

Министерство науки и высшего образования Российской Федерации
Пензенский государственный университет
Кафедра «вычислительная техника»

Пояснительная записка

К курсовому проектированию
по курсу «Логика и основы алгоритмизации в инженерных задачах»
на тему «Реализация алгоритма Форда – Беллмана»

18.12.24
отметка
фвд

Выполнил:
студент группы 23BBB4
Святов Иван
Принял:
к.т.н. Юрова О. В.

Пенза, 2024

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Факультет Вычислительной техники

Кафедра "Вычислительная техника"

"УТВЕРЖДАЮ"

Зав. кафедрой ВТ

«___» _____ 20__

ЗАДАНИЕ

на курсовое проектирование по курсу

«Алгоритмы и методы алгоритмизации в инженерных задачах»
Студенту Венков Н. Н. Группа 23 ВВВ 4
Тема проекта Решение алгоритма Гурца - Биллмана

Исходные данные (технические требования) на проектирование

Разработка алгоритмов и программное обеспечение в соответствии с заданием задания курсового проекта.

Необходимая документация должна содержать:

1. Постановка задачи;
2. Анализ задачи (часть задания);
3. Вывод алгоритма поставленной задачи;
4. Форма решения задачи и выводов (на определенном этапе работы алгоритма);
5. Выводы общей программы;
6. Выводы;
7. Список литературы;
8. Выводы программы;
9. Выводы работы программы.

Объем работы по курсу

1. Расчетная часть

Дан расчет работы аккумулятора.

2. Графическая часть

Сила аккумулятора в функции времени.

3. Экспериментальная часть

Измерение параметров.

Выработка работы аккумулятора на типовых данных

Срок выполнения проекта по разделам

- 1 Измерение параметров цепи курсового
- 2 Выработка алгоритмов программы
- 3 Разработка программы
- 4 Измерение и завершение разработки программы
- 5 Выработка окончательного текста
- 6
- 7
- 8

Дата выдачи задания "18" сентября 2024 г.

Дата защиты проекта " " "

Руководитель Ирбана О.В. ФГО

Задание получил "18" сентября 2024 г.

Студент Винов И.И.

Содержание

Реферат.....	5
Введение.....	6
Постановка задачи.....	7
Теоретическая часть задания.....	8
Описание алгоритма.....	10
Описание программы.....	19
Тестирование.....	34
Ручной расчет программы.....	40
Заключение.....	45
Список литературы.....	46
Приложение А. Листинг программы.....	47

Реферат

Отчет 48 страниц, 28 рисунков, 1 таблица, 1 приложение.

АЛГОРИТМ ФОРДА – БЕЛЛМАНА, ТЕОРИЯ ГРАФОВ, ГРАФ, КРАТЧАЙШИЕ ПУТИ, ВЕС РЕБЕР.

Цель исследования – разработка программы, предназначенная для нахождения кратчайших путей как в ориентированном, так и неориентированном графе между начальной вершиной и всеми остальными, как с отрицательным весом рёбер, так и с положительным, используя алгоритм Форда – Беллмана.

В работе рассматривается алгоритм Форда – Беллмана, на основе которого находятся кратчайшие пути между вершиной, которая является источником, и всеми другими вершинами. Установлено, что с помощью используемого в проекте алгоритма можно найти расстояния между вершинами графа, которые имеют положительный вес рёбер, а также отрицательный.

Введение

Алгоритм Форда – Беллмана – это алгоритм поиска кратчайшего пути во взвешенном графе. За время $O(|V| * |E|)$ алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. Предложен независимо Ричардом Беллманом и Лестером Фордом.

Отличие алгоритма Форда – Беллмана от алгоритма Дейкстры заключается в том, что алгоритм допускает рёбра с отрицательным весом. Тем не менее, в графе не должно быть циклов отрицательного веса, достижимых из начальной вершины, иначе вопрос о кратчайших путях является бессмысленным.

Однако алгоритм Дейкстры использует очередь с приоритетом, чтобы жадно выбрать ближайшую вершину, которая ещё не была обработана, и выполняет этот процесс релаксации для всех исходящих из неё рёбер. В отличие от этого, алгоритм Форда – Беллмана просто ослабляет все рёбра делает это $|V| - 1$ раз, где $|V|$ – количество вершин в графе. Также неэффективен на больших разреженных графах.

В качестве среды разработки была выбрана среда Microsoft Visual Studio 2019, а также язык программирования – Си.

Целью данной курсовой работы является разработка программы на языке Си, который является широко используемым. Именно с его помощью в данном курсовом проекте реализуется алгоритм Форда – Беллмана, осуществляющий поиск кратчайших путей в графе.

1 Постановка задачи

Требуется разработать программу, которая будет находить кратчайшие пути до каждой вершины сгенерированного, вводимого или считываемого графа, используя алгоритм Форда – Беллмана.

В программе должен выводиться исходный граф, причем при генерации данных должны быть предусмотрены границы, чтобы пользователь не мог задать граф с 0 вершин или отрицательным значением. Программа должна работать так, чтобы пользователь вводил количество вершин для генерации графа. После обработки вводимых данных на экран должен выводиться сам граф, в виде начальная вершина – конечная вершина – вес, а также выполненный алгоритм, то есть вершина и кратчайшее расстояние. Необходимо предусмотреть различные исходы поиска, чтобы программа не выдавала ошибок и работала правильно. Устройство ввода – клавиатура и мышь.

Задания выполняются в соответствии с вариантом № 14.

2 Теоретическая часть задания

На рисунке 1 изображен граф G , который представляет собой ориентированный взвешенный граф с шестью вершинами (A, B, C, D, s, t) и множествами ребер, соединяющими эти вершины. Каждое ребро имеет вес, как положительный, так и отрицательный, указанный рядом с ним. Рёбра, представленные на графе ориентированы, что показывается стрелкой, которая указывает достижимость данной вершины.

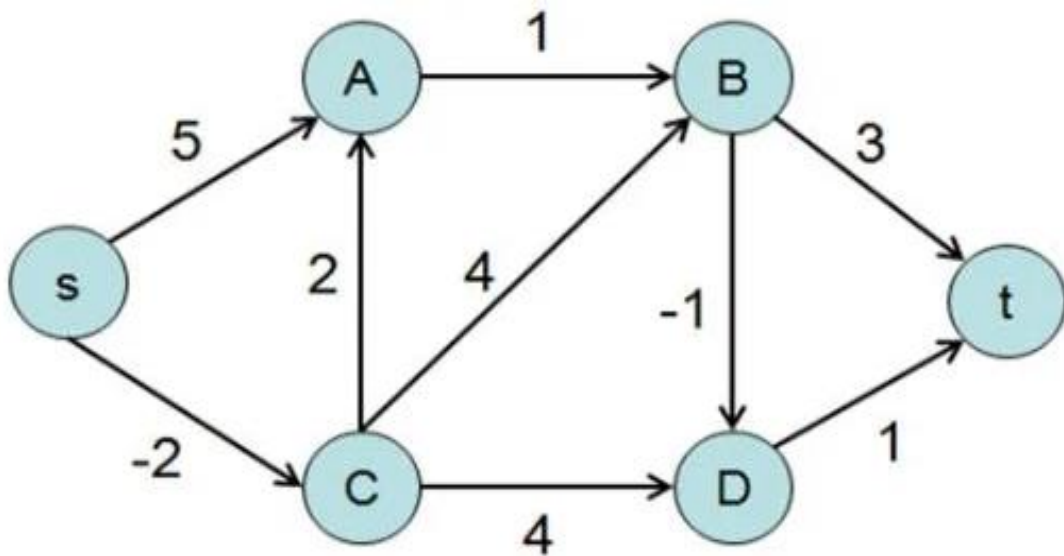


Рисунок 1 – Пример графа

При представлении графа информация о ребрах между вершинами хранится в векторном виде в формате, начальная вершина – конечная вершина – вес ребра.

