



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

КУРСОВАЯ РАБОТА
по дисциплине
«Методы программирования»

Тема курсовой работы
«Реализация алгоритмов на двудольном графе»

Студентка группы КТСО-03-22

Ягодарова С.Р.

Руководитель курсовой работы
доцент кафедры Высшей математики
к.ф.-м.н.

Петрусевич Д.А.

Работа представлена к
защите

«21» дек 2023 г.

(подпись студентки)

«Допущена к защите»

«21» дек 2023 г.

(подпись руководителя)

МОСКВА — 2023



МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

Институт искусственного интеллекта

Кафедра высшей математики

Утверждаю
Исполняющий обязанности заведующего
кафедрой А.В. Шатина

«01» октября 2023 г.

ЗАДАНИЕ
на выполнение курсовой работы
по дисциплине «Методы программирования»

Студентка *Ягодарова С.Р.*

Группа *КТСО-03-22*

1. Тема: «Реализация алгоритмов на двудольном графе»

2. Исходные данные:

Реализовать набор классов для хранения информации о вершинах и ребрах графа.
Перегрузить операторы ввода/вывода в поток $>>$, $<<$ так, чтобы можно было вводить и
выводить информацию о графе и из файла, и из консоли.

Реализовать алгоритмы:

- определения двудольности графа, используя поиск в ширину и глубину;
- раскраски графа;
- венгерский алгоритм для задачи о назначениях;
- нахождения наибольшего паросочетания.

Произвести обзор других алгоритмов и задач для двудольных графов

Оптимизировать построенный код: 1) по производительности, 2) по потреблению памяти

3. Перечень вопросов, подлежащих разработке, и обязательного графического материала:


Продемонстрировать работу алгоритмов

Проиллюстрировать оценки решения по памяти и производительности до и после оптимизации кода

4. Срок представления к защите курсовой работы: до «22» декабря 2023 г.


Задание на курсовую
работу выдал

«01» октября 2023 г.

 (Петрусеви́ч Д.А.)

Задание на курсовую
работу получила

«01» октября 2023 г.

 (Яго́дарова С.Р.)

Оглавление

Глава 1. Теоретическая часть	5
1.1. Алгоритмы на двудольном графе	5
1.1.1. Двудольный граф	5
1.1.2. Проверка графа на двудольность и его раскраска	6
1.1.3. Поиск наибольшего паросочетания	7
1.1.4. Задача о назначениях	9
1.2. Структуры данных	11
1.3. Вывод	13
Глава 2. Практическая часть	14
2.1. Классы и структуры	14
2.2. Проверка графа на двудольность и его раскраска	17
2.3. Реализация алгоритма Куна	20
2.4. Реализация венгерского алгоритма	24
2.5. Функция <code>main()</code>	28
Глава 3. Оптимизация кода	30
3.1. Оптимизация по времени выполнения	30
3.1.1. Оптимизация вызова функций	30
3.1.2. Оптимизация циклов	32
3.1.3. Оптимизация массивов	35
3.2. Оптимизация по потреблению памяти	36
3.3. Выводы к главе 3	39
Заключение	40
Список использованной литературы	41
Приложение 1. Неоптимизированный код	42
Приложение 2. Оптимизированный код	56

Глава 1. Теоретическая часть

1.1. Алгоритмы на двудольном графе

1.1.1. Двудольный граф

Двудольный граф — это граф, вершины которого можно разбить на два множества так, что каждое ребро будет соединять вершины из разных множеств. Иными словами, двудольный граф — это граф, в котором нет циклов нечётной длины. Разбитие графа на доли называется раскраской его вершин в два различных цвета. Двудольный граф называется полным, если каждая вершина из одного множества соединена с каждой вершиной из другого множества. В противном случае двудольный граф является неполным. В задании работы нужно решить следующие задачи для двудольных графов:

- Проверка графа на двудольность
- Раскраска графа
- Задача о назначениях
- Поиск наибольшего паросочетания

1.1.2. Проверка графа на двудольность и его раскраска

Для того, чтобы определить двудольность графа, можно использовать поиск в ширину и поиск в глубину.

Метод обхода графа в ширину основан на проверке каждой вершины U и всех её соседей. Если вершина U не является целевой, она добавляется в очередь, а затем в неё добавляются все соседние вершины. После проверки всех рёбер, выходящих из U , из очереди извлекается следующий узел, и так продолжается. Этот метод применяется для проверки двудольности графа и для раскраски его вершин: каждая вершина в очереди получает цвет, противоположный цвету предыдущей вершины, начиная с произвольного цвета для исходной вершины. Если при раскраске не возникает противоречий, граф считается двудольным.

Поиск в глубину представляет собой альтернативный метод обхода графа, имеющий рекурсивную и нерекурсивную реализации. В случае рекурсивной версии алгоритма все вершины изначально помечаются белым цветом. Начинается с выбора начальной вершины U , которая окрашивается в чёрный цвет. Если у выбранной вершины есть белые соседи, для каждого из них вызывается функция и они также окрашиваются в чёрный цвет. Этот процесс продолжается, пока все соседи выбранной вершины не окажутся чёрными. Во время выхода из рекурсии, уже посещённые вершины проверяют своих оставшихся соседей. Этот метод также применяется для проверки двудольности графа и для разделения его вершин на два разных цвета. Если во время проверки соседних вершин окажется, что какая-то вершина имеет тот же цвет, что и текущая, граф считается не двудольным.

1.1.3. Поиск наибольшего паросочетания

Паросочетание в двудольном графе представляет собой набор рёбер, каждые два из которых не соединены общей вершиной. Наибольшее паросочетание, в свою очередь, содержит наибольшее число рёбер по сравнению с другими паросочетаниями в графе.

Свободная вершина не включена в паросочетание, в то время как насыщенная вершина присутствует в паросочетании. Для поиска наибольшего паросочетания мы применим алгоритм Куна, который опирается на обнаружение увеличивающих цепей.

Цепь в графе представляет собой последовательность вершин и рёбер без повторов. Чередующаяся цепь чередует наличие и отсутствие рёбер в паросочетании. Увеличивающая цепь, в свою очередь, представляет чередующуюся цепь, в которой первое и последнее рёбра не входят в паросочетание.

Теорема Берга. Паросочетание является наибольшим тогда и только тогда, когда не существует увеличивающих относительно него цепей.

Алгоритм Куна будет искать любую из таких цепей с помощью поиска в глубину. Поскольку в увеличивающей цепи первое ребро не принадлежит паросочетанию, алгоритм начинает свою работу со свободной вершины. Он находит свободную вершину из первого множества, а затем просматривает все рёбра из этой вершины. Если какое-то из них ведёт к свободной вершине, то увеличивающая цепь найдена. Она состоит из одного ребра. Поскольку обе рассмотренные вершины были свободными, мы можем добавить это ребро в паросочетание, которое в результате увеличится на 1. Если же вторая вершина была насыщенной каким-то ребром, то нужно пройти вдоль этого ребра к следующей вершине. Далее поиск увеличивающей цепи будет происходить от неё. Если от этой вершины не исходит рёбер, не принадлежащих паросочетанию, то увеличивающей цепи, проходящей через выбранные

вершины, не существует. Вычислительная сложность алгоритма Куна равна $O(VE)$, где V - количество вершин, а E - количество рёбер.

1.1.4. Задача о назначениях

Задача о назначениях представляет собой распределение работ между исполнителями с минимизацией затрат. Каждый исполнитель способен выполнять любую работу с определенными издержками. Цель — обеспечить минимальные затраты, при этом каждый исполнитель берет на себя только одну работу. В основе этой задачи лежит поиск минимального полного паросочетания в двудольном графе с весами на рёбрах.

Веса взвешенного графа соответствуют числам, присвоенным каждому ребру. Полное паросочетание охватывает все вершины графа, и чтобы его найти, количество вершин в обеих частях графа должно совпадать. Если число вершин различно, можно дополнить одно из множеств нулевыми вершинами, что не влияет на решение.

Минимальное паросочетание представляет собой паросочетание с наименьшим общим весом. Для решения этой задачи используется венгерский алгоритм, основанный на двух принципах: если вычесть одно и то же число из элементов строки или столбца, решение не изменится; если существует решение с нулевой стоимостью, оно является оптимальным.

Алгоритм работает с матрицей, в которой каждая строка обозначает исполнителя, а каждый столбец - работу. Элемент матрицы в строке i и столбце j — это стоимость, за которую работник i выполняет работу j . Сперва алгоритм ищет минимальный элемент в каждой строке и вычитает его из этой строки. В результате в каждой строке появится как минимум один ноль. После этого шага нужно выбрать нули так, чтобы в каждой строке и в каждом столбце был всего один выбранный ноль. По сути, это задача поиска наибольшего паросочетания в двудольном графе. Если найденное паросочетание не будет полным, то кто-то из исполнителей остался без работы, или же одна из работ осталась без исполнителя. В таком случае нужно переходить к следующему шагу алгоритма. Он заключается в том, что первый шаг выполняется уже для столбцов матрицы: ищется минимальный элемент в каждом столбце и вычитается из него. После этого снова необходимо

применить алгоритм поиска наибольшего паросочетания. Если нужное паросочетание снова не будет найдено, переходим к шагу 3. Он заключается в покрывании всех строк и столбцов матрицы, в которых есть нули, минимальным количеством линий. Затем среди не покрытых линиями элементов нужно найти минимальный, вычесть его из всех не покрытых линиями строк и прибавить ко всем пересечениям выделенных столбцов и строк. Все вышеперечисленные шаги нужно повторять, пока назначение не станет возможным. Вычислительная сложность венгерского алгоритма равна $O(n^3)$.

Один из основных алгоритмов, который часто конкурирует с Венгерским алгоритмом при решении задачи о назначениях, — это метод ветвей и границ (Branch and Bound). Этот метод также применяется для решения задачи о назначениях и представляет собой комбинаторный подход к поиску оптимального назначения.

Метод ветвей и границ работает путем систематического перебора различных комбинаций назначений и последующего отсека (поиск границы) некоторых ветвей перебора, чтобы сократить пространство поиска и повысить эффективность алгоритма. Вычислительная сложность метода ветвей и границ для задачи о назначениях равна $O(2^n)$, где n — количество элементов в каждом из двух наборов.

1.2. Структуры данных

Были рассмотрены следующие варианты хранения графа в памяти:

1. Матрица смежности

Это матрица, строки и столбцы которой обозначают вершины. Если вершину i и вершину j соединяет ребро, то в $M[i][j]$ стоит 1 в случае невзвешенного графа и какое-то число в случае взвешенного. Иначе там будет стоять 0.

Удобно, что для хранения весов рёбер не нужно использовать отдельную структуру данных, потому что эта информация содержится в матрице.

Сложность по памяти матрицы смежности равна $O(V^2)$, сложность перечисления всех вершин, смежных с вершиной X равна $O(V)$, сложность проверки смежности вершин константная.

2. Матрица инцидентности

Это матрица, строки которой обозначают вершины, а столбцы - рёбра графа. Соответственно, если вершина i инцидентна ребру j , в $M[i][j]$ будет стоять 1, иначе 0. Очевидно, что в матрице инцидентности нельзя указать вес ребра, поэтому эту информацию придётся хранить в отдельной структуре данных. Сложность по памяти такой матрицы равна $O(VE)$ (что практически равно $O(V^2)$), сложность перечисления всех вершин, смежных с вершиной X равна $O(VE)$, что сильно больше $O(V)$ в случае матрицы смежности. Поскольку почти во всех алгоритмах придётся использовать какой-либо из поисков, время перечисления соседних вершин к текущей очень важно. Поэтому матрица инцидентности уступает матрице смежности по двум важным пунктам.

3. Перечень рёбер

Этот способ хранения графа основан на перечислении пар вершин, которые соединены ребром. Он занимает не очень много памяти относительно ранее рассмотренных методов. Однако, во-первых, понадобится отдельный

массив для хранения весов рёбер, а во-вторых, сложность проверки смежности вершин равна $O(E)$.

Для хранения графа в памяти мы решили использовать перечень рёбер. При этом будет существовать отдельный массив, в котором будут храниться данные о цвете вершин.

