ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 4

«Задача о покрытии множеств»

Выполнил работу

Ширкунова Мария

Академическая группа №J3114

Принято

Дунаев Максим Владимирович

Санкт-Петербург

2024

Содержание отчета

1. Введение

Цель работы: изучение сложности алгоритмов, относящихся к классу NP-полных задач, а также практическое применение комбинаторного подхода для решения одной из таких задач.

В рамках работы необходимо решить задачу о покрытии множества. Дан набор множеств sets и множество universe. Задача заключается в том, чтобы выбрать минимальное количество множеств из sets, которые полностью покрывают все элементы из universe.

Задачи:

- Реализовать алгоритм для нахождения минимального покрытия.
- Протестировать алгоритм на различных входных данных.
- Оценить сложность алгоритма.
- Оценить использование дополнительной памяти.
- Проанализировать влияние размера входных данных на время выполнения алгоритма.

2. Теоретическая подготовка

В данной работе используется комбинаторный подход для решения задачи о покрытии множества. Основные теоретические аспекты включают:

NP-полные задачи: Это задачи, для решения которых не существует детерминированного алгоритма, работающего за полиномиальное время.

Битовые маски: Для представления всех возможных комбинаций подмножеств используется битовая маска, где каждый бит указывает на включение или исключение соответствующего множества в будущее покрытие.

Структуры данных: Используются векторы (vector — контейнер, предоставляемый стандартной библиотекой C++, который представляет собой динамический массив) и множества (set — контейнер, который хранит уникальные элементы в отсортированном порядке) для хранения подмножеств и элементов универсума.

Библиотеки: iostream (предоставляет функциональность для ввода и вывода данных), cassert (предоставляет макросы для выполнения проверок условий), random (предоставляет функциональность для генерации случайных чисел), chrono (предоставляет функциональность для работы с временем и временными интервалами).

3. Реализация

Процесс выполнения работы включает несколько этапов:

1. Определение структуры данных: Для хранения множеств и универсума были выбраны стандартные контейнеры STL — vector для хранения подмножеств и set для универсума.
2. Определение используемых библиотек: использую <cassert> для написания тестов, <chrono> для подсчета времени выполнения алгоритма, <random> для генерации случайного набора входных данных.

```
#include <iostream>
#include <vector>
#include <set>
#include <cassert>
#include <chrono>
#include <random>
```

3. Реализация функции minimum_sets_to_cover:
    3.1. Определение входных параметров: Функция принимает на вход вектор множеств и множество универсума по ссылке как константу (нельзя менять в теле функции).
    3.2. Определение выходных параметров: вектор множеств – лучшая комбинация – ответ на задачу.
    3.3. Инициализация переменных: размер входного вектора, минимальный размер полученной комбинации, лучшая комбинация.
    3.4. Для каждой комбинации подмножеств (их 2^N), представленной битовой маской, функция проверяет, покрывает ли объединение этих подмножеств элементы универсума. Для этого мы заводим вектор, хранящий текущую комбинацию, кладя в него те множества, удовлетворяющие текущей маске (1 – множество включено в комбинацию, 0 – не включено).
    3.5. Все элементы, содержащиеся в текущей комбинации, кладем в множество union_set. Проверяем, покрывает ли это множество универсум с помощью функции includes.

3.6. Обновляем переменные минимального размера выходной комбинации и лучшую комбинацию.

3.7. Возвращаем удовлетворяющую условию комбинацию.

```
vector<set<int>> minimum_sets_to_cover(const vector<set<int>>& sets, const set<int>&
universe) {
    int n = sets.size();
    vector<set<int>> best_combination;
    int min_size = n + 1;

    for (int mask = 1; mask < (1 << n); ++mask) {
        set<int> union_set;
        vector<set<int>> current_combination;

        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) {
                union_set.insert(sets[i].begin(), sets[i].end());
                current_combination.push_back(sets[i]);
            }
        }

        if (universe.size() == union_set.size() && includes(union_set.begin(),
union_set.end(), universe.begin(), universe.end())) {
            if (current_combination.size() < min_size) {
                min_size = current_combination.size();
                best_combination = current_combination;
            }
        }
    }

    return best_combination;
}
```

4. Расчет алгоритмической сложности и объема дополнительной памяти. Подробнее на стр. 11-12.