В настоящее время существует достаточно много алгоритмов на графах, в основе которых лежит перебор вершин данного графа. Причем каждая вершина может пересматриваться как один, так и более раз, а переход между вершинами осуществляется по ребрам. Это могут быть алгоритмы поиска в глубину, поиска в ширину или же другие известные алгоритмы. В данном проекте остановимся на одном из таких методов – алгоритм Форда – Беллмана.

Данный алгоритм является методом для нахождения кратчайших путей от одной вершины графа к другим, в отличие от алгоритма Дейкстры может работать с графами, которые содержат рёбра с отрицательным весом. Данный метод подходит как для ориентированных, так и неориентированных графов разного размера.

Алгоритм Форда – Беллмана работает следующим образом:

1. Сначала необходимо задать начальную вершину из которой будет начинаться поиск кратчайших путей и определить расстояние до самой себя равным нулю, а расстояние до всех остальных вершин равным бесконечности.
2. Далее идет обновление информации, то есть для каждого ребра в графе алгоритм будет обновлять расстояния до вершин, если найден более короткий путь через данное ребро. Этот процесс повторяется $V - 1$ раз, где V – количество вершин в графе. Данное действие необходимо, поскольку наибольшее количество рёбер в любом кратчайшем пути не может превышать $V - 1$.
3. Далее идет проверка на отрицательные циклы. После $V - 1$ итерации алгоритм выполняет еще один проход по всем рёбрам. Если на этом этапе какое – либо расстояние может быть улучшено, значит, в графе существует отрицательный цикл, что делает невозможным нахождение кратчайшего пути.

После выполнения алгоритма будут выведены кратчайшие расстояния от начальной вершины до всех остальных вершин, если данные вершины достижимы, в противном же случае будет выведена информация о наличии отрицательных циклов. Алгоритм Форда – Беллмана является эффективным инструментом в задачах на графах, где могут встречаться рёбра с отрицательными весами.

3 Описание алгоритма программы

Для программной реализации алгоритма понадобится: структура `Edge` – для хранения информации о ребре, в методе `main` переменные `maxEdgesCount` – для хранения максимального количества ребер, `isDirected` – для хранения информации о том, ориентированный граф или нет, `vertices (int)` – для хранения количества вершин и `source` – для хранения начальной вершины, в методе `bellmanFord` массив `distance` – для хранения расстояний от исходной вершины до всех остальных. С самого начала расстояние до каждой вершины графа из начальной определено как бесконечность, а расстояние до самой начальной вершина, как 0.

Если пользователь выбирает случайную генерацию графа, тогда сначала определяется максимальное количество рёбер, которые записываются в переменную `maxEdgesCount` и выделяется память для графа. Количество данных рёбер зависит от того, какой тип графа был выбран ранее. Если был выбран неориентированный граф, то рёбер будет в два раза больше, так как необходимы также обратные. После чего создаются начальные рёбра между вершинами, если выбран неориентированный тип то создаются также обратные. После чего добавляются случайные рёбра между разными вершинами. После вызывается функция `bellmanFord`, в котором создается массив `distance`. Далее каждое ребро графа расслабляется `vertices – 1` раз, если текущая вершина не бесконечна и ребро можно расслабить, тогда расстояние до целевой вершины обновляется. Затем происходит проверка на наличие отрицательных циклов и выводится кратчайшее расстояние от источника до всех вершин. После чего пользователю будет предложено сохранить полученный результат в файл. Если в консоли будет введено `yes`, то необходимо ввести название файла, в который будет сохранен результат, иначе данный этап будет пропущен.

Если пользователь выбирает ручной ввод графа, тогда сначала определяется максимальное количество рёбер, зависящие от количества вводимых вершин и типа графа, которые записываются в переменную

maxEdgesCount и выделяется память для графа. Затем в цикле пользователь вводит ребра графа в формате, источник – вершина – вес, после чего идет запись в граф. Если выбран неориентированный граф, то также создаются обратные рёбра к вершинам. Данный цикл продолжается до тех пор, пока в консоли не будет введено «-1 -1 -1», что означает конец ввода. Затем вызывается алгоритм, работа которого описана выше, и граф выводится на экран.

Если пользователь выбирает считывание графа из файла, тогда необходимо ввести название заранее созданного файла, с записанным графом внутри. После чего выделяется память для графа, и программа считывает значения из файла в соответствующие поля структуры, если же граф не ориентированный то добавляются обратные рёбра. Затем вызывается функция расчета алгоритма, описанная выше. После чего граф и результат выводятся на экран.

Ниже представлен псевдокод методов `int main()`, `bellmanFord()`, `readGraphFromFile()`.

int main()

1. Установить Русский язык
2. Инициализировать переменные `vertices`, `op`, `filename`, `filenameForRead`, `isDirected`
3. Ввести указатели на файл `filenameForRead2`
4. Цикл: пока истина
5. Вывести сообщение "Выберите тип графа:"
6. Вывести сообщение " 1 - Ориентированный граф"
7. Вывести сообщение " 2 - Неориентированный граф"
8. Вывести сообщение "Введите номер операции:"
9. Считать `op`
10. Если `op == 1`
11. `isDirected = true;`
12. Конец Если Начало иначе если `op == '2'`
13. `isDirected = false;`
14. Конец иначе если Начало иначе

15. Вывести сообщение " Некорректный ввод, попробуйте снова."
16. continue;
17. Конец Иначе
18. Вывести сообщение "Выберите способ ввода графа:"
19. Вывести сообщение "1 - Автоматический ввод графа"
20. Вывести сообщение "2 - Ручной ввод графа"
21. Вывести сообщение "3 - Чтение графа из файла"
22. Вывести сообщение "0 - Завершение программы"
23. Вывести сообщение "Введите номер операции: "
24. Читать op
25. В зависимости от op
26. случай '1':
27. Инициализировать генерацию случайных чисел
28. Вывести сообщение "Введите количество вершин в графе: "
29. Читать количество вершин в переменную vertices
30. Если vertices <= 0
31. Вывести сообщение "Введено некорректное количество вершин!
Попробуйте снова"
32. Прервать цикл
33. Конец Если
34. int maxEdgesCount = isDirected ? vertices * (vertices - 1) : vertices * (vertices - 1) / 2;
35. Edge* graph = (Edge*)malloc(maxEdgesCount * sizeof(Edge));
36. Если graph == NULL
37. Вывести сообщение "Ошибка при выделении памяти."
38. Вернуть 1
39. КонецЕсли
40. edgesCount = 0
41. Цикл: for i = 0 пока i < vertices - 1 выполнить ++i
42. graph[edgesCount].source = i