1.3. Вывод

Таблица 1

Алгоритм	Вычислительная сложность	Пространственная сложность	Задача
Поиск в ширину	$O(V + E)$	$O(V)$	Раскраска графа
Поиск в глубину	$O(V + E)$	$O(V)$	Раскраска графа
Алгоритм Куна	$O(VE)$	$O(V + E)$	Поиск наибольшего паросочетания
Венгерский алгоритм	$O(n^3)$	$O(n^2)$	Решение задачи о назначениях
Метод ветвей и границ	$O(2^n)$	$O(2^n)$	Решение задачи о назначениях

Из таблицы видно, что поиск в ширину и поиск в глубину имеют одинаковую вычислительную и пространственную сложность, то есть нет разницы, какой из них использовать для раскраски. Для решения задачи о назначениях были рассмотрены два алгоритма: метод ветвей и границ и венгерский алгоритм. По вычислительной и пространственной сложности венгерский алгоритм лучше метода ветвей и границ.

Глава 2. Практическая часть

2.1. Классы и структуры

Для хранения графа в памяти нужно хранить его рёбра и данные о цветах вершин. Для этого была создана структура Edge (ребро) и перечисляемый тип Color.

```
struct Edge //Ребро
{
    int weight;
    int vertex1;
    int vertex2;
    Edge(int w = 0, int v1 = 0, int v2 = 0)
    {
        weight = w;
        vertex1 = v1;
        vertex2 = v2;
    }
    bool operator==(Edge e)
    {
        return weight == e.weight and vertex1 == e.vertex1 and vertex2 ==
e.vertex2;
    }

    Edge operator=(Edge e)
    {
        weight = e.weight;
        vertex1 = e.vertex1;
        vertex2 = e.vertex2;
        return *this;
    }
};
```

Листинг 1. Структура Edge

У ребра есть три поля: вершины, которые оно соединяет, и вес ребра. Также для удобства работы со структурой были перегружены операторы присваивания и проверки на равенство.

```
enum Color //Цвет вершины
{
    uncolor,
    blue,
    pink
};
```

Листинг 2. Перечисляемый тип Color

Тип Color может принимать три значения: uncolor (без цвета), blue и pink. До раскраски все вершины графа не имеют цвета.

Класс BipartiteGraph (двудольный граф) имеет следующие приватные поля: vector, хранящий рёбра графа, vector, хранящий их цвета и int, обозначающий количество вершин. Программа устроена таким образом, что сами вершины нумеруются с нуля, но нигде не хранятся. Поэтому цвет вершины хранится в массиве цветов по индексу, равному номеру вершины.

```
vector<Edge> edges; //Массив ребер
int CountVertices; //Количество вершин
vector<Color> colors; //Цвета вершин
```

Листинг 3. Приватные поля класса

В классе BipartiteGraph есть только один конструктор – по умолчанию.

```
BipartiteGraph()
{
    CountVertices = 0;
    colors.resize(100, uncolor);
}
```

Листинг 4. Конструктор по умолчанию

Чтобы добавить в граф рёбра, нужно воспользоваться функцией operator>>(), которая считывает граф из файла или из консоли, или функцией push(). Описание функции push() будет в следующем пункте.

```
istream& operator>>(istream& stream, BipartiteGraph& graph)
{
    int countE;
    stream >> countE;
    for (int i = 0; i < countE; i++)
    {
        Edge n;
        stream >> n;
        graph.push(n.vertex1, n.vertex2, n.weight);
    }
    return stream;
}
```

Листинг 5. Operator>>

Эта функция принимает объект класса istream и граф. Поскольку класс ifstream наследуется от istream, функция может считывать граф как из консоли, так и из файла. Для её корректной работы был перегружен operator>> для структуры Edge.

```
istream& operator>>(istream& stream, Edge& e)
{
    stream >> e.vertex1;
    stream >> e.vertex2;
    stream >> e.weight;
}
```

```
return stream;
}
```

Листинг 6. Operator>> для Edge

Для вывода графа в консоль был перегружен operator<< для BipartireGraph и для Edge.

```
ostream& operator<<(ostream& s, Edge e)
{
    s << '\n' << "[" << e.vertex1 << ", " << e.vertex2 << "], " <<
    "weight = " << e.weight;
    return s;
}
```

Листинг 7. Operator<< для Edge

```
ostream& operator<<(ostream& stream, BipartiteGraph& graph)
{
    for (int i = 0; i < graph.edges.size(); i++)
    {
        stream << graph.edges[i] << ", " << "color vertex1: " <<
        graph.colors[graph.edges[i].vertex1] << ", color vertex2: " <<
        graph.colors[graph.edges[i].vertex2];
    }
    return stream;
}
```

Листинг 8. Operator<< для графа

2.2. Проверка графа на двудольность и его раскраска

Для проверки графа на двудольность и его раскраски были использованы алгоритмы поиска в ширину и поиска в глубину. Обе функции находятся в `private`. Поиск в ширину был разделён на две функции: `BFS()` и `fBFS()`. Первая функция находит всех соседей вершины `cur_vertex`, а вторая добавляет их в очередь и распределяет по множествам. Поскольку граф двудольный, какие-либо вершины могут быть соединены ребром только между собой. Поэтому `BFS()` работает, пока все вершины не будут раскрашены и пока очередь не станет пустой (см. приложение). Функция находит очередного соседа `cur_vertex` и вызывает для них `fBFS()`.

```
for (int i = 0; i < edges.size(); i++)
{
    if (edges[i].vertex1 == cur_vertex)
    {
        if (!fBFS(cur_vertex, edges[i].vertex2, q))
            return false;
    }
    else if (edges[i].vertex2 == cur_vertex)
    {
        if (!fBFS(cur_vertex, edges[i].vertex1, q))
            return false;
    }
}
```

Листинг 9. Вызов функции `fBFS()`

```
bool fBFS(int vertex1, int vertex2, queue<int>& q)
{
    if (colors[vertex2] == uncolor)
    {
        q.push(vertex2);
        colors[vertex2] = colors[vertex1] == blue ? pink : blue;
        return true;
    }
    if (colors[vertex1] == colors[vertex2])
    {
        for (int j = 0; j < CountVertices; j++)
            colors[j] = uncolor;
        return false;
    }
    return true;
}
```

Листинг 10. Функция `fBFS()`

Обе функции меняют массив `colors` и возвращают `bool`. Если `fBFS()` найдёт противоречие, она изменит цвета всех вершин на `uncolor` и вернёт `false`.

Раскраска с помощью поиска в глубину (DFS()) работает рекурсивно. У этой функции есть два параметра: цвет предыдущей вершины и индекс текущей.

```
colors[vertex] = p == blue ? pink : blue;
for (int i = 0; i < edges.size(); i++)
{
    if (edges[i].vertex1 == vertex)
    {
        if (colors[edges[i].vertex2] == colors[vertex])
        {
            for (int i = 0; i < CountVertices; i++)
                colors[i] = uncolor;
            return false;
        }
        else
            if (colors[edges[i].vertex2] == uncolor)
                return DFS(colors[vertex], edges[i].vertex2);
    }
}
```

Листинг 11. Фрагмент функции DFS()

Функция назначает текущей вершине цвет, противоположный цвету предыдущей вершины. Затем она в цикле ищет всех соседей текущей вершины. Если какая-либо соседняя вершина не раскрашена, то функция рекурсивно вызывается для неё. А если одна из соседних вершин раскрашена в тот же цвет, то все элементы массива colors принимают значение uncolor, а DFS() возвращает false.

Функции раскраски с помощью поиска в ширину и в глубину вызываются в функции push() (см. приложение). Эта функция добавляет в граф новое ребро. Там рассматривается три варианта: ребро соединяет две новые вершины, ребро соединяет существующую вершину с новой и ребро соединяет две вершины, существовавшие ранее. Нарушение двудольности возможно только в последнем варианте, поэтому там проверяется возвращённое функцией раскраски значение, и вызывается исключение в случае false.

```
if (v1 < CountVertices and v2 < CountVertices)
{
    Edge newEdge(w, v1, v2);
    edges.push_back(newEdge);
    bool r = BFS();
    if (!r)
    {
        //Исключение
    }
}
```

```
        throw Exception("You have violated the bipartiteness of the  
graph!");  
    }  
}
```

Листинг 12. Вызов функции BFS()

2.3. Реализация алгоритма Куна

Этот алгоритм был использован для поиска наибольшего паросочетания. Для удобства алгоритм был разделён на две функции: `MaxMatch()` и `FindMaxMatch()`. У первой функции нет параметров, а находится она в `public`. Вторая функция находится в `private`.

В `MaxMatch()` создаются следующие переменные: векторы рёбер `res`, `M` и `M2`, векторы целых чисел `busy_V` и `busy_V2`. Ссылка на `res` передаётся в функцию `FindMaxMatch()`. Она добавит в этот вектор рёбра увеличивающей цепи. `M` – это текущее паросочетание, относительно которого `FindMaxMatch()` будет искать увеличивающую цепь. `Busy_V` – это массив, заполненный нулями и единицами. Если под индексом `i` в этом массиве стоит 1, то вершина графа `i` принадлежит паросочетанию, иначе – не принадлежит. `Busy_V2` заполняется аналогично, но нули и единицы указывают на принадлежность вершины `i` к увеличивающей цепи. В функции `MaxMatch()` в цикле вызывается `FindMaxMatch()` для каждой свободной вершины.

```
r = FindMaxMatch(res, M, busy_V, busy_V2, i);  
if (!r)  
{  
    for (int j = 0; j < CountVertices; j++)  
        busy_V2[j] = 0;  
    res.clear();  
    continue;  
}
```

Листинг 13. Вызов `FindMaxMatch()`

`FindMaxMatch()` добавляет в `res` рёбра увеличивающей цепи и возвращает `bool`, который укажет, удалось ли найти корректную увеличивающую цепь от вершины `i`. Если функция вернула `false`, то `res` очищается, `busy_V2` заполняется нулями, а алгоритм продолжает поиск других свободных вершин. Если же функция вернула `true`, то необходимо составить новое паросочетание, в котором, во-первых, не будет рёбер, которые есть и в `res`, и в `M`, а во-вторых, будут рёбра, которые присутствуют только в `res`, и которые присутствуют только в `M`.

```
for (int j = 0; j < M.size(); j++)  
{  
    bool p = true;  
    for (int k = 0; k < res.size(); k++)
```

```

{
    if (M[j] == res[k])
        p = false;
}
if (p)
    M2.push_back(M[j]);
}

```

Листинг 14. Добавление в M2 рёбер, которые есть только в M

```

for (int j = 0; j < res.size(); j++)
{
    bool p = true;
    for (int k = 0; k < M.size(); k++)
    {
        if (res[j] == M[k])
            p = false;
    }
    if (p)
        M2.push_back(res[j]);
}

```

Листинг 15. Добавление в M2 рёбер, которые есть только в res

Таким образом, новое паросочетание хранится в M2, и необходимо приравнять M к M2. После этого функция обнуляет busy_V2 и меняет busy_V в соответствии с новым паросочетанием.

