МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«МИРЭА – Российский технологический университет»

РТУ МИРЭА

Отчет по выполнению практического задания № 8, часть 2

Тема:

«Реализация алгоритмов на основе сокращения числа переборов»

Дисциплина: «Структуры и алгоритмы обработки данных»

Выполнил студент: Лисовский И.В

Группа: ИКБО-21-23

Москва – 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ 3

2 ЗАДАНИЕ 4

2.1 Формулировка задачи 4

2.2 Решение 4

2.3 Код программы 11

3 ВЫВОД 19

4 ЛИТЕРАТУРА 22

1 ЦЕЛЬ

Разработать и реализовать алгоритмы для решения задачи раскраски вершин графа с акцентом на сокращение числа переборов, необходимых для нахождения оптимального решения. В частности, необходимо применить данные алгоритмы к практической задаче управления светофорами на сложном перекрёстке.

2 ЗАДАНИЕ

2.1 Формулировка задачи

Необходимо решить задачу о раскраске вершин графа и применить ее к задаче управления светофорами на сложном перекрестке. Цель состоит в том, чтобы определить минимальное количество фаз светофора, при котором будут исключены конфликтующие направления движения.

Вариант №19:

Задача	Метод
Решить задачу о раскраске вершин графа. Применить к задаче управления светофорами на сложном перекрестке. (См. Ахо А., Хопкрофт Д., Ульман Дж. Структуры данных и алгоритмы).	Жадный алгоритм

2.2 Решение

Для решения задачи требуется:

1. Использовать **жадный алгоритм** для раскраски вершин графа, представляющего конфликтующие повороты на перекрестке.
2. Оценить количество переборов при решении задачи стратегией **«в лоб»** (методом грубой силы).
3. Сравнить число переборов при использовании жадного алгоритма и метода грубой силы.

Перекресток моделируется в виде графа, где вершины представляют возможные повороты или движения на перекрестке, а ребра соединяют конфликтующие движения, которые не могут осуществляться одновременно.

Повороты на перекрестке:

Список дорог: A, B, C, D, E

Возможные повороты: AB, AC, AD, BA, BC, BD, DA, DB, DC, EA, EB, EC, ED

Конфликтующие повороты определены заранее и представлены в виде списка для каждого поворота.

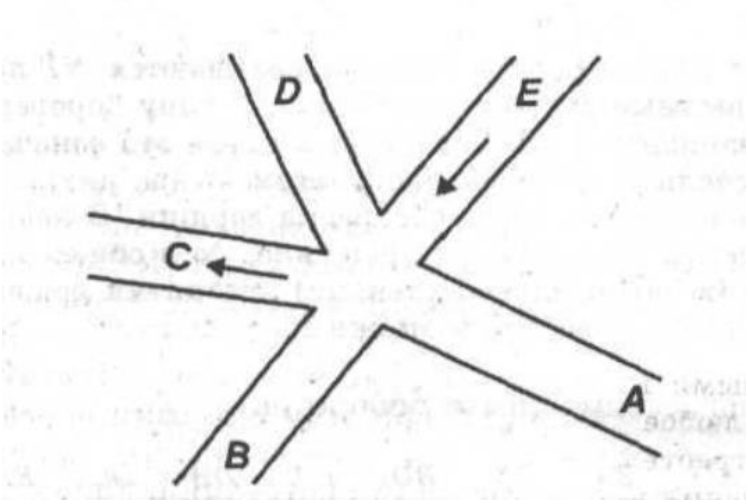

Рисунок 1 – сложный перекресток, представленный для решения задачи

Для построения модели этой задачи можно применить математическую структуру – граф, где вершины будут представлять повороты, а ребра соединят ту часть вершин-поворотов, которые нельзя выполнить одновременно. Для нашего перекрестка (рис. 1) соответствующий граф показан на рис. 2, а на рис. 3 дано другое представление графа — в виде матрицы, где на пересечении строки i и столбца j стоит 1 тогда и только тогда, когда существует ребро между вершинами i и j.

