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»

Выполнила работу

Луценко Елена

Академическая группа J3112

Принято

Дунаев Максим Владимирович

Санкт-Петербург

2024

**Структура отчёта:**

1. Введение

Цель работы: кластеризовать массив посредством полного перебора всех комбинаций значений этого массива.

Задачи:

1. Реализовать алгоритм полного перебора для нахождения всех возможных разбиений массива на заданное количество кластеров.
2. Оценить работу алгоритма на различных наборах данных
3. Проанализировать результаты работы алгоритма, включая время выполнения и использование памяти.
4. Теоретическая подготовка
5. Кластеризация:

Метод анализа данных для группировки объектов по их характеристикам и схожести.

Основная цель — минимизация расстояний между объектами внутри одного кластера и максимизация различий между кластерами.

1. Класс:

Объединяет данные (поля, атрибуты) и функции (методы), работающие с этими данными, в одну структуру

Используемые уровни доступа:

* **public** — члены класса доступны извне (для других частей программы).
* **private** — члены класса доступны только внутри самого класса.

1. Функция:

Блок кода, который выполняет определенную задачу, принимает входные параметры (если они есть), выполняет вычисления и, возможно, возвращает результат; функции помогают организовать код, повышая его читаемость и повторное использование.

1. Векторы:

Использование std::vector для динамического хранения элементов массива, текущих масок, кластеров, центров кластеров, скоров - суммы разностей каждого элемента и центров кластеров (средних значений), «лучших» кластеров.

1. Типы данных:

double: применяется для представления дробных значений элементов;

int: применяется для представления целых значений элементов, используется для индексов массивов и счётчиков элементов;

bool: применяется для прекращения генерации всех возможных разбиений массива.

1. Алгоритм:

Полный перебор разбиений массива для создания ровно k кластеров.

1. Реализация
2. Класс ProductIterator:

class ProductIterator {  
public:  
 ProductIterator(vector<int> elements, int repeat) : indices(repeat, 0) { // + n \* 4 байт  
 this->elements = elements;  
 this->repeat = repeat;  
 }  
  
 bool is\_done() {  
 return done;  
 }  
  
 vector<int> next() {  
 if (is\_done()) {  
 throw out\_of\_range("No more combinations");  
 }  
 vector<int> current\_combination(repeat); // + n \* 4 байт  
 for (int i = 0; i < repeat; i++) { // O(repeat)  
 current\_combination[i] = elements[indices[i]];  
 }  
 increment\_indices();  
 return current\_combination;  
 }  
  
private:  
 vector<int> elements;  
 int repeat;  
 vector<int> indices;  
 bool done = false; // + 1 байт  
  
 void increment\_indices() {  
 for (int i = repeat - 1; i >= 0; i--) { // O(repeat)  
 if (indices[i] < elements.size() - 1) {  
 indices[i]++;  
 return;  
 }  
 indices[i] = 0;  
 }  
 done = true;  
 }  
};

Используется для получения масок массивов (комбинаций индексов с возможными значениями от 0 до k – 1 включительно, где k – требуемое количество кластеров)

1. Функция mean\_val:

double mean\_val(vector<double> arr) {  
 int n = arr.size(); // + 4 байт  
 double sum = 0; // + 8 байт  
 for (int i = 0; i < n; i++) { // O(n)  
 sum += arr[i];  
 }  
 return sum / n;  
}

Используется для подсчёта центра кластера (среднего значения)

1. Функция score\_cluster:

double score\_cluster(vector<double> arr, double cluster\_center) {  
 int n = arr.size(); // + 4 байт  
 double score = 0; // + 8 байт  
 for (int i = 0; i < n; i++) { // O(n)  
 score += abs(arr[i] - cluster\_center);  
 }  
 return score;  
}

Используется для подсчета скора - суммы разностей каждого элемента и центра кластера (среднего значения)