5. Тестирование: Реализация функции тестирования с помощью cassert.

5.1. Проверяем тест из условия, случаи с пустыми sets, universe, best_combinations (отсутствие решения), одноэлементными подмножествами, 2 вариантами ответа (проверка выбора минимальной по длине комбинации множеств).

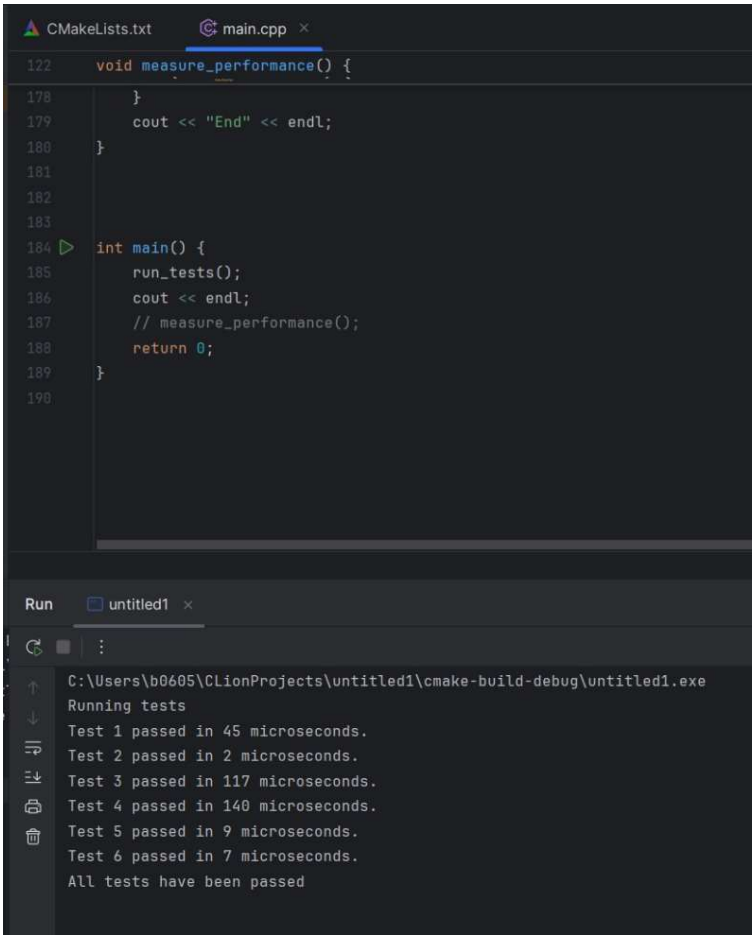
5.2. Подсчитываем время выполнения каждого из тестов с помощью chrono.

5.3. Выводим полученные результаты выполнения: изображение №1.

```
void run_tests() {
    cout << "Running tests" << endl;

    // Тест 1. Пример из условия
    vector<set<int>> sets1 = { {1, 2}, {2, 3}, {3, 4} };
    set<int> universe1 = { 1, 2, 3 };

    auto start1 = chrono::high_resolution_clock::now();
```

```
    auto result1 = minimum_sets_to_cover(sets1, universe1);
    auto end1 = chrono::high_resolution_clock::now();

    assert((result1 == vector<set<int>>{ {1, 2}, { 2, 3 }}));
    cout << "Test 1 passed in " << chrono::duration_cast<chrono::microseconds>(end1 -
start1).count() << " microseconds." << endl;


    // Тест 2. Пустое множество
    vector<set<int>> sets2 = { {} };
    set<int> universe2 = { 1, 2, 5 };

    auto start2 = chrono::high_resolution_clock::now();
    auto result2 = minimum_sets_to_cover(sets2, universe2);
    auto end2 = chrono::high_resolution_clock::now();

    assert((result2 == vector<set<int>>{}));
    cout << "Test 2 passed in " << chrono::duration_cast<chrono::microseconds>(end2 -
start2).count() << " microseconds." << endl;


    // Тест 3. Одноэлементное множество
    vector<set<int>> sets3 = { {1}, {2}, {3}, {4}, {5} };
    set<int> universe3 = { 1, 3, 5 };

    auto start3 = chrono::high_resolution_clock::now();
    auto result3 = minimum_sets_to_cover(sets3, universe3);
    auto end3 = chrono::high_resolution_clock::now();

    assert((result3 == vector<set<int>>{ {1}, { 3 }, { 5 }}));
    cout << "Test 3 passed in " << chrono::duration_cast<chrono::microseconds>(end3 -
start3).count() << " microseconds." << endl;


    // Тест 4. 2 варианта (а нужен минимальный)
    vector<set<int>> sets4 = { {1, 2}, {3}, {5}, {1, 3, 5}, {5} };
    set<int> universe4 = { 1, 3, 5 };

    auto start4 = chrono::high_resolution_clock::now();
    auto result4 = minimum_sets_to_cover(sets4, universe4);
    auto end4 = chrono::high_resolution_clock::now();

    assert((result4 == vector<set<int>>{ {1, 3, 5}}));
    cout << "Test 4 passed in " << chrono::duration_cast<chrono::microseconds>(end4 -
start4).count() << " microseconds." << endl;


    // Тест 5. Пустой вывод
    vector<set<int>> sets5 = { {1}, {3, 5} };
    set<int> universe5 = { 2, 4, 6 };

    auto start5 = chrono::high_resolution_clock::now();
    auto result5 = minimum_sets_to_cover(sets5, universe5);
    auto end5 = chrono::high_resolution_clock::now();

    assert((result5 == vector<set<int>>{}));
    cout << "Test 5 passed in " << chrono::duration_cast<chrono::microseconds>(end5 -
start5).count() << " microseconds." << endl;


    // Тест 6. Пустое множество
    vector<set<int>> sets6 = { {1}, {3, 5} };
    set<int> universe6 = {};
```

