ГУАП

КАФЕДРА № 42

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| доцент |  |  |  | В. А. Кузнецов |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 4.2 |
| РАЗРАБОТКА АЛГОРИТМА ИСЧЕРПЫВАЮЩЕГО ПОИСКА (ПОЛНОГО ПЕРЕБОРА) |
| по курсу: |
| ОСНОВЫ ПРОГРАММИРОВАНИЯ |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ гр. № | 4326 |  |  |  | Г. С. Томчук |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2024

СОДЕРЖАНИЕ

[1 Постановка задачи 3](#__RefHeading___Toc801_316780651)

[2 Схема алгоритма решения 4](#__RefHeading___Toc809_316780651)

[3 Полное описание реализованных функций 5](#__RefHeading___Toc807_316780651)

[3.1 unite\_skills 5](#__RefHeading___Toc3907_3112333877)

[3.2 find\_min\_set\_cover 5](#__RefHeading___Toc3913_3112333877)

[3.3 main 6](#__RefHeading___Toc3909_3112333877)

[4 Листинг программы 7](#__RefHeading___Toc805_316780651)

[5 Результаты тестирования программы 9](#__RefHeading___Toc803_316780651)

[6 Результаты измерения времени работы и оценки сложности алгоритма 10](#__RefHeading___Toc803_3167806511)

1. Постановка задачи

Задача: реализовать алгоритм на языке C/С++, выполняющий поставленную задачу. Алгоритм должен быть реализован путем перебора всех возможных вариантов решения. Вариант задания, пример входных и выходных данных представлен в таблице 1. Глобальные параметры использовать запрещено; допустимо использование дополнительных функций.

* Разработанный алгоритм должен быть реализован в виде цельной программной функции (или нескольких функций) так, чтобы мог быть многократно применим с различными исходными данными и при этом не включал команды, не относящиеся к решаемой задаче, например, ввод и вывод исходных данных на консоль или в файл.
* Произвести экспериментальную проверку времени работы разработанного алгоритма, определив его класс сложности для среднего случая. Измерить среднее время для Test\_Count повторений при различных размерностях входных данных.

Таблица 1 – Вариант

|  |  |  |  |
| --- | --- | --- | --- |
| N | Текст задания | Вход | Выход |
| 2 | **Данные:**   * Множество претендентов на вакансию Pi, количество претендентов N. * Каждый претендент обладает множеством навыков {Sj}, j – индекс навыка. * Количество не повторяющихся общих навыков M. * У разных претендентов могут быть пересекающиеся множества навыков. * Любой навык из общего множества навыков присутствует по крайней мере в одном множестве навыков претендента.   **Задача:**  Выбрать как можно меньше претендентов таким образом, чтобы все навыки были охвачены. |  |  |

