Московский авиационный институт (национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Курсовая работа по курсу «Дискретный анализ»

Тема: Поиск ближайших соседей

Студент: Д. Ляшун
Преподаватель: С. Сорокин
Группа: М8О-207Б

Дата:
Оценка:
Подпись:

Москва, 2021

1 Постановка задачи

Дано множество точек в многомерном пространстве. Для каждой точки из файла с запросами вам необходимо вывести номер ближайшей к ней точки из исходного множества в смысле простого евклидова расстояния. Если ближайших точек несколько, то выведите номер любой из них.

Формат входных файлов при построении структуры:

```
./prog build --input <input file>/
--output <stats file>
```

Ключ	Значение
–input	входной файл с точками
–output	выходной файл со структурой для поиска

Формат входного файла:

```
<dimensions>
<x_1,1><x_1,2>... <x_1,dimensions>
<x_2,1><x_2,2>... <x_2,dimensions>
...
```

Формат запуска программы в режиме поиска:

```
./prog search --struct <struct file>/
--input <input file>/
--output <output file>
```

Ключ	Значение
–struct	файл со структурой для поиска, полученной на предыдущем этапе
–input	входной файл с запросами
–output	выходной файл с ответами на запросы

Формат входного файла:

```
<dimensions>
<q_1,1><q_1,2>... <q_1,dimensions>
<q_2,1><q_2,2>... <q_2,dimensions>
...
```

2 Описание

Требуется написать оптимальный алгоритм поиска ближайшего соседа, который мог бы работать быстрее наивного линейного поиска за $O(kn)$, где n – общее число отмеченных точек, k – размерность пространства. Для решения этого можно воспользоваться такой структурой данных как K-d дерево. Как сказано в [1], K-d дерево – это структура данных с разбиением пространства для упорядочивания точек в k-мерном пространстве. K-d деревья используются для некоторых приложений, таких как поиск в многомерном пространстве ключей (поиск диапазона и поиск ближайшего соседа).

K-d деревья – особый вид двоичных деревьев поиска. В каждом узле K-d дерева хранится элемент с k-мерным ключом и указателем на то, по какому значению ключа (подпространству) необходимо начинать сравнивать при проведении поиска, вставки или удаления элемента, т.е. сравнение двух ключей в узле с указателем на подпространство i происходит с замыканием, начиная от i до k-ой координаты и затем от 1 до $i-1$. При этом, если ключ поиска или вставки/удаления больше ключа в текущем узле, то происходит переход в правое поддерево, иначе, если ключ меньше – в левое поддерево соответственно.

При вставке произвольных элементов в K-d дерево, как и в обычное бинарное дерево, не гарантируется, что оно будет сбалансированным, т.е. таким, что при хранении n элементов его высота составляет $O(logn)$. Если K-d Дерево не сбалансировано, то поиск в нём в худшем случае будет выполняться за $O(kn)$, что является неприемлемо медленным. Для решения этой проблемы необходимо воспользоваться алгоритмом построения сбалансированного K-d дерева [2], придуманным американским ученым в области компьютерных наук – Расселом А. Брауном.

Идея данного построения состоит в следующем. Сперва нужно разложить входные n точек в k различных массивов, каждый из которых необходимо отсортировать по возрастанию координат таким образом, что в i массиве по порядку сперва сравнивается значения по i координате, $i+1$ и т. д. до k, а затем от 1 до $i-1$ соответственно. Полученные таким образом массивы будут использоваться далее при рекурсивном построении, которое работает так: на каждом шаге работы будем брать медианный элемент в заданной рабочей области с индексами $[l, r]$ (т.е. индекс медианного элемента будет $m = (l + r)/2$) в отсортированном ранее i-ом массиве (i – текущее подпространство, от которого будет начинаться сравнение ключей). Данный медианный элемент вставляется в K-d дерево. Далее необходимо разбить рабочую область на два подотрезка $[l, m-1]$ и $[m+1, r]$, причём в первом будут находиться все точки, которые меньше медианной, а в другом – больше соответственно. При этом необходимо, чтобы это условие выполнялось на всех k отсортированных массивах, поскольку следующая вставка будет производиться уже в подпространстве $j = (i + 1)$ (если j превышает k – происходит переход к 1 и т.д.) на отрезках $[l, m-1]$ и $[m+1, r]$. Ал-

горитм выполняется, пока не будут вставлены все точки в каждом из порождаемых отрезков. Стоит отметить, что на начальном этапе рабочий отрезок имеет границы $[1, n]$.