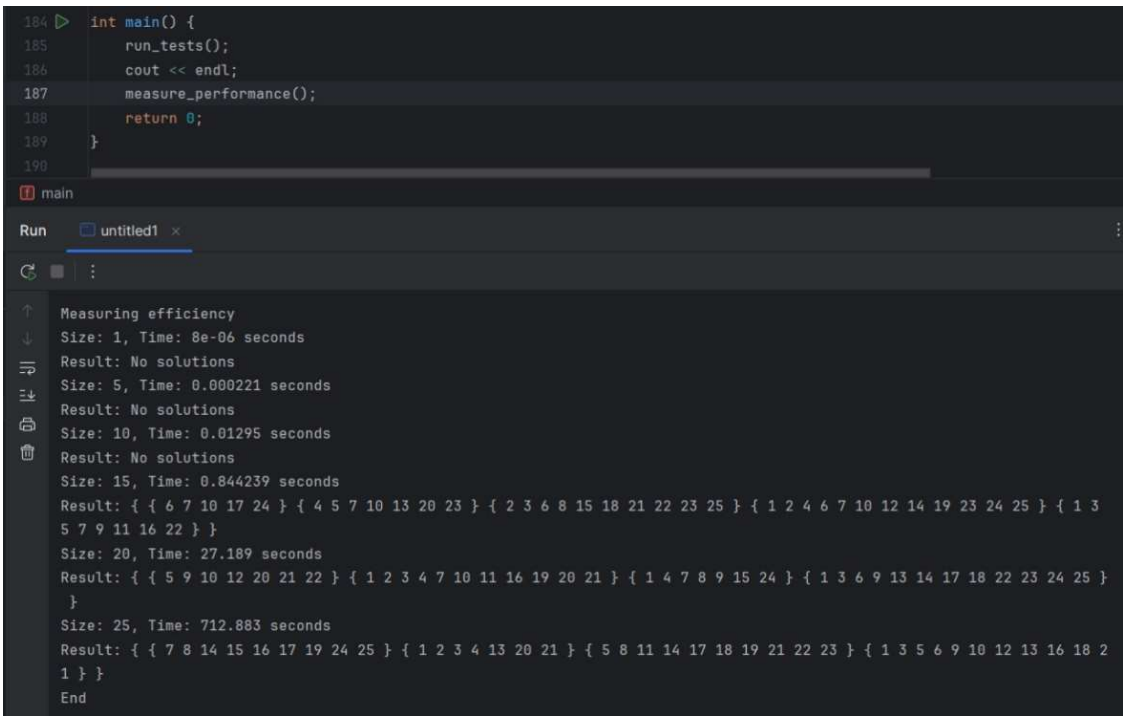
```
    auto start6 = chrono::high_resolution_clock::now();
    auto result6 = minimum_sets_to_cover(sets6, universe6);
    auto end6 = chrono::high_resolution_clock::now();

    assert((result6 == vector<set<int>>{}));
    cout << "Test 6 passed in " << chrono::duration_cast<chrono::microseconds>(end6 -
start6).count() << " microseconds." << endl;

    cout << "All tests have been passed" << endl;
}
```



