data Structure.
- Xét về số lần so sánh : Số lần so sánh của **Normal Kd-Tree** và **Balanced Kd-Tree** cũng có sự chênh lệch bởi vì lúc này **Normal Kd-Tree** chưa cân bằng. Số lần so sánh của **Hash** tăng rõ rệt hơn so với trước, 37 -42 lần nhưng chưa tính số **traveled bucket**. Số lần so sánh tăng do lúc này độ gộp giữa các điểm xảy ra nhiều hơn, lúc này chứng tỏ độ gộp đã xảy ra nhiều hơn trong một bucket. Dựa vào searchNNP Time ta có thể dự đoán số **traveled bucket** là không đáng kể.
- Xét về SearchNNP Time: Với tập dữ liệu lớn, ta thấy seachNNP Time của cả **Hash**, **Balanced KdTree**, **KdTree** đều **xấp xỉ 0s**. Lý do vì với tập dữ liệu lớn thì các điểm phân phối vào các bucket đều hơn, số bucket rỗng giảm, số **traveled bucket** giảm, performance của **Hash** tăng. Cả 3 cấu trúc dữ liệu đều đảm bảo được search 1 điểm NNP trong khoảng thời gian **xấp xỉ 0s**
⇒ Với size = 1000000, **Hash** ≥ **Balanced Kd-Tree** > **Normal KdTree** về searchNNP. Với dataSet cỡ lớn, cả 3 cấu trúc dữ liệu đều đảm bảo được search 1 điểm NNP trong khoảng thời gian **xấp xỉ 0s**.

***Kết luận chung:**

- Xét về **Build Time**, với mọi tập dữ liệu, **Hash** > **Normal KdTree** > **Balanced Kd-Tree** về build Data Structure.
- Xét về **SearchNNP Time**:

- Với tập dữ liệu vừa và nhỏ (size từ 100 - 10000): **Balanced KdTree** > **Normal KdTree** > **Hash** về searchNNP Time.
 - Với tập dữ liệu tương đối lớn và lớn (từ 10000 trở lên): **Hash** \geq **Balanced Kd-Tree** > **Normal KdTree** về searchNNP
3. Còn việc so sánh giữa **searchNNP Time** của **Balanced Kd-Tree** và **Hash** ta sẽ so sánh khi search 100 điểm ở phần sau.