43. graph[edgesCount].destination = i + 1
44. graph[edgesCount].weight = случайное число от -5 до 15
45. edgesCount++
46. Если (!isDirected)
47. graph[edgesCount].source = i + 1;
48. graph[edgesCount].destination = i;
49. graph[edgesCount].weight = graph[edgesCount - 1].weight;
50. edgesCount++;
51. Конец Если
52. Конец Цикла
53. Цикл: for i = vertices пока i < maxEdgesCount выполнить ++i
54. source = rand() % vertices
55. destination = rand() % vertices;
56. вызвать ensurePathToVertex(graph, &edgesCount, source, destination, isDirected)
57. Конец Цикла
58. вызвать printGraph(graph, edgesCount)
59. вызвать bellmanFord(graph, vertices, edgesCount, 0)
60. Освободить память graph
61. Прервать цикл
62. Конец Случая
63. случай '2':
64. Вывести сообщение "Введите количество вершин в графе: "
65. Считать vertices
66. Если vertices <= 0
67. Вывести сообщение "Введено некорректное количество вершин!
Попробуйте снова"
68. Прервать случай
69. Конец Если


```
70.  int maxEdgesCount = isDirected ? vertices * (vertices - 1) : vertices * (vertices
- 1) / 2;
71.  Edge* graph = (Edge*)malloc(maxEdgesCount * sizeof(Edge));
72.  Если graph == NULL
73.  Вывести сообщение "Ошибка при выделении памяти."
74.  Вернуть 1
75.  Конец Если
76.  edgesCount = 0
77.  int source, destination, weight
78.  Вывести сообщение "Введите рёбра в формате \"источник конечная
вершина вес\" (например, 0 1 5).\"
79.  Вывести сообщение "Введите \"-1 -1 -1\", чтобы завершить ввод.\"
80.  Цикл: пока истина
81.  Вывести сообщение "Введите ребро: "
82.  Считать source, destination, weight
83.  Если source == -1 и destination == -1 и weight == -1
84.  Прервать цикл
85.  Конец Если
86.  Если source < 0 или source >= vertices или destination < 0 или destination >=
vertices
87.  Вывести сообщение "Некорректные вершины. Попробуйте снова."
88.  continue;
89.  Конец Если
90.  graph[edgesCount].source = source
91.  graph[edgesCount].destination = destination
92.  graph[edgesCount].weight = weight
93.  edgesCount++
94.  Если (!isDirected)
95.  graph[edgesCount].source = destination;
96.  graph[edgesCount].destination = source;
```

97. `graph[edgesCount].weight = weight;`
98. `edgesCount++;`
99. Конец Если
100. Конец Цикла
101. Вызвать `printGraph(graph, edgesCount)`
102. Вызвать `bellmanFord(graph, vertices, edgesCount, 0)`
103. Освободить память `graph`
104. Прервать случай
105. Конец случая 2
106. случай '3':
107. Вывести сообщение "Введите имя файла, для чтения из него графа: "
108. Считать имя файла
109. `filenameForRead2 = fopen(filenameForRead, "r");`
110. Если `filenameForRead2 == NULL`
111. Вывести сообщение "Не удалось открыть файл, так как файл пустой или имеет не соответствующий вид! Повторите ввод"
112. Прервать случай
113. Иначе
114. `Edge graph[MAX_VERTICES]`
115. `edgesCount = readGraphFromFile(filenameForRead, graph, &vertices, isDirected)`
116. Если `edgesCount < 0`
117. Вернуть 1
118. Конец Если
119. Вызвать `printGraph(graph, edgesCount)`
120. Вызвать `bellmanFord(graph, vertices, edgesCount, 0)`
121. Прервать случай
122. Конец Иначе
123. Закрыть файл `filenameForRead2`
124. случай '0':

125. Вернуть 0
126. Прервать цикл
127. случай 'default':
128. Вывести сообщение "Некорректный ввод, попробуйте снова."
129. Прервать цикл
130. Конец случая
131. Конец Пока
132. Вернуть 0
133. Конец метода main

void bellmanFord(Edge edges[], int vertices, int edgesCount, int source)

1. Создать массив расстояний длиной в зависимости от количества вершин
2. Цикл: для $i = 0$ пока $i < \text{vertices}$ выполнить $++i$
3. расстояние[i] = Бесконечность
4. Конец цикла
5. расстояние[source] = 0
6. Цикл: для $i = 0$ пока $i < \text{vertices} - 1$ выполнить $++i$
7. Цикл: для $j = 0$ пока $j < \text{edgesCount}$ выполнить $++j$
8. Если distance[edges[j].source] не равно Бесконечность и
distance[edges[j].destination] > distance[edges[j].source] + edges[j].weight
9. Тогда distance[edges[j].destination] = distance[edges[j].source] +
edges[j].weight
10. Конец Если
11. Конец Цикла
12. Конец Цикла
13. Цикл: для $i = 0$ пока $i < \text{edgesCount}$ выполнить $++i$
14. Если distance[edges[j].source] не равно Бесконечность и
distance[edges[j].destination] > distance[edges[j].source] + edges[j].weight
15. Вывести сообщение "Присутствует отрицательный цикл"
16. Освободить память массива расстояний
17. Выход из функции

18. Конец Если
19. Конец Цикла
20. Вывести сообщение "Вершина Расстояние от источника"
21. Цикл: для $i = 0$ пока $i < \text{vertices}$ выполнить $++i$
22. Вывести i , $\text{distance}[i] == \text{INF}$? -1 иначе $\text{distance}[i]$
23. Конец Цикла
24. Проинициализировать переменную filename
25. Вывести сообщение " Хотите сохранить результат в файл? (yes/no):"
26. Проинициализировать переменную response
27. Считать переменную response
28. Если $\text{strcmp}(\text{response}, \text{"yes"}) == 0$
29. Вывести сообщение "Введите название файла: "
30. Считать переменную filename
31. $\text{file} = \text{открыть файл}(\text{filename}, \text{"w"})$
32. Если $\text{file} == \text{NULL}$
33. Вывести сообщение " Ошибка при открытии файла.\n "
34. Конец Если
35. Иначе
36. Записать в file "Результаты алгоритма Форда-Беллмана:\n"
37. Записать в file " Вершина Расстояние от источника\n"
38. Цикл: для $i = 0$ пока $i < \text{vertices}$ выполнить $++i$
39. Записать в файл "\%d \t\t \%d\n" , i , $(\text{distance}[i] == \text{INF}) ? -1 : \text{distance}[i]$
40. Конец Цикла
41. Вызвать функцию printGraphInFile
42. Закрывать файл
43. Вывести сообщение "Результат успешно сохранен в файл \%s\n ", filename
44. Конец Иначе
45. Конец Если
46. Освободить память массива расстояний

int readGraphFromFile(const char* filename, Edge edges[], int* verticesCount, bool isDirected)

1. Открыть файл с именем filename в режиме чтения
2. Если файл не удалось открыть
3. Вывести сообщение об ошибке открытия файла
4. Вернуть -1
5. КонецЕсли
6. edgesCount = 0
7. Считать количество_вершин из файла и сохранить в указателе_на_количество_вершин
8. Пока Считываем три целых числа из файла (source, destination, weight)
9. edges[edgesCount].source = source, edges[edgesCount].destination = destination, edges[edgesCount].weight = weight
10. Увеличить edgesCount на 1
11. Если (!isDirected)
12. edges[edgesCount].source = edges[edgesCount - 1].destination;
13. edges[edgesCount].destination = edges[edgesCount - 1].source;
edges[edgesCount].weight = edges[edgesCount - 1].weight;
14. edgesCount++;
15. Конец Если
16. Конец Пока
17. Закрывать файл
18. Вернуть edgesCount

4 Описание программы

Для написания данной программы был выбран язык программирования Си. Так как этот язык используется в операционных системах и различном прикладном программном обеспечении для множества устройств, а также это один из наиболее универсальных языков программирования, так как сочетает в себе возможности языков программирования высокого и низкого уровней.

