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:

Зырянов Виталий

Академическая группа:

J3111

Принято:

Ментор, Вершинин Владислав Константинович

Санкт-Петербург

2024

1. Введение

Цель работы:

Написать программу, которая будет разбивать массив на 5 кластеров, таким образом, что расстояние между элементами подмассива и средним значением было минимально возможным. Метрика для задачи равна сумме модулей разности элемента подмассива и средним значением этого подмассива. Элементы массива являются вещественными числами.

Задачи:

* Решить задачу с помощью полного перебора всех комбинаций кластеров.
* Сделать анализ затрачиваемой памяти и времени работы алгоритма.
* Определить сложность алгоритма.

1. Теоретическая подготовка

Используемые типы данных:

* int - целое число, используемый в работе для индексов элементов массивов.
* double - вещественное число с плавающей точки, для работы с элементами массива.
* vector - контейнер, который хранит элементы массива и подмассива.

Используемые функции:

* any\_of - функций из библиотеки algorithm, который позволяет проверить, удовлетворяет ли хотя бы один элемент диапазона заданному условию.
* random\_device -  это компонент стандартной библиотеки C++, который используется для получения случайных чисел.
* chrono::high\_resolution\_clock - функция библиотеки chrono, используемая для получения текущего времени.

Используемые алгоритмы:

Рекурсивный перебор всех возможных вариантов кластеризации массива на 5 кластера.

1. Реализация

По условию задачи, решение должно перебирать все возможные  разбиение на кластеры. Поэтому решение было реализовано рекурсией, которая принимает разбиение на кластеры до определённого элемента и смотрит следующие разбиения, с новым элементом. Как только все значения массива будут использованы в кластеризация, мы ищем расстояние для этого разбиения и если расстояние меньше текущего оптимально, обновляем оптимальный способ кластеризации.

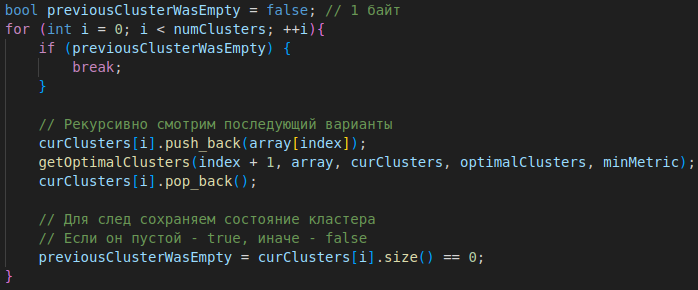


Рисунок. 1 - Алгоритм, который перебирает разбиения на кластеры.

На рисунке 1 изображён рекурсивный перебор всех разбиений массива на заданное число кластеров, в данной лабораторной работе рассматриваем разбиение на 5 кластеров. На каждой итерации цикла мы добавляем элемент в один из кластеров и вызываем рекурсивно функцию, чтобы узнать все последующие варианты разбиения на кластеры. Для того, чтобы никакая комбинация не проверялась более одного раза, я делаю проверку для следующих способов, при которых мы кладём текущий элемент в другой кластер. Программа проверяет был ли предыдущий кластер пустым, если да, то нам не имеет смотреть все следующие разбиения, так как от перемены мест кластеров, само разбиение не изменяется. Таким образом, можно перебрать все возможные уникальные комбинации.

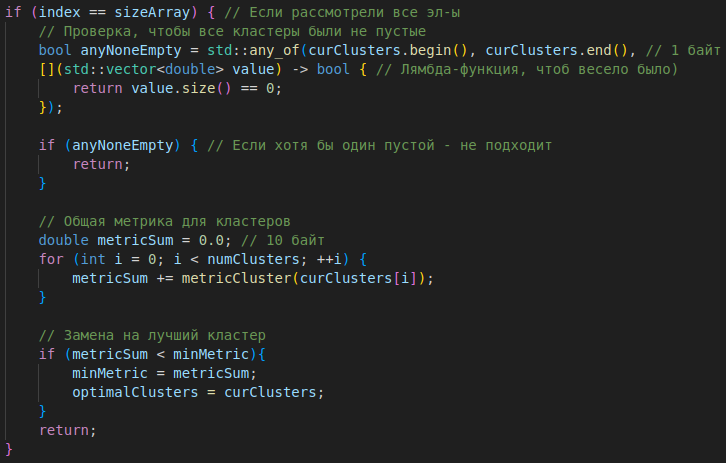


Рисунок 2 – Алгоритм для определения итогового разбиения.

На рисунке 2 изображён код, определяющий корректность разбиения. С помощью функции any\_of и lambda выражения мы проверяем все ли кластеры непустые. Если есть хотя бы один пустой, то разбиение считается некорректным, и мы не учитываем его. В ином случае мы определяем метрику для разбиения и перезаписываем оптимальное разбиение в случае лучшей метрики.