8 So sánh hiệu năng giữa KdTree và Hash ở các tập dữ liệu tăng dần

Chương trình sẽ tự động random dữ liệu với các dataSet với size theo thứ tự sau : **100,500,1000,5000,10000,50000,100000, 500000, 1000000** và search liên tiếp 100 điểm random để đo hiệu năng giữa các giải thuật trên.

```
Enter your choice!
2
In option 2, program automatically random 100 query point to test with randomly automatical dataset.
*Note: searchNNP = Total searchNNP time of 100 point continously.

Size: 100
*Kd-Tree: Build - 0.000995, SearchNNP - 0.003986
*Balanced Kd-Tree: Build - 0.001993, SearchNNP - 0.002039
*Hash: Build - 0, SearchNNP - 0.156102

Size: 500
*Kd-Tree: Build - 0.003986, SearchNNP - 0.005995
*Balanced Kd-Tree: Build - 0.020456, SearchNNP - 0.002974
*Hash: Build - 0.000996, SearchNNP - 0.151138

Size: 1000
*Kd-Tree: Build - 0.006932, SearchNNP - 0.007974
*Balanced Kd-Tree: Build - 0.053014, SearchNNP - 0.003976
*Hash: Build - 0.002003, SearchNNP - 0.014939

Size: 5000
*Kd-Tree: Build - 0.043298, SearchNNP - 0.021921
*Balanced Kd-Tree: Build - 0.388538, SearchNNP - 0.011429
*Hash: Build - 0.00598, SearchNNP - 0.064783

Size: 10000
*Kd-Tree: Build - 0.086959, SearchNNP - 0.018936
*Balanced Kd-Tree: Build - 0.908832, SearchNNP - 0.011963
*Hash: Build - 0.012957, SearchNNP - 0.030903

Size: 50000
*Kd-Tree: Build - 0.547361, SearchNNP - 0.023936
*Balanced Kd-Tree: Build - 6.45713, SearchNNP - 0.015367
*Hash: Build - 0.065778, SearchNNP - 0.015942

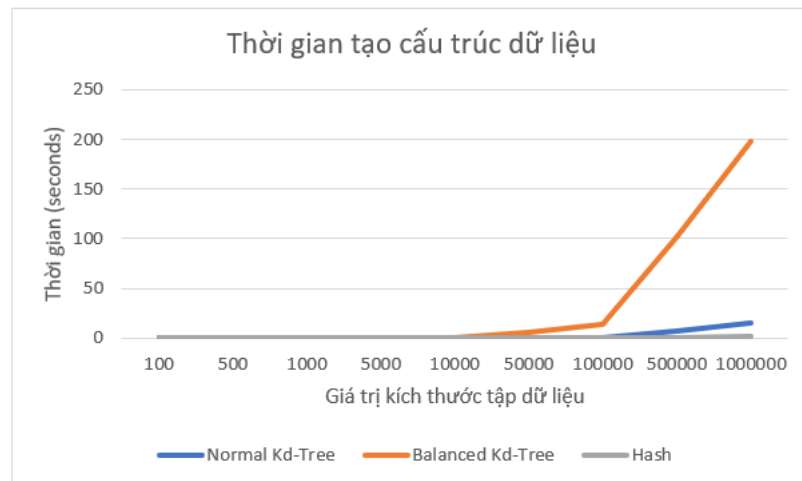
Size: 100000
*Kd-Tree: Build - 1.15443, SearchNNP - 0.028905
*Balanced Kd-Tree: Build - 13.9764, SearchNNP - 0.013954
*Hash: Build - 0.129564, SearchNNP - 0.004978

Size: 500000
*Kd-Tree: Build - 7.21237, SearchNNP - 0.03389
*Balanced Kd-Tree: Build - 102.11, SearchNNP - 0.014937
*Hash: Build - 0.711694, SearchNNP - 0.004983

Size: 1000000
*Kd-Tree: Build - 15.6328, SearchNNP - 0.034855
*Balanced Kd-Tree: Build - 197.467, SearchNNP - 0.016943
*Hash: Build - 1.45916, SearchNNP - 0.006977
```

