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

Отчёт по лабораторной работе № 4

«NP полные алгоритмы (Задача о покрытии множеств)»

Выполнил работу
Тищенко Павел
Академическая группа J3112
Принято
Лектор, Ходненко Иван Владимирович

Санкт-Петербург
2024

1. Введение

Цель работы: Разработка NP полного алгоритма решения задачи о покрытии множеств, а также проведение анализа производительности данного алгоритма на различных наборах данных.

Задачи:

Изучение теоретических основ задачи о покрытии множеств

Реализация алгоритма на языке программирования C++

Проведение тестов работы алгоритма с различными размерами наборов данных для оценки эффективности предложенного алгоритма с помощью библиотеки Google Tests.

Анализ результатов

2. Теоретическая подготовка

Для решения задачи о минимальном покрытии универсума используется метод перебора всех возможных подмножеств исходных множеств. Это делается с помощью битовых масок, что позволяет генерировать все возможные комбинации множества.

По шагово, для каждого из 2^n подмножеств осуществлялись следующие действия:

- Формирование подмножества: (O(n))

- Нахождение объединения подмножества: (O(N*K)\), где (N) — количество множеств, а (K) — средний размер подмножества.

- Проверка покрытия универсума: (O(U + log U)), где (U) — размер универсума.

Типы данных

- vector: используется для хранения динамических массивов. Позволяет изменять размер в процессе выполнения программы.

- set: представляет собой упорядоченное множество. Позволяет хранить уникальные элементы и обеспечивает быструю проверку наличия элемента.

-ifstream и ostringstream: используются для работы с файлами и преобразования данных соответственно (использовались для чтения данных для тестов с файлов).

5. Тестирование

Для проверки корректности работы алгоритма я реализовал юнит-тесты с использованием Google Test. Тесты охватывают разные случаи, включая базовые, случаи с пустыми множествами и универсума, а также случаи, когда покрытие невозможно. Это позволяет убедиться в правильности работы алгоритма в различных ситуациях.

3. Реализация

В этом разделе вам необходимо описать процесс выполнения работы, что вы сделали и какие этапы при этом выполняли, выжимки из кода, библиотеки и особенности реализации. Важно, не бывает 2 этапов выполнения задачи, Начали – закончили.

1. Осознание

Необходимо было понять, что от меня требуют, в чем заключается сложность данной мне задачи и что необходимо сделать для ее выполнения.

2. Выбор структур данных + реализация необходимых функций

Практически сразу было очевидно, что для решения данной задачи необходимо пользоваться **std::set<int>**: для хранения множеств. И **std::vector<int>**: для хранения индексов подмножеств.

Также сразу была реализована функция вывода результата алгоритма.

После чего, началась реализация findMinCover() (в последствии разбитая на еще 2 дополнительные: setInSet и unionOfSets, для лучшей читабельности

кода) В ходе ее разработки, необходимо было реализовать перебор всех 2^n случаем подмассивов, для этого было необходимо использовать битовые маски, также различные переборы и проверки на наличие одного множества в другом.