Для поиска ближайшего соседа в полученном сбалансированном K-d дереве можно воспользоваться следующим алгоритмом [3]. Сперва необходимо произвести обход в дереве в попытке поиска точки, которая равна входной, это можно сделать обычным спуском по дереву. В случае перехода в листовой узел его хранимая точка запоминается, и далее происходит так называемое распутывание рекурсии, которая возникает при обходе. В ходе него мы поднимаемся вверх по дереву и смотрим, меньше ли высота к гиперплоскости, образуемой точкой в узле и прямыми, параллельными осям координат (кроме оси, от которой начинается сравнение координат в узле), чем текущее найденное расстояние до ближайшего соседа (хранимой точки). Если да, то это значит, что если мы пойдём в другую область, которую разделяет эта гиперплоскость, то есть шанс найти более ближайшего соседа, при этом перед переходом также сравниваются расстояния между хранимой точкой и той, которая находится в текущем узле, и выбирается ближайшая до входной точки для хранения. При переходе в другую область гиперплоскости происходит сперва обычный спуск по дереву с входной точкой, а затем такое же распутывание рекурсии по тем же правилам. Стоит сказать, что если расстояние до гиперплоскости всё же больше расстояния до текущего ближайшего соседа, то тогда необходимо дальше подниматься вверх по дереву и не переходить в другую область.

Оценим время работы полученного алгоритма. Как сказано в [2], время построения K-d дерева составляет $O(k^2 nlogn)$ – сперва выполняется сортировка n точек в k массивах за $O(k^2 nlogn)$, далее выполняется последовательная вставка каждой из n точек в K-d дерево за $O(klogn)$, а также обновление порядка содержимого k массивов с $r - l + 1$ элементами за время $O(k(r - l + 1))$, при этом при заполнении каждой высоты K-d дерева сумма просмотренных отрезков $[l, r]$ будет составлять n – итого $O(knlogn)$.

Что касается времени поиска ближайшего соседа, то утверждается [1], что в среднем он будет выполняться за $O(klogn)$, если координаты точек определяются случайным образом и их число n во много раз больше размерности – $n >> 2k$. Стоит сказать, что современные распространённые идеи решения задачи о поиске ближайших соседей [4] не застрахованы от так называемого «проклятия размерности», когда незначительное изменение размерности пространства увеличивает время работы поиска почти до линейного.

3 Исходный код

Для представления входных точек в программе была описана структура $TPoint$:

```
1 struct TPoint {
2     TPoint(const std::vector<int> &coords);
3     void Output();
4     std::vector<int> Coords;
5 };
```

Для работы с точками $TPoint$ были описаны следующие методы:

kdtree.hpp	
TPoint::Output()	Выводит в стандартный поток вывода значения координат точки.
long long SquareDist(const TPoint& first, const TPoint& second)	Находит квадрат евклидова расстояния между двумя точками.

Для представления узла в K-d дереве была описана структура $TNode$:

```
1 struct TNode {
2     TNode(const TPoint key);
3     ~TNode();
4     TPoint Key;
5     int Demension;
6     TNode* Left;
7     TNode* Right;
8 };
```

Для работы с узлами $TNode$ используются следующие процедуры и функции:

kdtree.hpp	
void Insert(TNode*& root, TNode* key)	Производит вставку узла *key* в K-d дерево с корнем в узле *root*.
TNode* Search(TNode* root, const TPoint& point)	Производит поиск ближайшего соседа для точки *point* в K-d дереве с корнем в узле *root*.
void ConsoleOutput(TNode* root)	Выводит в стандартный поток вывода содержимое K-d дерева с корнем в узле *root*.
TNode* WhichBest(TNode* first, TNode* second, const TPoint& point)	Выбирает среди двух узлов K-d дерева *first* и *second* тот, в котором хранимая точка ближе к входной точке *point*.
void SaveInFile(TNode* root, std::ofstream& out)	Производит запись содержимого K-d дерева с корнем в узле *root* в файл с потоком вывода *out*.

void ReadFromFile(TNode*& root, std::ifstream& in, const int demensions)	Производит чтение информации о K-d дереве из файла с потоком чтения *in* и запись в K-d дерево, хранящее точки размерности *demensions*, с корнем в узле *root*.