Hình 18: Kết quả searchNNP 100 điểm liên tiếp với các tập dữ liệu khác nhau

8.1 So sánh hiệu năng tạo cấu trúc dữ liệu

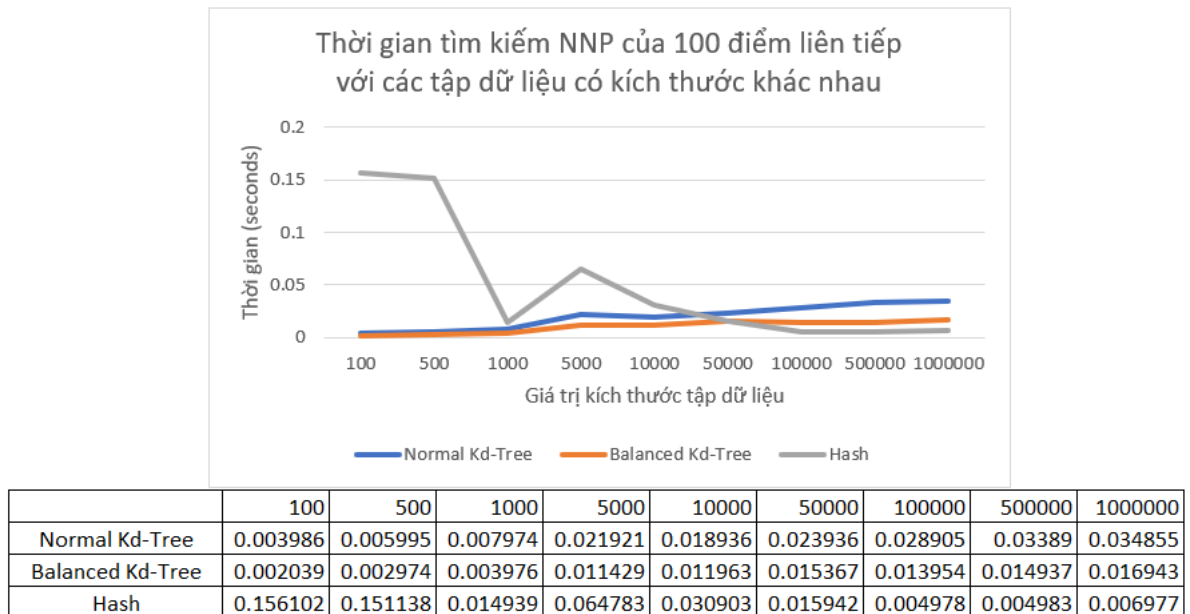


Hình 19: Thời gian tạo cấu trúc dữ liệu (tính theo seconds)

Nhận xét:

- Time complexity khi build một cây **Balanced KD-Tree** là $O(N(\log N)^2)$, cây **Normal Kd-Tree** là $O(N \log N)$, còn build **Hash** là $O(N)$ điều này thể hiện rõ trên đồ thị với các tập dữ liệu khác nhau.
- Nhìn chung, hiển nhiên thời gian build của cả 3 cấu trúc dữ liệu đều tăng khi kích thước tập dữ liệu tăng dần.
- Với tập dữ liệu nhỏ (từ 0 đến 10000), thời gian build của cả 3 cấu trúc dữ liệu gần như tương đương nhau, gần như build time xấp xỉ 0 giây.
- Với tập dữ liệu cỡ vừa (từ 10000 đến 100000), sự chênh lệch giữa thời gian build của **Balanced Kd-Tree** bắt đầu có sự thay đổi, cao hơn so với hai cấu trúc dữ liệu còn lại. **Build Time** giữa **Normal Kd-Tree** và **Hash** gần như tương đương nhau dao động trong khoảng 0 - 1 giây.
- Với tập dữ liệu lớn (từ 100000 trở lên), **Build Time** của **Balanced Kd-Tree** tăng đáng kể, dốc lên trên, cao hơn tuyệt đối so với **Normal Kd-Tree** và **Hash**. Thời gian build của **Normal Kd-Tree** cũng bắt đầu có sự chênh lệch, cao hơn so với **Hash**.
⇒ Với mọi tập dữ liệu, ta có thể kết luận performance xét về **Build Time** thì **Hash > Normal KdTree > Balanced Kd-Tree**

8.2 So sánh hiệu năng search Nearest Neighbor Point



Hình 20: Thời gian tìm kiếm NNP của 100 điểm liên tiếp với các tập dữ liệu có kích thước khác nhau

Nhận xét:

- Nhìn chung, thời gian **SearchNNP Time** của **Normal Kd-Tree** và **Balanced Kd-Tree** tương đối ổn định, dao động trong khoảng 0 - 0.03 giây với mọi tập dữ liệu. Tuy nhiên, **SearchNNP Time** của **Hash** có sự biến đổi đáng kể đối với các tập dữ liệu khác nhau.
- Với tập dữ liệu nhỏ (từ 0 đến 10000): thời gian build của **Hash** có sự thay đổi đáng kể thành nhiều giai đoạn khác nhau. Đối với các tập dữ liệu rất nhỏ (0 đến 750), **Search NNP Time** của **Hash** cao hơn đáng kể so với **Normal Kd-Tree** và **Balanced Kd-Tree**, search 100 điểm mất khoảng 0.35s. Khi các tập dữ liệu tương đối lớn hơn (từ 1000 đến 10000) thời gian Search của **Hash** dao động trong khoảng 0 - 0.05s, vẫn cao hơn nhiều so với **Normal Kd-Tree** và **Balanced Kd-Tree**. Bên cạnh đó, **SearchNNP Time** của **Normal Kd-Tree** và **Balanced Kd-Tree** có sự chênh lệch không đáng kể chỉ dao động từ 0 - 0.02s, trong đó performance của **Balanced Kd-Tree** có phần tốt hơn so với **Normal Kd-Tree**.
⇒ Với tập dữ liệu nhỏ (từ 0 đến 1000), ta có thể kết luận performance xét về **SearchNNP Time** thì **Balanced Kd-Tree** > **Normal KdTree** > **Hash**.
- Với tập dữ liệu cỡ vừa (từ 10000 đến 100000): **SearchNNP Time** của **Hash** có sự thay đổi rõ rệt, giảm mạnh so với các giai đoạn trước đó. **SearchNNP Time** của **Hash** giảm dần theo độ lớn của tập dữ liệu, nghĩa là với tập dữ liệu càng lớn thì performance của **Hash** càng tốt hơn. **SearchNNP Time** của **Balanced Kd-Tree** vẫn giữ tương đối ổn định, tuy nhiên time search của **Normal Kd-Tree** tăng dần trong giai đoạn này, chứng tỏ hoạt động của **Balanced Kd-Tree** ổn định hơn so với **Normal Kd-Tree** khi search nhiều điểm.
⇒ Với tập dữ liệu cỡ vừa (từ 10000 đến 100000), ta có thể kết luận performance xét về **SearchNNP Time** thì **Balanced KdTree** ≥ **Hash** > **Normal KdTree**.

