МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное учреждение высшего образования
"МИРЭА - Российский технологический университет"

РТУ МИРЭА

Институт информационных технологий (ИТ)
Кафедра математического обеспечения и стандартизации информационных технологий (МОСИТ)

ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ 8_1
по дисциплине
«Структуры и алгоритмы обработки данных»

Тема. Основные алгоритмы работы с графами. Применение графа в решении практических задач.

Выполнил студент группы ИКБО-60-23 Шеенко В.А

Принял старший преподаватель Скворцова Л.А.

Москва 2024

СОДЕРЖАНИЕ

1 ЦЕЛЬ РАБОТЫ

Получение практических навыков по выполнению операций над структурой данных граф.

ЗАДАНИЕ №1

2.1 Постановка задачи

Разработать структуру данных «Граф» в соответствии с вариантом индивидуального задания, обеспечивающий хранение и работу со структурой данных «граф». Вид графа – ориентированный.

Разработать и реализовать алгоритмы операций управления графом:

1. ввод графа с клавиатуры, наполнение графа осуществлять с помощью добавления одного ребер;
2. вывод графа монитор, представляя его списками смежных вершин;
3. алгоритмы задач, определенные вариантом индивидуального задания.

Таблица 1 – Задание индивидуального варианта

Номер варианта	Представление графа в памяти	Задачи
28	Матрица смежности	1) Определить количество простых циклов в графе. 2) Найти кратчайший путь между двумя заданными вершинами графа.

2.2 Ход решения

Для представления графа в программе будет использоваться матрица смежности в соответствии с зданием

Матрица смежности — это способ представления графа в виде таблицы (двумерного массива), где строки и столбцы соответствуют вершинам графа.

Структура матрицы смежности:

1. Размер матрицы:

   Матрица имеет размер $n \times n$, где n – количество вершин графа.

2. Элементы матрицы:

   a. Если между вершинами i и j есть связь (ребро), то в ячейке $A[i][j]$ стоит 1 (или вес ребра, если граф взвешенный).

b. Если связи нет, то записывается 0.

Метод Graph::print выводит матрицу смежности графа в консоль, чтобы можно было увидеть, как вершины связаны между собой. Он проходит по всем строкам и столбцам матрицы, выводя элементы через пробел. После каждой строки матрицы выполняется переход на новую строку, чтобы результат выглядел как таблица.

Метод Graph::addEdge добавляет ребро между вершинами графа, используя матрицу смежности. Она проверяет, что индексы вершин находятся в допустимом диапазоне. Если индексы некорректны, выбрасывается исключение. Если индексы правильные, в матрице по соответствующим позициям устанавливается вес ребра. Для неориентированного графа связь между вершинами записывается в обе позиции матрицы, чтобы связь была двусторонней.

В решении поставленных заданий будет использоваться *неориентированный граф*.

Описание алгоритмов, решающих задачи индивидуального варианта (28):

Метод Graph::cycleDfs выполняет поиск всех простых циклов в графе с использованием алгоритма поиска в глубину (DFS). Он рекурсивно обходит граф, начиная с заданной вершины, и сохраняет найденные циклы в множество уникальных циклов.

Метод Graph::getCountOfSimpleCycles вызывает cycleDfs для каждой вершины графа, чтобы найти все уникальные простые циклы. Он возвращает количество найденных уникальных циклов.

Метод Graph::shortestPath находит кратчайший путь между двумя вершинами в графе, используя алгоритм Дейкстры:

- Инициализирует векторы distance (расстояния до каждой вершины), parent (родительские вершины для восстановления пути) и visited (посещенные вершины).

- Устанавливает расстояние до начальной вершины равным 0.
- В цикле находит вершину с минимальным расстоянием, которая еще не посещена.
- Обновляет расстояния до смежных вершин, если найден более короткий путь.
- После завершения цикла восстанавливает путь от конечной вершины до начальной, используя вектор parent. В методе shortestPath для восстановления кратчайшего пути используется вектор parent. Этот вектор хранит информацию о родительских вершинах для каждой вершины, что позволяет восстановить путь от конечной вершины до начальной. Когда алгоритм Дейкстры находит более короткий путь до вершины, он обновляет значение вектора parent для этой вершины, указывая, из какой вершины был достигнут этот путь. После завершения алгоритма путь восстанавливается, начиная с конечной вершины и следуя по вектору parent до начальной вершины. Вершины добавляются в вектор path, который затем разворачивается, чтобы получить путь от начальной вершины до конечной.
- Возвращает пару, где первый элемент — это путь, а второй элемент - длина пути. Если путь не найден, возвращает пустой вектор и -1.

