ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 3

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

Выполнил работу

Воробьев Андрей

Академическая группа №J3110

Принято

Ментор, Вершинин Владислав

Санкт-Петербург

2024

1. Введение

Цель лабораторной работы — на практике изучить влияние высокой сложности NP-полных задач на время выполнение.

Задачи – изучить решение NP-полных задач, реализовать комбинаторный алгоритм, создать измерить время работы алгоритма, написать отчёт.

1. Теоретическая подготовка

Для реализации этой лабораторной работы потребовалось

* знание контейнеров vector, unordered\_set
* умение работы с рекурсивными алгоритмами для решения комбинаторных задач
* знание файловых и строковых потоков
* знание библиотеки измерения времени
* знание передачи функций и элементов по ссылке

1. Реализация

* Реализация алгоритма нахождения метрики набора кластеров
* Реализация рекурсивного алгоритма перебора всех возможных
* Реализация тестирующей системы
* Оптимизация перебора, чтобы немного приблизиться к требуемому времени работы

1. Экспериментальная часть

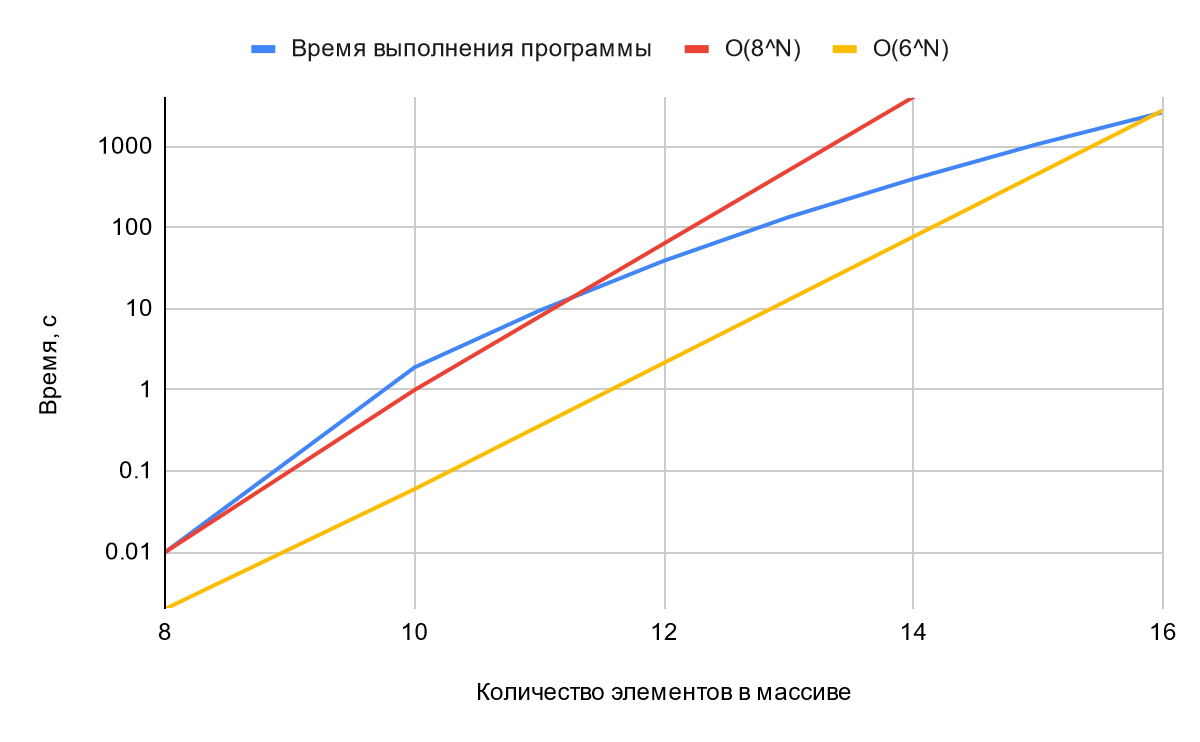
Согласно требованиям моего варианта, на вход к моему алгоритму подаётся до 25 элементов. Требуемая сложность алгоритма, согласно заданию, должна быть больше 2^N. Но в задании говорится о том, что решением должен быть перебор всех возможных наборов кластеров. Такое решение будет иметь сложность K^N, где K – число кластеров, N – число элементов. Но алгоритм со сложностью 8^25 будет выполняться вечно, так как это 10^22. По условию алгоритм должен работать меньше часа, таким образом, чтобы найти баланс между условиями по времени и по устройству алгоритма, было принято решение оптимизировать код, обрезая на некоторых моментах ненужный перебор.