Соответственно для реализации данных функций мне потребовались

<iostream> - для работы с выводом и вводом

<vector> - для работы с динам. массивами

<set> - для работы с множествами

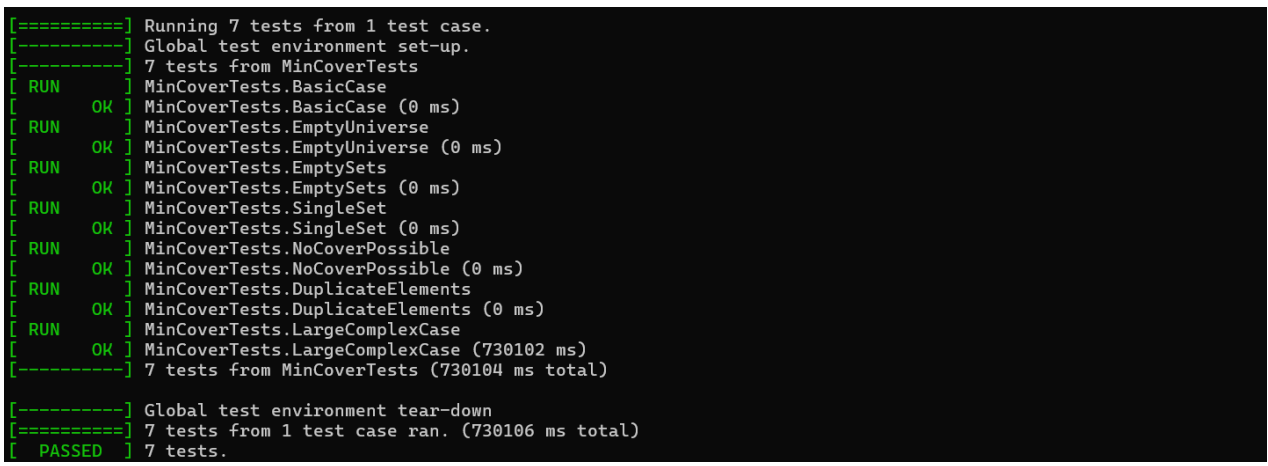
<fstream> - для чтения файлов

<sstream> - для чтения файлов

3. Тестирование

Для проверки правильности работы алгоритма были написаны юнит-тесты с использованием библиотеки Google Test. Тесты охватывали различные случаи показанные на Изображении 1:

- Базовые случаи с известными результатами.
- Случаи с пустыми множествами и универсума.
- Случаи, когда покрытие невозможно.



Изображение 1 – вывод GoogleTests

4. Экспериментальная часть

unionOfSets() | Память: 24 + ((24+4*k)*n) байт

Асимптотика: O(n*k)

Изображено на Изображении 2

```cpp
// Функция для объединения элементов подмножества множеств | O(N*K), N = subset.size(), K = unionSet.size()
std::set<int> unionOfSets(const std::vector<std::set<int>>& sets, const std::vector<int>& subset) {
    std::set<int> unionSet; // 24 + ((24+4*k)*n) байт
    for (int index : subset) { //4 байт
        unionSet.insert(sets[index].begin(), sets[index].end());
    }
    return unionSet;
```

Изображение 2 – код функции unionOfSets

findMinCover() | Память: 36 + 2^n*(64 + 8*k)

Асимптотика: O(2^n * (n*k))

Показано на изображении 3

```cpp
std::vector<int> findMinCover(const std::vector<std::set<int>>& sets, const std::set<int>& universe) {
    int n = sets.size(); //4 байта
    std::vector<int> bestCover;//24 байта
    int minSize = 25; //4 байта

    // Перебираем все подмножества с битовой маской
    for (int mask = 1; mask < (1 << n); ++mask) { //4 байта
        std::vector<int> subset;// каждый из 2**n массивов по 24байта

        // Формируем подмножество для текущей маски
        for (int i = 0; i < n; ++i) { //4 байта
            if (mask & (1 << i)) {
                subset.push_back(i); // 4 байта каждый эл-нт
            }
        }

        // Находим объединение подмножества
        std::set<int> unionSet = unionOfSets(sets, subset); // 24 + (4*k) байт

        // Проверяем, покрывает ли подмножество универсум
        if (setInSet(unionSet, universe)) {
            if (subset.size() < minSize) {
                minSize = subset.size();
                bestCover = subset;
            }
        }
    }

    return bestCover;
```