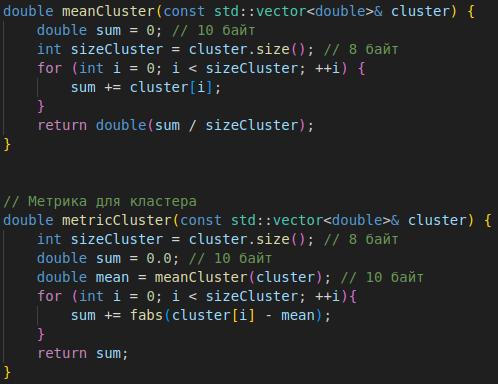


Рисунок 3 - Вычисление расстояние по заданной метрике.

Для определения расстояние для разбиения мы суммируем расстояние по заданной метрики для каждого кластера, которые в итоге получились. Метрика для кластеры вычисляется как сумма модулей разности элемента лежащего в кластере и среднего значения в кластере. Алгоритм показан на рисунке 3.

1. Экспериментальная часть

На данном этапе работы проводилась оценка затрачиваемой памяти, асимптотики и времени программы. Замер времени производился с помощью библиотеки chrono. На основе времени был построен график зависимости размера входного массива и времени выполнения.

Массив, которые мы разбиваем на 5 кластеров. Итоговый размер будет 8 \* N байт, где N - количество элементов массива. То есть в худшем случаем 8 \* 25 = 200 байт.

Кластеры представлены в виде двухмерного массива, у которого 1 мерность равна 5. В таком случае размер будет 8 \* 5 \* N, в худшем случае - 1000 байт.

В ходе кода будут объявлены и другие, промежуточные переменные, которые в конечном итоге будут суммарно занимать 56 байтов.

Размер памяти для одного разбиения: 200 + 1000 + 56 = 1256 байт

Всего разбиений в худшем случае может быть число сочетаний из 25 по 5. Таким образом, мы получаем 53130 комбинаций.

Полная оценка: 1256 \* 53130 = 66731280 байт = 54 Мбайт.

Код состоит из рекурсивного перебора всех разбиений. В случае разбиения на 5 кластеров, в худшем случае у нас функции вызывает себя 5 раз, но в среднем, так как мы не рассматриваем случаи, при которых кластеры меняются местами, этот алгоритм показан на рисунке 1. Поэтому асимптотика примерно равна O(4.5^N).

Для тестирования алгоритма была собрана статистика, приведенная в таблице 1.

Таблица 1 - Подсчёт сложности реализованного алгоритма.

|  |  |  |
| --- | --- | --- |
| Размер входного массива, N: | Время выполнения, с: | O(4.5^N), c: |
| 5 | 0.000306826 | 0.0003125 |
| 7 | 0.00189309 | 0.003736694531 |
| 9 | 0.0412644 | 0.07566806426 |
| 11 | 1.00328 | 1.532278301 |
| 13 | 24.1099 | 31.0286356 |
| 15 | 624.59 | 628.3298709 |

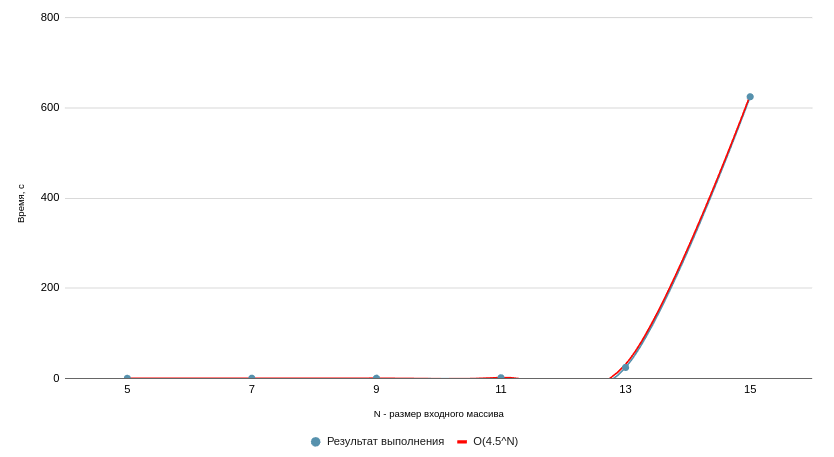


Рис. 4 - График работы алгоритма.

Анализируя графики на Рис. 4 можно увидеть, что сложность алгоритма была определена корректно. А также практикой доказывается наша теоретическая асимптотика и наоборот, равна O(4.5^N).

В этом разделе вам необходимо привести результаты работы вашего алгоритма, с таблицами и графиками, демонстрирующими выполнения алгоритма с различными условиями и наборами данных. Оценивается производительность и сравниваются результаты с теоретическими оценками.