1. Функция clustering:

vector<vector<double>> clustering(vector<double> numbers) {  
 int n = numbers.size(); // + 4 байт  
 int count = 5; // + 4 байт  
 ProductIterator masks\_iter(**{**0, 1, 2, 3, 4**}**, n); // + 4 + 4 \* count  
 double best\_score = numeric\_limits<double>::infinity(); // + 8 байт  
 vector<vector<double>> best\_cluster(count);  
 while (!masks\_iter.is\_done()) { // O(5^n)  
 vector<int> mask = masks\_iter.next(); // + n \* 4 байт // O(n)  
 vector<vector<double>> cluster(count);  
 for (int i = 0; i < n; i++) { // O(n^2)  
 cluster[mask[i]].push\_back(numbers[i]); // + n \* 8 байт  
 }  
 vector<double> cluster\_center(count);  
 vector<double> score(count);  
 for (int i = 0; i < count; i++) { // O(n\*count)  
 if (cluster[i].size() > 0) {  
 cluster\_center[i] = mean\_val(cluster[i]); // + 8 \* k, где k - кол-во ненулевых кластеров  
 score[i] = score\_cluster(cluster[i], cluster\_center[i]); // + 8 \* k, где k - кол-во ненулевых кластеров  
 } else {  
 score[i] = numeric\_limits<double>::infinity(); // + 8 \* (count - k) байт  
 }  
 }  
 double score\_end = 0; // + 8 байт  
 for (int i = 0; i < count; i++) { // O(count)  
 score\_end += score[i];  
 }  
 if (score\_end < best\_score) {  
 best\_score = score\_end;  
 for (int j = 0; j < count; j++) { // O(count)  
 best\_cluster[j] = cluster[j]; // + n \* 8 байт  
 }  
 }  
 }  
 return best\_cluster;  
};

Используется для поиска лучшего разбиения массива на 5 кластеров с использованием получаемых масок, минимизируя общую стоимость кластеров, определяемую суммой абсолютных отклонений элементов от средних значений кластера.

Используемые библиотеки:

<iostream>: для ввода и вывода.

<vector>: для использования динамических массивов (векторов).

<limits>: для получения значения положительной бесконечности.

<cassert>: для выполнения утверждений (проверки условий на истину).

Тестирование: в функции main реализовано тестирование алгоритма на нескольких наборах данных с фиксированным количеством кластеров (количество кластеров – 5 штук). Выводятся результаты кластеризации.

int main() {  
 vector<double> numbers1 = **{**-9, 78, 76, 14, 38**}**; // + 8 \* 5 байт  
 vector<vector<double>> res1 = clustering(numbers1); // + 8 \* n байт  
 vector<vector<double>> test1 = **{**{-9},  
 {78},  
 {76},  
 {14},  
 {38}**}**; // + 8 \* 5 байт  
 assert(res1 == test1);  
 vector<double> numbers2 = **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10**}**; // + 8 \* 10 байт  
 vector<vector<double>> res2 = clustering(numbers2); // + 8 \* n байт  
 vector<vector<double>> test2 = **{**{1, 2, 3},  
 {4, 5, 6},  
 {7, 8},  
 {9},  
 {10}**}**; // + 8 \* 10 байт  
 assert(res2 == test2);  
 vector<double> numbers3 = **{**-9, 78, 76, 14, 38, 4, 15, 100, 46, 0**}**; // 8 \* 10 байт  
 vector<vector<double>> res3 = clustering(numbers3); // + 8 \* n байт  
 vector<vector<double>> test3 = **{**{-9, 4, 0},  
 {78, 76},  
 {14, 15},  
 {38, 46},  
 {100}**}**; // + 8 \* 10 байт  
 assert(res3 == test3);  
 vector<double> numbers4 = **{**-9, 7.5, 6.6, 14, 38, 17, 15, 100, -46.2, 0**}**; // 8 \* 10 байт  
 vector<vector<double>> res4 = clustering(numbers4); // + 8 \* n байт  
 vector<vector<double>> test4 = **{**{-9, 7.5, 6.6, 0},  
 {14, 17, 15},  
 {38},  
 {100},  
 {-46.2}**}**; // + 8 \* 10 байт  
 assert(res4 == test4);  
 return 0;

1. Экспериментальная часть

Временная сложность: O (5^N \* N^2), где N — количество входных элементов. Это связано с экспоненциальным числом возможных масок (разбиений), которые необходимо проверить.