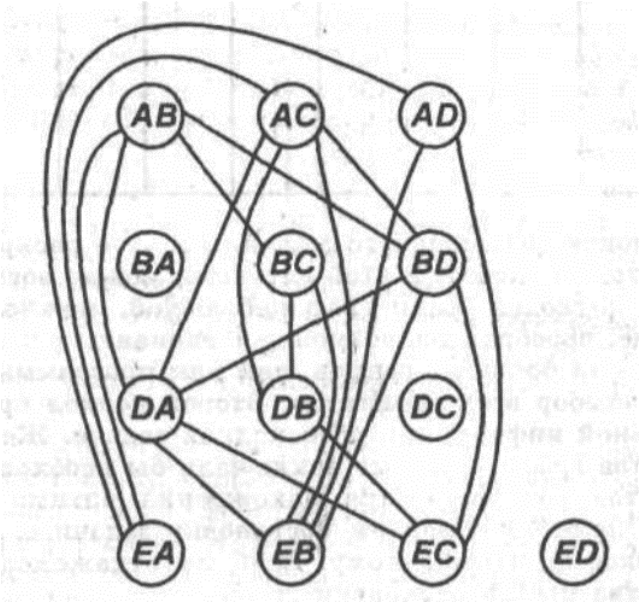

Рисунок 2 – Граф, отображающий конфликты поворотов на перекрестке

матрица смежности (1 означают конфликт дорог на перекрестке)

	AB	AC	AD	BA	BC	BD	DA	DB	DC	EA	EB	EC	ED
AB	0	0	0	0	1	1	1	0	0	1	0	0	0
AC	0	0	0	0	0	1	1	1	0	1	1	0	0
AD	0	0	0	0	0	0	0	0	0	1	1	1	0
BA	0	0	0	0	0	0	0	0	0	0	0	0	0
BC	1	0	0	0	0	0	0	1	0	0	1	0	0
BD	1	1	0	0	0	0	1	0	0	0	1	1	0
DA	1	1	0	0	0	1	0	0	0	0	1	1	0
DB	0	1	0	0	1	0	0	0	0	0	0	1	0
DC	0	0	0	0	0	0	0	0	0	0	0	0	0
EA	1	1	1	0	0	0	0	0	0	0	0	0	0
EB	0	1	1	0	1	1	1	0	0	0	0	0	0
EC	0	0	1	0	0	1	1	1	0	0	0	0	0
ED	0	0	0	0	0	0	0	0	0	0	0	0	0

Рисунок 3 – Матрица смежности, отображающая конфликты поворотов на перекрестке

Жадный алгоритм — алгоритм, заключающийся в принятии локально оптимальных решений на каждом этапе, допуская, что конечное решение также окажется оптимальным. Жадный алгоритм раскраски графа пытается последовательно назначать цвета вершинам, выбирая наименьший возможный цвет, который не конфликтует с уже раскрашенными соседями.

```
Для каждой вершины u в графе:
    Создать список доступных цветов.
    Для каждого соседа v вершины u:
        Если сосед v уже раскрашен в цвет c:
            Исключить цвет c из списка доступных цветов.
    Назначить вершине u наименьший доступный цвет.
```

Листинг 1 – псевдокод «Жадного алгоритма»

Структуры данных для реализации программы:

1. **Список дорог** (`roads`): хранит названия дорог на перекрестке.
2. **Список поворотов** (`turns`): содержит все возможные повороты.
3. **Список конфликтов** (`conflictData`): словарь, где ключ — поворот, а значение — список конфликтующих поворотов.
4. **Список смежности** (`adjList`): для представления графа конфликтов.

Основные функции

- greedyColoring: Реализует жадный алгоритм. Параметры: число вершин, список смежности, вектор для хранения цветов. Возвращает раскраску графа.

```
void greedyColoring(int numTurns, const vector<vector<int>>& adjList,
vector<int>& colors)
{
    colors.assign(numTurns, -1); //инициализировать все цвета как не
назначенные (-1)

    for (int u = 0; u < numTurns; ++u)
    {
        //множество для хранения уже использованных цветов соседей
        unordered_set<int> assignedColors;

        //проверить цвета смежных вершин
        for (int v : adjList[u])
        {
            if (colors[v] != -1)
            {
                assignedColors.insert(colors[v]);
            }
        }

        //найти минимальный доступный цвет
        int color = 0;
        while (assignedColors.find(color) != assignedColors.end())
        {
            ++color;
        }
```

```
            colors[u] = color; //назначаем цвет вершине
        }
    }
```