Проект был создан в виде консольного приложения Win32 (Visual C++).

Созданная программа является многомодульной, так как состоит из таких функций, как: `main`, `readGraphFromFile`, `bellmanFord`, `compareEdges`, `printGraph`, `printGraphInFile`, `edgeExists`, `ensurePathToVertex`.

Функция `readGraphFromFile` считывает граф представленный в файле в структуру, которая содержит начальную вершину, конечную, а также вес ребра. В методе открывается файл введенный пользователем, после чего считывается сначала количество вершин, а после чего уже все вершины с ребрами. Если граф неориентированный, то рёбра будут продублированы. Пример представления графа в файле для считывания представлен на рисунке 2. Код функции `readGraphFromFile` представлен ниже.

```
int readGraphFromFile(const char* filename, Edge edges[], int* verticesCount, bool isDirected) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        printf("Ошибка при открытии файла %s!\n", filename);
        return -1;
    }
    int edgesCount = 0;
    fscanf(file, "%d", verticesCount); // Чтение количества вершин
    while (fscanf(file, "%d %d %d", &edges[edgesCount].source,
        &edges[edgesCount].destination,
        &edges[edgesCount].weight) == 3) {
        edgesCount++;
        if (!isDirected) {
            edges[edgesCount].source = edges[edgesCount - 1].destination;
            edges[edgesCount].destination = edges[edgesCount - 1].source;
            edges[edgesCount].weight = edges[edgesCount - 1].weight;
            edgesCount++;
        }
    }
}
```

```

    }
}
fclose(file);
return edgesCount;
}

```

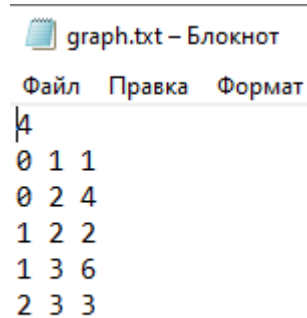


Рисунок 2 – Пример графа в файле для считывания

Функция `compareEdges` предназначена для сравнения двух рёбер графа. Метод используется в качестве функции сравнения для сортировки массива рёбер. Функция принимает два указателя на элементы `Edge`, которые будут сравниваться. Передаваемые указатели преобразуются в указатели на структуру, чтобы получить доступ к полю `source` каждого из рёбер. Возвращаемым значением будет разница между значениями полей `source` двух рёбер. Ниже представлен код функции `compareEdges`.

```

int compareEdges(const void* a, const void* b) {
    return ((Edge*)a)->source - ((Edge*)b)->source;
}

```

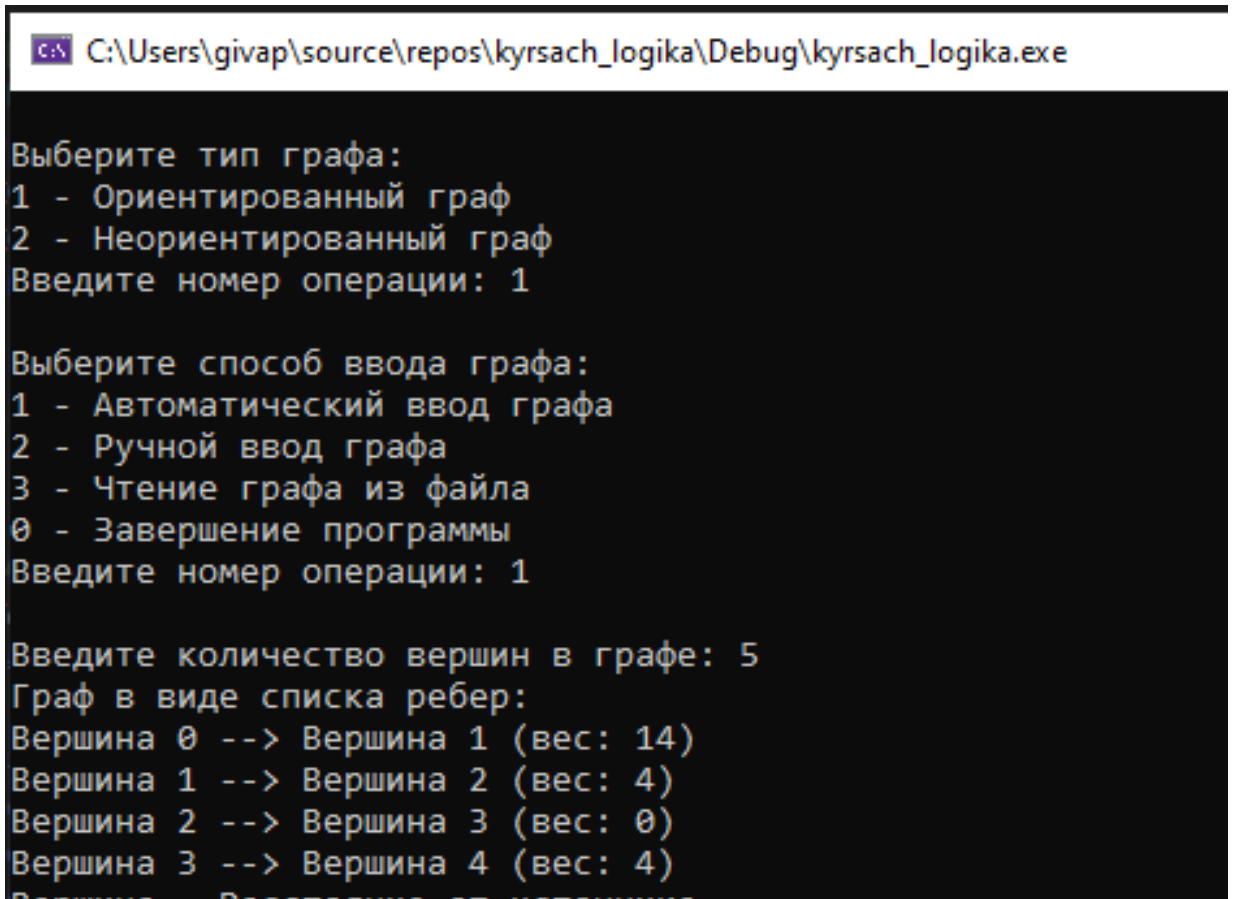
Функция `printGraph` выводит граф в виде списка рёбер. Сначала выводится заголовок, затем с помощью функции `qsort` происходит сортировка массива `edges` на основе значений `source` рёбер. После чего в консоль выводится каждое ребро. Пример реализации данного метода представлен на рисунке 3. Код функции `printGraph` приведен ниже.

```

void printGraph(Edge edges[], int edgesCount) {
    printf("Граф в виде списка ребер:\n");
    qsort(edges, edgesCount, sizeof(Edge), compareEdges);
    for (int i = 0; i < edgesCount; i++) {
        printf("Вершина   %d   -->   Вершина   %d   (вес:   %d)\n", edges[i].source,
edges[i].destination, edges[i].weight);
    }
}