Потребление памяти: 473 + 60 \* N байт на выходе

Согласно требованиям моего варианта, на вход к моему алгоритму подаётся до 25 элементов. Теоретически заданная сложность задачи составляет O (5^N \* N^2). Для тестирования алгоритма была собрана статистика, приведенная в таблице №1.

Таблица №1 - Подсчёт времени реализации алгоритма

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер входного набора | 5 | 6 | 7 | 8 | 9 | 10 |
| Время выполнения программы, с | 0.0078 | 0.068 | 0.459 | 2.617 | 16.09 | 100.2 |

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

Изображение выглядит как линия, диаграмма, График

Автоматически созданное описание

Изображение №1 - График работы алгоритма

Анализ таблицы: в ходе тестирования алгоритма кластеризации была построена зависимость времени выполнения от количества элементов в массиве. Результаты показали, что время выполнения алгоритма экспоненциально увеличивается с ростом числа элементов.

С учётом полученных данных, можно наблюдать, что:

Для небольших массивов (до 9 элементов включительно) время выполнения остаётся приемлемым и укладывается в разумные рамки.

С увеличением объема массива (начиная с 10 элементов) наблюдается резкое и значительное увеличение времени выполнения.

1. Заключение

В ходе работы был реализован алгоритм кластеризации, используя полный перебор всех возможных разбиений на заданное количество кластеров. Цель работы была достигнута путём тестирования на массивах с различным набором данных. Тестирование на различных наборах данных показало, что время выполнения алгоритма существенно возрастает с увеличением числа элементов (начиная с 10), из чего можно сделать вывод, что алгоритм не подойдёт для работы с массивами, превышающими 9 элементов.

Результаты тестов подтвердили теоретические оценки временной сложности алгоритма, которая равна O (5^N \* N^2).

В качестве дальнейших исследований можно предложить оптимизацию алгоритма с точки зрения уменьшения сложности алгоритма для возможности использования для большего числа элементов.

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла lab\_4.cpp (без функций подсчёта времени)

#include <iostream>  
#include <vector>  
#include <limits>  
#include <cassert>  
  
using namespace std;  
  
class ProductIterator {  
public:  
 ProductIterator(vector<int> elements, int repeat) : indices(repeat, 0) { // + n \* 4 байт  
 this->elements = elements;  
 this->repeat = repeat;  
 }  
  
 bool is\_done() {  
 return done;  
 }  
  
 vector<int> next() {  
 if (is\_done()) {  
 throw out\_of\_range("No more combinations");  
 }  
 vector<int> current\_combination(repeat); // + n \* 4 байт  
 for (int i = 0; i < repeat; i++) { // O(repeat)  
 current\_combination[i] = elements[indices[i]];  
 }  
 increment\_indices();  
 return current\_combination;  
 }  
  
private:  
 vector<int> elements;  
 int repeat;  
 vector<int> indices;  
 bool done = false; // + 1 байт  
  
 void increment\_indices() {  
 for (int i = repeat - 1; i >= 0; i--) { // O(repeat)  
 if (indices[i] < elements.size() - 1) {  
 indices[i]++;  
 return;  
 }  
 indices[i] = 0;  
 }  
 done = true;  
 }  
};  
  
  
double mean\_val(vector<double> arr) {  
 int n = arr.size(); // + 4 байт  
 double sum = 0; // + 8 байт  
 for (int i = 0; i < n; i++) { // O(n)  
 sum += arr[i];  
 }  
 return sum / n;  
}  
  