- graphColoringUtil: Вспомогательная рекурсивная функция для метода грубой силы. Пытается раскрасить граф с заданным числом цветов.

```
bool graphColoringUtil(int vertex, int numTurns, int numColors, const
vector<vector<int>>& adjList, vector<int>& colors)
{
    bruteForceIterations++; //увеличиваем счетчик итераций для метода грубой
силы

    if (vertex == numTurns)
        return true;

    if (showLogs)
    {
        cout << "\nПытаемся раскрасить вершину " << vertex << "\n";
    }

    //пробуем все цвета от 0 до numColors - 1
    for (int color = 0; color < numColors; ++color)
    {
        bool canColor = true;

        //проверяем, конфликтует ли цвет с соседями
        for (int neighbor : adjList[vertex])
        {
            if (colors[neighbor] == color)
            {
                canColor = false;
                break;
            }
            bruteForceIterations++; //счетчик итераций
        }

        if (showLogs2)
        {
            cout << "Пробуем цвет " << color << " для вершины " << vertex <<
": ";
            cout << (canColor ? "подходит" : "не подходит") << "\n";
        }

        if (canColor)
        {
            colors[vertex] = color;
            if (graphColoringUtil(vertex + 1, numTurns, numColors, adjList,
colors))
                return true;
            colors[vertex] = -1; //назначение не привело к решению,
откатываемся

            if (showLogs2)
            {
                cout << "Откатываемся с вершины " << vertex << ", цвет " <<
color << "\n";
            }
        }
    }
```

```
    return false;
}
```

- bruteForceColoring: Реализует метод грубой силы. Итерирует по возможному числу цветов от 1 до числа вершин. Возвращает минимальное число цветов, при котором граф можно корректно раскрасить.

```
int bruteForceColoring(int numTurns, const vector<vector<int>>& adjList,
vector<int>& colors)
{
    colors.assign(numTurns, -1); //инициализируем все цвета как не назначенные
(-1)

    //начинаем с 1 цвета и увеличиваем число цветов, пока не найдем корректную
раскраску
    for (int numColors = 1; numColors <= numTurns; ++numColors)
    {
        bruteForceIterations = 0; //сбрасываем счетчик итераций для каждого
числа цветов

        if (showLogs2)
        {
            cout << "\nПытаемся раскрасить граф с " << numColors << "
цветами\n";
        }

        if (graphColoringUtil(0, numTurns, numColors, adjList, colors))
        {
            return numColors; //найдено минимальное число цветов
        }
    }
    return numTurns; //в худшем случае потребуется столько же цветов, сколько
вершин
}
```

Для подсчета количества итераций, введены глобальные переменные greedyIterations и bruteForceIterations для подсчета числа операций в каждом методе. В жадном алгоритме счетчик увеличивается при просмотре соседей и поиске доступного цвета. В методе грубой силы счетчик увеличивается при каждом рекурсивном вызове и проверке соседей.