```

}



```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe

Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1

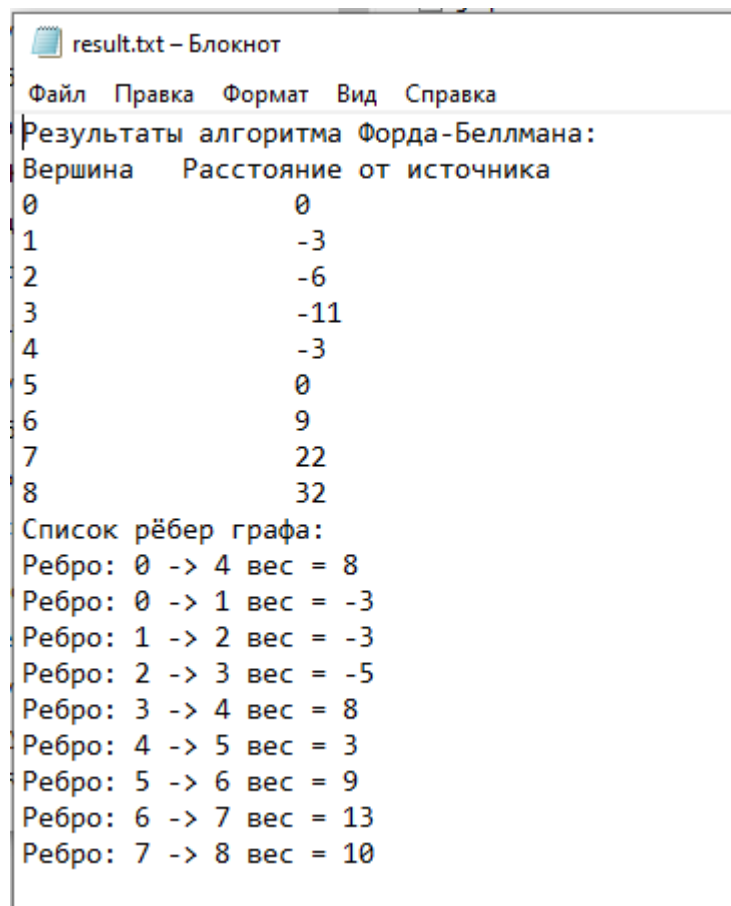
Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 1

Введите количество вершин в графе: 5
Граф в виде списка ребер:
Вершина 0 --> Вершина 1 (вес: 14)
Вершина 1 --> Вершина 2 (вес: 4)
Вершина 2 --> Вершина 3 (вес: 0)
Вершина 3 --> Вершина 4 (вес: 4)
Вершина 4 --> Вершина 0 (вес: 4)
```

Рисунок 3 – Вывод графа на экран

Функция `printGraphInFile` работает аналогично функции `printGraph`. Отличием является то, что данный метод записывает результат в файл, а не консоль. Пример реализации данной функции представлен на рисунке 4. Код метода `printGraphInFile` приведен ниже.

```
void printGraphInFile(Edge edges[], int edgesCount, FILE* file) {
    fprintf(file, "Список рёбер графа:\n");
    qsort(edges, edgesCount, sizeof(Edge), compareEdges);
    for (int i = 0; i < edgesCount; i++) {
        fprintf(file, "Ребро: %d -> %d вес = %d\n", edges[i].source, edges[i].destination,
edges[i].weight);
    }
}
```



```
result.txt – Блокнот
Файл  Правка  Формат  Вид  Справка
Результаты алгоритма Форда-Беллмана:
Вершина    Расстояние от источника
0           0
1          -3
2          -6
3         -11
4          -3
5           0
6           9
7          22
8          32
Список рёбер графа:
Ребро: 0 -> 4 вес = 8
Ребро: 0 -> 1 вес = -3
Ребро: 1 -> 2 вес = -3
Ребро: 2 -> 3 вес = -5
Ребро: 3 -> 4 вес = 8
Ребро: 4 -> 5 вес = 3
Ребро: 5 -> 6 вес = 9
Ребро: 6 -> 7 вес = 13
Ребро: 7 -> 8 вес = 10
```

Рисунок 4 – Сохранение результата в файле

Функция `bellmanFord` предназначена для выполнения соответствующего алгоритма. Первоначально выделяется память для хранения массива расстояний, после чего расстояние до каждой вершины инициализируется как бесконечность, а до самой себя равное 0. Затем каждое ребро расслабляется `vertices - 1` раз, и если текущая вершина не бесконечна и это ребро можно расслабить, то обновляется расстояние до целевой вершины. Затем идет проверка на наличие отрицательных циклов, после чего выводится результат. После вывода пользователь имеет возможность сохранить данный результат в файл, для этого необходимо согласиться с вопросом. Если пользователь вводит `no`, результат сохранен не будет, иначе необходимо ввести название файла, в который сохранится результат. Пример реализации данного метода приведен на рисунке 5.

```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe

Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 1

Введите количество вершин в графе: 5
Граф в виде списка ребер:
Вершина 0 --> Вершина 1 (вес: -2)
Вершина 1 --> Вершина 4 (вес: 13)
Вершина 1 --> Вершина 2 (вес: -1)
Вершина 2 --> Вершина 4 (вес: 3)
Вершина 2 --> Вершина 3 (вес: 3)
Вершина 3 --> Вершина 4 (вес: 7)
Вершина    Расстояние от источника
0           0
1           -2
2           -3
3           0
4           0
Хотите сохранить результат в файл? (yes/no): yes
Введите название файла: result2.txt
Результат успешно сохранен в файл result2.txt
```

Рисунок 5 – Выполнение алгоритма Форда – Беллмана

Ниже представлен код функции `bellmanFord`.

```
void bellmanFord(Edge edges[], int vertices, int edgesCount, int source) {
    int* distance = (int*)malloc(vertices * sizeof(int)); // массив для хранения расстояний
    от исходной вершины до всех остальных

    // инициализация расстояния от источника до всех вершин как бесконечность
    for (int i = 0; i < vertices; ++i)
        distance[i] = INF;

    distance[source] = 0; // дистанция из источника до самого себя = 0
    // расслабление ребер vertices - 1 раз
    for (int i = 0; i < vertices - 1; ++i) {
        for (int j = 0; j < edgesCount; ++j) {
            // если текущая вершина не бесконечна и это ребро можно расслабить
            if (distance[edges[j].source] != INF &&
                distance[edges[j].destination] > distance[edges[j].source] +
edges[j].weight) {
                // обновление расстояния до целевой вершины
            }
        }
    }
}
```



```

        distance[edges[j].destination] = distance[edges[j].source] +
edges[j].weight;
    }
}

// проверка на наличие циклов с отрицательным весом
for (int i = 0; i < edgesCount; ++i) {
    // если можно улучшить расстояние - значит есть отрицательный цикл
    if (distance[edges[i].source] != INF &&
        distance[edges[i].destination] > distance[edges[i].source] + edges[i].weight)
    {
        printf("Присутствует отрицательный цикл\n");
        free(distance);
        return; // выход из функции, если цикл есть
    }
}

// выводятся кратчайшие расстояния от источника до всех вершин
printf("Вершина    Расстояние от источника\n");
for (int i = 0; i < vertices; ++i)
    // вывод номера вершины и расстояние до неё (-1 если она не достижима)
    printf("%d \t\t %d\n", i, (distance[i] == INF) ? -1 : distance[i]);

// Сохранение результатов в файл
char filename[100];
printf("Хотите сохранить результат в файл? (yes/no): ");
char response[4];
scanf("%s", response);

if (strcmp(response, "yes") == 0) {
    printf("Введите название файла: ");
    scanf("%s", filename);
    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        printf("Ошибка при открытии файла.\n");
    }
    else {
        fprintf(file, "Результаты алгоритма Форда-Беллмана:\n");
        fprintf(file, "Вершина    Расстояние от источника\n");
        for (int i = 0; i < vertices; ++i) {
            fprintf(file, "%d \t\t %d\n", i, (distance[i] == INF) ? -1 : distance[i]);
        }
        printGraphInFile(edges, edgesCount, file);
    }
}