Изображение 3 – код функции findMinCover

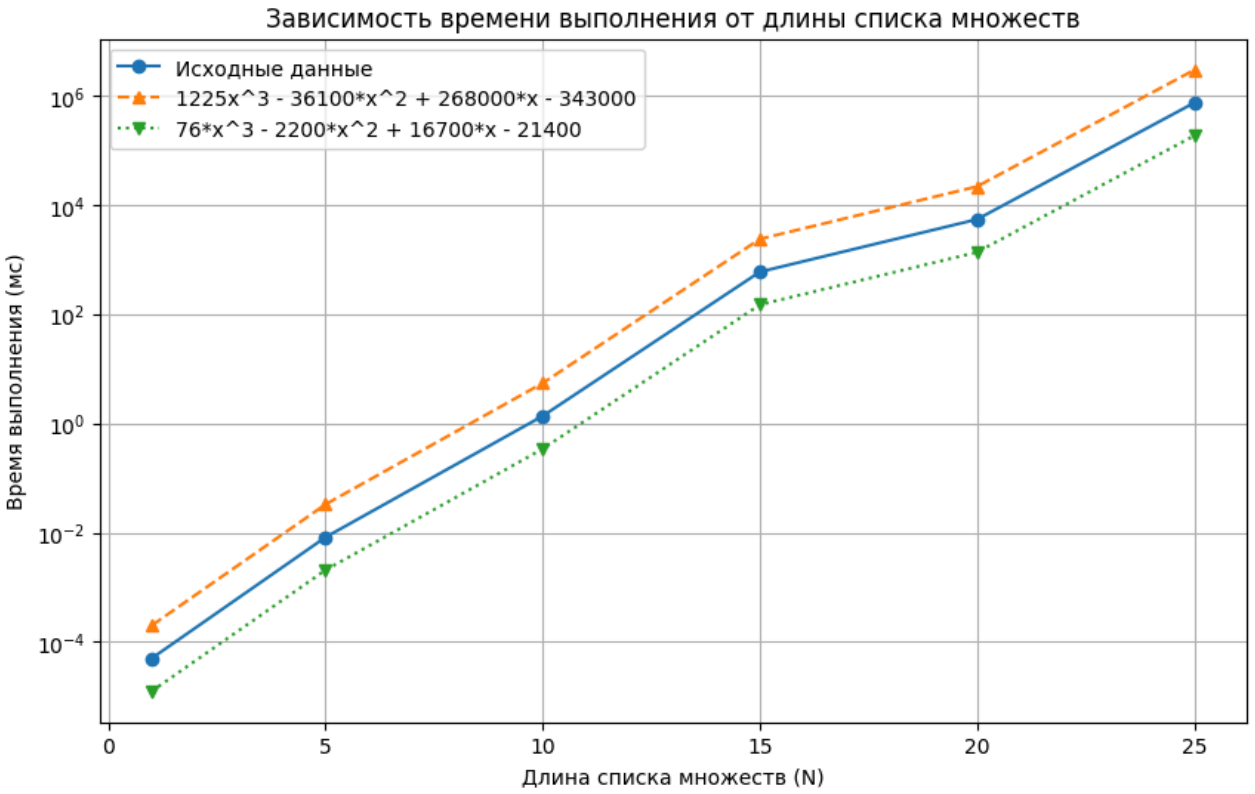
В соответствии с требованиями моего варианта, на вход алгоритма подается до 25 элементов. Теоретическая сложность задачи оценивается как O(2^N) и выше. Для тестирования алгоритма была собрана статистика, которая представлена в таблице №1.

Возьмем среднее universe.size() = 10; union.size() = 8 и рассмотрим зависимость скорости алгоритма от N = sets.size()

Таблица №1 - Подсчет сложности реализованного алгоритма

Размер входного набора	1	5	10	15	20	25
Время выполнения программы, с	0.0000001	0.0008	3	600	5500	750000
O( ), с	0.000001	0.008	15	1000	10000	100000
O( ), с	0.00000001	0.0002	0.66	300	3000	350000

График представляющий визуально удобный формат данных из таблицы №1 представлен на изображении 4.



Изображение 4 - График работы алгоритма

Исходя из графика и таблицы мы можем понять, что данный алгоритм является очень сложным по скорости и памяти и требует очень большого количества средств для его выполнения, При значениях N требуется все больше и больше времени для выполнения алгоритма

5. Заключение

В данной работе был реализован алгоритм нахождения минимального покрытия множества с использованием метода перебора всех возможных подмножеств. Алгоритм основан на применении битовых масок для генерации всех возможных комбинаций подмножеств и последующей проверке их покрытия универсума.