Функция FindMaxMatch() работает следующим образом: сперва она проверяет всех соседей вершины vertex. Если среди них есть свободная вершина, то ребро, соединяющее эти вершины, добавляется в res, и функция возвращает true

```

if (edges[i].vertex1 == vertex)
{
    if (busy_V[edges[i].vertex2] == 0)
    {
        res.push_back(edges[i]);
        busy_V2[edges[i].vertex2] = 1;
        busy_V2[vertex] = 1;
        return true;
    }
}

```

Листинг 16. Добавление увеличивающей цепи из одного ребра

Если все соседние вершины не свободны, то функция проверяет, есть ли среди соседей вершина, ещё не участвующая в увеличивающей цепи.

```

bool a = false;
for (int i = 0; i < edges.size(); i++)
{
    if (edges[i].vertex1 == vertex)
        if (busy_V2[edges[i].vertex2] == 0)
            a = true;
    if (edges[i].vertex2 == vertex)
        if (busy_V2[edges[i].vertex1] == 0)
            a = true;
}

```

```
}
```

Листинг 17. Поиск вершины, не добавленной в цепь

Если такая вершина не найдена, то от текущей вершины `vertex` не существует увеличивающей цепи. Тогда функция удалит из `res` два последних ребра и вернёт `false`. Если вершина найдена, то нужно сохранить значение `vertex` в другую переменную, а затем изменить `vertex` на индекс найденной вершины. После этого от нового `vertex` нужно пройти по занятому паросочетанием ребру к другой соседней вершине и снова изменить `vertex` на её индекс.

```
int ver = vertex;
vertex = edges[i].vertex2;
res.push_back(edges[i]);
busy_V2[vertex] = 1;
for (int j = 0; j < M.size(); j++)
{
    if (M[j].vertex1 == vertex)
    {
        vertex = M[j].vertex2;
        res.push_back(M[j]);
        break;
    }
    else if (M[j].vertex2 == vertex)
    {
        vertex = M[j].vertex1;
        res.push_back(M[j]);
        break;
    }
}
busy_V2[vertex] = 1;
```

Листинг 18. Изменение значения `vertex`

Ребра, к которым принадлежат все три рассмотренные вершины, добавляются в `res`. Затем для новой вершины функция вызывается рекурсивно. После вызова рекурсии она вернёт `true`, если в процессе создания увеличивающей цепи найдёт свободную вершину.

```
bool lucky = FindMaxMatch(res, M, busy_V, busy_V2, vertex);
if (!lucky)
{
    vertex = ver;
    continue;
}
else
    return true;
```

Листинг 19. Рекурсивный вызов функции

Если функция вернула `false`, то последняя вершина не была свободной, и от неё не отходило рёбер, не принадлежащих цепи. Тогда нужно вернуться

к сохранённому в переменную `ver` индексу вершины, и продолжить от неё поиск других путей.

2.4. Реализация венгерского алгоритма

Задачу о назначениях решает функция `appointment()`. Кроме того, для неё были написаны некоторые вспомогательные функции: `NewGraph()`, `operator=`, `perfect()` и `transpose()`. `NewGraph()` формирует граф на основе матрицы, соединяя ребром вершины, у которых значение веса обнулилось. `Perfect()` – это функция, которая проверяет, является ли паросочетание совершенным. `Transpose()` – это функция вне класса. Она принимает матрицу и транспонирует её.

Сперва функция `appointment()` проверяет, равно ли количество вершин первого множества количеству вершин второго. Если множества не равны, функция дополняет граф рёбрами нулевого веса. Например, если в первом множестве больше вершин:

```
if (countP > countB)
{
    for (int i = c; i < c + countP - countB; i++)
        for (int j = 0; j < c; j++)
            if (colors[j] == pink)
                push(i, j);
    countB = countP;
}
```

Листинг 20. Дополнение графа рёбрами

Затем функция создаёт двумерный массив размером $(n+1) \times (n+1)$, где n – количество вершин одного множества. Нулевая строка матрицы заполняется индексами вершин одного множества, а нулевой столбец – индексами другого множества. Затем сама матрица заполняется соответственно весами рёбер.

```
for (int i = 0; i < edges.size(); i++) //Заполнение матрицы
{
    int z = 0;
    int z2 = 0;
    for (int j = 1; j < countB + 1; j++)
        if (matrix[0][j] == edges[i].vertex1 or matrix[0][j] ==
edges[i].vertex2)
            z = j;
    for (int j = 1; j < countB + 1; j++)
        if (matrix[j][0] == edges[i].vertex1 or matrix[j][0] ==
edges[i].vertex2)
            z2 = j;
    matrix[z2][z] = edges[i].weight;
}
```

Листинг 21. Заполнение матрицы

Затем алгоритм вычитает из каждой строки минимальный элемент этой строки и формирует новый граф. Это делает отдельная функция `NewGraph()`.

```
vector<int> temp(CountVertices);  
BipartiteGraph graph = NewGraph(countB, matrix, temp);  
  
vector<Edge> max_m = graph.MaxMatch();  
vector<Edge> max_m_2(max_m.size());
```

Листинг 22. Вызов функции `NewGraph()`

Эта функция принимает количество вершин одного множества, ссылку на матрицу и вектор `temp`. Этот вектор нужен для корректной индексации вершин: при создании нового графа его вершины нумеруются по порядку с нуля, то есть в новом графе те же самые рёбра будут соединять вершины с другими индексами. В массив `temp` на соответствующий индекс, равный индексу вершины в изначальном графе, ставится индекс этой вершины в другом графе. Это поможет вернуть корректное паросочетание в конце функции. `NewGraph()` сперва вычитает из каждой строки минимальный элемент этой строки.

```
for (int i = 1; i < countB + 1; i++)  
{  
    int min = matrix[i][1];  
    for (int j = 2; j < countB + 1; j++)  
        if (matrix[i][j] < min) //Находим минимум в каждой строке  
            min = matrix[i][j];  
    for (int j = 1; j < countB + 1; j++)  
        matrix[i][j] = matrix[i][j] - min; //Вычитаем минимум из каждого  
        элемента этой строки  
}
```

Листинг 23. Начало функции `NewGraph()`

Затем функция формирует новый граф, соединяя рёбрами нужные вершины и заполняя массив `temp` (см. приложение). После создания графа в функции `appointment()` для него вызывается `MaxMatch()`. То есть ищется наибольшее паросочетание в новом графе. Если оно будет совершенным (то есть все вершины графа будут ему принадлежать), то решение найдено. Осталось лишь заменить индексы вершин на изначальные, используя массив `temp`. Для проверки паросочетания на совершенность используется функция `perfect()` (см. приложение).

```
if (graph.perfect(max_m))  
{  
    for (int i = 0; i < max_m.size(); i++)
```

```

{
    for (int j = 0; j < CountVertices; j++)
    {
        if (temp[j] == max_m[i].vertex1)
            max_m_2[i].vertex1 = j;
        else if (temp[j] == max_m[i].vertex2)
            max_m_2[i].vertex2 = j;
    }
}
for (int i = 0; i < edges.size(); i++)
    for (int j = 0; j < max_m_2.size(); j++)
        if (edges[i].vertex1 == max_m_2[j].vertex1 and
edges[i].vertex2 == max_m_2[j].vertex2)
            max_m_2[j].weight = edges[i].weight;
return max_m_2;
}

```

Листинг 24. Проверка на совершенность и возвращение результата

Если же найденное паросочетание не было совершенным, то алгоритм продолжает работу. Теперь он находит минимум в каждом столбце и вычитает его из этого столбца. Для упрощения реализации была написана функция `transpose()`, которая транспонирует матрицу. Тогда для неё можно будет вызвать ту же функцию `NewGraph()`, которая находит в каждой строке минимум и вычитает его из строки. В остальном же эта часть кода ничем не отличается от предыдущей (см. приложение). Если после этого шага решение всё ещё не было найдено, то алгоритм продолжает работу.

Наибольшее паросочетание сохраняется в массив `max_m2_2`. В контексте задачи это значит, что мы назначили как можно больше работ как можно большему количеству работников. Далее нужно транспонировать матрицу обратно и покрыть все нули в ней минимальным количеством линий. Для этого алгоритм сохраняет в массивы индексы всех строк без назначений, всех столбцов с нулями в этих строках, всех строк с выделенными (то есть участвующими в текущем назначении) нулями в этих столбцах. В вектор `lwa` добавляются все непокрытые линиями строки, а в вектор `ncwz` — все непокрытые столбцы (см. приложение). Затем алгоритм находит минимальный из непокрытых элементов и вычитает его из них. К элементам на пересечениях выделенных строк и столбцов он прибавляет его.

```

int min = 100000000;
for (int i = 0; i < lwa.size(); i++)
    for (int j = 0; j < ncwz.size(); j++)
        if (matrix[lwa[i]][ncwz[j]] < min)
            min = matrix[lwa[i]][ncwz[j]];

```

```

for (int i = 0; i < lwa.size(); i++)
    for (int j = 0; j < ncwz.size(); j++)
        matrix[lwa[i]][ncwz[j]] = matrix[lwa[i]][ncwz[j]] - min;
//Вычитаем из всех непокрытых элементов минимальный из них

for (int i = 0; i < unmarked.size(); i++)
    for (int j = 0; j < cwz.size(); j++)
        matrix[unmarked[i]][cwz[j]] = matrix[unmarked[i]][cwz[j]] +
min;

```

Листинг 25. Нахождение минимума и изменение элементов матрицы

В массиве `unmarked` находятся все выделенные строки, а в массиве `cwz` – все выделенные столбцы. Очевидно, что после этих действий в матрице появились новые нули. Это значит, что назначение могло стать возможным. Поэтому мы повторяем действия из шага 1, вызывая `NewGraph()`, создавая наибольшее паросочетание и проверяя его на совершенство (см. приложение).

Всё это повторяется, пока назначение не станет возможным, то есть пока функция не вернёт паросочетание. Поэтому все вышеописанные шаги написаны в бесконечном цикле `while(true)`.

2.5. Функция main()

```
BipartiteGraph testP;  
testP.push(0, 1);  
testP.push(2, 3);  
testP.push(3, 4);  
testP.push(4, 5);  
testP.push(2, 1);  
testP.push(5, 6);  
testP.push(0, 7);  
testP.push(1, 4);  
  
cout << testP;  
  
vector<Edge> r = testP.MaxMatch();  
cout << '\n';  
for (int i = 0; i < r.size(); i++)  
    cout << r[i];
```

Листинг 26. Создание графа и вызов MaxMatch()

```
[0, 1], weight = 0, color vertex1: 1, color vertex2: 2  
[2, 3], weight = 0, color vertex1: 1, color vertex2: 2  
[3, 4], weight = 0, color vertex1: 2, color vertex2: 1  
[4, 5], weight = 0, color vertex1: 1, color vertex2: 2  
[2, 1], weight = 0, color vertex1: 1, color vertex2: 2  
[5, 6], weight = 0, color vertex1: 2, color vertex2: 1  
[0, 7], weight = 0, color vertex1: 1, color vertex2: 2  
[1, 4], weight = 0, color vertex1: 2, color vertex2: 1  
  
[5, 6], weight = 0  
[3, 4], weight = 0  
[2, 1], weight = 0  
[0, 7], weight = 0
```

Рисунок 1. Вывод графа и его наибольшего паросочетания

```
BipartiteGraph error4;  
error4.push(0, 1, 7);  
error4.push(0, 2, 3);  
error4.push(0, 3, 6);  
error4.push(0, 4, 9);  
error4.push(0, 5, 5);  
error4.push(6, 1, 7);  
error4.push(6, 2, 5);  
error4.push(6, 3, 7);  
error4.push(6, 4, 5);  
error4.push(6, 5, 6);  
error4.push(7, 1, 7);  
error4.push(7, 2, 6);  
error4.push(7, 3, 8);  
error4.push(7, 4, 8);  
error4.push(7, 5, 9);  
cout << '\n'; cout << error4;  
  
vector<Edge> app = error4.appointment();  
cout << '\n';  
for (int i = 0; i < app.size(); i++)  
    cout << app[i];
```

Листинг 27. Создание графа и вызов appointment()

В этом графе множества вершин не равны друг другу.

```
[0, 1], weight = 7, color vertex1: 1, color vertex2: 2
[0, 2], weight = 3, color vertex1: 1, color vertex2: 2
[0, 3], weight = 6, color vertex1: 1, color vertex2: 2
[0, 4], weight = 9, color vertex1: 1, color vertex2: 2
[0, 5], weight = 5, color vertex1: 1, color vertex2: 2
[6, 1], weight = 7, color vertex1: 1, color vertex2: 2
[6, 2], weight = 5, color vertex1: 1, color vertex2: 2
[6, 3], weight = 7, color vertex1: 1, color vertex2: 2
[6, 4], weight = 5, color vertex1: 1, color vertex2: 2
[6, 5], weight = 6, color vertex1: 1, color vertex2: 2
[7, 1], weight = 7, color vertex1: 1, color vertex2: 2
[7, 2], weight = 6, color vertex1: 1, color vertex2: 2
[7, 3], weight = 8, color vertex1: 1, color vertex2: 2
[7, 4], weight = 8, color vertex1: 1, color vertex2: 2
[7, 5], weight = 9, color vertex1: 1, color vertex2: 2

[0, 2], weight = 3
[6, 4], weight = 5
[7, 1], weight = 7
[8, 3], weight = 0
[9, 5], weight = 0
```

Рисунок 2. Вывод графа и решения задачи о назначениях

Глава 3. Оптимизация кода

3.1. Оптимизация по времени выполнения

3.1.1. Оптимизация вызова функций

Вызовы функций требуют дополнительного времени, так как для них нужно передать управление из точки вызова в функцию, вернуть управление после окончания работы функции, а также скопировать параметры, извлечь их в функции и вернуть значение. Для оптимизации этого процесса можно использовать макросы. Однако в нашей работе отсутствуют функции, которые было бы удобно заменить макросами. Почти все функции работы являются решениями поставленных задач, и их замена макросами приведёт к множеству ошибок.

