

Санкт-Петербург 2021

Содержание

Содержание	2
1. Постановка Задачи	3
2. Алгоритм	4
2.1 Описание переборного алгоритма.....	4
2.2 Реализация переборного алгоритма	5
2.3 Описание жадного алгоритма	6
2.4 Реализация жадного алгоритма.....	8
3. Инструкция пользователя	9
4. Текстовые примеры	9
5. Список источников	11

1. Постановка Задачи

Задачей данной курсовой работы является разработка программы, которая решает задачу о назначениях методом полного перебора и с помощью жадного алгоритма. Задача состоит в отыскании назначения, при котором каждая работа выполняется некоторым человеком и которое имеет минимальную стоимость.

На вход программы подается матрица работник-работа, в которой номер строки - это номер работника, а числа в данной строке это стоимость назначения данного работника на одну из работ (номер столбца – это номер работы). Один работник может быть назначен только на одну из работ, следовательно, в одной строке и одном столбце не может быть несколько назначений.

Подробнее постановка задачи описана в книге – [1]

Покажем, что задача имеет смысл на примере:

Возьмем следующую матрицу:

Составим два назначения:

2	3	1
4	5	6
3	1	2

Назначение 1 -

2	3	W
W	5	6
3	W	2

Буквой W будем обозначать назначенного работника (от слова Worker)

В таком случае суммарная стоимость назначений будет равна $S = 1 + 4 + 1 = 6$

Назначение 2 –

2	W	1
W	5	6
3	1	W

В таком случае суммарная стоимость назначений будет равна $S = 3 + 4 + 2 = 9$

Первое назначение имеет стоимость меньше, чем второе назначение, следовательно, данная задача имеет смысл.

2. Алгоритм

2.1 Описание переборного алгоритма

Переборный алгоритм реализован на основе рекурсии. Структуры данных, использованные в данном алгоритме, следующие:

```
std::vector<std::vector<int>> Matrix;  
int min_summ;  
std::vector<int> min_comb;  
std::vector<int> comb;  
std::vector<int> used_idx;
```

Рис. 1 – Структуры данных, используемые в переборном алгоритме

`std::vector<std::vector<int>> Matrix` – исходная матрица работник-работа.

`int min_summ` – запоминаем сюда стоимость минимального назначения.

`std::vector<int> min_comb` – запоминаем сюда индексы нашей матрицы, на которых располагаются элементы, входящие в минимальное назначение.

`std::vector<int> comb` – здесь хранится текущая комбинация индексов.

`std::vector<int> used_idx` – здесь хранятся уже использованные индексы в матрице(с помощью этого вектора проверяем, чтобы в одной строке и одном столбце не было несколько назначений).

Шаги для реализации переборного алгоритма следующие:

1. Проходим по строкам матрицы работник-работа и помечаем индексы назначений. Попутно проверяем не использовали ли мы ранее данные индексы. После того, как назначили работника в одной строке – переходим на следующий уровень рекурсии, т.е. переходим к следующей строке матрицы и назначаем следующего работника.
2. После того, как прошли по всем строкам матрицы и назначили всех работников, высчитываем сумму получившегося назначения, если она меньше, чем минимальная сумма, то запоминаем получившуюся комбинацию как минимальную и запоминаем как промежуточное решение.
3. В конце сохраняем нашу матрицу с помеченными назначениями и сумму минимального назначения в текстовый файл.

Рассмотрим пошаговое выполнение алгоритма на простом примере:

Возьмем матрицу

	2	3	1
	4	5	6
Комбинация 1	3	1	2

После первого прохода по матрице будет записано мы назначили работников на позиции [0][0], [1][1], [2][2], а промежуточная сумма будет равна $S = 2 + 5 + 2 = 9$, т.к. это первая комбинация, то эта сумма и эта комбинация будет минимальной.

Комбинация 2

Опускаемся на один уровень рекурсии (вторая строка матрицы) и берем индекс 2, переходим на следующую строку(третью) и берем индекс 1. Теперь мы еще раз прошлись по всем строкам матрицы и назначили работников на позиции [0][0] [1][2] [2][1], а промежуточная сумма будет равна $S = 2 + 6 + 3 = 12$. Сравнивая с минимальной суммой ($S = 9$) мы понимаем, что данная комбинация больше по стоимости и отбрасываем ее.

И так далее мы перебираем все возможные комбинации и находим минимальную. Хотя переборный алгоритм имеет большую сложность, но при этом всегда выдает оптимальное решение, в отличие от жадного алгоритма.