2.3 Программная реализация

2.3.1 Graph.h

```
#ifndef CODE_GRAPH_H
#define CODE_GRAPH_H

#include <vector>
#include <set>

class Graph {
private:
    int verticesCount;
    std::vector<std::vector<int>> adjMatrix;

    void cycleDfs(int start, int current, std::vector<bool>& visited,
std::vector<int>& path, std::set<std::vector<int>>& uniqueCycles, int depth);
public:
    explicit Graph(int verticesCount);
    void addEdge(int start, int end, int distance);
```

```
    int getCountOfSimpleCycles();
    std::pair<std::vector<int>, int> shortestPath(int start, int end);
    void print();

    ~Graph() = default;
};


#endif //CODE_GRAPH_H
```

2.3.2 Graph.cpp

```
#include <stdexcept>
#include <iostream>
#include <algorithm>
#include "Graph.h"

Graph::Graph(int verticesCount) {
    adjMatrix.resize(verticesCount, std::vector<int>(verticesCount, 0));
    this->verticesCount = verticesCount;
}

void Graph::addEdge(int start, int end, int distance) {
    if (start > 0 && start <= verticesCount && end > 0 && end <=
verticesCount) {
        adjMatrix[start - 1][end - 1] = distance;
        adjMatrix[end - 1][start - 1] = distance;
    } else {
        throw std::invalid_argument("Invalid vertex index");
    }
}

void Graph::print() {
    for (int i = 0; i < verticesCount; i++) {
        for (int j = 0; j < verticesCount; j++) {
            std::cout << adjMatrix[i][j] << " ";
        }
        std::cout << std::endl;
    }
}

void Graph::cycleDfs(int start, int current, std::vector<bool>& visited,
std::vector<int>& path, std::set<std::vector<int>>& uniqueCycles, int depth)
{
    visited[current] = true;
    path.push_back(current);

    for (int next = 0; next < verticesCount; ++next) {
        if (adjMatrix[current][next] != 0) {
            if (!visited[next]) {
                cycleDfs(start, next, visited, path, uniqueCycles, depth +
1);
            } else if (next == start && depth >= 2) {
                // Найден простой цикл
                std::vector<int> cycle = path;
                cycle.push_back(start);
                uniqueCycles.insert(cycle);
            }
        }
```

```
    }

    visited[current] = false; // Снимаем посещение для поиска других циклов
    path.pop_back();
}

int Graph::getCountOfSimpleCycles() {
    std::set<std::vector<int>> uniqueCycles;
    std::vector<bool> visited(verticesCount, false);
    std::vector<int> path;

    for (int i = 0; i < verticesCount; ++i) {
        cycleDfs(i, i, visited, path, uniqueCycles, 0);
    }

    return uniqueCycles.size();
}

std::pair<std::vector<int>, int> Graph::shortestPath(int start, int end) {
    std::vector<int> distance(verticesCount, INT_MAX);
    std::vector<int> parent(verticesCount, -1);
    std::vector<bool> visited(verticesCount, false);

    distance[start - 1] = 0;

    for (int i = 0; i < verticesCount - 1; i++) {
        int minDistance = INT_MAX;
        int minVertex = -1;

        for (int j = 0; j < verticesCount; j++) {
            if (!visited[j] && distance[j] < minDistance) {
                minDistance = distance[j];
                minVertex = j;
            }
        }

        visited[minVertex] = true;

        for (int j = 0; j < verticesCount; j++) {
            if (!visited[j] && adjMatrix[minVertex][j] != 0 &&
distance[minVertex] + adjMatrix[minVertex][j] < distance[j]) {
                distance[j] = distance[minVertex] + adjMatrix[minVertex][j];
                parent[j] = minVertex;
            }
        }
    }

    std::vector<int> path;
    for (int v = end - 1; v != -1; v = parent[v]) {
        path.push_back(v + 1);
    }
    std::reverse(path.begin(), path.end());

    int pathLength = distance[end - 1];
    if (path.size() == 1 && path[0] != start) {
        return {{}, -1};
    }

    return {path, pathLength};
}
```