  
double score\_cluster(vector<double> arr, double cluster\_center) {  
 int n = arr.size(); // + 4 байт  
 double score = 0; // + 8 байт  
 for (int i = 0; i < n; i++) { // O(n)  
 score += abs(arr[i] - cluster\_center);  
 }  
 return score;  
}  
  
  
vector<vector<double>> clustering(vector<double> numbers) {  
 int n = numbers.size(); // + 4 байт  
 int count = 5; // + 4 байт  
 ProductIterator masks\_iter(**{**0, 1, 2, 3, 4**}**, n); // + 4 + 4 \* count  
 double best\_score = numeric\_limits<double>::infinity(); // + 8 байт  
 vector<vector<double>> best\_cluster(count);  
 while (!masks\_iter.is\_done()) { // O(5^n)  
 vector<int> mask = masks\_iter.next(); // + n \* 4 байт // O(n)  
 vector<vector<double>> cluster(count);  
 for (int i = 0; i < n; i++) { // O(n^2)  
 cluster[mask[i]].push\_back(numbers[i]); // + n \* 8 байт  
 }  
 vector<double> cluster\_center(count);  
 vector<double> score(count);  
 for (int i = 0; i < count; i++) { // O(n\*count)  
 if (cluster[i].size() > 0) {  
 cluster\_center[i] = mean\_val(cluster[i]); // + 8 \* k, где k - кол-во ненулевых кластеров  
 score[i] = score\_cluster(cluster[i], cluster\_center[i]); // + 8 \* k, где k - кол-во ненулевых кластеров  
 } else {  
 score[i] = numeric\_limits<double>::infinity(); // + 8 \* (count - k) байт  
 }  
 }  
 double score\_end = 0; // + 8 байт  
 for (int i = 0; i < count; i++) { // O(count)  
 score\_end += score[i];  
 }  
 if (score\_end < best\_score) {  
 best\_score = score\_end;  
 for (int j = 0; j < count; j++) { // O(count)  
 best\_cluster[j] = cluster[j]; // + n \* 8 байт  
 }  
 }  
 }  
 return best\_cluster;  
};  
  
int main() {  
 vector<double> numbers1 = **{**-9, 78, 76, 14, 38**}**; // + 8 \* 5 байт  
 vector<vector<double>> res1 = clustering(numbers1); // + 8 \* n байт  
 vector<vector<double>> test1 = **{**{-9},  
 {78},  
 {76},  
 {14},  
 {38}**}**; // + 8 \* 5 байт  
 assert(res1 == test1);  
 vector<double> numbers2 = **{**1, 2, 3, 4, 5, 6, 7, 8, 9, 10**}**; // + 8 \* 10 байт  
 vector<vector<double>> res2 = clustering(numbers2); // + 8 \* n байт  
 vector<vector<double>> test2 = **{**{1, 2, 3},  
 {4, 5, 6},  
 {7, 8},  
 {9},  
 {10}**}**; // + 8 \* 10 байт  
 assert(res2 == test2);  
 vector<double> numbers3 = **{**-9, 78, 76, 14, 38, 4, 15, 100, 46, 0**}**; // 8 \* 10 байт  
 vector<vector<double>> res3 = clustering(numbers3); // + 8 \* n байт  
 vector<vector<double>> test3 = **{**{-9, 4, 0},  
 {78, 76},  
 {14, 15},  
 {38, 46},  
 {100}**}**; // + 8 \* 10 байт  
 assert(res3 == test3);  
 vector<double> numbers4 = **{**-9, 7.5, 6.6, 14, 38, 17, 15, 100, -46.2, 0**}**; // 8 \* 10 байт  
 vector<vector<double>> res4 = clustering(numbers4); // + 8 \* n байт  
 vector<vector<double>> test4 = **{**{-9, 7.5, 6.6, 0},  
 {14, 17, 15},  
 {38},  
 {100},  
 {-46.2}**}**; // + 8 \* 10 байт  
 assert(res4 == test4);  
 return 0;  
}