Для тестирования алгоритма была собрана статистика, приведенная в таблице №1. С помощью предварительного теста было выяснено, что программа сложностью 10^9 выполняется за 1 секунду.

Таблица №1 - Подсчёт сложности реализованного алгоритма

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 8 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| Время выполнения программы, с | 0.01 | 1,9 | 9,5 | 39,2 | 135,4 | 401,1 | 1079,6 | 2653,9 |
| O(8^N), с | 0,01 | 1 | 8 | 64 | 512 | 4096 | 32768 | 262144 |
| O(6^N), с | 0,002 | 0,06 | 0,36 | 2,16 | 12,96 | 77,76 | 466,56 | 2799,36 |

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



Изображение 1 - График работы алгоритма

По графикам заметно, что при небольших значениях время работы алгоритма и её скорость роста сравнимы с алгоритмом сложности 8^N, но позже время выполнения программы уже больше сближается с графиком 4^N, за счёт оптимизации алгоритма.

1. Заключение.

В ходе выполнения работы мною был реализован алгоритм кластеризации массива, по заданной метрике. В задании были противоречивые требования одновременной быстроты работы алгоритма и его устройстве – полном переборе, что вынудило искать баланс между условиями.

Полный перебор был реализован с помощью рекурсии, но для ускорения были добавлены условия остановки рекурсии, основываясь на уже полученном лучше результате. Это позволило приблизить сложность алгоритма к 4^N, в отличие от 8^N при полном переборе.

Также в ходе работы потребовалось модифицировать созданную мной тестирующую систему, так как обычного сравнения ответов в этой задаче недостаточно. Нужно было реализовать проверку равенства метрик.

Алгоритм и тестирующая система были реализованы на языке C++. Если бы целью работы было бы реализовать именно эффективный по времени алгоритм, то сложность можно было бы уменьшить в разы воспользовавшись сортировкой исходного массива и реализацией выбора «мест разделения кластеров».

1. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла hw3.cpp

#include <cmath>  
#include <sstream>  
#include <unordered\_set>  
#include <vector>  
#include "../execute.h"  
  
  
*double* mean(*const* std::vector<*double*> &v) {  
 *// среднее арифметическое кластера  
 double* sum = 0;  
 *for* (*const double* &x: v) *// O(N), N = длина кластера* sum += x;  
 *return* sum / v.size();  
}  
  
*double* cluster\_metric(*const* std::vector<*double*> &v) {  
 *// метрика кластера  
 double* metric = 0;  
 *const double* current\_mean = mean(v);  
 *for* (*const double* &x: v) *// O(N), N = длина кластера* metric += std::abs((x - current\_mean));  
 *return* metric;  
}  
  
*double* clusters\_metric(*const* std::vector<std::vector<*double*> > &cluster\_distribution) {  
 *// метрика кластерного массива  
 double* metric = 0;  
 *for* (*const* std::vector<*double*> &cluster: cluster\_distribution) *// O(N), N = длина кластерного массива* metric += cluster\_metric(cluster);  
 *return* metric;  
}  
  
  
std::pair<*double*, std::vector<std::vector<*double*> > > clustering(*const* std::vector<*double*> &original\_array,  
 *const* size\_t number\_of\_element,  
 *const* size\_t &number\_of\_clusters, *double* &best\_metric,  
 std::vector<std::vector<*double*> > &  
 cluster\_distribution,  
 *double* &current\_clusters\_metric) {  
 *// рекурсивный алгоритм перебора кластеров  
  
 if* (number\_of\_element == original\_array.size()) {  
 *// условие завершения распределения  
 if* (current\_clusters\_metric < best\_metric)  
 best\_metric = current\_clusters\_metric;  
 *return* {current\_clusters\_metric, cluster\_distribution};  
 }  
  
 std::pair<*double*, std::vector<std::vector<*double*> > > best\_clusters = {10e20, std::vector<std::vector<*double*> >()};  
 *// память 2 байт + N \* 2 байт, N = количество элементов в исходном массиве* size\_t number\_of\_empty\_clusters = 0;  
 *for* (*const auto* &cluster: cluster\_distribution) *// подсчёт пустых кластеров, O(K), K = количество кластеров  
 if* (cluster.empty())  
 ++number\_of\_empty\_clusters;  
 *if* (number\_of\_empty\_clusters > original\_array.size() - number\_of\_element) *// условие того, что невозможно оставить кластеры не пустыми  
 return* best\_clusters;  
  