2.3.3 main.cpp

```cpp
#include <iostream>
#include "Graph.h"

int main() {
    int vertices, edges;
    std::cout << "Enter the number of vertices and edges: ";
    std::cin >> vertices >> edges;

    Graph graph(vertices);

    for (int i = 0; i < edges; i++) {
        int start, end, distance;
        std::cout << "Enter the start, end and distance of the edge: ";
        std::cin >> start >> end >> distance;
        graph.addEdge(start, end, distance);
    }

    graph.print();

    // Задача 1: Количество простых циклов
    int simpleCycles = graph.getCountOfSimpleCycles();
    std::cout << "Number of simple cycles in the graph: " << simpleCycles <<
std::endl;

    // Задача 2: Кратчайший путь между двумя вершинами
    int start, end;
    std::cout << "Enter start and end vertices to find shortest path: ";
    std::cin >> start >> end;

    auto [path, distance] = graph.shortestPath(start, end);
    if (path.empty()) {
        std::cout << "Path not found." << std::endl;
    } else {
        std::cout << "Shortest path: ";
        for (int v : path) {
            std::cout << v << " ";
        }
        std::cout << "with distance: " << distance << std::endl;
    }

    std::cin.get();
    return 0;
}
```

2.4 Тестирование

В качестве «подопытного» возьмем следующий граф (рис. 1):

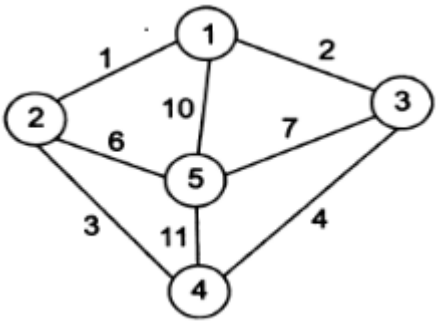

Рисунок 1 – Граф для теста №1

Кратчайший путь будем искать между вершинами 1 и 4. Результат работы программы показан на рисунке 2.

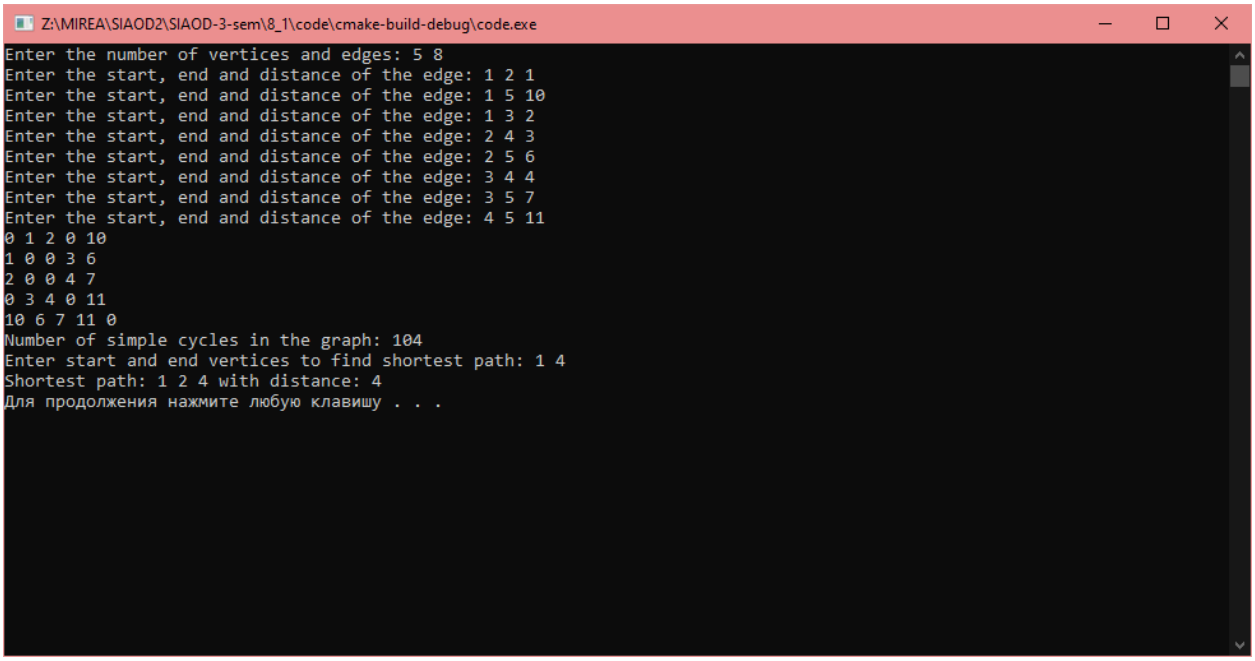

Рисунок 2 – Тестирование программы для теста №1

Для наглядности продемонстрируем работу алгоритма Дейкстры в таблице 2.