Итоговое решение:

Total summ = 6

2	3	W
W	5	6
3	W	2

Track:

(1;3) W = 1
(2;1) W = 4
(3;2) W = 1

Рис. 3 – Итоговое решение для тестового примера работы переборного алгоритма

Буквами W обозначены назначения работников.

2.2 Реализация переборного алгоритма

```
class AssignmentP{
private:

    int min_summ;
```

```

std::vector<int> min_comb;
std::vector<int> comb;
std::vector<std::vector<int>> Matrix;
std::vector<int> used_idx;

public:
friend bool check_used_idx(int idx, std::vector<int>& vec, int size); // для проверки использованных
индексов
void BruteForce(int idx, std::vector<int>& bruteforce, int work_idx);
void load(const std::string fname);
void save(const std::string fname);

}

void AssignmentP::BruteForce(int idx, std::vector<int>& bruteforce, int work_idx) {
    if (work_idx == this->size) {
        Суммируем полученные назначения
        Сравниваем с минимальной суммой, если меньше ее, то обновляем int min_summ
    }
    else {
        for (int j = 0; j < this->work[work_idx].size(); j++) {
            bruteforce[idx] = this->work[work_idx][j];
            if (Индекс в исходной матрице не был использован ранее) {
                this->used_idx.push_back(this->work[work_idx][j]);
                BruteForce(idx + 1, bruteforce, work_idx + 1);
            }
        }
    }
}

```

Метод load загружает данные из текстового файла. Метод save записывает получившийся результат в текстовый файл

Сверху приведен код реализации алгоритма полного перебора. Все переменные и структуры данных описаны в пункте 2.1. (Описание переборного алгоритма)

Сложность переборного алгоритма будет равна $O(N * N!)$, где N – размер нашей матрицы. Такую сложность мы получаем, т.к. в каждой строке имеем по $N!$ возможных перестановок, а всего строк N .

2.3 Описание жадного алгоритма

Идея жадного алгоритма заключается в том, проходя по строкам матрицы работник-работа, он выбирает наименьшее назначение по стоимости в надежде на то, что результат будет оптимальным.

Структуры данных, использованные в жадном алгоритме следующие:

```
int min_summ;  
std::vector<std::vector<int>> Matrix;  
std::vector<int> used_idx;
```

Рис. 3 – Структуры данных, используемые в жадном алгоритме

`int min_summ` – хранит в себе общую стоимость назначения работников

`std::vector<std::vector<int>> Matrix` – исходная матрица работник-работа.

`std::vector<int> used_idx` – здесь хранятся уже использованные индексы в матрице(с помощью этого вектора проверяем, чтобы в одной строке и одном столбце не было несколько назначений).

Шаги для реализации жадного алгоритма следующие:

1. Ищем минимальную стоимость назначения в строке, если эта работа не занята каким-либо работником, то назначаем туда работника и прибавляем стоимость данного назначения к текущей сумме
2. Если минимальная по стоимости назначения работа в данной строке матрицы уже занята, то ищем следующую по возрастанию стоимости назначения работу в данной строке и так до тех пор, пока не назначим работника
3. В конце сохраняем нашу матрицу с помеченными назначениями и сумму минимального назначения в текстовый файл.

Рассмотрим пошаговое выполнение алгоритма на простом примере:

Возьмем матрицу:

2	3	1
4	5	3
3	1	2

Шаг 1.

В первой строке исходной матрицы минимальное число это 1, оно расположено на позиции [0][2]. В первом столбце нет назначенных работников, следовательно, помечаем данную работу как занятую и прибавляем 1 к промежуточной сумме. Переходим к следующей строке матрицы.

Шаг 2

Во второй строке исходной матрицы минимальное число 3, но в третьем столбце уже имеется назначенный работник, следовательно, ищем другое число (это 4). Оно находится на позиции [1][0]. В первом столбце нет назначенных работников, следовательно, помечаем данную работу как занятую и прибавляем 4 к промежуточной сумме. Переходим к третьей строке матрицы.

Шаг 3

В третьей строке минимальное число 1. Во втором столбце нет назначенных работников, следовательно, помечаем данную работу как занятую и прибавляем 1 к промежуточной сумме

Шаг 4

Все работники назначены, общая стоимость назначения равна 6 ($1 + 4 + 1$). Выводим итоговую сумму и матрицу работник-работа в текстовый файл.