1. Заключение

В ходе выполнения лабораторной работы мною был реализован алгоритм перебора разбиения с самым маленьким расстоянием для заданной метрики. Также была проведен анализ затрачиваемой памяти и сложности, которая практически подтвердилась. На пример этой лабораторной можно понять, что при высокой сложности алгоритмы трудно тестировать, так как это занимают большое количество времени.

ПРИЛОЖЕНИЕ А

Листинг кода файла lab4.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <chrono>

#include <random>

// Средние значение элементов кластера

double meanCluster(const std::vector<double>& cluster) {

double sum = 0; // 10 байт

int sizeCluster = cluster.size(); // 8 байт

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

sum += cluster[i];

}

return double(sum / sizeCluster);

}

// Метрика для кластера

double metricCluster(const std::vector<double>& cluster) {

int sizeCluster = cluster.size(); // 8 байт

double sum = 0.0; // 10 байт

double mean = meanCluster(cluster); // 10 байт

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

sum += fabs(cluster[i] - mean);

}

return sum;

}

// Получение оптимального набора кластеров из массива по заданной метрике

// index - элемент, который мы рассматриваем

// array - заданный массив

// curClusters - кластер, который получился на данной итерации

// optimalClusters - ответ, самый оптимальный кластер на данной итерации

// minMetric - метрика для оптимального кластера на данный момент

void getOptimalClusters(int index, const std::vector<double>& array, std::vector<std::vector<double>> curClusters,

std::vector<std::vector<double>>& optimalClusters, double& minMetric) {

int sizeArray = array.size(); // Кол-во эл-ов массива 4 байта

int numClusters = curClusters.size(); // Кол-во кластеров 4 байта

if (index == sizeArray) { // Если рассмотрели все эл-ы

// Проверка, чтобы все кластеры были не пустые

bool anyNoneEmpty = std::any\_of(curClusters.begin(), curClusters.end(), // 1 байт

[](std::vector<double> value) -> bool { // Лямбда-функция, чтоб весело было)

return value.size() == 0;

});

if (anyNoneEmpty) { // Если хотя бы один пустой - не подходит

return;

}

// Общая метрика для кластеров

double metricSum = 0.0; // 10 байт

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

metricSum += metricCluster(curClusters[i]);

}

// Замена на лучший кластер

if (metricSum < minMetric){

minMetric = metricSum;

optimalClusters = curClusters;

}

return;

}

// Проверка на пустоту предыд кластера

// Если он пустой - не имеет смысла

// заполнять след, так как порядок

// не имеет смысла

bool previousClusterWasEmpty = false; // 1 байт

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

if (previousClusterWasEmpty) {

break;

}

// Рекурсивно смотрим последующий варианты

curClusters[i].push\_back(array[index]);

getOptimalClusters(index + 1, array, curClusters, optimalClusters, minMetric);

curClusters[i].pop\_back();

// Для след сохраняем состояние кластера

// Если он пустой - true, иначе - false

previousClusterWasEmpty = curClusters[i].size() == 0;

}

}

void test(std::vector<double> array, int sizeArray, int numClusters)

{

// array - 8 \* N байт, в худшем 200 байт

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

// Вывод массива

std::cout << "Размер массива: " << sizeArray << "\n";

for (double i : array) std::cout << i << ", ";

// Разбиения на кластеры

std::vector<std::vector<double>> clusters(numClusters, std::vector<double>()); // 8 \* 5 \* N = 40 \* N байт

// Метрика для кластеров

double metricClusters = std::numeric\_limits<double>::max(); // 10 байт

getOptimalClusters(0, array, clusters, clusters, metricClusters); // O(4.5^N) т. к. в среднем примерно 4.5 вызова функции рекурсивно

// Вывод разбиения на кластеры

std::cout << "\n| ";

for (std::vector<double> i : clusters){

for (double j : i){

std::cout << j << " ";

}

std::cout << "| ";

}

// Вывод метрики

std::cout << "\nМетрика: " << metricClusters;

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

std::chrono::duration<double> duration = end - start;

std::cout << "\nВремя выполнения программы: " << duration.count() << " секунд" << "\n\n";

}

// Генерация случайного массива длины size

std::vector<double> randArray(const int& size) {

std::vector<double> arr(size);

std::random\_device rd;

std::mt19937 seed(rd());

for (int i = 0; i < size; ++i) arr[i] = seed() % 999999;

return arr;

}

int main() {

std::vector<double> arr {1.1, 1.2, 1.3, 1.9, 2.0, 2.5, 2.9, 3.1, 4.2, 4.4, 5.2, 6.4};

int size = arr.size();

test(arr, size, 5); // 5 - кол-во кластеров

return 0;

}