Изображение №1 – Результаты выполнения тестов

6. Эффективность алгоритма на различных входных наборах: Реализация функции measure_performance.
    6.1. Генерируем универсум из 25 элементов.
    6.2. Смотрим на ситуации с 1, 5, 10, 15, 20, 25 подмножествами длиной от 1 до 12 и случайным наполнением.
    6.3. Считаем время выполнения функции minimum_sets_to_cover для каждой из ситуаций.
    6.4. Выводим на экран полученное время и результат выполнения алгоритма решения задачи: изображение №2.

```
void measure_performance() {
    cout << "Measuring efficiency" << endl;

    const int universe_size = 25;
    set<int> universe;

    for (int i = 1; i <= universe_size; ++i) {
        universe.insert(i);
    }

    vector<int> sizes = { 1, 5, 10, 15, 20, 25 };

    for (int size : sizes) {
        vector<set<int>> sets;

        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dis(1, universe_size);

        for (int i = 0; i < size; ++i) {
            set<int> subset;
            int subset_size = dis(gen) % (universe_size / 2) + 1;

            while (subset.size() < subset_size) {
                subset.insert(dis(gen));
            }

            sets.push_back(subset);
        }

        auto start = chrono::high_resolution_clock::now();
        vector<set<int>> result = minimum_sets_to_cover(sets, universe);
        auto end = chrono::high_resolution_clock::now();

        double elapsed_time = chrono::duration_cast<chrono::microseconds>(end -
start).count();

        cout << "Size: " << size << ", Time: " << elapsed_time / 1e6 << " seconds" <<
endl;

        cout << "Result: ";
        if (!result.empty()) {
            cout << "{ ";
            for (const auto& s : result) {
                cout << "{ ";
                for (const auto& elem : s) {
                    cout << elem << " ";
                }
                cout << "} ";
            }
            cout << "}" << endl;
        }
        else {
            cout << "No solutions" << endl;
        }
    }
    cout << "End" << endl;
}
```



Изображение №2 – Результат выполнения алгоритма

7. Построение таблицы и графика по полученным значениям. Сравнение с теоретическими значениями. Анализ полученных результатов. Подробнее на стр. 12-13. Построение выводов. Подробнее на стр. 14.

4. Экспериментальная часть

4.1. Подсчет по памяти.

```
vector<set<int>> minimum_sets_to_cover(const vector<set<int>>& sets, const set<int>&
universe) {
    int n = sets.size(); // + 4 байт (максимум 25 множеств => int)
    vector<set<int>> best_combination; // 4*M*N, где N - количество множеств, M - макс.
количество элементов в множествах
    int min_size = n + 1; // + 4 байт

    for (int mask = 1; mask < (1 << n); ++mask) { // + 4 байт
        set<int> union_set; // + 4*M
        vector<set<int>> current_combination; // + 4*N*M

        for (int i = 0; i < n; ++i) { // + 4 байт
            if (mask & (1 << i)) {
                union_set.insert(sets[i].begin(), sets[i].end());
                current_combination.push_back(sets[i]);
            }
        }

        if (universe.size() == union_set.size() && includes(union_set.begin(),
union_set.end(), universe.begin(), universe.end())) {
            if (current_combination.size() < min_size) {
                min_size = current_combination.size();
                best_combination = current_combination;
            }
        }
    }

    return best_combination;

    // Итого: 4 + 4*M*N + 4 + 4 + 4*M + 4*N*M + 4 = 4*4 + 2*4*M*N + 4*M = 8MN + 4M + 16 байт
    // Максимальное N=25, M=50 => 8*25*50 + 4*50 + 16 = 10216 байт
}
```