Итоговое решение:

```
Total summ = 6

      2      3      W
W      5      3
3      W      2

Track:
(1;3)  W = 1
(2;1)  W = 4
(3;2)  W = 1
```

Рис. 4 – Итоговое решение для тестового примера работы жадного алгоритма

2.4 Реализация жадного алгоритма

```
class AssignmentP {
private:
    int min_summ;
    std::vector<std::vector<int>>> Matrix;

    std::vector<int> used_idx;

public:
    void GreedyAlg();
```



```

    void load(const std::string fname);
    void save(const std::string fname);
    friend bool check_used_idx(int idx, const AssignmentP& pr, const std::vector<int>& itstr); // для
    проверки использованных индексов
}

void AssignmentP::GreedyAlg() {
    std::vector<int>::iterator it_min; // указывает на минимальное число в строке матрицы

    for(auto& it: this->Matrix) {
        it_min = std::min_element(it.begin(), it.end()); // ищем минимальный элемент в строке

        while (Проверяем использовался ли индекс ранее) {
            Помечаем индекс как недоступный
            it_min = std::min_element(it.begin(), it.end()); // ищем новый минимальный элемент
        }
        this->used_idx.push_back(min_idx); // помечаем наш индекс как использованный
        this->min_summ += it.at(min_idx); // добавляем стоимость назначения в итоговую сумму
    }
}

```

Метод load загружает данные из текстового файла. Метод save записывает получившийся результат в текстовый файл

Сложность жадного алгоритма будет равна $O(N)$, где N – размер исходной матрицы работник-работа. Такую сложность мы получаем, т.к. в каждой строке мы делаем по две операции: ищем минимальный элемент и прибавляем его к общей стоимости назначения – сложность $O(1)$, таких операций мы делаем N штук, следовательно, общая сложность $O(N)$.

3. Инструкция пользователя

Пользователю необходимо создать текстовый документ с расширением .txt, в который он записывает квадратную матрицу работник-работа.

Запустив программу, в консоли нужно указать путь файла с исходными данными и путь, по которому создастся <выходной_файл>.txt.

В консоли это будет выглядеть так:

```
D:\VS repositories\coursework\Debug> coursework input.txt output.txt
```

Где *input.txt* – это файл, в который пользователь записывает матрицу работник-работа, а *output.txt* – это файл, в который запишется результат выполнения программы


4. Текстовые примеры

Пример 1 (Жадный алгоритм)

Консоль:

```
D:\VS repositories\coursework\Debug> coursework input.txt output.txt
```

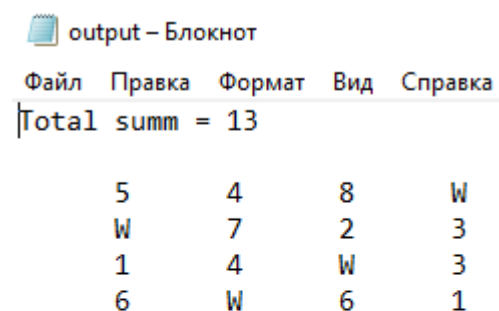
Входной файл:



Файл	Правка	Формат	Вид
5	4	8	1
2	7	2	3
1	4	3	3
6	7	6	1

Рис. 5 – Входной файл (пример 1)

Выходной файл:



Файл	Правка	Формат	Вид	Справка
Total summ = 13				
5	4	8	W	
W	7	2	3	
1	4	W	3	
6	W	6	1	

Track:

(1;4) W = 1
(2;1) W = 2
(3;3) W = 3
(4;2) W = 7

Рис. 6 – Выходной файл (пример 1)

Пример 2 (Полный перебор)

Консоль:

```
D:\VS repositories\coursework\Debug> coursework input.txt output.txt
```

Входной файл:



Файл	Правка	Формат	Вид
5	4	8	1
2	7	2	3
1	4	3	3
6	7	6	1

Рис. 7 – Входной файл (пример 2)



output – Блокнот

Файл	Правка	Формат	Вид	Справка
Total summ = 8				
5	W	8	1	
2	7	W	3	
W	4	3	3	
6	7	6	W	

Track:

(1;2) W = 4
(2;3) W = 2
(3;1) W = 1
(4;4) W = 1

Выходной файл:

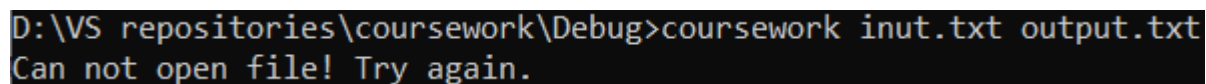
Рис. 8 – Выходной файл (пример 2)

Пример 3 (Некорректные параметры командной строки)

Консоль:

```
D:\VS repositories\coursework\Debug> coursework inut.txt output.txt
```

Результат выполнения:



```
D:\VS repositories\coursework\Debug>coursework inut.txt output.txt  
Can not open file! Try again.
```

Рис. 9 – Вывод ошибки в консоли

5. Список источников

Книги:

- [1] А. Кофман, Введение в прикладную комбинаторику, Наука, 1975 г.