Для улучшения читаемости кода и упрощения реализации алгоритмов раскраски графа и поиска оптимального назначения были добавлены функции fBFS() (см. листинг 10) и функция transpose() (см. приложение). В целях оптимизации вызовы этих функций были заменены на соответствующий код, в результате чего потерпели изменения функции BFS() и appointment().

```
for (int i = 0; i < CountEdges; ++i)
{
    if (edges[i].vertex1 == cur_vertex)
    {
        if (colors[edges[i].vertex2] == uncolor)
        {
            q.push(edges[i].vertex2);
            colors[edges[i].vertex2] = colors[cur_vertex] ==
blue ? pink : blue;
        }
        else if (colors[cur_vertex] == colors[edges[i].vertex2])
        {
            for (int j = 0; j < CountVertices; ++j)
                colors[j] = uncolor;
            return false;
        }
    }
}
```

Листинг 28. Фрагмент функции BFS()

Этот код ищет всех соседей текущей вершины и сравнивает их цвет с цветом cur_vertex. Он заменяет функцию fBFS().

```
int** res = new int* [countB1];
for (int i = 0; i < countB1; ++i)
    res[i] = new int[countB1];
for (int i = 0; i < countB1; ++i)
```

```
        for (int j = 0; j < countB1; ++j)
            res[i][j] = matrix[j][i];
    matrix = res;
```

Листинг 29. Транспонирование матрицы в appointment()

В функции perfect(), проверяющей, является ли паросочетание совершенным, отпала необходимость, так как для проверки достаточно сравнить количество рёбер паросочетания с количеством вершин одной доли графа. Таким образом, в функции appointment() вызов perfect() заменяется сравнением двух чисел.

```
if (max_m2_size == countB)
{
    Edge* ptr_edg = edges + CountEdges;
    for (Edge* p = edges; p < ptr_edg; ++p)
        for (int j = 0; j < max_m2_size; ++j)
            if ((*p).vertex1 == max_m2[j].vertex1 and (*p).vertex2 ==
max_m2[j].vertex2)
                max_m2[j].weight = (*p).weight;

    return max_m2;
}
```

Листинг 30. Возвращение результата, если паросочетание совершенное

3.1.2. Оптимизация циклов

Для оптимизации циклов были предприняты следующие шаги:

1. Замена вызовов функций и арифметических выражений в объявлениях циклов на константы.

В функции `appointment()` создавалось много массивов и переменных для хранения временных данных. Циклы `for` для перечисления элементов этих массивов вызывали функции `begin()` и `end()`. Таким образом, при каждой новой итерации функции вызывались заново, хотя размер массива в цикле не менялся. Для оптимизации по времени выполнения стоит сохранить значения в переменные и в цикле использовать их.

```
vector<int>::iterator it = lwa.begin();  
vector<int>::iterator it2 = lwa.end();  
for (; it != it2; it++)  
    if (*it == i)  
        ok = false;
```

Листинг 31. Сохранение значений `lwa.begin()` и `lwa.end()` и использование их в цикле

Также стоит сохранять значения арифметических выражений. Поскольку в нулевом столбце и нулевой строке квадратной матрицы, созданной в `appointment()`, хранятся индексы вершин разных множеств, сама матрица по размеру на 1 больше, чем количество вершин в одном множестве. Из-за этого все циклы для матрицы вычисляют значение `countB + 1` (см. листинг 23).

```
int countB1 = countB + 1;  
  
int** matrix = new int*[countB1];  
for (int i = 0; i < countB1; ++i)  
    matrix[i] = new int[countB1];
```

Листинг 32. Сохранение значения `countB + 1` и использование его в цикле

Таким образом были изменены все циклы внутри функции `appointment()` (см. приложение 2).

2. Выход из цикла, если дальнейшее его выполнение не необходимо.

Такие ситуации возникают, например, при проверке, полностью ли раскрашен граф. Если цвет хотя бы одной вершины является `uncolor`, стоит выйти из цикла и продолжить выполнение функции раскраски.


```

bool r = true;
for (int i = 0; i < CountEdges; ++i)
    if (colors[edges[i].vertex1] == uncolor or colors[edges[i].vertex2] ==
        uncolor)
    {
        r = false;
        break;
    }

```

Листинг 34. Проверка вершин в функции DFS()

Таким образом были переписаны некоторые циклы в функциях FindMaxMatch(), appointment(), BFS() и DFS() (см. приложение 2).

3. Объединение циклов.

Циклы возможно объединить, если у них одинаковое/кратное количество итераций. В функции appointment() такая возможность возникла при подсчёте количеств вершин правой и левой доли графа.

```

int countB = 0; //Количество вершин первой доли
int countP = 0; //Количество вершин второй доли
Color* ptr_end = colors + CountVertices;
for (Color* p = colors; p < ptr_end; ++p)
{
    if (*p == blue)
        countB++;
    if (*p == pink)
        countP++;
}

```

Листинг 35. Подсчёт количества вершин правой и левой доли

До оптимизации подсчёт происходил в двух разных циклах.

4. Уменьшение количества итераций в циклах.

Во многих циклах по массивам можно совершать действия с несколькими элементами сразу, чтобы уменьшить количество итераций в цикле в два раза. Например:

```

countB_2 = countB / 2 + 1;
for (int j = 1; j < countB_2; ++j)
{
    matrix[i][j] = matrix[i][j] - min;
    matrix[i][countB_2 + j - 1] = matrix[i][countB_2 + j - 1] - min;
}

```

Листинг 36. Фрагмент функции NewGraph()

В этом коде минимум вычитается из двух элементов строки матрицы, а цикл идёт до середины строки.

Также в функции `appointment()` при работе с квадратной матрицей можно проверять одновременно элементы, симметричные относительно главной диагонали. Например:

```
for (int j = i; j < countB; ++j)
{
    if (matrix[i][j] == 0)
        graph.push(matrix[i][0], matrix[0][j]);
    else if (matrix[j][i] == 0)
        graph.push(matrix[j][0], matrix[0][i]);
}
```

Листинг 37. Поиск нулевых элементов в матрице для добавления рёбер в граф.

5. Использование префиксного инкремента.

Префиксный инкремент работает немного быстрее постфиксного, так как не тратит время на сохранение значения переменной до её изменения. Все инкременты в работе были изменены на префиксные (см. приложение 2).

6. Использование библиотечных функций вместо циклов, например, для заполнения вектора одним и тем же значением.

7. С помощью дополнительных массивов были удалены некоторые вложенные циклы.

3.1.3. Оптимизация массивов

Оптимизация массивов состояла из двух частей: увеличения скорости добавления элемента в конец массива и оптимизация циклов, совершающих обход массивов. Для первой части стоит заменить все основные массивы на статические. Это поможет избежать траты времени на изменение размера массива в случае необходимости. Однако такой способ ведёт к расходу лишней памяти, так как значительная часть массива может остаться неиспользованной. В данной работе массивы-члены класса были заменены на статические, а для хранения временных данных в функциях использовался класс `vector`.

```
private:
    int CountEdges;
    Edge edges[225];
    Color colors[30];
```

Листинг 36. Поля класса `BipartiteGraph`

Суть оптимизации циклов, совершающих обход массивов, состоит в использовании указателя вместо `int`. Указатель перемещается от начала массива к его концу, благодаря чему не нужно каждый раз вычислять позицию элемента с помощью индекса. Указатель на место после последнего элемента массива вычисляется заранее.

```
Color* ptr_end = colors + CountVertices;
for (Color* p = colors; p < ptr_end; ++p)
    *p = uncolor;
```

Листинг 37. Обход массива `colors` и изменение значений его элементов

Почти все циклы в коде были изменены таким образом (см. приложение).

3.1.4. Оптимизация ветвлений

В этой работе практически не встречались сложные ветвления, поэтому их оптимизация ограничилась следующими действиями:

1. Использование тернарного оператора в случаях простых условных проверок. Это позволило сократить объём кода и улучшить его читаемость. Так, например, тернарный оператор использовался, когда нужно было назначить соседней вершине цвет, противоположный цвету текущей.

```
colors[(*p).vertex2] = colors[cur_vertex] == blue ? pink : blue;
```

Листинг 38. Изменение цвета вершины

2. Использование конструкции switch-case вместо if-else. Эта конструкция часто может быть более эффективной и понятной, особенно когда нужно обработать несколько возможных значений одной переменной или выражения. Например, switch-case был использован при подсчёте количества вершин в каждой доле графа.

```
Color* ptr_end = colors + CountVertices;
for (Color* p = colors; p < ptr_end; ++p)
{
    switch (*p)
    {
        case blue:
            countB++;
            break;
        case pink:
            countP++;
            break;
    }
}
```

Листинг 39. Конструкция switch-case

3. Расположение наиболее вероятного условия в самом начале списка условий в конструкции if-else if. Это улучшило производительность программы, так как если первое условие выполняется чаще остальных, то остальные условия проверяться не будут, что уменьшит количество операций проверок и ускорит выполнение кода.

3.2. Оптимизация по потреблению памяти

Из-за того, что мы хранили граф в виде перечня рёбер, вершины можно было добавлять только по порядку от 0 до CountVertices - 1. В результате в функции appointment() вершины нового графа имели индексы, отличающиеся от индексов вершин изначального графа. Из-за этого пришлось хранить несколько массивов (для каждого нового графа), в которых содержалась информация о соответствии индексов вершин старого графа индексам вершин нового графа. Поэтому было принято решение унаследовать класс BipartiteGraph от класса Graph.

```
class Graph
{
protected:
    int** mat;
    int CountVertices;
public:
    Graph(int c = 20)
    {
        CountVertices = c;
        mat = new int*[c * c];
        for (int i = 0; i < c; ++i)
        {
            mat[i] = new int[c * c];
            for (int j = 0; j < c; ++j)
                mat[i][j] = 0;
        }
    }
}
```

Листинг 40. Поля класса Graph и его конструктор

Обычный граф хранится как матрица смежности. Предполагаемое количество вершин передаётся в конструктор, в котором для матрицы выделяется память. Конструктор класса BipartiteGraph вызывает конструктор базового класса. В результате у объекта класса BipartiteGraph будет и перечень рёбер, и матрица смежности.

```
BipartiteGraph(int c = 20) : Graph(c)
{
    Color* ptr_end = colors + 30;
    for (Color* p = colors; p < ptr_end; ++p)
        (*p) = uncolor;
    CountEdges = 0;
}
```

Листинг 41. Конструктор класса BipartiteGraph

Всё это позволит добавлять любые вершины в любом порядке (в пределах выделенной для матрицы памяти), то есть необходимость в массивах

temp (см. листинги 22, 24) отпадает. Функция push() будет создавать новое ребро и добавлять его в массив, а затем вызывать push() базового класса, который изменит значения в матрице смежности(см. приложение). Кроме того, не будет тратиться дополнительное время на замену индексов паросочетания на корректные перед возвращением результата, то есть наследование от класса Graph улучшает показатели не только по памяти, но и по времени.

3.3. Выводы к главе 3

В результате изменений, описанных в пунктах 3.1-3.2, отпала необходимость в создании и использовании нескольких дополнительных массивов и переменных, что уменьшило количество используемой памяти. По результатам 40 измерений времени выполнения кода, производительность улучшилась в среднем в 1,8 раза.