 *for* (size\_t i = 0; i < number\_of\_clusters; ++i) { *// перебор кластеров, куда вставить новый элемент, O(K), K = количество кластеров  
 if* (number\_of\_element == original\_array.size() - 1 *and* number\_of\_empty\_clusters == 1 *and* !cluster\_distribution[i].empty()) *// условие последнего распределяемого элемента при на наличии пустого кластера  
 continue*;  
  
 current\_clusters\_metric -= cluster\_metric(cluster\_distribution[i]);  
 cluster\_distribution[i].push\_back(original\_array[number\_of\_element]);  
 current\_clusters\_metric += cluster\_metric(cluster\_distribution[i]); *// добавление в i-й кластер нового элемента и пересчёт его метрики  
  
 if* (current\_clusters\_metric < best\_metric) { *// условие того, что метрика текущего кластера меньше лучшей* std::pair<*double*, std::vector<std::vector<*double*> > > current\_clusters =  
 clustering(original\_array, number\_of\_element + 1, number\_of\_clusters, best\_metric,  
 cluster\_distribution,  
 current\_clusters\_metric); *// проверка нового набора кластеров  
 // O(8 ^ N - number\_of\_element), N - number\_of\_element = количество нераспределённых элементов  
 // память 2 байт + N \* 2 байт, N = количество элементов в исходном массиве  
  
 if* (current\_clusters.first < best\_clusters.first)  
 best\_clusters = current\_clusters;  
 }  
  
 current\_clusters\_metric -= cluster\_metric(cluster\_distribution[i]);  
 cluster\_distribution[i].pop\_back();  
 current\_clusters\_metric += cluster\_metric(cluster\_distribution[i]); *// возврат к моменту до распределения элемента* }  
  
 *return* best\_clusters;  
}  
  
std::vector<std::vector<*double*> > clusterization(*const* std::vector<*double*> &original\_array,  
 *const* size\_t &number\_of\_clusters) {  
 *// запуск алгоритма кластеризации* std::vector<std::vector<*double*> > cluster\_distribution(number\_of\_clusters); *// N \* 2 байт, N = длина исходного массива  
 double* best\_metric = 10e20, current\_clusters\_metric = 0;  
  
 *return* clustering(original\_array, 0, number\_of\_clusters, best\_metric, cluster\_distribution, current\_clusters\_metric)  
 .second;  
}  
  
std::string execute\_clusterization(*const* std::string &input) {  
 std::vector<*double*> original\_array;  
 size\_t number\_of\_clusters;  
  
 std::istringstream input\_ss(input);  
 input\_ss >> number\_of\_clusters;  
 *double* tmp;  
 *while* (input\_ss >> tmp)  
 original\_array.push\_back(tmp);  
  
 std::vector<std::vector<*double*> > result = clusterization(original\_array, number\_of\_clusters);  
 std::ostringstream output\_ss;  
  
 *for* (*auto* &cluster: result) {  
 *for* (*auto* &element: cluster)  
 output\_ss << element << ' ';  
 output\_ss << "| ";  
 }  
  
 *return* output\_ss.str();  
}  
  
  
std::vector<std::vector<*double*> > string\_to\_clusters(*const* std::string &input\_string) {  
 std::vector<std::vector<*double*> > cluster(1);  
 std::stringstream ss(input\_string);  
 std::string tmp;  
  
 *while* (ss >> tmp) {  
 *if* (tmp == "|")  
 cluster.emplace\_back();  
 *else* cluster.back().push\_back(std::stod(tmp));  
 }  
  
 *return* cluster;  
}  
  
  
*bool* clusters\_contain\_same\_elements(*const* std::vector<std::vector<*double*> > &v1,  
 *const* std::vector<std::vector<*double*> > &v2) {  
 std::unordered\_multiset<*double*> s1, s2;  
  
 *for* (*const* std::vector<*double*> &cluster: v1)  
 *for* (*const double* &element: cluster)  
 s1.insert(element);  
  
 *for* (*const* std::vector<*double*> &cluster: v2)  
 *for* (*const double* &element: cluster)  
 s2.insert(element);  
  
