ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 4

«Кластеризация массива посредством полного перебора всех комбинаций значений этого массива»

Выполнил работу

Маркелова Софья

Академическая группа С3100

Принято

Ментор, Вершинин Владислав

Санкт-Петербург

2024

Введение:

Цель лабораторной работы – решение задачи по кластеризации массива посредством полного перебора всех комбинаций значений этого массива. В процессе проведения данной работы требуется на практике убедиться в ресурсоемкости сложных алгоритмов. Необходимо оценить полученную сложность алгоритма и сопоставить время исполнения алгоритма с расчетным временем исполнения алгоритма в зависимости от количества элементов в исходном массиве.

Для достижения цели были установлены следующие задачи: разработка класса ProductIterator; вычисление среднего значения (функция mean\_val); оценка кластеров; генерация масок для кластеров; формирование кластеров; поиск оптимального разбиения; вывод результатов

1. Теоретическая подготовка:

Для выполнения работы необходимо обладать теоретической подготовкой в части понимания работы с классами, принципов кластеризации массива, вычисления среднего значения элементов массива, вычисления расстояния элементов массива до среднего значения, фиксации лучшего результата. Также потребуется понимание метода преобразования числа десятичной системы счисления в другую систему (двоичную, троичную, и.т.д). Необходимо знать принцип оценки времени исполнения программы и расчет оценки сложности алгоритма.

1. Реализация

В процессе реализации были использованы стандартная библиотека iostream для вывода результата, chrono для фиксации времени работы, cstdio для предоставления функции для ввода и вывода данных, vector для работы со структурой данных типа вектор.

Создание класса ProductIterator: реализован класс, который генерирует все возможные комбинации индексов для трех кластеров. Класс принимает вектор элементов и количество повторений (в данном случае, количество кластеров); метод next() возвращает следующую комбинацию индексов, а метод is\_done() проверяет, завершено ли генерирование всех комбинаций.

class ProductIterator {

public:

ProductIterator(std::vector<int> elements, int repeat)

: elements(elements), repeat(repeat), indices(repeat, 0) {}

std::vector<int> next() {

if (is\_done()) {

throw std::out\_of\_range("No more combinations");

}

std::vector<int> current\_combination(repeat);

for (int i = 0; i < repeat; ++i) {

current\_combination[i] = elements[indices[i]];

}

increment\_indices();

return current\_combination;

}

bool is\_done() const {

return done;

}

private:

std::vector<int> elements;

int repeat;

std::vector<int> indices;

bool done = false;

void increment\_indices() {

for (int i = repeat - 1; i >= 0; --i) {

if (indices[i] + 1 < elements.size()) {

indices[i]++;

return;

}

indices[i] = 0;

}

done = true;

}

};

Реализация функций для вычисления среднего значения и оценки кластеров: функция mean\_val() вычисляет среднее значение для заданного вектора; функция score\_cluster() вычисляет "скор" для каждого кластера, основываясь на сумме абсолютных отклонений от среднего значения.

double mean\_val(std::vector<int> arr) {

int N = arr.size();

int sum\_ = 0;

for (int i = 0; i < N; i++) {

sum\_ += arr[i];

}

return ((double)sum\_) / N;

}

double score\_cluster(std::vector<int> arr, double cluster\_center) {

int N = arr.size();

double score = 0;

for (int i = 0; i < N; i++) {

score += std::abs(((double)arr[i] - cluster\_center));

}

return score;

}

Вывод результатов: функция print\_clusters() отвечает за вывод информации о полученных кластерах и их оценке.

void print\_clusters(std::vector<int> cluster1, std::vector<int> cluster2, std::vector<int> cluster3, double score) {

std::cout << "Score: " << score << ". Cluster 1:";

for (int i : cluster1) {

std::cout << " " << i;

}

std::cout << ". Cluster 2:";

for (int i : cluster2) {

std::cout << " " << i;

