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

Тамразов Валерий Каренович

Академическая группа №J3113

Принято

Должность, звание Фамилия Имя преподавателя

Санкт-Петербург

2024

***Введение***

Цель работы – реализовать алгоритм кластеризации массива посредством полного перебора всех комбинаций значений этого массива на языке C++ без использования сторонних библиотек

Задачи:  
1) Разработать алгоритм, обеспечивающий минимизацию указанной метрики.

2) Гарантировать выполнение ограничения сложности алгоритма (O(2^N) и выше).

3) Провести анализ использования памяти и времени выполнения.

4) Продемонстрировать результаты работы алгоритма с различными наборами данных.

***Теоретическая Часть***

Алгоритм - Полный перебор всех возможных разбиений массива предполагает использование рекурсивного подхода с отслеживанием текущих и оставшихся кластеров. Для каждого возможного разбиения:

* Вычисляется метрика: сумма абсолютных отклонений элементов подмассива от среднего значения.
* Хранится минимальная стоимость разбиения и соответствующая конфигурация.

**Основные аспекты**:

1. **Метрика**: sum(abs(arr\_cluster[i] - arr\_cluster\_mean)).
2. **Условия**:
   * Все элементы массива должны быть распределены.
   * Каждый кластер содержит хотя бы один элемент.
3. **Сложность**: Экспоненциальная из-за полного перебора.

**Типы данных**

* **Массив входных данных**: std::vector<int> (основной контейнер для данных).
* **Кластеры**: std::vector<std::vector<int>> для хранения текущей конфигурации.

**Вычисление метрики**

Среднее значение рассчитывается как: mean= ∑cluster/ size of cluster ​ Стоимость кластера: cost=∑∣val−mean

**Особенности реализации**

1. **Рекурсия с обратным трекингом** для перебора комбинаций разбиения.
2. **Защита от пустых кластеров**: добавлено условие проверки.

***Реализация***

**Этапы работы**

1. **Определение задачи**:
   * Подготовка структуры данных.
   * Выбор алгоритма полного перебора.
2. **Кодирование**:
   * Реализация рекурсивного алгоритма с оптимизацией памяти.
3. **Тестирование**:
   * Проверка работы на массиве с 20 элементами и 7 кластерами.

**Используемые библиотеки**

* <vector> для работы с массивами и кластерами.
* <cmath> для вычисления абсолютного значения.
* <limits> для хранения минимальной стоимости.

**Особенности кода**

* Рекурсивная функция backtrack выполняет полный перебор всех возможных комбинаций разбиения.
* Функция calculateClusterCost вычисляет метрику для каждого кластера.

***Экспериментальная часть***

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Размер входного массива | 7 | 10 | 15 | 20 | 25 | 30 |
| Время выполнения программы, с | 0,0 | 0,0 | 0,1 | 1,4 | 7,8 | 30,9 |
| O(2^N), с | 2 | 16 | 512 | 16384 | 524288 | 16777216 |

**Анализ таблицы**:

1. Теоретические и экспериментальные результаты не совпадают
2. Алгоритм сохраняет экспоненциальный рост времени выполнения, как и ожидалось.

***Заключение***

В ходе работы был реализован алгоритм кластеризации методом полного перебора. Цель работы достигнута: разработан и протестирован алгоритм, минимизирующий сумму абсолютных отклонений в кластерах. Результаты соответствуют теоретическим оценкам сложности O(2^N).

**Направления дальнейшего исследования**:

1. Оптимизация потребления памяти, например, через итеративные подходы.
2. Исследование параллельной реализации алгоритма для уменьшения времени выполнения.