- Với tập dữ liệu lớn (từ 100000 trở lên), SearchNNP Time của Normal Kd-Tree vẫn tiếp tục tăng, do cây chưa được cân bằng nên performance không ổn định với tập dữ liệu lớn và khi search nhiều điểm. Trong khi đó, SearchNNP Time của Balanced Kd-Tree và Hash có sự ổn định kéo dài trong suốt các tập dữ liệu lớn, dao động trong khoảng 0 - 0.01s. nếu xét kĩ hơn về performance thì Hash search hiệu quả hơn một chút so với Kd-Tree trong giai đoạn này.

⇒ Với tập dữ liệu lớn (từ 100000 trở lên), ta có thể kết luận performance xét về SearchNNP Time thì Hash > Balanced Kd-Tree > Normal Kd-Tree hoặc Hash ≥ Balanced Kd-Tree > Normal Kd-Tree

9 Kết luận

9.1 Xét về Build Data Structure

- Với mọi tập dữ liệu, ta có thể kết luận performance xét về Build Data Structure thì Hash > Normal KdTree > Balanced Kd-Tree

9.2 Xét về Search Nearest Neighbor Point

- Với tập dữ liệu nhỏ (từ 0 đến 1000), ta có thể kết luận performance xét về Search Nearest Neighbor Point thì Balanced KdTree > Normal KdTree > Hash.
- Với tập dữ liệu cỡ vừa (từ 10000 đến 100000), ta có thể kết luận performance xét về Search Nearest Neighbor Point thì Balanced KdTree ≥ Hash > Normal KdTree.
- Với tập dữ liệu lớn (từ 100000 trở lên), ta có thể kết luận performance xét về Search Nearest Neighbor Point thì Hash > Balanced Kd-Tree > Normal Kd-Tree hoặc Hash ≥ Balanced Kd-Tree > Normal Kd-Tree.

10 Tài liệu tham khảo

- [1] Martin Skrodzki, *The k-d tree data structure and a proof for neighborhood computation in expected logarithmic time*, March 13, 2019.
- [2] OpenDSA Data Structures and Algorithms Modules Collection, *Chapter 20.3. KD-Trees*, Link: <https://opensa-server.cs.vt.edu/ODSA/Books/Everything/html/KDtree.html>
- [3] Yassen Hu, *KD-Tree*, Link: <https://yassenh.github.io/post/kd-tree/>.
- [4] OpenDSA Data Structures and Algorithms Modules Collection, *Chapter 10.5. Bucket Hashing*, Link: <https://opensa-server.cs.vt.edu/ODSA/Books/CS3/html/BucketHash.html>
- [5] Stable Sort Channel, *KD-Tree Nearest Neighbor Data Structure*, Youtube, Link: <https://youtu.be/Glp7THUpGow>
- [6] Wikipedia, *k-d tree*, Link: https://en.wikipedia.org/wiki/K-d_tree
- [7] Geekforgeeks, *KD Tree*, Link: <https://www.geeksforgeeks.org/k-dimensional-tree/>