Заключение

В данной курсовой работе были решены следующие задачи:

1. Проверка графа на двудольность;
2. Раскраска графа;
3. Задача о назначениях;
4. Поиск наибольшего паросочетания.

В задаче о назначениях был использован Венгерский алгоритм, основанный на поиске совершенного паросочетания минимального веса. Данный алгоритм конкурирует с методом ветвей и границ, который работает путем систематического перебора различных комбинаций назначений и последующего отсека (поиск границы) некоторых ветвей перебора. Был выбран именно Венгерский алгоритм, так как он более эффективен в ряде случаев, особенно для больших задач о назначениях.

В поиске наибольшего паросочетания был использован алгоритм Куна, основанный на поиске увеличивающих цепей, так как он эффективен и прост в реализации.

Для хранения графа в памяти был использован перечень рёбер. При этом существует отдельный массив, в котором хранятся данные о цвете вершин. Этот способ хранения графа основан на перечислении пар вершин, которые соединены ребром. Он занимает не очень много памяти относительно матрицы смежности и матрицы инцидентности.

В процессе оптимизации кода были внесены значительные изменения, которые привели к улучшению времени работы программы. Оптимизация вызова функций, улучшение структуры циклов и массивов, а также уменьшение условных ветвлений были проведены в целях повышения эффективности кода.

В заключение, двудольные графы имеют большое практическое применение, и реализация может быть полезна в разных областях, где требуется работа с данной структурой данных.

Список использованной литературы

1. Иванов, Б. Н. Дискретная математика. Алгоритмы и программы: учеб. Пособие / Б. Н. Иванов – Вологда: Изд-во Лаборатория Базовых Знаний, 2001 — 288 с. (185)
2. Липский, В. Комбинаторика для программистов / В. Липский — Москва: Изд-во Мир, 1988 – 217 с.
3. Деревенец, О. В. Графомания / О. В. Деревенец – Воронеж, 2022 – 737 с.
4. Емиличев, В. А. Лекции по теории графов / В. А. Емиличев, О. И. Мельников, В. И. Сарванов, Р. И. Тышкевич – Москва: Изд-во Ленанд, 2021 – 390 с.
5. Фог, А. Оптимизация программ на Си++. Руководство по оптимизации для платформ Windows, Linux и Mac / Агнер Фог – Изд-во Инженерный колледж Копенгагенского университета, 2011 – 145 с.

Приложение 1. Неоптимизированный код

```
#include <iostream>
#include <vector>
#include <queue>
#include <fstream>
#include <chrono>

using namespace std;

class Exception : public exception
{
private:
    string str;
public:
    Exception(string s)
    {
        str = s;
    }
    void print()
    {
        cout << "\nException: " << str;
    }
};

int** transpose(int** matrix, int n)
{
    int** res = new int* [n];
    for (int i = 0; i < n; i++)
        res[i] = new int[n];
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            res[i][j] = matrix[j][i];
    return res;
}

enum Color //Цвет вершины
{
    uncolor,
    blue,
    pink
};

struct Edge //Ребро
{
    int weight;
    int vertex1;
    int vertex2;
    Edge(int w = 0, int v1 = 0, int v2 = 0)
    {
        weight = w;
        vertex1 = v1;
        vertex2 = v2;
    }
    bool operator==(Edge e)
    {
        return weight == e.weight and vertex1 == e.vertex1 and vertex2 ==
e.vertex2;
    }

    Edge operator=(Edge e)
    {
        weight = e.weight;
        vertex1 = e.vertex1;
```

```

        vertex2 = e.vertex2;
        return *this;
    }
};

ostream& operator<<(ostream& s, Edge e)
{
    s << '\n' << "[" << e.vertex1 << ", " << e.vertex2 << "], " << "weight = " << e.weight;
    return s;
}

istream& operator>>(istream& stream, Edge& e)
{
    stream >> e.vertex1;
    stream >> e.vertex2;
    stream >> e.weight;
    return stream;
}

class BipartiteGraph //Класс двудольный граф
{
private:
    vector<Edge> edges; //Массив ребер
    int CountVertices; //Количество вершин
    vector<Color> colors; //Цвета вершин

    bool fBFS(int vertex1, int vertex2, queue<int>& q) //
    {
        if (colors[vertex2] == uncolor)
        {
            q.push(vertex2);
            colors[vertex2] = colors[vertex1] == blue ? pink : blue;
            return true;
        }
        if (colors[vertex1] == colors[vertex2])
        {
            for (int j = 0; j < CountVertices; j++)
                colors[j] = uncolor;
            return false;
        }
        return true;
    }

    bool BFS() //Раскраска с помощью поиска в ширину
    {
        for (int j = 0; j < CountVertices; j++)
            colors[j] = uncolor;
        queue<int> q;
        q.push(edges[0].vertex1);
        colors[edges[0].vertex1] = blue;
        bool r = false;
        while (true)
        {
            r = true;
            int s;
            for (int j = 0; j < CountVertices; j++)
            {
                if (colors[j] == uncolor)
                {
                    r = false;
                    s = j;
                    break;
                }
            }
        }
    }
};

```

```

    }
    if (r and q.empty())
        return true;
    int cur_vertex;
    if (!q.empty())
    {
        cur_vertex = q.front();
        q.pop();
    }
    else
    {
        cur_vertex = s;
        colors[cur_vertex] = blue;
    }
    for (int i = 0; i < edges.size(); i++)
    {
        if (edges[i].vertex1 == cur_vertex)
        {
            if (!fBFS(cur_vertex, edges[i].vertex2, q))
                return false;
        }
        else if (edges[i].vertex2 == cur_vertex)
        {
            if (!fBFS(cur_vertex, edges[i].vertex1, q))
                return false;
        }
    }
}
return true;
}

bool DFS(Color p, int vertex = 0) //Раскраска с помощью поиска в глубину
{
    colors[vertex] = p == blue ? pink : blue;
    for (int i = 0; i < edges.size(); i++)
    {
        if (edges[i].vertex1 == vertex)
        {
            if (colors[edges[i].vertex2] == colors[vertex])
            {
                for (int i = 0; i < CountVertices; i++)
                    colors[i] = uncolor;
                return false;
            }
            else
            {
                if (colors[edges[i].vertex2] == uncolor)
                    return DFS(colors[vertex], edges[i].vertex2);
            }
        }
        else if (edges[i].vertex2 == vertex)
        {
            if (colors[edges[i].vertex1] == colors[vertex])
            {
                for (int i = 0; i < CountVertices; i++)
                    colors[i] = uncolor;
                return false;
            }
            else
            {
                if (colors[edges[i].vertex1] == uncolor)
                    return DFS(colors[vertex], edges[i].vertex1);
            }
        }
    }
}
bool r = true;

```

```

        for (int i = 0; i < CountVertices; i++)
            if (colors[i] == uncolor)
                r = false;
        if (r)
            return true;
    }

    bool FindMaxMatch(vector<Edge>& res, vector<Edge> M, vector<int> busy_V,
vector<int>& busy_V2, int vertex)
    {
        if (busy_V[vertex] == 0)
        {
            for (int i = 0; i < edges.size(); i++)
            {
                if (edges[i].vertex1 == vertex)
                {
                    if (busy_V[edges[i].vertex2] == 0)
                    {
                        res.push_back(edges[i]);
                        busy_V2[edges[i].vertex2] = 1;
                        busy_V2[vertex] = 1;
                        return true;
                    }
                }
                if (edges[i].vertex2 == vertex)
                {
                    if (busy_V[edges[i].vertex1] == 0)
                    {
                        res.push_back(edges[i]);
                        busy_V2[edges[i].vertex1] = 1;
                        busy_V2[vertex] = 1;
                        return true;
                    }
                }
            }
        }

        busy_V2[vertex] = 1;
        bool a = false;
        for (int i = 0; i < edges.size(); i++)
        {
            if (edges[i].vertex1 == vertex)
                if (busy_V2[edges[i].vertex2] == 0)
                    a = true;
            if (edges[i].vertex2 == vertex)
                if (busy_V2[edges[i].vertex1] == 0)
                    a = true;
        }
        if (!a)
        {
            if (busy_V[vertex] == 0)
                return true;
            int ind = res.size() - 1;
            busy_V2[res[ind].vertex1] = 0;
            busy_V2[res[ind].vertex2] = 0;
            res.pop_back();
            res.pop_back();
            return false;
        }

        for (int i = 0; i < edges.size(); i++)
        {

```

```

0)         if (edges[i].vertex1 == vertex and busy_V2[edges[i].vertex2] ==
{
    int ver = vertex;
    vertex = edges[i].vertex2;
    res.push_back(edges[i]);
    busy_V2[vertex] = 1;
    bool norm = false;
    for (int j = 0; j < M.size(); j++)
    {
        if (M[j].vertex1 == vertex)
        {
            vertex = M[j].vertex2;
            res.push_back(M[j]);
            norm = true;
        }
        else if (M[j].vertex2 == vertex)
        {
            vertex = M[j].vertex1;
            res.push_back(M[j]);
            norm = true;
        }
    }
    if (!norm)
        return true;
    busy_V2[vertex] = 1;

    bool lucky = FindMaxMatch(res, M, busy_V, busy_V2, vertex);
    if (!lucky)
    {
        vertex = ver;
        continue;
    }
    else
        return true;
}

0)         if (edges[i].vertex2 == vertex and busy_V2[edges[i].vertex1] ==
{
    int ver = vertex;
    vertex = edges[i].vertex1;
    res.push_back(edges[i]);
    busy_V2[vertex] = 1;
    bool norm = false;
    for (int j = 0; j < M.size(); j++)
    {
        if (M[j].vertex1 == vertex)
        {
            vertex = M[j].vertex2;
            res.push_back(M[j]);
            norm = true;
        }
        else if (M[j].vertex2 == vertex)
        {
            vertex = M[j].vertex1;
            res.push_back(M[j]);
            norm = true;
        }
    }
    if (!norm)
        return true;
    busy_V2[vertex] = 1;

```

```

        bool lucky = FindMaxMatch(res, M, busy_V, busy_V2, vertex);
        if (!lucky)
        {
            vertex = ver;
            continue;
        }
        else
            return true;
    }
}
return false;
}

BipartiteGraph NewGraph(int countB, int**& matrix, vector<int>& temp)
{
    for (int i = 1; i < countB + 1; i++)
    {
        int min = matrix[i][1];
        for (int j = 2; j < countB + 1; j++)
            if (matrix[i][j] < min) //Находим минимум в каждой строке
                min = matrix[i][j];
        for (int j = 1; j < countB + 1; j++)
            matrix[i][j] = matrix[i][j] - min; //Вычитаем минимум из
каждого элемента этой строки
    }
    BipartiteGraph graph;
    //vector<int> temp(countVertices);
    vector<bool> change(countVertices, false);
    int w = 0;
    int w2, w3;
    for (int i = 1; i < countB + 1; i++)
    {
        for (int j = 1; j < countB + 1; j++)
        {
            if (matrix[i][j] == 0)
            {
                if (change[matrix[i][0]] == false)
                {
                    temp[matrix[i][0]] = w;
                    w2 = w;
                    w++;
                    change[matrix[i][0]] = true;
                }
                else
                {
                    w2 = temp[matrix[i][0]];
                }
                if (change[matrix[0][j]] == false)
                {
                    temp[matrix[0][j]] = w;
                    w3 = w;
                    w++;
                    change[matrix[0][j]] = true;
                }
                else
                {
                    w3 = temp[matrix[0][j]];
                }
                graph.push(w2, w3);
            }
        }
    }
}