1. Схема алгоритма решения

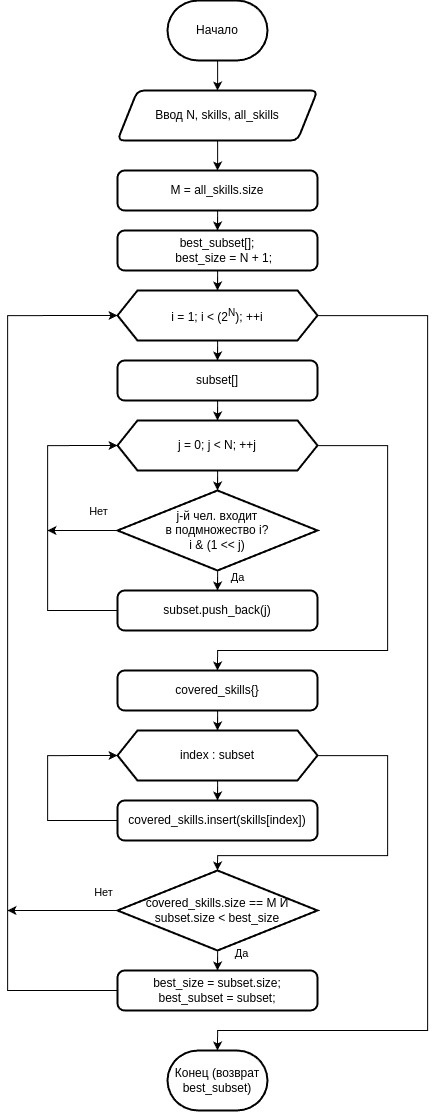


Рисунок 1 – Блок-схема алгоритма

1. Полное описание реализованных функций
   1. unite\_skills

Функция unite\_skills объединяет навыки нескольких претендентов в одно множество. Принимает следующие аргументы:

1. const std::vector<std::set<std::string>>& skills — вектор, содержащий множества навыков всех претендентов. Каждое множество представляет навыки одного претендента.
2. const std::vector<int>& indices — вектор, содержащий индексы претендентов, навыки которых нужно объединить.

Возвращает std::set<std::string> — множество, содержащее все уникальные навыки претендентов, указанных в indices. Работа функции происходит следующим образом:

1. Создается пустое множество result.
2. Проходит по каждому индексу в indices.
3. Для каждого индекса добавляет все навыки соответствующего претендента в множество result.
4. Возвращает объединенное множество всех навыков.
   1. find\_min\_set\_cover

Функция find\_min\_set\_cover находит минимальное подмножество претендентов, которое покрывает все уникальные навыки. Принимает следующие аргументы:

1. const std::vector<std::set<std::string>>& skills — вектор, содержащий множества навыков всех претендентов.
2. int M — количество уникальных навыков.

Возвращает std::vector<int> — вектор, содержащий индексы претендентов, составляющих минимальное подмножество, покрывающее все уникальные навыки. Работа функции происходит следующим образом:

1. Определяет количество претендентов N.
2. Инициализирует пустой вектор best\_subset для хранения лучшего подмножества.
3. Устанавливает best\_size на значение большее, чем количество претендентов (для сравнения).
4. Перебирает все возможные подмножества претендентов (от 1 до 2N - 1 — общее количество подмножеств множества из N элементов без пустого подмножества). Число i используется как битовая маска для представления подмножества претендентов.
5. Для каждого подмножества создает вектор subset, содержащий индексы претендентов, входящих в подмножество. Условие if (i & (1 << j)) проверяет, включен ли j-й претендент в подмножество, соответствующее числу i.
6. Проверяет, покрывают ли выбранные претенденты все уникальные навыки, используя функцию unite\_skills.
7. Если подмножество покрывает все навыки и его размер меньше текущего лучшего размера, обновляет best\_size и best\_subset.
   1. main
8. Запрашивает у пользователя количество претендентов N.
9. Создает вектор skills для хранения множеств навыков претендентов.
10. Запрашивает у пользователя навыки каждого претендента, разделенные пробелами, и заполняет вектор skills.
11. Определяет количество уникальных навыков M.
12. Вызывает функцию find\_min\_set\_cover для нахождения минимального подмножества претендентов, покрывающего все навыки.
13. Выводит количество претендентов в минимальном подмножестве и их порядковые номера.
14. Листинг программы

Листинг 1

#include <iostream>  
#include <vector>  
#include <set>  
#include <string>  
#include <sstream>  
  
// Функция для объединения навыков нескольких претендентов  
std::set<std::string> unite\_skills(const std::vector<std::set<std::string>> &skills, const std::vector<int> &indices) {  
 std::set<std::string> result;  
 for (int index : indices)  
 result.insert(skills[index].begin(), skills[index].end());  
 return result;  
}  
  
// Основная функция для решения задачи полного перебора  
std::vector<int> find\_min\_set\_cover(const std::vector<std::set<std::string>> &skills, int M) {  
 int N = skills.size();  
 std::vector<int> best\_subset;  
 int best\_size = N + 1;  
  