}

std::cout << ". Cluster 3:";

for (int i : cluster3) {

std::cout << " " << i;

}

std::cout << std::endl;

}

Основная часть программы: в функции main() определен входной массив, запущен таймер для измерения времени выполнения, инициализирован объект класса ProductIterator для генерации масок; инициализированы глобальные переменные для хранения лучшего результата (скора) и кластеров; используется генератор mask\_iter для генерации всех возможных распределений элементов по трем кластерам, цикл продолжается до тех пор, пока не будут исчерпаны все комбинации масок; создание локальных кластеров; проверка, что все три кластера имеют хотя бы один элемент ; подсчёт скора как суммы абсолютных отклонений от среднего значения, если текущий скор меньше лучшего найденного, обновляются переменные, хранящие лучший скор и соответствующие кластеры; измерение времени выполнения; вывод информация о лучшем найденном результате и времени выполнения.

int main() {

int arr[] = {0, 1, 3, 5, 6, 8, 9};

int N = sizeof(arr) / sizeof(arr[0]);

auto start = std::chrono::high\_resolution\_clock::now();

ProductIterator masks\_iter({0, 1, 2}, N);

double best\_score = std::numeric\_limits<double>::infinity();

std::vector<int> best\_cluster1;

std::vector<int> best\_cluster2;

std::vector<int> best\_cluster3;

while (!masks\_iter.is\_done()) {

std::vector<int> mask = masks\_iter.next();

std::vector<int> cluster1;

std::vector<int> cluster2;

std::vector<int> cluster3;

for (int i = 0; i < N; i++) {

if (mask[i] == 0) {

cluster1.push\_back(arr[i]);

} else if (mask[i] == 1) {

cluster2.push\_back(arr[i]);

} else if (mask[i] == 2) {

cluster3.push\_back(arr[i]);

}

}

if (cluster1.size() > 0 && cluster2.size() > 0 && cluster3.size() > 0) {

double cluster1\_center = mean\_val(cluster1);

double cluster2\_center = mean\_val(cluster2);

double cluster3\_center = mean\_val(cluster3);

double score1 = score\_cluster(cluster1, cluster1\_center);

double score2 = score\_cluster(cluster2, cluster2\_center);

double score3 = score\_cluster(cluster3, cluster3\_center);

double score = score1 + score2 + score3;

if (score < best\_score) {

best\_score = score;

best\_cluster1 = cluster1;

best\_cluster2 = cluster2;

best\_cluster3 = cluster3;

print\_clusters(cluster1, cluster2, cluster3, score);

}

auto end = std::chrono::high\_resolution\_clock::now();

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start);

std::cout << "Running time: " << duration.count() << " milliseconds" << std::endl;

print\_clusters(best\_cluster1, best\_cluster2, best\_cluster3, best\_score);

1. Экспериментальная часть:

В рамках реализации алгоритма были задействованы следующие структуры данных: исходный массив значений типа double (8\*N байт).

Два вектора для хранения лучших кластеров (3 \* (24 + 8 \* N/3)) байт,

Вектор текущей маски распределения массива по кластерам (24 + 4 \* N байта),

Три вектора текущих кластеров (3 \* (24 + 8 \* N/3)) байт.

Все остальные потребители памяти – переменные типа int и double, используемые в функциях подсчета среднего значения, расстояния и преобразования числа в маску.

В основном цикле происходит итерация по N^3 значениям, в теле цикла происходит несколько последовательных циклов перебора элементов массива размерности N. Итоговая сложность алгоритма O(3^N) \* O (N).

График зависимости времени от числа элементов.

Согласно требованиям моего варианта, на вход к моему алгоритму подаётся до 25 элементов. Теоретически заданная сложность задачи составляет O(3^N) и более. Для тестирования алгоритма была собрана статистика, приведенная в таблице №1.