4.2. Подсчет асимптотики.

```
vector<set<int>> minimum_sets_to_cover(const vector<set<int>>& sets, const set<int>&
universe) {
    int n = sets.size(); // O(1)
    vector<set<int>> best_combination;
    int min_size = n + 1; // O(1)

    for (int mask = 1; mask < (1 << n); ++mask) { // O(2^N)
        set<int> union_set;
        vector<set<int>> current_combination;

        for (int i = 0; i < n; ++i) { // O(N)
            if (mask & (1 << i)) { // O(1)
                union_set.insert(sets[i].begin(), sets[i].end()); // O(M), где M - средний
размер множества, M < N
                current_combination.push_back(sets[i]); // O(1)
            }
        }

        // O(M)
        if (universe.size() == union_set.size() && includes(union_set.begin(),
union_set.end(), universe.begin(), universe.end())) {
            if (current_combination.size() < min_size) { // O(1)
                min_size = current_combination.size(); // O(1)
                best_combination = current_combination;
```

```
            }
        }
    }

    return best_combination;

    // Итого: O((2^N)*(N*M + M)) = O((2^N)*M(N+1)) = O((2^N)*N*M) = O((2^N)*(N^2)) > O(2^N)
- условие выполнено
}
```

4.3. График зависимости времени от числа элементов.

Согласно требованиям моего варианта, на вход к моему алгоритму подаётся до 25 множеств в sets. Теоретически заданная сложность задачи составляет O(2^N) и более. Подсчитанная мною асимптотика составляет O((N^2)*(2^N)). Для тестирования алгоритма была собрана статистика, приведенная в таблице №1.

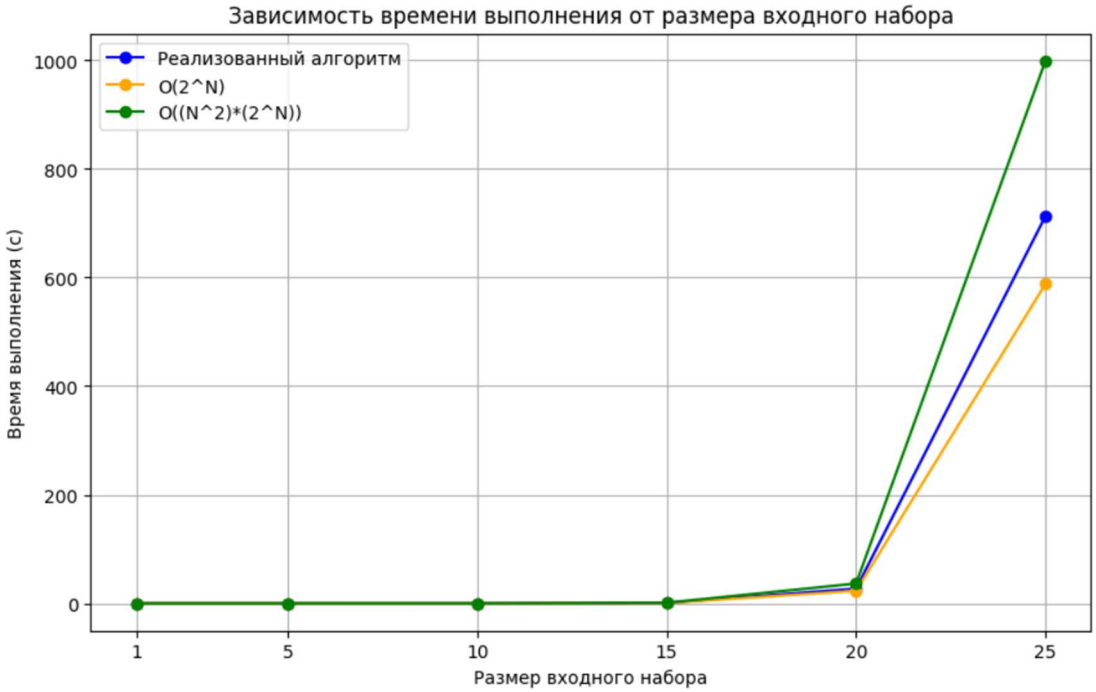
Таблица №1 - Подсчёт сложности реализованного алгоритма

Размер входного набора	1	5	10	15	20	25
Время выполнения программы, с	0.000006	0.000221	0.01295	0.844239	27.189	712.883
O(2^N), с	0	0.00002	0.00941	0.64812	23.246	587.895
O((N^2)*(2^N)), с	0.000013	0.000415	0.02273	1.23119	36.556	998.749

График представляющий визуально удобный формат данных из таблицы №1 представлен на изображении №3.

