ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчёт по лабораторной работе № 4

«NP-полные задачи. Задача на покрытие множеств.»

Выполнила работу

Таволжанская Полина Александровна

Академическая группа №С3100

Санкт-Петербург

2024

1. Введение

**Цель**: изучение влияние сложности алгоритмов для решения NP-полных задач на эффективность их выполнения.

**Задачи**:

Ознакомиться с понятием NP-полных задач, их свойствами и примерами. Понять, почему такие задачи сложно решать и какие подходы используются для их оптимизации.

Реализовать решение задачи о покрытии множеств, которая формулируется следующим образом: дан набор множеств B и множество A, необходимо выбрать минимальное количество множеств B, чтобы покрыть все элементы из множества A.

Применить рекурсивные подходы и итераторы для генерации всех возможных комбинаций подмножеств, что позволит глубже понять комбинаторные аспекты задачи.

Провести эксперимент по измерению времени выполнения разработанного алгоритма в зависимости от размера входных данных. Сравнить результаты для различных наборов множеств и выявить влияние количества подмножеств на время выполнения.

Рассмотреть возможные способы оптимизации алгоритма.

Зафиксировать результаты экспериментов, сделать выводы о сложности алгоритмов и их применимости к реальным задачам.

1. Теоретическая подготовка

Типы данных:

Множества (set): Используются для хранения уникальных элементов и предоставляют эффективные операции проверки вхождения и объединения. В C++ для реализации множеств используется контейнер std::set, который обеспечивает логарифмическое время доступа.

Векторы (vector): Используются для хранения последовательностей элементов. В C++ контейнер std::vector предоставляет динамический массив, который позволяет эффективно добавлять и удалять элементы.

Алгоритмы:

Алгоритм полного перебора: Для решения задачи о покрытии множеств применяется полный перебор всех возможных комбинаций подмножеств. Этот подход имеет экспоненциальную сложность O(2^n), где n — количество подмножеств.

Рекурсия: Используется для генерации всех возможных подмножеств и их комбинаций. Рекурсивные функции позволяют удобно обрабатывать вложенные структуры данных.

Проверка покрытия: Для проверки того, покрывает ли набор подмножеств все элементы из множества A, используется алгоритм, который объединяет все элементы подмножеств и проверяет наличие всех элементов из A в этом объединении.

1. Реализация

В этом разделе описывается процесс выполнения лабораторной работы, включая этапы, выполненные действия и особенности реализации.

Определение структуры данных: для реализации задачи о покрытии множеств были определены два основных типа данных — множество (set) для хранения элементов и вектор (vector) для хранения подмножеств, функция при запуске требует две переменные: вектор из множеств (подмножеств будущих комбинаций, составленных далее в generateSubsets()) и множество, которое необходимо покрыть.

Реализация функции проверки покрытия: создана функция covers(), которая принимает множество A и вектор подмножеств B. Функция объединяет все элементы подмножеств и проверяет, содержатся ли все элементы A в этом объединении.

Генерация всех комбинаций подмножеств: реализована рекурсивная функция generateSubsets(), которая генерирует все возможные комбинации подмножеств из заданного вектора

Основная функция: в функции CoveringSets\_comb() реализовано основное логическое построение алгоритма — перебор всех комбинаций подмножеств с последующей проверкой на покрытие. Тестирование: проведено тестирование алгоритма на различных наборах данных с целью оценки его производительности.

1. Экспериментальная часть

В этом разделе представлены результаты работы алгоритма с различными условиями и наборами данных. Результаты включают таблицы и графики, демонстрирующие выполнение алгоритма.

Асимптотическая сложность:

O(n \* 2^n + 2^n \* (n \* k + m)) = O(2^n \* n) = **O(2^n)**

Память:

**O(2^n \* n)**

Таблица №1 - Результаты тестирования алгоритма

| N подмножеств | Время выполнения (мс) | Ожидаемая сложность (O(2^N)) |
| --- | --- | --- |
| 1 | 0.0751 | O(2) |
| 2 | 0.2027 | O(4) |
| 5 | 1.976 | O(2^5) |
| 10 | 125.7 | O(2^10) |
| 12 | 3512.085 | O(2^12) |
| 14 | 7208.59 | O(2^14) |
| 15 | 119108.59 | O(2^15) |
| 20 | 451456.01 | O(2^20) |
| 25 | 14446392 | O(2^25) |

Изображение №\* 1- График работы алгоритма

В силу особенностей работы вычислительной машины, на которой производилось тестирование, время выполнения алгоритма при небольших размерах входных данных может быть неожиданным, однако общая картина временных результатов показывает, что временная сложность находится в рамках от O(2^N) до O(3^N). Результаты (временные) работы программы наглядно представлены на изображении №1.

1. Заключение

В ходе выполнения лабораторной работы мной был реализован алгоритм нахождения всех подмножеств множества для решения задачи о покрытии множества. Задачи работы были достигнуты путём написания двух алгоритмов: полного перебора (комбинаторного) рекурсией и жадного алгоритма для последующего подсчёта асимптотики и, в результате, формирования понимания важности написания кода с учётом сложности используемых алгоритмов. Также задачи лабораторной работы были достигнуты через экспериментальные запуски кода при различных входных данных и оценке влияния сложности алгоритма на время выполнения программы.

В качестве дальнейших исследований можно предложить оптимизацию алгоритма с точки зрения уменьшения затрат использования памяти, а также исследование влияния на время выполнения программу различных данных (например, не только входных подмножеств, но и входного множества для покрытия). В результате выполнения лабораторной работы удалось изучить влияние сложности алгоритмов для решения NP-полных задач на эффективность их выполнения.

1. Приложения

ПРИЛОЖЕНИЯ: КОД

Функция covers()

//функция для проверки на покрытие через includes

bool covers(const set<int>& universe, const vector<set<int>>& subsets) {

    set<int> joined\_subsets;

    for (const set<int>& subset : subsets) {

        joined\_subsets.insert(subset.begin(), subset.end());

    }

    return includes(joined\_subsets.begin(), joined\_subsets.end(), universe.begin(), universe.end());

}

Функция generateSubsets()

\*/функция для генерации всех возмоных комбинаций подмножеств для subsets, рекурсивная\*/

void generateSubsets(const vector<set<int>>& subsets, vector<set<int>>& current, int index, set<vector<set<int>>>& all\_subsets) {

    if (index == subsets.size()) { // все подмножества в наборе обработаны

        all\_subsets.insert(current);

        return;

    }

    current.push\_back(subsets[index]);

    generateSubsets(subsets, current, index + 1, all\_subsets); // снова вызов с включением текущего подмножества

    current.pop\_back();

    generateSubsets(subsets, current, index + 1, all\_subsets); //  без включения текущего подмножества.

}

Функция CoveringSets\_Comb()

set<vector<set<int>>> CoveringSets\_comb(const vector<set<int>>& subsets, const set<int>& universe) {

    set<vector<set<int>>> all\_subsets;

    vector<set<int>> current;

    generateSubsets(subsets, current, 0, all\_subsets);

    set<vector<set<int>>> covering\_sets;

    int min\_size = 10666; // рандомное большое на первые сравнния

    for (const vector<set<int>>& subset\_combination : all\_subsets) {

        if (covers(universe, subset\_combination)) {

            if (subset\_combination.size() < min\_size) {

                min\_size = subset\_combination.size();

                covering\_sets.clear(); // удаляю предыдущие минимальные

                covering\_sets.insert(subset\_combination);

            }  // отсуюда память O(2^n \* m(=n) \* k) в худшем случае (если мы сохраняем все минимальные подмножества)

        }

    }

    return covering\_sets;

}