Таблица 2 – Работа алгоритма Дейкстра для теста №1

1	2	3	4	5
0	∞	∞	∞	∞
	1	2	∞	10
		2	4	7
			4	7
				7

Для второго теста был выбран граф (рис. 3):

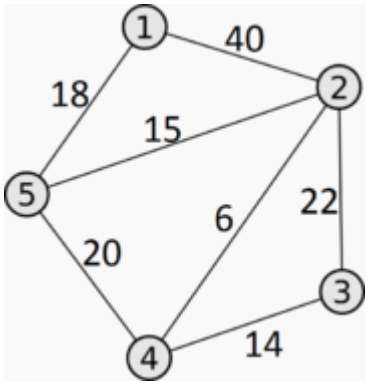

Рисунок 3 – Граф для теста №2

Кратчайший путь будем искать между вершинами 1 и 3. Результат работы программы показан на рисунке 4.

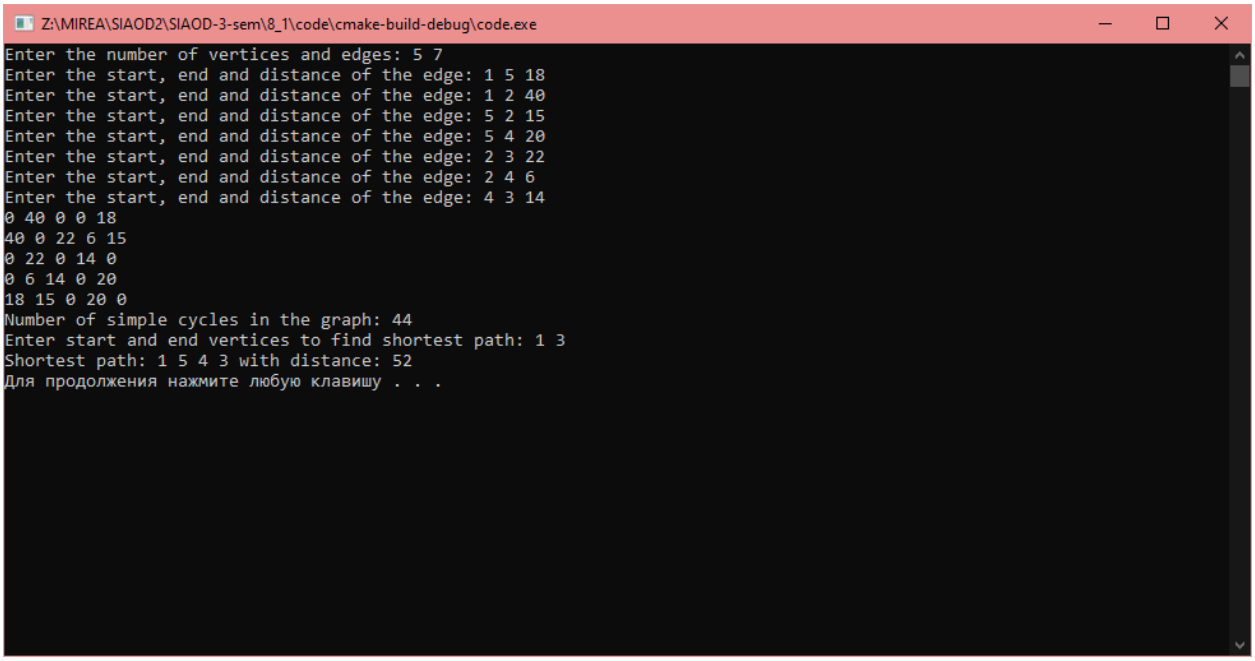

Рисунок 4 – Результат работы программы для теста №2

Также построим таблицу (табл. 3), демонстрирующую работу алгоритма Дейкстры.