Согласно результатам, мы можем сделать вывод о том, что алгоритм демонстрирует значительное увеличение времени выполнения с ростом размера входного набора. Время выполнения программы превышает теоретическую оценку O(2^N) и выполняет требуемое условие задачи. Это указывает на то, что реальная сложность алгоритма выше, чем просто экспоненциальная. Время выполнения программы соответствует ожидаемым значениям для данной асимптотики. График реализованного алгоритма находится между O(2^N) и O((N^2)*(2^N)). Для больших значений N, например, при N = 25, время выполнения приближается к 998.749 секунд, что согласуется с предположением о сложности. Данные результаты подчеркивают важность оптимизации

алгоритмов для работы с большими наборами данных, так как даже небольшое увеличение размера входного набора может привести к значительному увеличению времени выполнения.



Изображение №3 – График работы алгоритма

5. Заключение

В ходе выполнения лабораторной работы был реализован комбинаторный алгоритм для решения задачи о покрытии множества. Цель работы была достигнута путем тестирования алгоритма на различных наборах данных с различным количеством множеств и элементов этих множеств. Полученные результаты подтвердили теоретические оценки сложности алгоритма, демонстрируя экспоненциальный рост времени выполнения с увеличением числа подмножеств.

В качестве дальнейших исследований можно предложить оптимизацию алгоритма с точки зрения уменьшения временных затрат за счет использования жадных методов или эвристик. Также стоит рассмотреть возможность применения параллельных вычислений для обработки больших наборов данных, что может значительно ускорить выполнение алгоритма в условиях ограниченных ресурсов.