 *return* s1 == s2;  
}  
  
  
*bool* compare\_function(*const* std::string &s1, *const* std::string &s2) {  
 std::stringstream ss1(s1), ss2(s2);  
  
 std::vector<std::vector<*double*> > v1 = string\_to\_clusters(s1), v2 = string\_to\_clusters(s2);  
  
 *return* clusters\_metric(v1) == clusters\_metric(v2) *and* clusters\_contain\_same\_elements(v1, v2);  
}  
  
  
*int* main() {  
 tester(11, execute\_clusterization, "C:/Users/vorandpav/CLionProjects/polygon/3hw/tests/",  
 compare\_function);  
}

ПРИ ЛОЖЕНИЕ Б

Листинг кода файла execute.h

#ifndef **EXECUTE\_H**#define **EXECUTE\_H**#include <cstddef>  
#include <functional>  
#include <string>  
  
*void* tester(  
 *const* size\_t file\_count, std::string (&exec\_function)(*const* std::string &),  
 *const* std::string &test\_path,  
 *bool* (&compare\_function)(*const* std::string &, *const* std::string &)  
 );  
  
*bool* check\_answer(  
 std::string &answer\_file\_name,  
 std::string &prog\_result\_file\_name,  
 *bool* (&compare\_function)(*const* std::string &, *const* std::string &)  
 );  
  
*void* execute\_str\_to\_str\_function(  
 std::string &input\_file\_name,  
 std::string &output\_file\_name,  
 std::string (&exec\_function)(*const* std::string &)  
 );  
  
#endif

ПРИЛОЖЕНИЕ В

Листинг кода файла execute.cpp

#include "execute.h"  
#include <fstream>  
#include <iostream>  
  
  
*void* execute\_str\_to\_str\_function(std::string &input\_file\_name, std::string &output\_file\_name,  
 std::string (&exec\_function)(*const* std::string &)) {  
 std::ifstream input\_file;  
 input\_file.open(input\_file\_name);  
 std::string line;  
 std::string tmp;  
 *while* (input\_file >> tmp) {  
 line += tmp + ' ';  
 }  
 input\_file.close();  
  
 line = exec\_function(line);  
  
 std::ofstream output\_file;  
 output\_file.open(output\_file\_name);  
 output\_file << line;  
 output\_file.close();  
}  
  
  
*bool* check\_answer(std::string &answer\_file\_name, std::string &prog\_result\_file\_name,  
 *bool* (&compare\_function)(*const* std::string &, *const* std::string &)) {  
 std::ifstream answer\_file, prog\_result\_file;  
 answer\_file.open(answer\_file\_name);  
 prog\_result\_file.open(prog\_result\_file\_name);  
 std::string answer, prog\_result, tmp;  
 *while* (answer\_file >> tmp) {  
 answer += tmp + ' ';  
 }  
 *while* (prog\_result\_file >> tmp) {  
 prog\_result += tmp + ' ';  
 }  
 answer\_file.close();  
 prog\_result\_file.close();  
  
 *if* (compare\_function(answer, prog\_result)) {  
 std::cout << "OK" << std::endl;  
 *return true*;  
 }  
 std::cout << "WRONG ANSWER" << std::endl;  
 *return false*;  
}  
  
  
*void* tester(*const* size\_t file\_count, std::string (&exec\_function)(*const* std::string &), *const* std::string &test\_path,  
 *bool* (&compare\_function)(*const* std::string &, *const* std::string &)) {  
 size\_t right\_answers = 0;  
  
 *for* (*int* i = 0; i < file\_count; i++) {  
 std::string input\_file\_name = test\_path + "test" + std::to\_string(i) + ".txt";  
 std::string output\_file\_name = test\_path + "output" + std::to\_string(i) + ".txt";  
  
 clock\_t start = clock();  
 execute\_str\_to\_str\_function(input\_file\_name, output\_file\_name, exec\_function);  
 clock\_t end = clock();  
  
 std::string answer\_file\_name = test\_path + "answer" + std::to\_string(i) + ".txt";  
 std::string prog\_result\_file\_name = test\_path + "output" + std::to\_string(i) + ".txt";  
  
 std::cout << "TEST " << i << ": ";  
  
 check\_answer(answer\_file\_name, prog\_result\_file\_name, compare\_function) ? right\_answers++ : 0;  
  
 std::cout << "TIME: " << (*double*) (end - start) / **CLOCKS\_PER\_SEC** << " SECONDS" << std::endl << std::endl;  
 }  
  
 std::cout << "Right answers: " << right\_answers << "/" << file\_count << std::endl;  
}