```

```

        return graph;
    }

public:
    BipartiteGraph()
    {
        CountVertices = 0;
        colors.resize(100, uncolor);
    }

    void push(int v1, int v2, int w = 0)
    {
        if (v1 == v2)
        {
            //Исключение
            throw Exception("You have violated the bipartiteness of the
graph!");
        }
        if (v1 == CountVertices and v2 == CountVertices + 1 or v2 ==
CountVertices and v1 == CountVertices + 1)
        {
            if (CountVertices + 2 <= 100)
                CountVertices = CountVertices + 2;
            else
            {
                //Исключение
                throw Exception("The graph cannot have more than 100
vertices!");
            }
            Edge newEdge(w, v1, v2);
            edges.push_back(newEdge);
            BFS();
            return;
        }
        if (v1 == CountVertices and v2 < CountVertices or v2 ==
CountVertices and v1 < CountVertices)
        {
            if (CountVertices + 1 <= 100)
                CountVertices = CountVertices + 1;
            else
            {
                //Исключение
                throw Exception("The graph cannot have more than 100
vertices!");
            }
            Edge newEdge(w, v1, v2);
            edges.push_back(newEdge);
            BFS();
            return;
        }
        if (v1 < CountVertices and v2 < CountVertices)
        {
            Edge newEdge(w, v1, v2);
            edges.push_back(newEdge);
            bool r = BFS();
            if (!r)
            {
                //Исключение
                throw Exception("You have violated the bipartiteness of the
graph!");
            }
        }
    }
}

```



```

BipartiteGraph operator=(BipartiteGraph graph)
{
    edges = graph.edges;
    CountVertices = graph.CountVertices;
    colors = graph.colors;
    for (int i = 0; i < CountVertices; i++)
        colors[i] = graph.colors[i];
    return *this;
}

vector<Edge> MaxMatch()
{
    vector<Edge> res;
    vector<Edge> M;
    vector<Edge> M2;
    vector<int> busy_V(CountVertices);
    vector<int> busy_V2(CountVertices);
    for (int i = 0; i < CountVertices; i++)
    {
        bool r;
        if (busy_V[i] == 0)
        {
            r = FindMaxMatch(res, M, busy_V, busy_V2, i);
            if (!r)
            {
                for (int j = 0; j < CountVertices; j++)
                    busy_V2[j] = 0;
                res.clear();
                continue;
            }
            for (int j = 0; j < M.size(); j++)
            {
                bool p = true;
                for (int k = 0; k < res.size(); k++)
                {
                    if (M[j] == res[k])
                        p = false;
                }
                if (p)
                    M2.push_back(M[j]);
            }

            for (int j = 0; j < res.size(); j++)
            {
                bool p = true;
                for (int k = 0; k < M.size(); k++)
                {
                    if (res[j] == M[k])
                        p = false;
                }
                if (p)
                    M2.push_back(res[j]);
            }

            M = M2;

            for (int j = 0; j < CountVertices; j++)
                busy_V2[j] = 0;

            for (int j = 0; j < CountVertices; j++)
                busy_V[j] = 0;
        }
    }
}

```

```

        for (int j = 0; j < M.size(); j++)
        {
            busy_V[M[j].vertex1] = 1;
            busy_V[M[j].vertex2] = 1;
        }

        res.clear();
        M2.clear();
    }
}

return M;
}

bool perfect(vector<Edge> p)
{
    bool res = true;
    for (int i = 0; i < CountVertices; i++)
    {
        bool a = false;
        if (colors[i] == blue)
        {
            for (int j = 0; j < p.size(); j++)
            {
                if (p[j].vertex1 == i or p[j].vertex2 == i)
                    a = true;
            }
            if (!a)
            {
                res = false;
                break;
            }
        }
    }
    return res;
}

vector<Edge> appointment() //Задача о назначениях
{
    int countB = 0; //Количество вершин первой доли
    int countP = 0; //Количество вершин второй доли
    for (int i = 0; i < CountVertices; i++)
        if (colors[i] == blue)
            countB++;

    for (int i = 0; i < CountVertices; i++)
        if (colors[i] == pink)
            countP++;

    int c = CountVertices;

    if (countP > countB)
    {
        for (int i = c; i < c + countP - countB; i++)
            for (int j = 0; j < c; j++)
                if (colors[j] == pink)
                    push(i, j);
        countB = countP;
    }

    if (countB > countP)
    {
        for (int i = c; i < c + countB - countP; i++)
            for (int j = 0; j < c; j++)

```

```

        if (colors[j] == blue)
            push(i, j);
        countP = countB;
    }

    int** matrix = new int* [countB + 1];
    for (int i = 0; i < countB + 1; i++)
        matrix[i] = new int[countB + 1];

    int k = 1;
    for (int i = 0; i < CountVertices; i++) //Заполнение нулевого
        столбца матрицы номерами голубых вершин
        {
            if (colors[i] == blue)
            {
                matrix[k][0] = i;
                k++;
            }
        }

    int j = 1;
    for (int i = 0; i < CountVertices; i++) //Заполнение нулевой строки
        матрицы номерами розовых вершин
        {
            if (colors[i] == pink)
            {
                matrix[0][j] = i;
                j++;
            }
        }

    matrix[0][0] = -1;
    for (int i = 0; i < edges.size(); i++) //Заполнение матрицы
    {
        int z = 0;
        int z2 = 0;
        for (int j = 1; j < countB + 1; j++)
            if (matrix[0][j] == edges[i].vertex1 or matrix[0][j] ==
edges[i].vertex2)
                z = j;
        for (int j = 1; j < countB + 1; j++)
            if (matrix[j][0] == edges[i].vertex1 or matrix[j][0] ==
edges[i].vertex2)
                z2 = j;
        matrix[z2][z] = edges[i].weight;
    }

    while (true)
    {
        vector<int> temp(CountVertices);
        BipartiteGraph graph = NewGraph(countB, matrix, temp);

        vector<Edge> max_m = graph.MaxMatch();
        vector<Edge> max_m_2(max_m.size());
        if (graph.perfect(max_m))
        {
            for (int i = 0; i < max_m.size(); i++)
            {
                for (int j = 0; j < CountVertices; j++)
                {
                    if (temp[j] == max_m[i].vertex1)
                        max_m_2[i].vertex1 = j;
                    else if (temp[j] == max_m[i].vertex2)

```

```

        max_m_2[i].vertex2 = j;
    }
}
for (int i = 0; i < edges.size(); i++)
    for (int j = 0; j < max_m_2.size(); j++)
        if (edges[i].vertex1 == max_m_2[j].vertex1 and
edges[i].vertex2 == max_m_2[j].vertex2)
            max_m_2[j].weight = edges[i].weight;
    return max_m_2;
}
else
{
    matrix = transpose(matrix, countB + 1);
    vector<int> temp2(countVertices);
    BipartiteGraph graph = NewGraph(countB, matrix, temp2);
    vector<Edge> max_m2 = graph.MaxMatch();
    vector<Edge> max_m2_2(max_m2.size());
    bool p = graph.perfect(max_m2);
    for (int i = 0; i < max_m2.size(); i++)
    {
        for (int j = 0; j < countVertices; j++)
        {
            if (temp2[j] == max_m2[i].vertex1)
                max_m2_2[i].vertex1 = j;
            else if (temp2[j] == max_m2[i].vertex2)
                max_m2_2[i].vertex2 = j;
        }
    }
    if (p)
    {
        for (int i = 0; i < edges.size(); i++)
            for (int j = 0; j < max_m2_2.size(); j++)
                if (edges[i].vertex1 == max_m2_2[j].vertex1 and
edges[i].vertex2 == max_m2_2[j].vertex2)
                    max_m2_2[j].weight = edges[i].weight;
        return max_m2_2;
    }
    else
    {
        matrix = transpose(matrix, countB + 1);
        vector<int> lwa; //Строки без назначений
        for (int i = 1; i < countB + 1; i++)
        {
            bool ok = true;
            for (int j = 1; j < countB + 1; j++)
            {
                if (matrix[i][j] == 0)
                {
                    for (int k = 0; k < max_m2.size(); k++)
                    {
                        if (matrix[i][0] == max_m2_2[k].vertex1
and matrix[0][j] == max_m2_2[k].vertex2 or matrix[i][0] ==
max_m2_2[k].vertex2 and matrix[0][j] == max_m2_2[k].vertex1)
                        {
                            ok = false;
                        }
                    }
                }
            }
            if (ok)
                lwa.push_back(i);
        }
        vector<int> cwz; //Столбцы с нулями в этих строках
    }
}

```

```

        for (int i = 0; i < lwa.size(); i++)
            for (int j = 1; j < countB + 1; j++)
                if (matrix[lwa[i]][j] == 0)
                    cwz.push_back(j);

        vector<int> rz; //Строки с "красными" нулями в этих
        столбцах

        for (int i = 0; i < cwz.size(); i++)
            for (int j = 1; j < countB + 1; j++)
                if (matrix[j][cwz[i]] == 0)
                    for (int k = 0; k < max_m2_2.size(); k++)
                        if (matrix[j][0] == max_m2_2[k].vertex1
and matrix[0][cwz[i]] == max_m2_2[k].vertex2 or matrix[j][0] ==
max_m2_2[k].vertex2 and matrix[0][cwz[i]] == max_m2_2[k].vertex1)
                            rz.push_back(j);

        vector<int> unmarked; //Неотмеченные строки
        for (int i = 1; i < countB + 1; i++)
        {
            bool ok = true;
            for (int j = 0; j < lwa.size(); j++)
                if (lwa[j] == i)
                    ok = false;
            for (int j = 0; j < rz.size(); j++)
                if (rz[j] == i)
                    ok = false;
            if (ok)
                unmarked.push_back(i);
        }
        for (int i = 0; i < rz.size(); i++)
            lwa.push_back(rz[i]); //Все не покрытые линиями
        строки

        vector<int> ncwz; //Все не покрытые линиями столбцы
        for (int i = 1; i < countB + 1; i++)
        {
            bool ok = true;
            for (int j = 0; j < cwz.size(); j++)
            {
                if (i == cwz[j])
                    ok = false;
            }
            if (ok)
                ncwz.push_back(i);
        }

        int min = 100000000;
        for (int i = 0; i < lwa.size(); i++)
            for (int j = 0; j < ncwz.size(); j++)
                if (matrix[lwa[i]][ncwz[j]] < min)
                    min = matrix[lwa[i]][ncwz[j]];

        for (int i = 0; i < lwa.size(); i++)
            for (int j = 0; j < ncwz.size(); j++)
                matrix[lwa[i]][ncwz[j]] =
matrix[lwa[i]][ncwz[j]] - min; //Вычитаем из всех непокрытых элементов
минимальный из них

        for (int i = 0; i < unmarked.size(); i++)
            for (int j = 0; j < cwz.size(); j++)
                matrix[unmarked[i]][cwz[j]] =
matrix[unmarked[i]][cwz[j]] + min;

        vector<int> temp3(CountVertices);

```

```

BipartiteGraph graph = NewGraph(countB, matrix, temp3);
vector<Edge> max_m3 = graph.MaxMatch();
vector<Edge> max_m3_2(max_m3.size());
bool p1 = graph.perfect(max_m3);
for (int i = 0; i < max_m3.size(); i++)
{
    for (int j = 0; j < CountVertices; j++)
    {
        if (temp3[j] == max_m3[i].vertex1)
        {
            max_m3_2[i].vertex1 = j;
        }
        else if (temp3[j] == max_m3[i].vertex2)
        {
            max_m3_2[i].vertex2 = j;
        }
    }
}
if (p1)
{
    for (int i = 0; i < edges.size(); i++)
        for (int j = 0; j < max_m3_2.size(); j++)
            if (edges[i].vertex1 == max_m3_2[j].vertex1
and edges[i].vertex2 == max_m3_2[j].vertex2)
                max_m3_2[j].weight = edges[i].weight;
    return max_m3_2;
}
}
}
}

friend ostream& operator<<(ostream& stream, BipartiteGraph& graph);
friend istream& operator>>(istream& stream, BipartiteGraph& graph);
};

ostream& operator<<(ostream& stream, BipartiteGraph& graph)
{
    for (int i = 0; i < graph.edges.size(); i++)
    {
        stream << graph.edges[i] << ", " << "color vertex1: " <<
graph.colors[graph.edges[i].vertex1] << ", color vertex2: " <<
graph.colors[graph.edges[i].vertex2];
    }
    return stream;
}

istream& operator>>(istream& stream, BipartiteGraph& graph)
{
    int countE;
    stream >> countE;
    for (int i = 0; i < countE; i++)
    {
        Edge n;
        stream >> n;
        graph.push(n.vertex1, n.vertex2, n.weight);
    }
    return stream;
}

int main()
{
    try