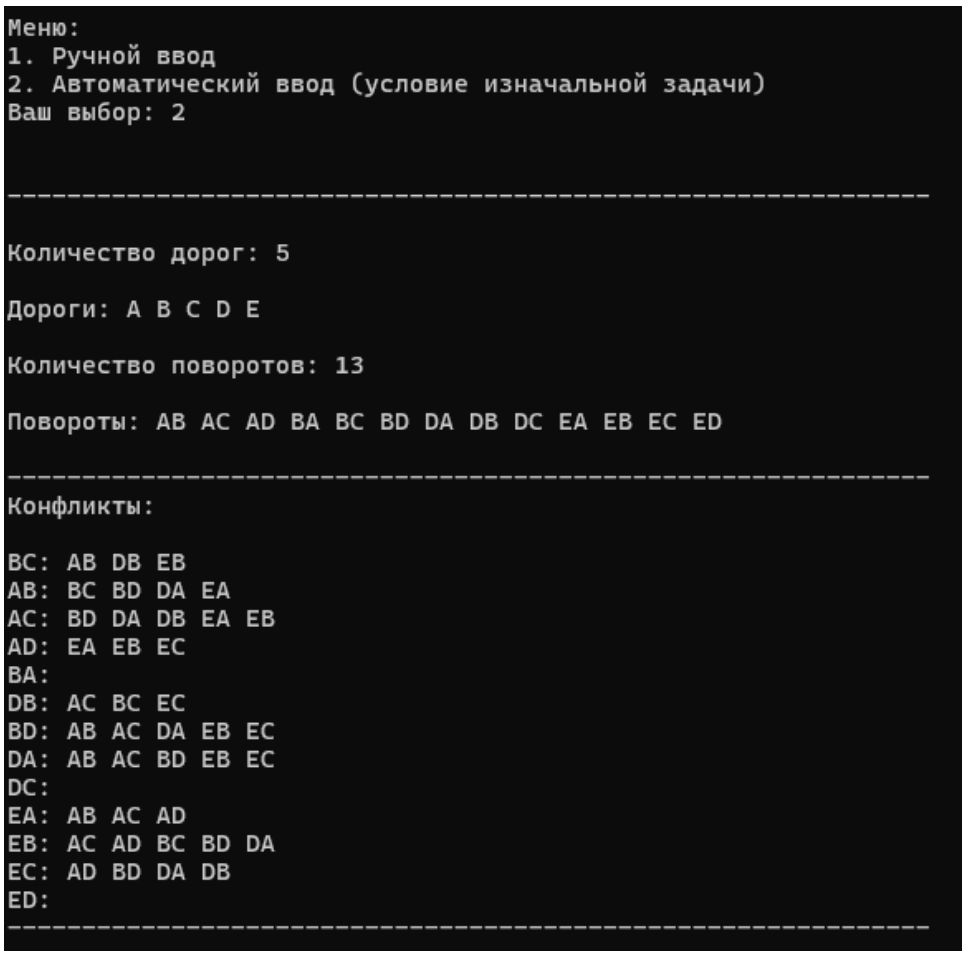

Рисунок 4 – запуск программы с входными значениями – схема дороги, представленная в работе

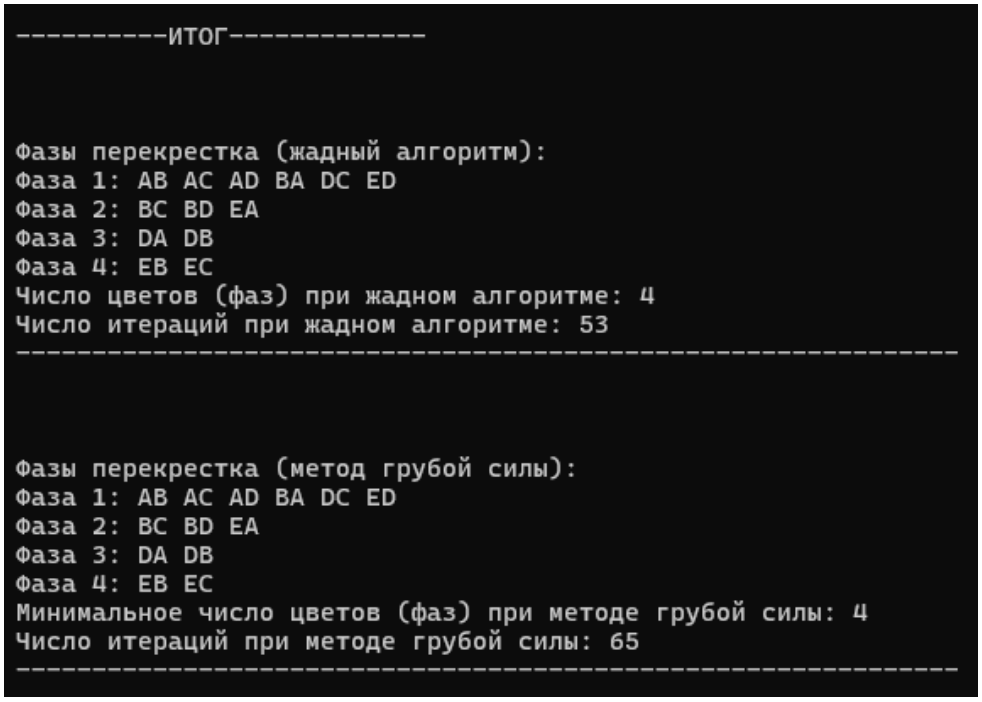

Рисунок 5 – запуск программы с входными значениями – схема дороги, представленная в работе (ч.2)

Оба метода показали, что минимальное число фаз для управления светофорами на данном перекрестке равно 4. Жадный алгоритм выполняет меньше операций благодаря своей природе и отсутствию полного перебора. Метод грубой силы требует большего числа итераций – 65, из-за полного перебора всех возможных комбинаций. В данном случае жадный алгоритм нашел оптимальное решение, что не всегда происходит на практике, и потребовал меньше итераций – 53.