Таблица 3 - Работа алгоритма Дейкстра для теста №2

1	2	3	4	5
0	∞	∞	∞	∞
	40	∞	∞	18
	33	∞	38	
		55	39	
		52		

ЗАДАНИЕ №2

2.1 Постановка задачи

Применение графа в решении задачи:

Раскрасить граф так что любые две смежные вершины были раскрашены в разные цвета, а число использованных цветов было минимально возможным.

Необходимо:

- обосновать применение графа как структуры данных;
- обосновать выбор алгоритмов для операций с графом;
- разработать и отладить программу.

2.2 Ход решения

Для решения задачи раскраски графа с минимальным числом цветов можно использовать алгоритм жадной раскраски. Этот алгоритм последовательно присваивает каждой вершине минимально возможный цвет, который не используется её соседями.

Алгоритм обрабатывает вершины графа одну за другой в заданном порядке. Каждой вершине назначается минимально возможный цвет, который еще не используется у её смежных вершин.

Для каждой вершины сначала отмечаются все цвета, занятые её соседями. Затем алгоритм ищет первый незанятый цвет и назначает его текущей вершине.

Для решения задачи раскраски графа с минимальным числом цветов можно использовать алгоритм жадной раскраски. Этот алгоритм последовательно присваивает каждой вершине минимально возможный цвет, который не используется её соседями.

2.3 Программная реализация

Для более простой и удобной реализации будем использовать список смежности, который будет представлять граф.

2.3.1 main.cpp:

```cpp
#include <iostream>
#include <vector>

void greedyColoring(const std::vector<std::vector<int>>& graph, int V) {
    std::vector<int> result(V, -1);
    result[0] = 0;

    std::vector<bool> available(V, false);

    for (int u = 1; u < V; ++u) {
        for (int i : graph[u]) {
            if (result[i] != -1) {
                available[result[i]] = true;
            }
        }

        int cr;
        for (cr = 0; cr < V; ++cr) {
            if (!available[cr]) {
                break;
            }
        }

        result[u] = cr;

        for (int i : graph[u]) {
            if (result[i] != -1) {
                available[result[i]] = false;
            }
        }
    }

    for (int u = 0; u < V; ++u) {
        std::cout << "Vertex " << u + 1 << " ---> Color " << result[u] <<
std::endl;
    }
}

int main() {
    int V = 8;
    std::vector<std::vector<int>> graph(V);

    graph[0] = {1, 2};
    graph[1] = {0, 2, 4, 7};
    graph[2] = {0, 1, 3};
    graph[3] = {2, 5};
    graph[4] = {1, 5, 6};
    graph[5] = {3, 5, 6};
    graph[6] = {4, 5, 7};
    graph[7] = {1, 6};

    greedyColoring(graph, V);

    return 0;
}
```

2.4 Тестирование

Для тестового примера возьмем следующий граф (рис. 5):

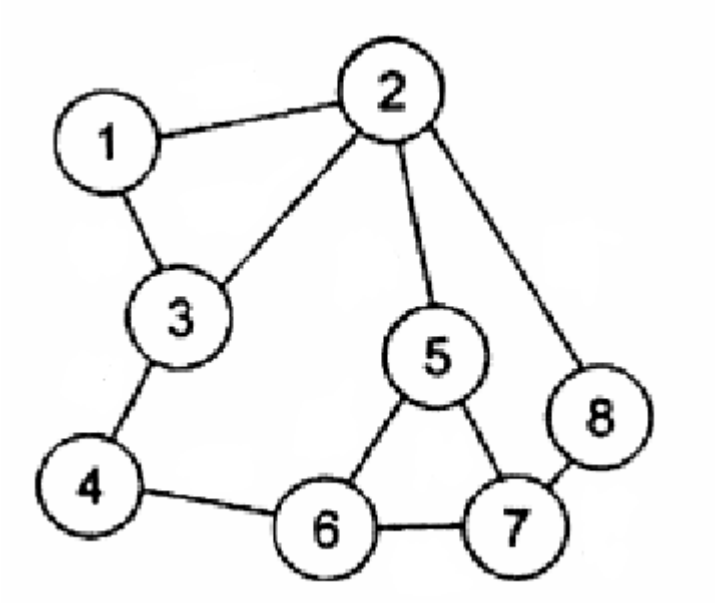

Рисунок 5 – Граф для тестов

Для начала попробуем решить данную задачу «руками» (рис. 6).