Таблица №1 - Подсчёт сложности реализованного алгоритма

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер входного набора | 3 | 5 | 10 | 15 | 20 | 25 |
| время выполнения программы, с | 0,0001 | 0,0003 | 0,0075 | 21,384 | 7000 | 1945702 |
| O(3^N)\*O(3N) | 0 | 0 | 0,6 | 215 | 69735 | 3956781 |
| O(3^N)\*O(N) | 0 | 0 | 0,004 | 10 | 4000 | 1478920 |

График представляющий визуально удобный формат данных из таблицы №1 представлен на изображении №1.

Рисунок №1 - График работы алгоритма

Время исполнения реализованного алгоритма находится между расчетными значениями для O(3^N)\*O(N) и O(3^N)\*O(3N) и совпадает с ними по профилю, что отражает сложность O(3^N) \* O(N)

Заключение

В ходе выполнения работы мною был реализован алгоритм кластеризации массива посредством полного перебора всех комбинаций значений этого массива. Цель работы была достигнута путём тестирования на массивах с различным числом и комбинациями входных значений. Полученные результаты также совпадают с теоретическими оценками сложности алгоритма. В целях работы не стояло задачи по оптимизации алгоритма. Однако, чтобы минимизировать затраты по времени и памяти, вместо того, чтобы создавать векторы для кластеров и заполнять их при каждой итерации, можно сначала подсчитать, сколько элементов будет в каждом кластере, а затем выделить память для кластеров заранее. Это позволит избежать многократного выделения памяти и улучшит производительность.

ПРИЛОЖЕНИЕ А

Листинг кода файла quick\_sort.cpp

#include <iostream>

#include <cstdio>

#include <vector>

#include <chrono>

// Класс, который будет использоваться для генерации всех возможных комбинаций элементов из массива

class ProductIterator {

public:

ProductIterator(std::vector<int> elements, int repeat)

: elements(elements), repeat(repeat), indices(repeat, 0) {}

std::vector<int> next() {

// Метод, который возвращает следующую комбинацию индексов

if (is\_done()) {

// Кинуть ошибку, если пользователь попытается вызвать next в случае,

// когда все комбинации сгенерированны

throw std::out\_of\_range("No more combinations");

}

// Сгенерировать текущую комбинацию индексов

std::vector<int> current\_combination(repeat); // O(N)

for (int i = 0; i < repeat; ++i) { // O(N)

current\_combination[i] = elements[indices[i]];

}

// Инкрементировать следующий индекс, внутри функции описание

increment\_indices(); // O(N)

return current\_combination;

}

bool is\_done() const {

return done;

}

private:

// Элементы на вход, [0, 1, 2]

std::vector<int> elements;

// Количество повторений, 3^(repeat)

int repeat;

// Текущее состояние для индексов, его на каждой итерации меняем и по нему ориентируемся, что вернуть пользователю

std::vector<int> indices;

bool done = false; // флаг завершения генерации комбинаций

// Метод, который увеличивает индексы для генерации след. комбинации

void increment\_indices() {

for (int i = repeat - 1; i >= 0; --i) { // O(N / 2) = O(N)

if (indices[i] + 1 < elements.size()) {

indices[i]++; // Тут инкрементируем и больше не идем

return;

}

indices[i] = 0; // Вот здесь вот зануляем

}

// Если не нашлось того значения, которое надо инкрементировать, считаем, что все комбинации сгенерированны

done = true;

}

};

double mean\_val(std::vector<int> arr) {

// Подсчет центра кластера (среднего значения)

int N = arr.size();

int sum\_ = 0;

for (int i = 0; i < N; i++) { // O(N)

sum\_ = sum\_ + arr[i];

}

return ((double) sum\_) / N;

}

double score\_cluster(std::vector<int> arr, double cluster\_center) {

// Подчет скора - сумма разностей каждого элемента и центра кластера (среднего значения)

int N = arr.size();

double score = 0;

for (int i = 0; i < N; i++) { // O(N)

score += std::abs(((double)arr[i] - cluster\_center));