2.3 Код программы

```
#include <iostream>
#include <iomanip>
#include <vector>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <algorithm>

using namespace std;

// Глобальные переменные для подсчета итераций
long long greedyIterations = 0;
long long bruteForceIterations = 0;

// Глобальная переменная для управления выводом логов
bool showLogs = false;
bool showLogs2 = false;

// Функция для выполнения жадной раскраски графа
void greedyColoring(int numTurns, const vector<vector<int>>& adjList, vector<int>&
colors)
{
    colors.assign(numTurns, -1); // Инициализировать все цвета как не назначенные (-
1)

    for (int u = 0; u < numTurns; ++u)
    {
        // Множество для хранения уже использованных цветов соседей
        unordered_set<int> assignedColors;

        // Проверить цвета смежных вершин
        if (showLogs)
        {
            cout << "\nОбрабатываем вершину " << u << "\n";
            cout << "Смежные вершины: ";
            for (int v : adjList[u])
            {
                cout << v << " ";
            }
            cout << "\n";
        }

        for (int v : adjList[u])
        {
```

```
            if (colors[v] != -1)
            {
                assignedColors.insert(colors[v]);
            }
            greedyIterations++; // Увеличиваем счетчик итераций для жадного
алгоритма
        }

        if (showLogs)
        {
            cout << "Использованные цвета соседей: ";
            for (int c : assignedColors)
            {
                cout << c << " ";
            }
            cout << "\n";
        }

        // Найти минимальный доступный цвет
        int color = 0;
        while (assignedColors.find(color) != assignedColors.end())
        {
            ++color;
            greedyIterations++; // Счетчик итераций
        }

        colors[u] = color; // Назначаем цвет вершине

        if (showLogs)
        {
            cout << "Назначенный цвет вершине " << u << ": " << color << "\n";
        }
    }
}

// Функция для проверки возможности раскраски графа с заданным числом цветов
bool graphColoringUtil(int vertex, int numTurns, int numColors, const
vector<vector<int>>& adjList, vector<int>& colors)
{
    bruteForceIterations++; //увеличиваем счетчик итераций для метода грубой силы

    if (vertex == numTurns)
        return true;

    if (showLogs)
    {
        cout << "\nПытаемся раскрасить вершину " << vertex << "\n";
    }

    //пробуем все цвета от 0 до numColors - 1
    for (int color = 0; color < numColors; ++color)
    {
        bool canColor = true;

        //проверяем, конфликтует ли цвет с соседями
        for (int neighbor : adjList[vertex])
        {
            if (colors[neighbor] == color)
            {
                canColor = false;
                break;
            }
            bruteForceIterations++; //счетчик итераций
        }
```

```
        if (showLogs2)
        {
            cout << "Пробуем цвет " << color << " для вершины " << vertex << ": ";
            cout << (canColor ? "подходит" : "не подходит") << "\n";
        }

        if (canColor)
        {
            colors[vertex] = color;
            if (graphColoringUtil(vertex + 1, numTurns, numColors, adjList, colors))
                return true;
            colors[vertex] = -1; //назначение не привело к решению, откатываемся

            if (showLogs2)
            {
                cout << "Откатываемся с вершины " << vertex << ", цвет " << color <<
"\n";
            }
        }
    }

    return false;
}

// Функция для поиска минимального числа цветов методом грубой силы
int bruteForceColoring(int numTurns, const vector<vector<int>>& adjList,
vector<int>& colors)
{
    colors.assign(numTurns, -1); //инициализируем все цвета как не назначенные (-1)

    //начинаем с 1 цвета и увеличиваем число цветов, пока не найдем корректную
раскраску
    for (int numColors = 1; numColors <= numTurns; ++numColors)
    {
        bruteForceIterations = 0; //сбрасываем счетчик итераций для каждого числа
цветов

        if (showLogs2)
        {
            cout << "\nПытаемся раскрасить граф с " << numColors << " цветами\n";
        }

        if (graphColoringUtil(0, numTurns, numColors, adjList, colors))
        {
            return numColors; //найдено минимальное число цветов
        }
    }
    return numTurns; //в худшем случае потребуется столько же цветов, сколько вершин
}
int main()
{
    setlocale(0, "");
    int choice;
    cout << "Меню:\n";
    cout << "1. Ручной ввод\n";
    cout << "2. Автоматический ввод (условие изначальной задачи)\n";
    cout << "Ваш выбор: ";