```

```

        fclose(file);
        printf("Результат успешно сохранен в файл %s\n", filename);
    }
}
free(distance);
}

```

Функция `edgeExists` используется для проверки существования ребра между двумя вершинами в графе, что позволяет избежать создания дублирующих рёбер. Ниже приведен код метода `edgeExists`.

```

bool edgeExists(Edge* edges, int edgesCount, int source, int dest) {
    for (int i = 0; i < edgesCount; i++) {
        if ((edges[i].source == source && edges[i].destination == dest) ||
            (edges[i].source == dest && edges[i].destination == source)) {
            return true; // если путь уже существует
        }
    }
    return false;
}

```

Функция `ensurePathToVertex` создает путь между двумя вершинами. Первоначально идет проверка на валидность, то есть если `source` больше или равен `destination`, функция завершает выполнение, что позволяет предотвратить дублирование ребер. После чего если ребра нет, функция будет создавать ребро. Если граф неориентированный добавляется также обратное ребро между вершинами. Код метода `ensurePathToVertex` представлен ниже.

```

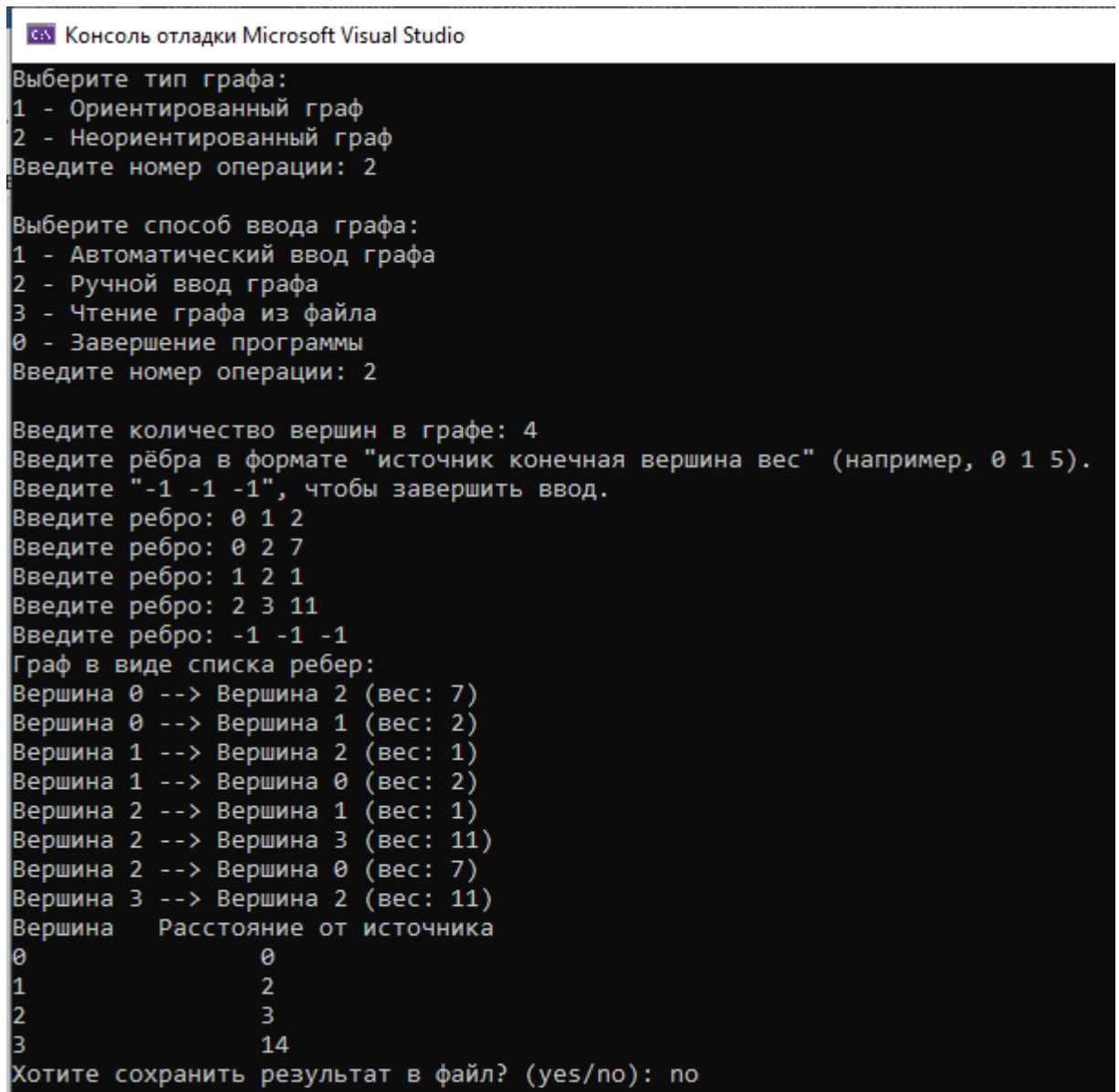
void ensurePathToVertex(Edge* edges, int* edgesCount, int source, int destination, bool
isDirected) {
    if (source >= destination) return; // проверка на валидность

    if (!edgeExists(edges, *edgesCount, source, destination)) {
        edges[*edgesCount].source = source;
        edges[*edgesCount].destination = destination;
        edges[*edgesCount].weight = (rand() % 21) - 2; // вес рёбер
        (*edgesCount)++;
        if (!isDirected) {
            edges[*edgesCount].source = destination;
            edges[*edgesCount].destination = source;
            edges[*edgesCount].weight = edges[*edgesCount - 1].weight;
            (*edgesCount)++;
        }
    }
}

```

}

Всё взаимодействие программы с пользователем содержится в функции main. Работа программы начинается с выбора типа графа пользователем. Далее пользователь может выбрать как ввести граф. Всего есть 3 варианта, это автоматический ввод, ручной ввод и считывание уже созданного графа из файла. Результат выбора типа графа и вывод дальнейшего меню представлен на рисунке 6.



```
Консоль отладки Microsoft Visual Studio
Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 2

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 2

Введите количество вершин в графе: 4
Введите рёбра в формате "источник конечная вершина вес" (например, 0 1 5).
Введите "-1 -1 -1", чтобы завершить ввод.
Введите ребро: 0 1 2
Введите ребро: 0 2 7
Введите ребро: 1 2 1
Введите ребро: 2 3 11
Введите ребро: -1 -1 -1
Граф в виде списка ребер:
Вершина 0 --> Вершина 2 (вес: 7)
Вершина 0 --> Вершина 1 (вес: 2)
Вершина 1 --> Вершина 2 (вес: 1)
Вершина 1 --> Вершина 0 (вес: 2)
Вершина 2 --> Вершина 1 (вес: 1)
Вершина 2 --> Вершина 3 (вес: 11)
Вершина 2 --> Вершина 0 (вес: 7)
Вершина 3 --> Вершина 2 (вес: 11)
Вершина    Расстояние от источника
0           0
1           2
2           3
3          14
Хотите сохранить результат в файл? (yes/no): no
```

Рисунок 6 – Выбор неориентированного графа

Если пользователь выбрал автоматический ввод графа, то необходимо ввести только количество вершин в графе. Далее программа рассчитает

количество рёбер в зависимости от типа графа, после чего расставит первоначальные рёбра, также если выбран неориентированный граф, то рёбра про дублируются. После уже будут добавлены дополнительные. Затем выводится сам граф и выполняется алгоритм Форда – Беллмана. Реализация случайной генерации графа представлена на рисунке 7.