Полученный результат демонстрирует корректность работы алгоритма при тестировании на различных наборах данных. Однако стоит отметить, что текущая реализация имеет экспоненциальную сложность

Для дальнейшего улучшения алгоритма можно разделять задачи на несколько потоков обработки . Или исходно использовать жадный алгоритм, который может уменьшить количество рассматриваемых вариантов без значительного ухудшения качества решения.

Таким образом, существует потенциал для дальнейшей оптимизации и адаптации к задачам с большим числом входных данных.

6. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла lab4.cpp

```
#include "pch.h"

#include <iostream>
#include <vector>
#include <set>
#include <fstream>
#include <sstream>

// Функция для объединения элементов подмножества множеств | O(N*K), N = subset.size(), K = unionSet.size()
std::set<int> unionOfSets(const std::vector<std::set<int>>& sets, const std::vector<int>& subset) {
    std::set<int> unionSet; // 24 + ((24+4*k)*n) байт
    for (int index : subset) { //4 байт
                    unionSet.insert(sets[index].begin(), sets[index].end());
```

```
    }
    return unionSet;
}

// проверка на покрытие мн-ва | O(U+logU), U = universe.size()
bool setInSet(const std::set<int>& unionSet, const std::set<int>& universe) {
    for (int num : universe) {//4 байта
        if (unionSet.count(num) == 0) { // O(logU)
            return false;
        }
    }
    return true;
}

// Функция для поиска минимального покрытия | O(2^n * (n+(N*K+U+logU))) , n sets.size()
std::vector<int> findMinCover(const std::vector<std::set<int>>& sets, const std::set<int>& universe) {
    int n = sets.size(); //4 байта
    std::vector<int> bestCover;//24 байта
    int minSize = 25; //4 байта

    // Перебираем все подмножества с битовой маской
     for (int mask = 1; mask < (1 << n); ++mask) { //4 байта
```

```
            std::vector<int> subset;// каждый из 2**n
массивов по 24байта

        // Формируем подмножество для текущей маски
        for (int i = 0; i < n; ++i) { //4 байта
            if (mask & (1 << i)) {
                subset.push_back(i); // 4 байта каждый
эл-нт
            }
        }

        // Находим объединение подмножества
            std::set<int> unionSet = unionOfSets(sets,
subset); // 24 + (4*k) байт

            // Проверяем, покрывает ли подмножество
универсум
        if (setInSet(unionSet, universe)) {
            if (subset.size() < minSize) {
                minSize = subset.size();
                bestCover = subset;
            }
        }
    }

    return bestCover;
}

int NotMain() {
```

```
        std::vector<std::set<int>>  sets;  //24+(n*(24+k*4
байт))байт
    std::set<int> universe; // 24+4*k байт
    std::vector<int> uniset; // 24 + 4*k байт
    if (universe.size() > 0) {
         uniset = findMinCover(sets, universe); //24+4*k
байт
    }
    else {
        uniset = { 0 };
    }


    ///вывод | O(n*k)
    std::cout << "[";
    for (size_t i = 0; i < uniset.size(); i++) { //8байт
        std::cout << "{";
               for (auto it = sets[i].begin(); it !=
sets[i].end(); ++it) { //4байта
            std::cout << *it;
            if (std::next(it) != sets[i].end()) {
                std::cout << ",";
            }
        }
        std::cout << "}";
        if ((i + 1) != uniset.size()) {
            std::cout << ", ";
        }
    }
```

```
    std::cout << "]" << std::endl;

    return 0;
}
class Pupipam : public testing::Test {
protected:
    void SetUp() override {}
    void TearDown() override {}
};
// Тест на базовый случай
TEST(MinCoverTests, BasicCase) {
    std::vector<std::set<int>> sets = { {1, 2}, {11, 3}, {3, 4}, {14, 2}, {2, 18}, {3, 9}, {1, 2}, {21, 33}, {3, 4}, {15, 2}, {2, 3}, {3, 9} };