    cin >> choice;
    int numRoads;
    vector<string> roads;
    int numTurns;
    vector<string> turns;
    unordered_map<string, int> turnIndices; // Отображение названия поворота на его
индекс
    vector<vector<int>> adjList;
```

```
    if (choice == 1)
    {
        // Ручной ввод данных
        cout << "Количество дорог: ";
        cin >> numRoads;

        roads.resize(numRoads);
        cout << "Названия дорог: ";
        for (int i = 0; i < numRoads; ++i)
        {
            cin >> roads[i];
        }

        cout << "Количество поворотов: ";
        cin >> numTurns;

        turns.resize(numTurns);
        cout << "Введите повороты (пример: AB AC AD BA ...): ";
        for (int i = 0; i < numTurns; ++i)
        {
            cin >> turns[i];
            turnIndices[turns[i]] = i;
        }

        // Инициализируем список смежности для графа конфликтов
        adjList.resize(numTurns);

        // Ввод конфликтов для каждого поворота
        cout << "Введите конфликты для каждого поворота.\n";

        for (int i = 0; i < numTurns; ++i)
        {
            string turn = turns[i];
            int numConflicts;
            cout << "Количество конфликтов на повороте " << turn << ": ";
            cin >> numConflicts;
            cout << "Введите конфликтующие повороты для поворота " << turn << ": ";
            for (int j = 0; j < numConflicts; ++j)
            {
                string conflictTurn;
                cin >> conflictTurn;
                // Добавляем ребро между поворотом и конфликтующим поворотом
                if (turnIndices.find(conflictTurn) != turnIndices.end())
                {
                    int v = turnIndices[conflictTurn];
                    // Избегаем дублирования ребер
                    if (find(adjList[i].begin(), adjList[i].end(), v) ==
adjList[i].end())
                    {
                        adjList[i].push_back(v);
                        adjList[v].push_back(i); // Т.к граф неориентированный
                    }
                }
                else
                {
                    cerr << "Поворот " << conflictTurn << " не найден\n";
                }
            }
        }
    }
    else
    {
        // Используем готовые данные исходной задачи
        numRoads = 5;
```

```
        roads = { "A", "B", "C", "D", "E" };

        turns = { "AB", "AC", "AD", "BA", "BC", "BD", "DA", "DB", "DC", "EA", "EB",
"EC", "ED" };
        numTurns = turns.size();

        for (int i = 0; i < numTurns; ++i)
        {
            turnIndices[turns[i]] = i;
        }

        // Инициализируем список смежности для графа конфликтов
        adjList.resize(numTurns);

        // Данные о конфликтах
        unordered_map<string, vector<string>> conflictData = {
            {"AB", {"BC", "BD", "DA", "EA"}},
            {"AC", {"BD", "DA", "DB", "EA", "EB"}},
            {"AD", {"EA", "EB", "EC"}},
            {"BA", {}},
            {"BC", {"AB", "DB", "EB"}},
            {"BD", {"AB", "AC", "DA", "EB", "EC"}},
            {"DA", {"AB", "AC", "BD", "EB", "EC"}},
            {"DB", {"AC", "BC", "EC"}},
            {"DC", {}},
            {"EA", {"AB", "AC", "AD"}},
            {"EB", {"AC", "AD", "BC", "BD", "DA"}},
            {"EC", {"AD", "BD", "DA", "DB"}},
            {"ED", {}},
        };

        // Заполняем adjList на основе conflictData
        for (const auto& entry : conflictData)
        {
            int u = turnIndices[entry.first];
            for (const string& conflictTurn : entry.second)
            {
                int v = turnIndices[conflictTurn];
                // Избегаем дублирования ребер
                if (find(adjList[u].begin(), adjList[u].end(), v) ==
adjList[u].end())
                {
                    adjList[u].push_back(v);
                    adjList[v].push_back(u);
                }
            }
        }
        cout << "\n\n-----------------------------------------------------------
\n";
        cout << "\nКоличество дорог: " << numRoads << "\n\n";
        cout << "Дороги: ";
        for (const string& road : roads)
        {
            cout << road << " ";
        }
        cout << "\n\nКоличество поворотов: " << numTurns << "\n";
        cout << "\nПовороты: ";
        for (const string& turn : turns)
        {
            cout << turn << " ";
        }
        cout << "\n\n-----------------------------------------------------------
\n";
        cout << "Конфликты:\n\n";
        for (const auto& entry : conflictData)
```

```
        {
            cout << entry.first << ": ";
            for (const string& conflict : entry.second)