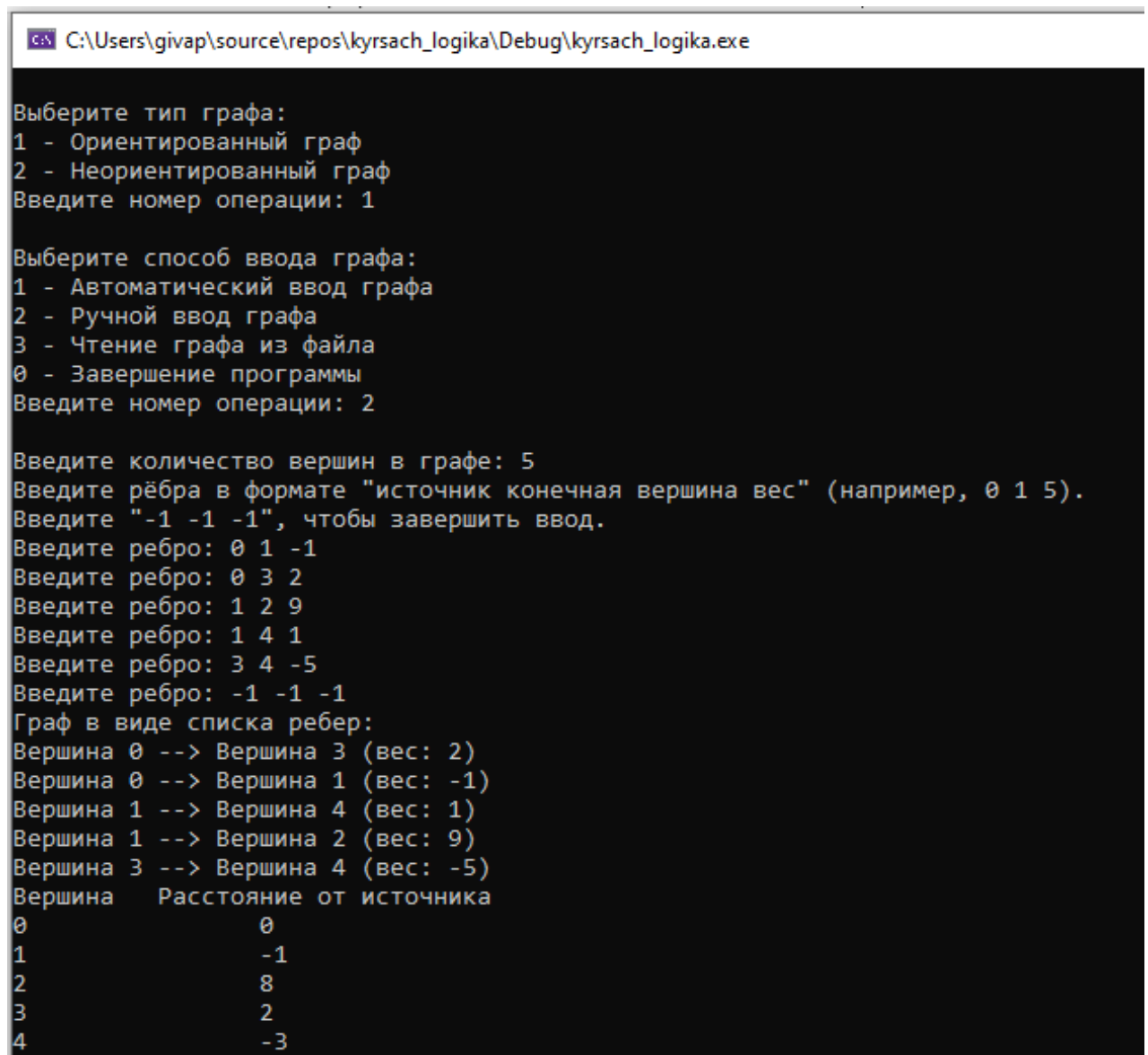
```

C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe
Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1
Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 1
Введите количество вершин в графе: 7
Граф в виде списка ребер:
Вершина 0 --> Вершина 1 (вес: 3)
Вершина 0 --> Вершина 6 (вес: 16)
Вершина 0 --> Вершина 5 (вес: 1)
Вершина 0 --> Вершина 4 (вес: 11)
Вершина 0 --> Вершина 2 (вес: 18)
Вершина 1 --> Вершина 3 (вес: 1)
Вершина 1 --> Вершина 2 (вес: -5)
Вершина 1 --> Вершина 4 (вес: 8)
Вершина 1 --> Вершина 6 (вес: 4)
Вершина 2 --> Вершина 3 (вес: 2)
Вершина 2 --> Вершина 5 (вес: 14)
Вершина 3 --> Вершина 5 (вес: 12)
Вершина 3 --> Вершина 4 (вес: 11)
Вершина 4 --> Вершина 6 (вес: -1)
Вершина 4 --> Вершина 5 (вес: 5)
Вершина 5 --> Вершина 6 (вес: 5)
Вершина    Расстояние от источника
0           0
1           3
2          -2
3           0
4          11
5           1
6           6
Хотите сохранить результат в файл? (yes/no): no
  
```

Рисунок 7 – Выбор случайной генерации графа

Если же пользователь выбрал ручной ввод графа, то необходимо первоначально ввести количество вершин, после чего программа определит максимальное количество рёбер графа в зависимости от типа графа. Далее пользователю необходимо вводить рёбра в формате «Источник – Конечная вершина – Вес», а для завершения ввода «-1 -1 -1». После чего программа

проверяет рёбра, чтобы начальная и конечная вершины не были меньше 0 или больше максимальной вершины, затем данные вершина заносятся в граф, а если граф неориентированный, то вершины дублируются. После чего уже выполняется алгоритм Форда – Беллмана и выводится граф. Пример ввода графа вручную с клавиатуры представлен на рисунке 8.



```
C:\Users\givap\source\repos\kysach_logika\Debug\kysach_logika.exe

Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 2

Введите количество вершин в графе: 5
Введите рёбра в формате "источник конечная вершина вес" (например, 0 1 5).
Введите "-1 -1 -1", чтобы завершить ввод.
Введите ребро: 0 1 -1
Введите ребро: 0 3 2
Введите ребро: 1 2 9
Введите ребро: 1 4 1
Введите ребро: 3 4 -5
Введите ребро: -1 -1 -1
Граф в виде списка ребер:
Вершина 0 --> Вершина 3 (вес: 2)
Вершина 0 --> Вершина 1 (вес: -1)
Вершина 1 --> Вершина 4 (вес: 1)
Вершина 1 --> Вершина 2 (вес: 9)
Вершина 3 --> Вершина 4 (вес: -5)
Вершина    Расстояние от источника
0          0
1          -1
2          8
3          2
4          -3
```

Рисунок 8 – Выбор ручного ввода графа

Если же пользователь выбирает чтение графа из файла, то необходимо ввести название заранее созданного файла, и тогда программа считывает граф из введенного файла. Пример реализации чтения графа из файла представлен на рисунке 9.

```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe

Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 3

Введите имя файла, для чтения из него графа: graph.txt
Граф в виде списка ребер:
Вершина 0 --> Вершина 2 (вес: 4)
Вершина 0 --> Вершина 1 (вес: 1)
Вершина 1 --> Вершина 3 (вес: 6)
Вершина 1 --> Вершина 2 (вес: 2)
Вершина 2 --> Вершина 3 (вес: 3)
Вершина    Расстояние от источника
0           0
1           1
2           3
3           6
Хотите сохранить результат в файл? (yes/no): no
```