Для представления K-d дерева в программе был описан класс $TKDTree$:

```
1  class TKDTree {
2      public:
3          TKDTree(const std::vector<TPoint>& points);
4          TKDTree(std::ifstream& in);
5          ~TKDTree();
6          TPoint Search(const TPoint& point);
7          void ConsoleOutput();
8          void SaveInFile(std::ofstream& out);
9      private:
10         TNode* Root;
11         void BuildRec(const std::vector<TPoint>& points, std::vector<std::vector<int> >
               &indices, const std::vector<std::function<bool(const int&, const int&)> >&
               comparators, const int d, const int l, const int r);
12 };
```

Для работы с K-d деревом используются следующие встроенные процедуры и функции класса $TKDTree$:

kdtree.hpp	
TPoint Search(const TPoint& point)	Производит поиск точки *point* в K-d дереве.
void ConsoleOutput()	Выводит содержимое K-d дерева в стандартный поток вывода.
void SaveInFile(std::ofstream& out)	Записывает содержимое K-d дерева в файл с потоком вывода *out*.

Ниже приведён исходный код main.cpp основной программы:

```
1  #include "kdtree.hpp"
2  #include <fstream>
3  #include <cstdio>
4  #include <iostream>
5  #include <string>
6  int main(int argc, char *argv[]) {
7      std::string inputFileName;
8      std::string outputFileName;
9      std::string structFileName;
10     std::string typeWork;
```

```
    for (int i = 1; i < argc; ++i) {
        std::string check(argv[i]);
        if (i == 1) {
            typeWork = check;
        }
        else if (i % 2 == 0) {
            if (check == "--input") {
                check = std::string(argv[i + 1]);
                inputFileName = check;
            }
            else if (check == "--output") {
                check = std::string(argv[i + 1]);
                outputFileName = check;
            }
            else if (check == "--struct") {
                check = std::string(argv[i + 1]);
                structFileName = check;
            }
            else {
                std::cout << "Error! Wrong program key input!\n";
                return -1;
            }
        }
    }
    if (typeWork != "build" && typeWork != "search") {
        std::cout << "Wrong work key input!\n";
        return -1;
    }
    std::streambuf *cinbuf = std::cin.rdbuf();
    std::streambuf *coutbuf = std::cout.rdbuf();
    if (!inputFileName.empty()) {
        static std::ifstream in(inputFileName);
        std::cin.rdbuf(in.rdbuf());
    }
    int demensions;
    std::cin >> demensions;
    std::vector<NMyStd::TPoint> points;
    std::vector<int> coords(demensions);
    while (std::cin >> coords[0]) {
        for (int i = 1; i < demensions; ++i) {
            std::cin >> coords[i];
        }
        points.push_back(NMyStd::TPoint(coords));
    }
    if (typeWork == "build") {
        NMyStd::TKDTree tree(points);
        if (outputFileName.empty()) {
            tree.ConsoleOutput();
        }
```

```
60          else {
61              std::ofstream out(outputFileName, std::ios::binary);
62              tree.SaveInFile(out);
63              out.close();
64          }
65      }
66      else {
67          if (structFileName.empty()) {
68          std::cout << "Error! No struct file name input!\n";
69          return -1;
70        }
71        if (!outputFileName.empty()) {
72              static std::ofstream out(outputFileName);
73              std::cout.rdbuf(out.rdbuf());
74          }
75        std::ifstream treeStructure(structFileName, std::ios::binary);
76        NMyStd::TKDTree tree(treeStructure);
77        treeStructure.close();
78        for (int i = 0; i < points.size(); ++i) {
79            NMyStd::TPoint result = tree.Search(points[i]);
80            std::cout << "Nearest neighbor for ";
81            points[i].Output();
82            std::cout << " is ";
83            result.Output();
84            std::cout << '\n';
85        }
86        if (!outputFileName.empty()) {
87            std::cout.rdbuf(coutbuf);
88        }
89      }
90      if (!inputFileName.empty()) {
91          std::cin.rdbuf(cinbuf);
92      }
93      return 0;
94  }
```