}

return score;

}

void print\_clusters(std::vector<int> cluster1, std::vector<int> cluster2, std::vector<int> cluster3, double score) {

// Вывод скора и соответствующего кластера

std::cout << "Score: " << score << ". Cluster 1:";

for (int i = 0; i < cluster1.size(); i++) {

std::cout << " " << cluster1[i];

}

std::cout << ". Cluster 2:";

for (int i = 0; i < cluster2.size(); i++) {

std::cout << " " << cluster2[i];

}

std::cout << ". Cluster 3:";

for (int i = 0; i < cluster3.size(); i++) {

std::cout << " " << cluster3[i];

}

std::cout << std::endl;

}

int main()

{

// Данные на вход - массив, который нужно разбить на кластера

int arr[] = {0, 1, 3, 6, 7, 11, 40, 60, 3, 54, 100, 89, 34, 30, 1};

int N = sizeof(arr) / sizeof(arr[0]);

//Записываем текущее время работы

auto start = std::chrono::high\_resolution\_clock::now();

// Генерация масок размерностью (3^N, N), чтобы учесть вме комбинации элементов

ProductIterator masks\_iter({0, 1, 2}, N);

// Создание глобальных переменных для хранения лучшего скора и кластеров, изначальный "лучший скор" - очень большое число

double best\_score = std::numeric\_limits<double>::infinity();

std::vector<int> best\_cluster1;

std::vector<int> best\_cluster2;

std::vector<int> best\_cluster3;

Sonya Markelova, [25.11.2024 16:14]

// И теперь идем по всем комбинациям масок

while (!masks\_iter.is\_done()) { // O(3^N)

// Получаем следующую маску

std::vector<int> mask = masks\_iter.next(); // O(3\*N) = O(N)

// Создание локальных переменных для хранения кластеров

std::vector<int> cluster1;

std::vector<int> cluster2;

std::vector<int> cluster3;

// Переспределение значений в кластера согласно новой маски

for (int i = 0; i < N; i++) { // O(N)

if (mask[i] == 0) {

cluster1.push\_back(arr[i]); // O(N / 3) = O(N)

} else if (mask[i] == 1) {

cluster2.push\_back(arr[i]); // O(N / 3) = O(N)

}

else if (mask[i] == 2) {

cluster3.push\_back(arr[i]);

}

}

if (cluster1.size() > 0 && cluster2.size() > 0 && cluster3.size() > 0) {

// Подсчёт центров кластеров

double cluster1\_center = mean\_val(cluster1); // O(N)

double cluster2\_center = mean\_val(cluster2); // O(N)

double cluster3\_center = mean\_val(cluster3);

// Подсчёт скора для каждого кластера, как сумма разностей элементов и центра

double score1 = score\_cluster(cluster1, cluster1\_center); // O(N)

double score2 = score\_cluster(cluster2, cluster2\_center); // O(N)

double score3 = score\_cluster(cluster3, cluster3\_center);

double score = score1 + score2 + score3;

// Чем меньше скор, тем лучше получились кластера

if (score < best\_score) {

// Сохраняем лучший скор

best\_score = score;

// Сохраняем новые полученные кластера

best\_cluster1 = cluster1;

best\_cluster2 = cluster2;

best\_cluster3 = cluster3;

// Отладочная информация

print\_clusters(cluster1, cluster2, cluster3, score); // O(N)

}

}

}

// Текущее время окончания выполнения работы

auto end = std::chrono::high\_resolution\_clock::now();

// Продолжительность выполнения работы

auto duration = std::chrono::duration\_cast<std::chrono::milliseconds>(end - start);

std::cout << std::endl;

std::cout << "Running time: " << duration.count() << "milliseconds" << std::endl;

print\_clusters(best\_cluster1, best\_cluster2, best\_cluster3, best\_score);

return 0;

}