6. Приложения

ПРИЛОЖЕНИЕ А

Листинг кода файла main.cpp

```
#include <iostream>
#include <vector>
#include <set>
#include <cassert>
#include <chrono>
#include <random>

using namespace std;

// Функция для нахождения минимального количества множеств для покрытия
// Множества передаю по ссылке как константу (нельзя менять в теле
функции)
vector<set<int>> minimum_sets_to_cover(const vector<set<int>>& sets,
const set<int>& universe) {
    int n = sets.size();
    vector<set<int>> best_combination;
    int min_size = n + 1; // Инициализируем минимальный размер большим
значением

    // Комбинаторно проверяем все возможные комбинации подмножеств
    // Их 2^n = 1 << n
    // 1 - включен, 0 - не включен
    for (int mask = 1; mask < (1 << n); ++mask) {
        set<int> union_set;
        vector<set<int>> current_combination;

        for (int i = 0; i < n; ++i) {
            if (mask & (1 << i)) { // Если i-й элемент включен в
комбинацию
                union_set.insert(sets[i].begin(), sets[i].end());
                current_combination.push_back(sets[i]);
            }
        }

        // Проверяем, покрывает ли объединение множество universe
        if (universe.size() == union_set.size() &&
includes(union_set.begin(), union_set.end(), universe.begin(),
universe.end())) {
            if (current_combination.size() < min_size) {
                min_size = current_combination.size();
                best_combination = current_combination;
            }
        }
    }
```

```
    return best_combination;
}

// Функция для тестирования
void run_tests() {
    cout << "Running tests" << endl;

    // Тест 1. Пример из условия
    vector<set<int>> sets1 = { {1, 2}, {2, 3}, {3, 4} };
    set<int> universe1 = { 1, 2, 3 };

    auto start1 = chrono::high_resolution_clock::now();
    auto result1 = minimum_sets_to_cover(sets1, universe1);
    auto end1 = chrono::high_resolution_clock::now();

    assert((result1 == vector<set<int>>{ {1, 2}, { 2, 3 }}));
    cout << "Test 1 passed in " <<
chrono::duration_cast<chrono::microseconds>(end1 - start1).count() << "
microseconds." << endl;


    // Тест 2. Пустое множество
    vector<set<int>> sets2 = { {} };
    set<int> universe2 = { 1, 2, 5 };

    auto start2 = chrono::high_resolution_clock::now();
    auto result2 = minimum_sets_to_cover(sets2, universe2);
    auto end2 = chrono::high_resolution_clock::now();

    assert((result2 == vector<set<int>>{}));
    cout << "Test 2 passed in " <<
chrono::duration_cast<chrono::microseconds>(end2 - start2).count() << "
microseconds." << endl;


    // Тест 3. Одноэлементное множество
    vector<set<int>> sets3 = { {1}, {2}, {3}, {4}, {5} };
    set<int> universe3 = { 1, 3, 5 };

    auto start3 = chrono::high_resolution_clock::now();
    auto result3 = minimum_sets_to_cover(sets3, universe3);
    auto end3 = chrono::high_resolution_clock::now();

    assert((result3 == vector<set<int>>{ {1}, { 3 }, { 5 }}));
    cout << "Test 3 passed in " <<
chrono::duration_cast<chrono::microseconds>(end3 - start3).count() << "
microseconds." << endl;


    // Тест 4. 2 варианта (а нужен минимальный)
    vector<set<int>> sets4 = { {1, 2}, {3}, {5}, {1, 3, 5}, {5} };
```

```
    set<int> universe4 = { 1, 3, 5 };

    auto start4 = chrono::high_resolution_clock::now();
    auto result4 = minimum_sets_to_cover(sets4, universe4);
    auto end4 = chrono::high_resolution_clock::now();

    assert((result4 == vector<set<int>>{ {1, 3, 5}}));
    cout << "Test 4 passed in " <<
chrono::duration_cast<chrono::microseconds>(end4 - start4).count() << "
microseconds." << endl;


    // Тест 5. Пустой вывод
    vector<set<int>> sets5 = { {1}, {3, 5} };
    set<int> universe5 = { 2, 4, 6 };

    auto start5 = chrono::high_resolution_clock::now();
    auto result5 = minimum_sets_to_cover(sets5, universe5);
    auto end5 = chrono::high_resolution_clock::now();

    assert((result5 == vector<set<int>>{}));
    cout << "Test 5 passed in " <<
chrono::duration_cast<chrono::microseconds>(end5 - start5).count() << "
microseconds." << endl;


    // Тест 6. Пустое множество
    vector<set<int>> sets6 = { {1}, {3, 5} };
    set<int> universe6 = {};

    auto start6 = chrono::high_resolution_clock::now();
    auto result6 = minimum_sets_to_cover(sets6, universe6);
    auto end6 = chrono::high_resolution_clock::now();

    assert((result6 == vector<set<int>>{}));
    cout << "Test 6 passed in " <<
chrono::duration_cast<chrono::microseconds>(end6 - start6).count() << "
microseconds." << endl;

    cout << "All tests have been passed" << endl;
}

// Функция для генерации подмножеств и измерения времени выполнения
void measure_performance() {
    cout << "Measuring efficiency" << endl;

    const int universe_size = 25;
    set<int> universe;

    // Заполняем множество universe числами от 1 до 25
    for (int i = 1; i <= universe_size; ++i) {
```

```
        universe.insert(i);
    }

    // Размеры подмножеств для тестирования
    vector<int> sizes = { 1, 5, 10, 15, 20, 25 };

    for (int size : sizes) {
        vector<set<int>> sets;

        // Генерация подмножеств
        random_device rd;
        mt19937 gen(rd());
        uniform_int_distribution<> dis(1, universe_size);

        for (int i = 0; i < size; ++i) {
            set<int> subset;
            int subset_size = dis(gen) % (universe_size / 2) + 1; //
Размер подмножества от 1 до 12

            while (subset.size() < subset_size) {
                subset.insert(dis(gen)); // Добавляем случайные элементы
из universe
            }

            sets.push_back(subset);
        }

        auto start = chrono::high_resolution_clock::now();
        vector<set<int>> result = minimum_sets_to_cover(sets, universe);
        auto end = chrono::high_resolution_clock::now();

        double elapsed_time =
chrono::duration_cast<chrono::microseconds>(end - start).count();

        cout << "Size: " << size << ", Time: " << elapsed_time / 1e6 << "
seconds" << endl;

        // Выводим ответ
        cout << "Result: ";
        if (!result.empty()) {
            cout << "{ ";
            for (const auto& s : result) {
                cout << "{ ";
                for (const auto& elem : s) {
                    cout << elem << " ";
                }
                cout << "} ";
            }
            cout << "}" << endl;
        }
        else {
```

```
            cout << "No solutions" << endl;
        }
    }
    cout << "End" << endl;
}



int main() {
    run_tests();
    cout << endl;
    measure_performance();
    return 0;
}
```