4 Демонстрация работы

```
dmitry@dmitry-VirtualBox:~/Work_place/DA_labs/KP$ make
g++ -c kdtree.cpp
g++ kdtree.o benchmark.cpp -o benchmark
g++ kdtree.o main.cpp -o prog
dmitry@dmitry-VirtualBox:~/Work_place/DA_labs/KP$ cat testI.txt
3
2 3 3
5 4 2
9 6 7
4 7 9
8 1 5
7 2 6
9 4 1
8 4 2
9 7 8
6 3 1
dmitry@dmitry-VirtualBox:~/Work_place/DA_labs/KP$ ./prog build --input testI.txt
--output search.bin
dmitry@dmitry-VirtualBox:~/Work_place/DA_labs/KP$ cat testS.txt
3
2 5 3
3 7 1
5 4 4
6 3 1
dmitry@dmitry-VirtualBox:~/Work_place/DA_labs/KP$ ./prog search --input testS.txt
--output answer.txt --struct search.bin
dmitry@dmitry-VirtualBox:~/Work_place/DA_labs/KP$ cat answer.txt
Nearest neighbor for (2,5,3) is (2,3,3)
Nearest neighbor for (3,7,1) is (5,4,2)
Nearest neighbor for (5,4,4) is (5,4,2)
Nearest neighbor for (6,3,1) is (6,3,1)
```

5 Тест производительности

Тест производительности представляет из себя сравнение времени работы алгоритма поиска ближайшего соседа с использованием сбалансированного K-d дерева и наивного поиска. В ходе проведения теста производительности также происходила проверка того, что полученный нами алгоритм даёт правильные ответы.

Размерность пространства k	max значение координат по модулю	Число эталонных точек n	Число точек для поиска m	Время работы алгоритма с использованием K-d дерева	Время работы наивного алгоритма
3	100	100	50	3275 мкс.	10258 мкс.
7	100	100	50	19717 мкс.	25341 мкс.
3	1000	1000	500	21971 мкс.	389303 мкс.
7	1000	1000	500	631013 мкс.	727315 мкс.
3	10000	10000	5000	330994 мкс.	39604542 мкс.
7	10000	10000	5000	24658018 мкс.	71341272 мкс.

Как видно, время работы алгоритма поиска ближайшего соседа с использованием сбалансированного K-d дерева в целом быстрее наивного алгоритма поиска, однако в случае увеличения размерности пространства k, особенно если число точек для поиска небольшое, время работы становится больше. Это связано с тем, что при работе на больших размерностях в ходе поиска часто происходит переход в другую область, разделяемую гиперплоскостью, при распутывании рекурсии.

Как сказано в [1], проблема ухудшения времени работы при поиске на пространствах с большой размерностью известна как «проклятие размерности», и решается обычно аппроксимацией т.е. поиском приближённого ближайшего соседа. В случае работы со сбалансированным K-d деревом можно наложить ограничение на число изменений текущего найденного ближайшего соседа в ходе распутывания рекурсии и тогда время работы алгоритма будет оптимальным.

6 Выводы

Выполнив курсовую работу по курсу «Дискретный анализ», я познакомился с идеей использования сбалансированного K-d дерева при решении задачи поиска ближайшего соседа в многомерном пространстве. Основную трудность в ходе выполнения работы для меня составил поиск материала для решения задачи, а также правильная реализация алгоритма. Например, я неправильным образом обрабатывал рабочие массивы при рекурсивном построении K-d дерева – в каждом из них сравнивал элементы только по правилу сравнения для текущего вставляемого ключа. Также у меня возникла проблема в ходе написания алгоритма поиска – при распутывании рекурсии я не рассматривал элемент в узле как возможного ближайшего соседа, отчего программа не всегда давала правильные ответы.

Стоит отметить, что данный алгоритм имеет множество практических применений. Например, с его помощью можно находить ближайшие к нам интересуемые места на карте, или же найти похожую фотографию для входной – для этого можно задать ряд числовых параметров, которые будут как-то характеризовать наши фотографии, и таким образом представить их в виде точек, по которым будет выполняться поиск.

Список литературы

[1] *Wikipedia: k-d tree*
URL: `https://en.wikipedia.org/wiki/K-d_tree` (дата обращения: 3.05.2021).

[2] Russell A. Brown. *Building a Balanced k-d Tree in O(kn log n) Time* — Journal of Computer Graphics Techniques (JCGT), vol. 4, no. 1, 50-68, 2015.

[3] Stable Sort. *KD-Tree Nearest Neighbor Data Structure*
URL: `https://www.youtube.com/watch?v=Glp7THUpGow&list=LL&index=2` (дата обращения: 3.05.2021).

[4] Rajendra Prasad Mahapatra, Partha Sarathi Chakraborty. *Comparative Analysis of Nearest Neighbor Query Processing Techniques* — 3rd International Conference on Recent Trends in Computing 2015 (ICRTC-2015).