 // Перебираем все возможные подмножества претендентов  
 for (int i = 1; i < (1 << N); ++i) {  
 std::vector<int> subset;  
 for (int j = 0; j < N; ++j)  
 if (i & (1 << j)) subset.push\_back(j);  
  
 // Проверяем, покрывают ли выбранные претенденты все навыки  
 std::set<std::string> covered\_skills = unite\_skills(skills, subset);  
 if (covered\_skills.size() == M && subset.size() < best\_size) {  
 best\_size = subset.size();  
 best\_subset = subset;  
 }  
 }  
  
 return best\_subset;  
}  
  
int main() {  
 int N;  
 std::cout << "Введите количество претендентов: ";  
 std::cin >> N;  
 std::cin.ignore(); // Для игнорирования символа новой строки после ввода числа  
  
 std::vector<std::set<std::string>> skills(N);  
 std::set<std::string> all\_skills;  
  
 for (int i = 0; i < N; ++i) {  
 std::cout << "Введите навыки претендента " << i + 1 << " через пробел: ";  
 std::string line;  
 getline(std::cin, line);

Продолжение листинга 1

std::stringstream ss(line);  
 std::string skill;  
 while (ss >> skill) {  
 skills[i].insert(skill);  
 all\_skills.insert(skill);  
 }  
 }  
  
 int M = all\_skills.size(); // Количество уникальных навыков  
  
 std::vector<int> result = find\_min\_set\_cover(skills, M);  
  
 std::cout << "Минимальное количество претендентов для покрытия всех навыков: " << result.size() << std::endl;  
 std::cout << "Претенденты: ";  
 for (int index : result) {  
 std::cout << index + 1 << " ";  
 }  
  
 return 0;  
}

1. Результаты тестирования программы

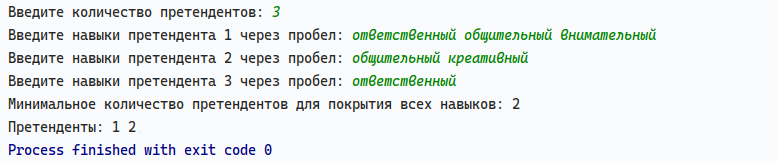


Рисунок 2

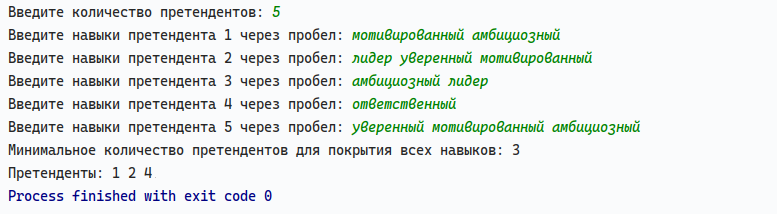


Рисунок 3

1. Результаты измерения времени работы и оценки сложности алгоритма

Количество всего возможных подмножеств множества из N элементов — 2N. Для каждого подмножества необходимо объединить навыки претендентов и проверить их количество. Объединение навыков для каждого подмножества требует O(N) времени, так как нужно просмотреть до N претендентов и объединить их навыки. Проверка покрытия всех навыков требует проверки размера объединенного множества навыков, что требует O(1) времени. Таким образом, общая временная сложность алгоритма: O(2N ⋅ N).

Для экспериментальной проверки времени работы были написаны две дополнительные функции: generate\_random\_skills — генерирует случайное количество случайно выбранных навыков для каждого претендента, test\_find\_min\_set\_cover — замеряет и выводит среднее время выполнения. Для измерения времени была использована библиотека <chrono>. Чтобы получить среднее время выполнения, функция вызывается test\_count=5 раз.