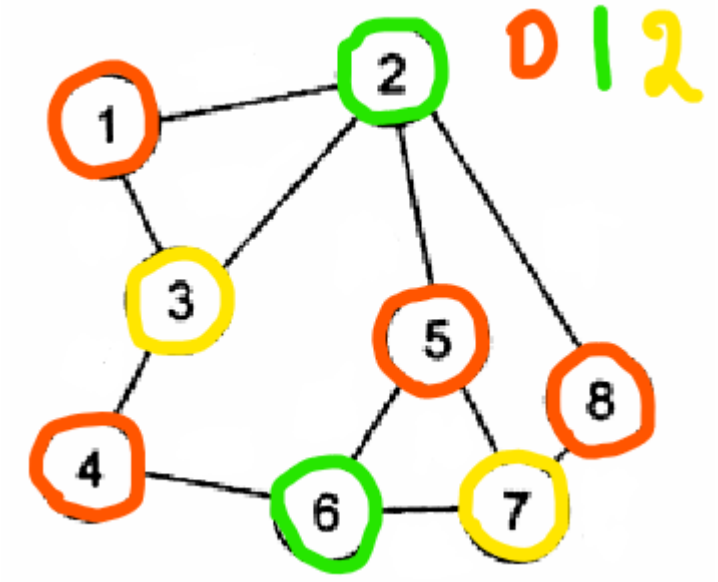

Рисунок 6 – Раскрашенный «руками» граф

Теперь протестируем работу алгоритма, протестировав на данном графе (рис. 7)

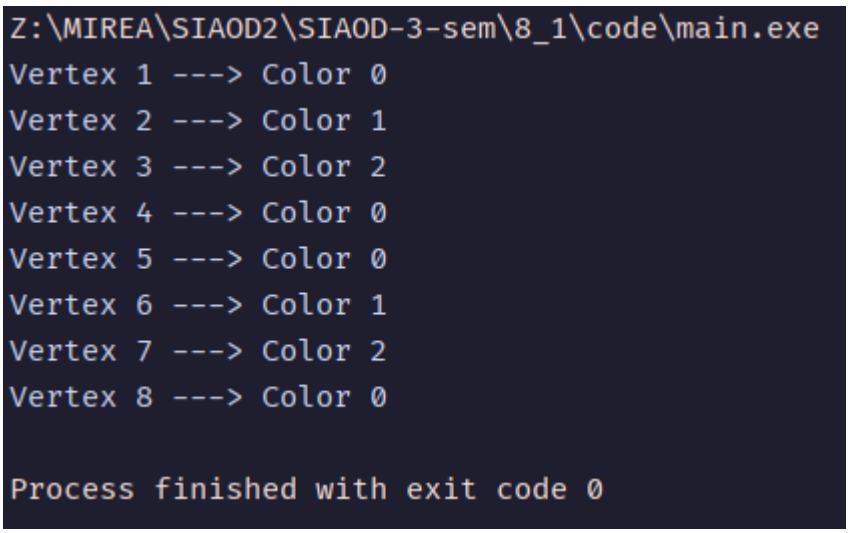

Рисунок 7 – Результат работы программы

Экспериментальные данные совпали с теоретическими.

ВЫВОД

Получили практические навыки по выполнению операций над структурой данных - граф.

СПИСОК ИНФОРМАЦИОННЫХ ИСТОЧНИКОВ

1. Рысин, М. Л. Введение в структуры и алгоритмы обработки данных : учебное пособие / М. Л. Рысин, М. В. Сартаков, М. Б. Туманова. — Москва : РТУ МИРЭА, 2022 — Часть 2 : Поиск в тексте. Нелинейные структуры данных. Кодирование информации. Алгоритмические стратегии — 2022. — 111 с. — Текст : электронный // Лань : электронно-библиотечная система. — URL: https://e.lanbook.com/book/310826 (дата обращения: 10.09.2024).
2. ГОСТ 19.701-90. Единая система программной документации. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения : межгосударственный стандарт : дата введения 1992-01- 01 / Федеральное агентство по техническому регулированию. – Изд. официальное. – Москва : Стандартинформ, 2010. – 23 с