            {
                cout << conflict << " ";
            }
            cout << "\n";
        }
    }

    // Построение матрицы смежности
    vector<vector<int>> adjMatrix(numTurns, vector<int>(numTurns, 0));
    for (int u = 0; u < numTurns; ++u)
    {
        for (int v : adjList[u])
        {
            adjMatrix[u][v] = 1;
        }
    }

    cout << "------------------------------------------------------------\n";
    cout << "\nМатрица смежности (1 означает конфликт дорог на перекрестке)\n\n";

    // Вывод заголовка
    cout << setw(5) << "";
    for (int i = 0; i < numTurns; ++i)
    {
        cout << setw(5) << turns[i];
    }
    cout << "\n";

    for (int i = 0; i < numTurns; ++i)
    {
        cout << setw(5) << turns[i];
        for (int j = 0; j < numTurns; ++j)
        {
            cout << setw(5) << adjMatrix[i][j];
        }
        cout << "\n";
    }
    cout << "\n------------------------------------------------------------\n";
    char logChoice;
    cout << "Вывести подробные логи жадного алгоритма? (y/n): ";
    cin >> logChoice;
    showLogs = (logChoice == 'y' || logChoice == 'Y');
    // Выполнение жадной раскраски (жадный алгоритм)
    vector<int> greedyColors;
    greedyIterations = 0; // Сбрасываем счетчик итераций для жадного алгоритма
    greedyColoring(numTurns, adjList, greedyColors);

    // Группировка поворотов по цветам (фазам светофоров) для жадного алгоритма
    unordered_map<int, vector<string>> greedyPhases;
    for (int i = 0; i < numTurns; ++i)
    {
        greedyPhases[greedyColors[i]].push_back(turns[i]);
    }

    // Вывод фаз для жадного алгоритма
    cout << "\n\nФазы перекрестка (жадный алгоритм):\n";
    for (const auto& phase : greedyPhases)
    {
        cout << "Фаза " << phase.first + 1 << ": ";
        for (const string& turn : phase.second)
        {
            cout << turn << " ";
```

```
        }
        cout << "\n";
    }
    cout << "Число цветов (фаз) при жадном алгоритме: " << greedyPhases.size() <<
"\n";
    cout << "Число итераций при жадном алгоритме: " << greedyIterations << "\n";
    cout << "-----------------------------------------------------------\n";
    char logChoice2;
    cout << "Вывести подробные логи алгоритма грубой силы? (y/n): ";
    cin >> logChoice2;
    showLogs2 = (logChoice2 == 'y' || logChoice2 == 'Y');
    // Выполнение раскраски методом грубой силы
    vector<int> bruteForceColors;
    int minColors = bruteForceColoring(numTurns, adjList, bruteForceColors);

    // Группировка поворотов по цветам (фазам светофоров) для метода грубой силы
    unordered_map<int, vector<string>> bruteForcePhases;
    for (int i = 0; i < numTurns; ++i)
    {
        bruteForcePhases[bruteForceColors[i]].push_back(turns[i]);
    }

    // Вывод фаз для метода грубой силы
    cout << "\nФазы перекрестка (метод грубой силы):\n";
    for (const auto& phase : bruteForcePhases)
    {
        cout << "Фаза " << phase.first + 1 << ": ";
        for (const string& turn : phase.second)
        {
            cout << turn << " ";
        }
        cout << "\n";
    }
    cout << "Минимальное число цветов (фаз) при методе грубой силы: " << minColors
<< "\n";
    cout << "Число итераций при методе грубой силы: " << bruteForceIterations <<
"\n";
    cout << "-----------------------------------------------------------\n";

    cout << "\n\n----------ИТОГ-------------\n\n";
    cout << "\n\nФазы перекрестка (жадный алгоритм):\n";
    for (const auto& phase : greedyPhases)
    {
        cout << "Фаза " << phase.first + 1 << ": ";
        for (const string& turn : phase.second)
        {
            cout << turn << " ";
        }
        cout << "\n";
    }
    cout << "Число цветов (фаз) при жадном алгоритме: " << greedyPhases.size() <<
"\n";
    cout << "Число итераций при жадном алгоритме: " << greedyIterations << "\n";
    cout << "-----------------------------------------------------------\n";

    // Вывод фаз для метода грубой силы
    cout << "\n\n\nФазы перекрестка (метод грубой силы):\n";
    for (const auto& phase : bruteForcePhases)
    {
        cout << "Фаза " << phase.first + 1 << ": ";
        for (const string& turn : phase.second)
        {
            cout << turn << " ";
        }
        cout << "\n";
```

```
    }
    cout << "Минимальное число цветов (фаз) при методе грубой силы: " << minColors
<< "\n";
    cout << "Число итераций при методе грубой силы: " << bruteForceIterations <<
"\n";
    cout << "-----------------------------------------------------------\n";
    return 0;
}
```