```

```

{
    BipartiteGraph error4;
    error4.push(0, 1, 7);
    error4.push(0, 2, 3);
    error4.push(0, 3, 6);
    error4.push(0, 4, 9);
    error4.push(0, 5, 5);
    error4.push(6, 1, 7);
    error4.push(6, 2, 5);
    error4.push(6, 3, 7);
    error4.push(6, 4, 5);
    error4.push(6, 5, 6);
    error4.push(7, 1, 7);
    error4.push(7, 2, 6);
    error4.push(7, 3, 8);
    error4.push(7, 4, 8);
    error4.push(7, 5, 9);
    error4.push(8, 1, 3);
    error4.push(8, 2, 1);
    error4.push(8, 3, 6);
    error4.push(8, 4, 5);
    error4.push(8, 5, 7);
    error4.push(9, 1, 2);
    error4.push(9, 2, 4);
    error4.push(9, 3, 9);
    error4.push(9, 4, 9);
    error4.push(9, 5, 5);

    vector<Edge> app = error4.appointment();
    cout << error4;
    cout << '\n';

    for (int i = 0; i < app.size(); ++i)
        cout << app[i];
}
catch (Exception e)
{
    e.print();
}
char c; cin >> c;
return 0;
}

```

Листинг 42. Неоптимизированный код

Приложение 2. Оптимизированный код

```
#include <iostream>
#include <vector>
#include <queue>
#include <fstream>
#include <chrono>

using namespace std;

class Exception : public exception
{
private:
    string str;
public:
    Exception(string s)
    {
        str = s;
    }
    void print()
    {
        cout << "\nException: " << str;
    }
};

class Graph
{
protected:
    int** mat;
    int CountVertices;
public:
    Graph(int c = 20)
    {
        CountVertices = c;
        mat = new int* [c];
        for (int i = 0; i < c; ++i)
        {
            mat[i] = new int[c];
            for (int j = 0; j < c; ++j)
                mat[i][j] = 0;
        }
    }

    void push(int v1, int v2, int w = 1)
    {
        if (v1 < CountVertices and v2 < CountVertices)
        {
            mat[v1][v2] = w;
            mat[v2][v1] = w;
        }
    }

    ~Graph()
    {
        for (int i = 0; i < CountVertices; ++i)
            delete[] mat[i];
        delete[] mat;
    }
};

enum Color //Цвет вершины
{
    uncolor,
    blue,
```



```

    pink
};

struct Edge //Ребро
{
    int weight;
    int vertex1;
    int vertex2;
    Edge(int w = 0, int v1 = 0, int v2 = 0)
    {
        weight = w;
        vertex1 = v1;
        vertex2 = v2;
    }
    bool operator==(Edge e)
    {
        return weight == e.weight and vertex1 == e.vertex1 and vertex2 ==
e.vertex2;
    }

    Edge operator=(Edge e)
    {
        weight = e.weight;
        vertex1 = e.vertex1;
        vertex2 = e.vertex2;
        return *this;
    }
};

ostream& operator<<(ostream& s, Edge e)
{
    s << '\n' << "[" << e.vertex1 << ", " << e.vertex2 << "], " << "weight =
" << e.weight;
    return s;
}

istream& operator>>(istream& stream, Edge& e)
{
    stream >> e.vertex1;
    stream >> e.vertex2;
    stream >> e.weight;
    return stream;
}

class BipartiteGraph : public Graph //Класс двудольный граф
{
private:
    int CountEdges;
    Edge edges[225];
    Color colors[30];

    bool BFS() //Раскраска с помощью поиска в ширину
    {
        Color* ptr_end = colors + CountVertices;
        for (Color* p = colors; p < ptr_end; ++p)
            *p = uncolor;
        queue<int> q;
        q.push(edges[0].vertex1);
        colors[edges[0].vertex1] = blue;
        bool r = false;
        while (true)
        {
            r = true;

```

```

int s;
bool m;
for (int j = 0; j < CountVertices; ++j)
{
    m = false;
    for (int i = 0; i < CountVertices; ++i)
    {
        if (mat[j][i] != 0)
            m = true;
    }
    if (colors[j] == uncolor and m)
    {
        r = false;
        s = j;
        break;
    }
}
if (r and q.empty())
    return true;
int cur_vertex;
if (!q.empty())
{
    cur_vertex = q.front();
    q.pop();
}
else
{
    cur_vertex = s;
    colors[cur_vertex] = blue;
}

Edge* ptr_end = edges + CountEdges;

for (Edge* p = edges; p < ptr_end; ++p)
{
    if ((*p).vertex1 == cur_vertex)
    {
        if (colors[(*p).vertex2] == uncolor)
        {
            q.push((*p).vertex2);
            colors[(*p).vertex2] = colors[cur_vertex] == blue ?
pink : blue;
        }
        else if (colors[cur_vertex] == colors[(*p).vertex2])
        {
            for (int j = 0; j < CountVertices; ++j)
                colors[j] = uncolor;
            return false;
        }
    }
    else if ((*p).vertex2 == cur_vertex)
    {
        if (colors[(*p).vertex1] == uncolor)
        {
            q.push((*p).vertex1);
            colors[(*p).vertex1] = colors[cur_vertex] == blue ?
pink : blue;
        }
        else if (colors[cur_vertex] == colors[(*p).vertex1])
        {
            for (int j = 0; j < CountVertices; ++j)
                colors[j] = uncolor;
            return false;
        }
    }
}

```

```

    }
    }
    }
    return true;
}

bool DFS(Color p, int vertex = 0) //Раскраска с помощью поиска в глубину
{
    colors[vertex] = p == blue ? pink : blue;
    for (int i = 0; i < CountEdges; ++i)
    {
        if (edges[i].vertex1 == vertex)
        {
            if (colors[edges[i].vertex2] == colors[vertex])
            {
                for (int i = 0; i < CountVertices; ++i)
                    colors[i] = uncolor;
                return false;
            }
            else
            {
                if (colors[edges[i].vertex2] == uncolor)
                    return DFS(colors[vertex], edges[i].vertex2);
            }
        }
        else if (edges[i].vertex2 == vertex)
        {
            if (colors[edges[i].vertex1] == colors[vertex])
            {
                for (int i = 0; i < CountVertices; ++i)
                    colors[i] = uncolor;
                return false;
            }
            else
            {
                if (colors[edges[i].vertex1] == uncolor)
                    return DFS(colors[vertex], edges[i].vertex1);
            }
        }
    }
    bool r = true;
    for (int i = 0; i < CountEdges; ++i)
        if (colors[edges[i].vertex1] == uncolor or
            colors[edges[i].vertex2] == uncolor)
        {
            r = false;
            break;
        }
    if (r)
        return true;
}

bool FindMaxMatch(vector<Edge>& res, vector<Edge> M, vector<int> busy_V,
vector<int>& busy_V2, int vertex)
{
    if (busy_V[vertex] == 0)
    {
        Edge* ptr_end = edges + CountEdges;
        for (Edge* p = edges; p < ptr_end; ++p)
        {
            if ((*p).vertex1 == vertex)
            {
                if (busy_V[(*p).vertex2] == 0)
                {
                    res.push_back((*p));
                    busy_V2[(*p).vertex2] = 1;
                }
            }
        }
    }
}

```

```

        busy_V2[vertex] = 1;
        return true;
    }
}
if ((*p).vertex2 == vertex)
{
    if (busy_V[(*p).vertex1] == 0)
    {
        res.push_back((*p));
        busy_V2[(*p).vertex1] = 1;
        busy_V2[vertex] = 1;
        return true;
    }
}
}
}

busy_V2[vertex] = 1;
bool a = false;
Edge* ptr_end = edges + CountEdges;

for (Edge* p = edges; p < ptr_end; ++p)
{
    if ((*p).vertex1 == vertex)
        if (busy_V2[(*p).vertex2] == 0)
        {
            a = true;
            break;
        }
    if ((*p).vertex2 == vertex)
        if (busy_V2[(*p).vertex1] == 0)
        {
            a = true;
            break;
        }
}
if (!a)
{
    if (busy_V[vertex] == 0)
        return true;
    int ind = res.size() - 1;
    busy_V2[res[ind].vertex1] = 0;
    busy_V2[res[ind].vertex2] = 0;
    res.pop_back();
    res.pop_back();
    return false;
}
int ver;
bool norm;
int M_size = M.size();

for (Edge* p = edges; p < ptr_end; ++p)
{
    if ((*p).vertex1 == vertex and busy_V2[(*p).vertex2] == 0)
    {
        ver = vertex;
        vertex = (*p).vertex2;
        res.push_back((*p));
        busy_V2[vertex] = 1;
        norm = false;
        for (int j = 0; j < M_size; ++j)
        {
            if (M[j].vertex1 == vertex or M[j].vertex2 == vertex)

```

```

        {
            vertex = M[j].vertex1 == vertex ? M[j].vertex2 :
M[j].vertex1;
            res.push_back(M[j]);
            norm = true;
            break;
        }
    }
    if (!norm)
        return true;
    busy_V2[vertex] = 1;

    bool lucky = FindMaxMatch(res, M, busy_V, busy_V2, vertex);
    if (!lucky)
    {
        vertex = ver;
        continue;
    }
    else
        return true;
}

if ((*p).vertex2 == vertex and busy_V2[(*p).vertex1] == 0)
{
    ver = vertex;
    vertex = (*p).vertex1;
    res.push_back((*p));
    busy_V2[vertex] = 1;
    norm = false;

    for (int j = 0; j < M_size; ++j)
    {
        if (M[j].vertex1 == vertex or M[j].vertex2 == vertex)
        {
            vertex = M[j].vertex1 == vertex ? M[j].vertex2 :
M[j].vertex1;
            res.push_back(M[j]);
            norm = true;
            break;
        }
    }
    if (!norm)
        return true;
    busy_V2[vertex] = 1;

    bool lucky = FindMaxMatch(res, M, busy_V, busy_V2, vertex);
    if (!lucky)
    {
        vertex = ver;
        continue;
    }
    else
        return true;
}

}

if (res.size() > 1)
{
    int ind = res.size() - 1;
    busy_V2[res[ind].vertex1] = 0;
    busy_V2[res[ind].vertex2] = 0;
}
if (!res.empty())

```

```

        res.pop_back();
        if (!res.empty())
            res.pop_back();
        return false;
    }

BipartiteGraph NewGraph(int countB, int**& matrix, int** mt2)
{
    countB += 1;
    for (int i = 1; i < countB; ++i)
    {
        int min = matrix[i][1];
        for (int j = 2; j < countB; ++j)
            if (matrix[i][j] < min) //Находим минимум в каждой строке
                min = matrix[i][j];
        int countB_2;
        int os = countB % 2;
        switch(os)
        {
            case 0:
                countB_2 = countB / 2;
                matrix[i][countB_2] = matrix[i][countB_2] - min;
                for (int j = 1; j < countB_2; ++j)
                {
                    matrix[i][j] = matrix[i][j] - min;
                    matrix[i][countB_2 + j] = matrix[i][countB_2 + j] - min;
                }
                break;
            default:
                countB_2 = countB / 2 + 1;
                for (int j = 1; j < countB_2; ++j)
                {
                    matrix[i][j] = matrix[i][j] - min;
                    matrix[i][countB_2 + j - 1] = matrix[i][countB_2 + j -
1] - min;
                }
                break;
        }
    }
    BipartiteGraph graph(CountVertices);
    for (int i = 1; i < countB; ++i)
    {
        for (int j = i; j < countB; ++j)
        {
            if (matrix[i][j] == 0)
                graph.push(matrix[i][0], matrix[0][j], mt2[i][j]);
            else if (matrix[j][i] == 0)
                graph.push(matrix[j][0], matrix[0][i], mt2[j][i]);
        }
    }
    return graph;
}

public:
    BipartiteGraph(int c = 20) : Graph(c)
    {
        Color* ptr_end = colors + 30;
        for (Color* p = colors; p < ptr_end; ++p)
            (*p) = uncolor;
        CountEdges = 0;
    }

    void push(int v1, int v2, int w = 1)