***Приложения***

Код:  
#include <iostream>

#include <vector>

#include <cmath>

#include <limits>

#include <algorithm>

#include <numeric>

#include <functional>

#include <chrono>

#include <iomanip>

class Clusterizer {

private:

size\_t memory\_usage = 0;

void update\_memory\_usage(const std::vector<int>& arr, const std::vector<std::vector<int>>& clusters) {

memory\_usage = 0;

// Подсчёт памяти для массива arr

memory\_usage += arr.capacity() \* sizeof(int);

// Подсчёт памяти для кластеров

for (const auto& cluster : clusters) {

memory\_usage += cluster.capacity() \* sizeof(int);

}

}

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

if (cluster.empty()) return std::numeric\_limits<double>::max(); // Изменено для избежания пустых кластеров

double mean = std::accumulate(cluster.begin(), cluster.end(), 0.0) / cluster.size();

double cost = 0;

for (int val : cluster) {

cost += std::abs(val - mean);

}

return cost;

}

public:

std::vector<std::vector<int>> bruteForceClustering(const std::vector<int>& arr, int K) {

if (static\_cast<int>(arr.size()) < K) {

throw std::invalid\_argument("Количество элементов должно быть не меньше K");

}

int n = arr.size();

double min\_cost = std::numeric\_limits<double>::max();

std::vector<std::vector<int>> best\_clusters;

auto backtrack = [&](auto&& self, int start, int remaining\_clusters,

std::vector<std::vector<int>> current\_clusters) -> void {

// Проверяем, можем ли мы создать оставшиеся кластеры с минимум 1 элементом

int remaining\_elements = n - start;

if (remaining\_elements < remaining\_clusters) {

return;

}

if (remaining\_clusters == 0) {

if (start < n) return; // Остались необработанные элементы

double total\_cost = 0;

for (const auto& cluster : current\_clusters) {

if (cluster.empty()) return; // Пропускаем решения с пустыми кластерами

total\_cost += calculateClusterCost(cluster);

}

if (total\_cost < min\_cost) {

min\_cost = total\_cost;

best\_clusters = current\_clusters;

}

return;

}

// Оставляем элементы для остальных кластеров

int max\_elements = n - remaining\_clusters + 1 - start;

for (int end = start; end < start + max\_elements; ++end) {

std::vector<int> new\_cluster(arr.begin() + start, arr.begin() + end + 1);

current\_clusters.push\_back(new\_cluster);

self(self, end + 1, remaining\_clusters - 1, current\_clusters);

current\_clusters.pop\_back();

}

};

backtrack(backtrack, 0, K, {});

update\_memory\_usage(arr, best\_clusters);

return best\_clusters;

}

[[nodiscard]]

size\_t get\_memory\_usage() const {

return memory\_usage;

}

static void performanceTest() {

const std::vector<int> sizes = {10, 15, 20, 25, 30};

const int K = 7; // Фиксированное количество кластеров для тестов

std::cout << "Размер массива\tВремя (с)\tO(2^N) (теор.)\n";

for (const int size : sizes) {

// Генерируем тестовый массив

std::vector<int> test\_arr(size);

std::iota(test\_arr.begin(), test\_arr.end(), 1); // Заполняем числами от 1 до size

// Замеряем время выполнения

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

Clusterizer clusterizer;

clusterizer.bruteForceClustering(test\_arr, K);

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

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

// Теоретическое время O(2^N), нормализованное к первому измерению

const double theoretical = 0.5 \* std::pow(2, size - 5);

std::cout << size << "\t\t"

<< std::fixed << std::setprecision(1) << duration.count() << "\t\t"

<< theoretical << "\n";

}

}

};

// тестовая функция main для проверки времени работы

int main() {

Clusterizer::performanceTest();

return 0;

}

// // тестовая функция main

// int main() {

// std::vector<int> arr = {1,2,3,4,5, 6, 10, 20, 30, 40, 50, 110, 230, 340, 450, 560, 670, 780, 890, 900}; // Тест с разными числами

// int K = 7;

// Clusterizer clusterizer;

// try {

// std::vector<std::vector<int>> clusters = clusterizer.bruteForceClustering(arr, K);

// std::cout << "Clusters:" << std::endl;

// for (const auto& cluster : clusters) {

// std::cout << "[ ";

// for (int val : cluster) {

// std::cout << val << " ";

// }

// std::cout << "] ";

// }

// std::cout << std::endl;

// std::cout << "Memory usage: " << clusterizer.get\_memory\_usage() << " bytes" << std::endl;

// // 344 bytes

// } catch (const std::exception& e) {

// std::cout << "Error: " << e.what() << std::endl;

// }

// return 0;

// }