Листинг 2 – код решения задачи

3 ВЫВОД

В ходе данной работы была рассмотрена задача раскраски вершин графа, применимая к управлению светофорами на сложном перекрёстке. Задача заключалась в минимизации количества фаз светофора, при которых исключаются конфликтующие направления движения. Для её решения были применены два подхода: жадный алгоритм и метод грубой силы. Проведён сравнительный анализ этих методов с точки зрения эффективности, сложности и количества переборов.

Жадный алгоритм:

Жадный алгоритм раскраски графа последовательно назначает каждой вершине наименьший доступный цвет, не конфликтующий с соседями. Алгоритм обходится без полного перебора всех возможных комбинаций, делая выбор на каждом шаге на основе локальной информации.

1. **Сложность:** Временная сложность жадного алгоритма составляет $O(V^2)$, где V — количество вершин графа. Это обусловлено тем, что для каждой вершины требуется просмотреть все её смежные вершины для определения доступных цветов.

В рамках нашей задачи жадный алгоритм успешно раскрасил граф в 4 цвета (фазы), что соответствует минимальному возможному значению для данного графа конфликтов. Количество итераций составило 53, что демонстрирует его эффективность.

2. **Преимущества и недостатки:**

- *Преимущества:* Простота реализации, сравнительно низкая вычислительная сложность, быстрое получение решения.
- *Недостатки:* Не гарантирует нахождение оптимального решения во всех случаях; результат может зависеть от порядка обхода вершин.

Метод грубой силы:

Метод грубой силы (полный перебор) пытается раскрасить граф, перебирая все возможные комбинации цветов для вершин, начиная с 1 цвета и увеличивая их количество до тех пор, пока не будет найдена корректная раскраска.

1. **Сложность:** Временная сложность метода грубой силы экспоненциальна и составляет $O(C^V)$, где C — количество цветов, а V — количество вершин.

   Это связано с тем, что проверяются все возможные варианты раскрасок.

Метод грубой силы также раскрасил граф в 4 цвета, подтвердив минимальное число фаз, необходимое для решения задачи. Однако количество итераций составило 65, что превышает показатель жадного алгоритма.

2. **Преимущества и недостатки:**

- *Преимущества:* Гарантированное нахождение оптимального решения, полное исследование пространства решений.
- *Недостатки:* Высокая вычислительная сложность, неэффективность при большом количестве вершин, значительное потребление ресурсов.

Сравнительный анализ и выводы

Эффективность и сложность:

Жадный алгоритм показал высокую эффективность и справился с задачей за меньшее количество итераций. Его полиномиальная сложность делает его предпочтительным для графов с большим числом вершин.

Метод грубой силы, несмотря на гарантированное нахождение оптимального решения, продемонстрировал высокую вычислительную сложность и значительное увеличение числа переборов, что делает его менее практичным для больших графов.

Качество решений:

В данном случае оба алгоритма достигли одинакового результата, найдя минимальное число фаз для управления светофорами.

Тем не менее, жадный алгоритм не всегда гарантирует оптимальное решение в других задачах, что стоит учитывать при его применении.

Практическая применимость:

Для задач, требующих быстрого принятия решений с допустимым уровнем оптимальности, жадный алгоритм является предпочтительным.

Метод грубой силы может быть полезен для небольших графов или для валидации решений, полученных другими методами.

4 ЛИТЕРАТУРА

1. Вирт Н. Алгоритмы и структуры данных. Новая версия для Оберона, 2010.
2. Кнут Д. Искусство программирования. Тома 1-4, 1976-2013.
3. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих, 2017.
4. Кормен Т.Х. и др. Алгоритмы. Построение и анализ, 2013.
5. Лафоре Р. Структуры данных и алгоритмы в Java. 2-е изд., 2013.
6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., 2018.
7. Скиена С. Алгоритмы. Руководство по разработке, 2011.
8. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2017.
9. Гасфилд Д. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология, 2003.

По языку C++:

10. Страуструп Б. Программирование. Принципы и практика с использованием C++. 2-е изд., 2016.
11. Павловская Т.А. C/C++. Программирование на языке высокого уровня, 2003.
12. Прата С. Язык программирования C++. Лекции и упражнения. - 6-е изд., 2012.
13. Седжвик Р. Фундаментальные алгоритмы на C++, 2001-2002
14. Хортон А. Visual C++ 2010. Полный курс, 2011.
15. Шилдт Г. Полный справочник по C++. 4-е изд., 2006.