```

```

{
    if (v1 == v2)
    {
        //Исключение
        throw Exception("You have violated the bipartiteness of the
graph!");
    }

    if (v1 < CountVertices and v2 < CountVertices)
    {
        Edge newEdge(w, v1, v2);
        edges[CountEdges] = newEdge;
        CountEdges++;
        Graph::push(v1, v2, w);
        Color* ptr_end = colors + CountVertices;
        for (Color* p = colors; p < ptr_end; ++p)
            (*p) = uncolor;
        bool r = BFS();
        if (!r)
        {
            //Исключение
            throw Exception("You have violated the bipartiteness of the
graph!");
        }
    }
    else
        throw Exception("You can not add more vertices to the graph!");
}

BipartiteGraph operator=(BipartiteGraph graph)
{
    CountEdges = graph.CountEdges;
    for (int i = 0; i < CountEdges; ++i)
        edges[i] = graph.edges[i];
    CountVertices = graph.CountVertices;
    for (int i = 0; i < CountVertices; ++i)
        colors[i] = graph.colors[i];
    return *this;
}

vector<Edge> MaxMatch()
{
    vector<Edge> res;
    vector<Edge> M;
    vector<Edge> M2;
    vector<int> busy_V(CountVertices);
    vector<int> busy_V2(CountVertices);
    bool r, p;
    for (int i = 0; i < CountVertices; ++i)
    {
        if (busy_V[i] == 0)
        {
            r = FindMaxMatch(res, M, busy_V, busy_V2, i);
            if (!r)
            {
                for (int j = 0; j < CountVertices; ++j)
                    busy_V2[j] = 0;
                res.clear();
                continue;
            }
        }

        int ms = M.size();

```

```

        int rs = res.size();

        for (int j = 0; j < ms; ++j)
        {
            p = true;
            for (int k = 0; k < rs; ++k)
            {
                if (M[j] == res[k])
                {
                    p = false;
                    break;
                }
            }
            if (p)
                M2.push_back(M[j]);
        }

        for (int j = 0; j < rs; ++j)
        {
            p = true;
            for (int k = 0; k < ms; ++k)
            {
                if (res[j] == M[k])
                {
                    p = false;
                    break;
                }
            }
            if (p)
                M2.push_back(res[j]);
        }

        M = M2;

        fill(busy_V2.begin(), busy_V2.end(), 0);
        fill(busy_V.begin(), busy_V.end(), 0);

        ms = M.size();
        for (int j = 0; j < ms; ++j)
        {
            busy_V[M[j].vertex1] = 1;
            busy_V[M[j].vertex2] = 1;
        }

        res.clear();
        M2.clear();
    }
}

return M;
}

vector<Edge> appointment() //Задача о назначениях
{
    int countB = 0; //Количество вершин первой доли
    int countP = 0; //Количество вершин второй доли
    Color* ptr_end = colors + CountVertices;
    for (Color* p = colors; p < ptr_end; ++p)
    {
        switch (*p)
        {
            case blue:
                countB++;
                break;

```



```

        case pink:
            countP++;
            break;
    }
}

int c = countB + countP - 1;
if (countP > countB)
{
    int cpb = c + countP - countB;
    for (int i = c; i < cpb; ++i)
        for (int j = 0; j < c; ++j)
            if (colors[j] == pink)
                push(i, j, 0);
    countB = countP;
}

else if (countB > countP)
{
    int cbp = c + countB - countP;
    for (int i = c; i < cbp; ++i)
        for (int j = 0; j < c; ++j)
            if (colors[j] == blue)
                push(i, j, 0);
    countP = countB;
}

int countB1 = countB + 1;

int** matrix = new int* [countB1];
for (int i = 0; i < countB1; ++i)
    matrix[i] = new int[countB1];

int k = 1;
int j = 1;
for (int i = 0; i < CountVertices; ++i) //Заполнение нулевого
столбца матрицы номерами голубых вершин
{
    switch (colors[i])
    {
        case blue:
            matrix[k][0] = i;
            ++k;
            break;
        case pink:
            matrix[0][j] = i;
            ++j;
            break;
    }
}

matrix[0][0] = -1;
for (int i = 0; i < CountEdges; ++i) //Заполнение матрицы
{
    int z = 0;
    int z2 = 0;
    for (int j = 1; j < countB1; ++j)
    {
        if (matrix[0][j] == edges[i].vertex1 or matrix[0][j] ==
edges[i].vertex2)
        {
            z = j;
        }
    }
}

```

```

        if (matrix[j][0] == edges[i].vertex1 or matrix[j][0] ==
edges[i].vertex2)
        {
            z2 = j;
        }
        matrix[z2][z] = edges[i].weight;
    }

    int** matrix2 = new int* [countB1];
    for (int i = 0; i < countB1; ++i)
    {
        matrix2[i] = new int[countB1];
        for (int j = 0; j < countB1; ++j)
        {
            matrix2[i][j] = matrix[i][j];
        }
    }

    while (true)
    {
        BipartiteGraph graph = NewGraph(countB, matrix, matrix2);

        vector<Edge> max_m = graph.MaxMatch();
        int max_m_size = max_m.size();
        if (max_m_size == countB)
        {
            return max_m;
        }
        int** res = new int* [countB1];
        for (int i = 0; i < countB1; ++i)
        {
            res[i] = new int[countB1];
            for (int j = 0; j < countB1; ++j)
                res[i][j] = matrix[j][i];
        }

        matrix = res;
        BipartiteGraph graph2 = NewGraph(countB, matrix, matrix);
        vector<Edge> max_m2 = graph2.MaxMatch();
        int max_m2_size = max_m2.size();
        if (max_m2_size == countB)
        {
            Edge* ptr_edg = edges + CountEdges;
            for (Edge* p = edges; p < ptr_edg; ++p)
                for (int j = 0; j < max_m2_size; ++j)
                    if ((*p).vertex1 == max_m2[j].vertex1 and
(*p).vertex2 == max_m2[j].vertex2)
                        max_m2[j].weight = (*p).weight;
            return max_m2;
        }
        int** res2 = new int* [countB1];
        for (int i = 0; i < countB1; ++i)
        {
            res2[i] = new int[countB1];
            for (int j = 0; j < countB1; ++j)
                res2[i][j] = matrix[j][i];
        }

        matrix = res2;
        vector<int> lwa; //Строки без назначений
        vector<int> cwz; //Столбцы с нулями в этих строках
        for (int i = 1; i < countB1; i++)

```

```

    {
        bool ok = true;
        for (int k = 0; k < max_m2_size; ++k)
        {
            if (matrix[i][0] == max_m2[k].vertex2)
            {
                ok = false;
                break;
            }
        }
        if (ok)
        {
            lwa.push_back(i);
            for (int j = 1; j < countB1; ++j)
                if (matrix[i][j] == 0)
                    cwz.push_back(j);
        }
    }

    int lwa_size = lwa.size();

    int cwz_size = cwz.size();
    vector<int> rz; //Строки с "красными" нулями в этих столбцах
    for (int i = 0; i < cwz_size; ++i)
        for (int j = 1; j < countB1; ++j)
            if (matrix[j][cwz[i]] == 0)
                for (int k = 0; k < max_m2_size; ++k)
                    if (matrix[j][0] == max_m2[k].vertex2 and
matrix[0][cwz[i]] == max_m2[k].vertex1)
                        rz.push_back(j);

    int rz_size = rz.size();
    vector<int> unmarked; //Неотмеченные строки
    for (int i = 1; i < countB1; ++i)
    {
        bool ok = true;
        vector<int>::iterator it = lwa.begin();
        vector<int>::iterator it2 = lwa.end();
        for (; it != it2; it++)
            if (*it == i)
                ok = false;
        for (int j = 0; j < rz_size; ++j)
            if (rz[j] == i)
                ok = false;
        if (ok)
            unmarked.push_back(i);
    }
    int unmarked_size = unmarked.size();

    for (int i = 0; i < rz_size; ++i)
        lwa.push_back(rz[i]); //все не покрытые линиями строки
    lwa_size = lwa.size();
    vector<int> ncwz; //Все не покрытые линиями столбцы
    for (int i = 1; i < countB1; ++i)
    {
        bool ok = true;
        for (int j = 0; j < cwz_size; ++j)
        {
            if (i == cwz[j])
                ok = false;
        }
        if (ok)
            ncwz.push_back(i);
    }

```

```

    }

    int ncwz_size = ncwz.size();

    int min = 1000000000;
    for (int i = 0; i < lwa_size; ++i)
        for (int j = 0; j < ncwz_size; ++j)
            if (matrix[lwa[i]][ncwz[j]] < min)
                min = matrix[lwa[i]][ncwz[j]];

    for (int i = 0; i < lwa_size; ++i)
        for (int j = 0; j < ncwz_size; ++j)
            matrix[lwa[i]][ncwz[j]] = matrix[lwa[i]][ncwz[j]] - min;
    //Вычитаем из всех непокрытых элементов минимальный из них

    for (int i = 0; i < unmarked_size; ++i)
        for (int j = 0; j < cwz_size; ++j)
            matrix[unmarked[i]][cwz[j]] =
matrix[unmarked[i]][cwz[j]] + min;

    BipartiteGraph graph3(CountVertices);
    for (int i = 1; i < countB1; ++i)
        for (int j = 1; j < countB1; ++j)
            if (matrix[i][j] == 0)
                graph3.push(matrix[i][0], matrix[0][j],
matrix2[i][j]);

    vector<Edge> max_m3 = graph3.MaxMatch();
    int max_m3_size = max_m3.size();
    if (max_m3_size == countB)
    {
        return max_m3;
    }
}

friend ostream& operator<<(ostream& stream, BipartiteGraph& graph);
friend istream& operator>>(istream& stream, BipartiteGraph& graph);
};

ostream& operator<<(ostream& stream, BipartiteGraph& graph)
{
    for (int i = 0; i < graph.CountEdges; ++i)
    {
        stream << graph.edges[i] << ", " << "color vertex1: " <<
graph.colors[graph.edges[i].vertex1] << ", color vertex2: " <<
graph.colors[graph.edges[i].vertex2];
    }
    stream << '\n' << "::~ ";
    for (int i = 0; i < graph.CountVertices; ++i)
    {
        stream << i << " ";
    }
    for (int i = 0; i < graph.CountVertices; ++i)
    {
        stream << '\n' << i << ": ";
        for (int j = 0; j < graph.CountVertices; ++j)
        {
            stream << graph.mat[i][j] << " ";
        }
    }
    return stream;
}

```

```

istream& operator>>(istream& stream, BipartiteGraph& graph)
{
    int countE;
    stream >> countE;
    for (int i = 0; i < countE; ++i)
    {
        Edge n;
        stream >> n;
        graph.push(n.vertex1, n.vertex2, n.weight);
    }
    return stream;
}

int main()
{
    try
    {
        BipartiteGraph error4(11);
        error4.push(0, 1, 7);
        error4.push(0, 2, 3);
        error4.push(0, 3, 6);
        error4.push(0, 4, 9);
        error4.push(0, 5, 5);
        error4.push(6, 1, 7);
        error4.push(6, 2, 5);
        error4.push(6, 3, 7);
        error4.push(6, 4, 5);
        error4.push(6, 5, 6);
        error4.push(7, 1, 7);
        error4.push(7, 2, 6);
        error4.push(7, 3, 8);
        error4.push(7, 4, 8);
        error4.push(7, 5, 9);
        error4.push(8, 1, 3);
        error4.push(8, 2, 1);
        error4.push(8, 3, 6);
        error4.push(8, 4, 5);
        error4.push(8, 5, 7);
        error4.push(9, 1, 2);
        error4.push(9, 2, 4);
        error4.push(9, 3, 9);
        error4.push(9, 4, 9);
        error4.push(9, 5, 5);

        vector<Edge> app = error4.appointment();
        cout << error4;

        for (int i = 0; i < app.size(); ++i)
            cout << app[i];
        cout << '\n';
    }
    catch (Exception e)
    {
        e.print();
    }
    char c; cin >> c;
    return 0;
}

```

Листинг 43. Оптимизированный код