Листинг 2

// Функция для генерации случайных данных  
std::vector<std::set<std::string>> generate\_random\_skills(int N, int M) {  
 std::vector<std::string> all\_skills(M);  
 std::vector<std::set<std::string>> skills(N);  
  
 for (int i = 0; i < M; ++i)  
 all\_skills[i] = "skill" + std::to\_string(i);  
  
 std::random\_device rd;  
 std::mt19937 gen(rd());  
 std::uniform\_int\_distribution<> skill\_quantity\_dist(1, M);  
 std::uniform\_int\_distribution<> skill\_index\_dist(0, M - 1);  
  
 for (int i = 0; i < N; ++i) {  
 int skills\_quantity = skill\_quantity\_dist(gen);  
 for (int j = 0; j < skills\_quantity; ++j)  
 skills[i].insert(all\_skills[skill\_index\_dist(gen)]);  
 }  
  
 return skills;  
}  
  
void test\_find\_min\_set\_cover() {  
 const int test\_count = 5;  
 std::vector<int> sizes = {5, 6, 7, 8, 9, 10, 15,  
 20, 21, 22, 23, 24}; // Размерности входных данных (количество претендентов)

Продолжение листинга 2

for (int size : sizes) {  
 int N = size;  
 int M = 10; // Количество уникальных навыков  
  
 double total\_duration = 0.0;  
  
 for (int i = 0; i < test\_count; ++i) {  
 std::vector<std::set<std::string>> skills = generate\_random\_skills(N, M);  
  
 auto start = std::chrono::high\_resolution\_clock::now();  
 std::vector<int> result = find\_min\_set\_cover(skills, M);  
 auto end = std::chrono::high\_resolution\_clock::now();

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

total\_duration += duration.count();  
 }  
  
 double average\_duration = total\_duration / test\_count;  
 std::cout << "Среднее время для N = " << N << ": " << std::fixed << std::setprecision(10) << average\_duration  
 << " секунд." << std::endl;  
 }  
}

Результаты измерения среднего времени работы представлены на рисунке 4. Из измеренного времени видно, что при незначительном увеличении количества претендентов N, время выполнения стремительно увеличивается. На рисунке 5 изображены графики, из которых видно, что экспериментально определенная сложность близко соотносится с теоретической.

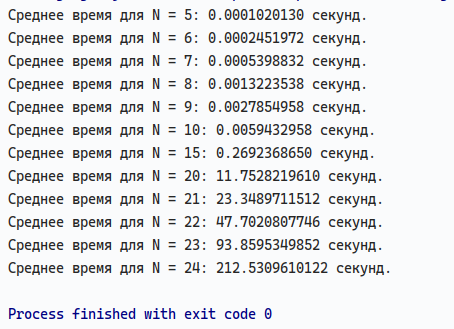


Рисунок 4

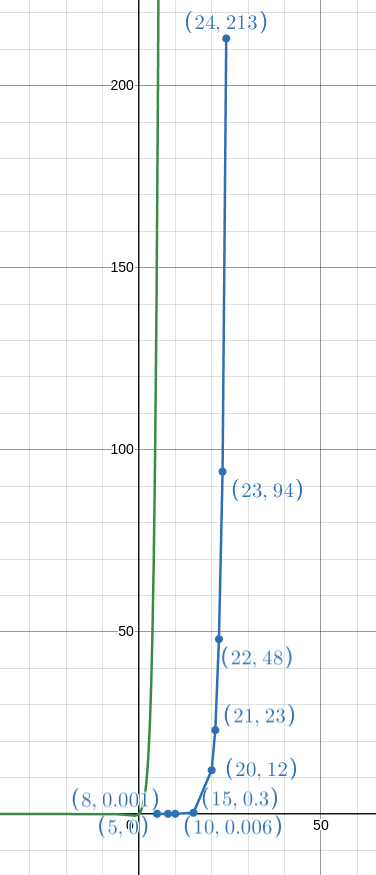


Рисунок 5 - Зеленый график — 2x ⋅ x