    std::set<int> universe = { 1, 2, 3, 9, 21, 15, 11, 18, 33, 14 };
        std::vector<int> result = findMinCover(sets, universe);
    std::vector<int> expected = { 0, 1, 3, 4, 5, 7, 9 };
// Ожидаем, что будут выбраны множества {1, 2} и {3, 9}
    EXPECT_EQ(result, expected);
}

// Тест на случай с пустым универсумом
TEST(MinCoverTests, EmptyUniverse) {
    std::vector<std::set<int>> sets = { {1, 2}, {2, 3}, {3, 4} };
    std::set<int> universe = {};
```

```
        std::vector<int> result = findMinCover(sets, universe);
       std::vector<int> expected = { 0 }; // Пустой универсум требует пустого покрытия
    EXPECT_EQ(result, expected);
}

// Тест на случай с пустыми множествами
TEST(MinCoverTests, EmptySets) {
    std::vector<std::set<int>> sets = { {}, {}, {} }; // Три пустых множества
    std::set<int> universe = { 1, 2, 3 };
        std::vector<int> result = findMinCover(sets, universe);
       std::vector<int> expected = {}; // Невозможно покрыть универсум
    EXPECT_EQ(result, expected);
}

// Тест на случай, когда есть только одно множество
TEST(MinCoverTests, SingleSet) {
    std::vector<std::set<int>> sets = { {1, 2} };
    std::set<int> universe = { 1, 2 };
        std::vector<int> result = findMinCover(sets, universe);
    std::vector<int> expected = { 0 }; // Одно множество покрывает универсум
    EXPECT_EQ(result, expected);
}
```

```
// Тест на случай, когда покрытие невозможно
TEST(MinCoverTests, NoCoverPossible) {
    std::vector<std::set<int>> sets = { {1}, {2}, {3} };
       std::set<int> universe = { 4 }; // Универсум, который не может быть покрыт
        std::vector<int> result = findMinCover(sets, universe);
       std::vector<int> expected = {}; // Невозможно покрыть
    EXPECT_EQ(result, expected);
}

// Тест на случай с дублирующимися элементами
TEST(MinCoverTests, DuplicateElements) {
     std::vector<std::set<int>> sets = { {1, 2}, {2, 3}, {3, 1} };
    std::set<int> universe = { 1, 2, 3 };
        std::vector<int> result = findMinCover(sets, universe);
      std::vector<int> expected = { 0, 1 }; // Можно выбрать любые два из трех
     EXPECT_TRUE((result == std::vector<int>{0, 1}) || (result == std::vector<int>{0, 2}) || (result == std::vector<int>{1, 2}));
}
```

```
std::vector<std::set<int>>  readSets(const  std::string&
filename) {
    std::vector<std::set<int>> sets;
    std::ifstream file(filename);
    std::string line;

    while (std::getline(file, line)) {
        std::set<int> currentSet;
        std::istringstream iss(line);
        int num;
        while (iss >> num) {
            currentSet.insert(num);
        }
        sets.push_back(currentSet);
    }

    return sets;
}

std::set<int>  readUniverse(const  std::string&  filename)
{
    std::set<int> universe;
    std::ifstream file(filename);
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream iss(line);
        int num;
        while (iss >> num) {
```

```
            universe.insert(num);
        }
    }

    return universe;
}

// Тест на случай с большим числом множеств и элементов универсума
TEST(MinCoverTests, LargeComplexCase) {
                std::vector<std::set<int>>    sets    = readSets("C:/Users/agafo/Desktop/lab42/sets.txt");
                    std::set<int>     universe     = readUniverse("C:/Users/agafo/Desktop/lab42/universum.txt");
    // Ожидаемое покрытие

     std::vector<int> expected = { 0, 1, 4, 17 }; // Ожидаем, что будут выбраны множества {1, 2, 3, 4, 5}, {3, 4, 5, 6, 7}, {10, 11, 12, 13, 14}, и {1, 20, 30, 40, 50}

    // Выполнение поиска минимального покрытия
        std::vector<int>  result  =  findMinCover(sets, universe);

    // Проверка результата
    EXPECT_EQ(result, expected);
}
```

```
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);
    return RUN_ALL_TESTS();
}
```