Рисунок 9 – Считывание графа из файла

Ниже представлен код функции main.

```
int main() {
    setlocale(LC_ALL, "Russian");
    int vertices;
    char op;
    char filenameForRead[100];
    FILE* filenameForRead2;
    bool isDirected;

    while (1) {
        printf("\nВыберите тип графа:\n");
        printf("1 - Ориентированный граф\n");
        printf("2 - Неориентированный граф\n");
        printf("Введите номер операции: ");
        scanf(" %c", &op);
        if (op == '1') {
            isDirected = true;
        }
        else if (op == '2') {
```

```

        isDirected = false;
    }
    else {
        printf("Некорректный ввод, попробуйте снова.\n");
        continue;
    }

    printf("\nВыберите способ ввода графа:\n");
    printf("1 - Автоматический ввод графа\n");
    printf("2 - Ручной ввод графа\n");
    printf("3 - Чтение графа из файла\n");
    printf("0 - Завершение программы\n");
    printf("Введите номер операции: ");
    scanf(" %c", &op);
    switch (op) {
    case '1':
    {
        srand(time(NULL));

        printf("\nВведите количество вершин в графе: ");
        scanf("%d", &vertices);

        if (vertices <= 0) {
            printf("Введено некорректное количество вершин! Попробуйте снова\n");
            break;
        }

        // определение макс количества рёбер
        int maxEdgesCount = isDirected ? vertices * (vertices - 1) : vertices *
(vertices - 1) / 2;
        Edge* graph = (Edge*)malloc(maxEdgesCount * sizeof(Edge));
        if (graph == NULL) {
            printf("Ошибка при выделении памяти.\n");
            return 1;
        }

        int edgesCount = 0;

        // создание минимального остовного дерева, добавляя рёбра между
последовательными вершинами
        for (int i = 0; i < vertices - 1; ++i) {
            graph[edgesCount].source = i;
            graph[edgesCount].destination = i + 1;
            graph[edgesCount].weight = (rand() % 21) - 5; // вес

```



```

        edgesCount++;
        if (!isDirected) {
            graph[edgesCount].source = i + 1;
            graph[edgesCount].destination = i;
            graph[edgesCount].weight = graph[edgesCount - 1].weight;
            edgesCount++;
        }
    }

    // добавление доп рёбер
    for (int i = vertices; i < maxEdgesCount; ++i) {
        int source = rand() % vertices;
        int destination = rand() % vertices;
        ensurePathToVertex(graph, &edgesCount, source, destination,
isDirected);
    }

    printGraph(graph, edgesCount); // вызов функции вывода
    bellmanFord(graph, vertices, edgesCount, 0); // вызов функции выполнения
алгоритма

    free(graph); // освобождение памяти
    break;
}
case '2': { // Ручной ввод графа
    printf("\nВведите количество вершин в графе: ");
    scanf("%d", &vertices);

    if (vertices <= 0) {
        printf("Введено некорректное количество вершин! Попробуйте снова\n");
        break;
    }

    // макс количество рёбер
    int maxEdgesCount = isDirected ? vertices * (vertices - 1) : vertices *
(vertices - 1) / 2;

    Edge* graph = (Edge*)malloc(maxEdgesCount * sizeof(Edge));
    if (graph == NULL) {
        printf("Ошибка при выделении памяти.\n");
        return 1;
    }

    int edgesCount = 0;
    int source, destination, weight;

```

```

0 1 5).\n");

printf("Введите рёбра в формате \"источник конечная вершина вес\" (например,

printf("Введите \"-1 -1 -1\", чтобы завершить ввод.\n");

while (1) {
    printf("Введите ребро: ");
    scanf("%d %d %d", &source, &destination, &weight);

    // Проверка на завершение
    if (source == -1 && destination == -1 && weight == -1) {
        break;
    }

    // Проверка на валидность введённого ребра
    if (source < 0 || source >= vertices || destination < 0 || destination
    >= vertices) {

        printf("Некорректные вершины. Попробуйте снова.\n");
        continue;
    }

    // Запись в граф
    graph[edgesCount].source = source;
    graph[edgesCount].destination = destination;
    graph[edgesCount].weight = weight;
    edgesCount++;
    if (!isDirected) {
        graph[edgesCount].source = destination;
        graph[edgesCount].destination = source;
        graph[edgesCount].weight = weight;
        edgesCount++;
    }
}

printGraph(graph, edgesCount);
bellmanFord(graph, vertices, edgesCount, 0);

free(graph);
break;
}
case '3':
{
    printf("\nВведите имя файла, для чтения из него графа: ");
    scanf("%s", filenameForRead);
    filenameForRead2 = fopen(filenameForRead, "r");

```

```

        if (filenameForRead2 == NULL) {
            printf("Не удалось открыть файл, так как файл пустой или имеет не
соответствующий вид! Повторите ввод\n\n");
            break;
        }
        else {
            Edge graph[MAX_VERTICES];
            int edgesCount = readGraphFromFile(filenameForRead, graph, &vertices,
isDirected);

            if (edgesCount < 0) {
                return 1;
            }
            printGraph(graph, edgesCount);
            bellmanFord(graph, vertices, edgesCount, 0);
            break;
        }
        fclose(filenameForRead2);
    }
    case '0':
    {
        return 0;
        break;
    }
    default: {
        printf("Некорректный ввод, попробуйте снова.\n");
        break;
    }
}
return 0;
}

```

5 Тестирование

Среда разработки Microsoft Visual Studio 2019 обладает всеми средствами необходимыми при разработке и отладке программы, а именно: возможность поставить точки останова, пошаговое выполнение программы, а также отслеживание и анализ содержимого переменных.

Тестирование проводилось в рабочем порядке, в процессе разработки и после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем. Все тесты, проводимые с программой представлены в таблице 1.

Таблица 1 – Описание поведения программы при тестировании

№	Описание	Предусловие	Тестирование	Ожидаемый результат
1	Работа меню	Программа запущена	Запуск программы с помощью Visual Studio	Вывод в консоли меню выбора типа графа
2	Выбор функции	Программа запущена и выбран тип графа	Ввод номера функции	Переход к выполнению выбранной функции
3	Создание ориентированного графа	Ввод в меню выбора типа графа 1	Выбор способа создания и ввод числа вершин	Создание ориентированного графа и вывод на экран
4	Создание неориентированного графа	Ввод в меню выбора типа графа 2	Выбор способа создания и ввод числа вершин	Создание неориентированного графа и вывод на экран
5	Генераций графа случайным образом	Ввод в меню функций 1	Ввод числа вершин	Создание графа случайным образом
6	Ввод графа вручную	Ввод в меню функций 2	Ввод числа вершин и последующее создание ребер	Создание графа с рёбрами, которые вводил пользователь

Продолжение таблицы 1.

№	Описание	Предусловие	Тестирование	Ожидаемый результат
7	Загрузка графа из файла	Ввод в меню функций 3	Ввод имени файла	Считывание графа из файла и вывод на экран, с последующим выполнением алгоритма
8	Сохранение в файл результата	Выбор любого типа графа и создание графа любым способом	Ввод названия файла сохранения	Сохранение в файле полученных результатов

Результаты проводимого тестирования продемонстрированы на рисунках ниже.

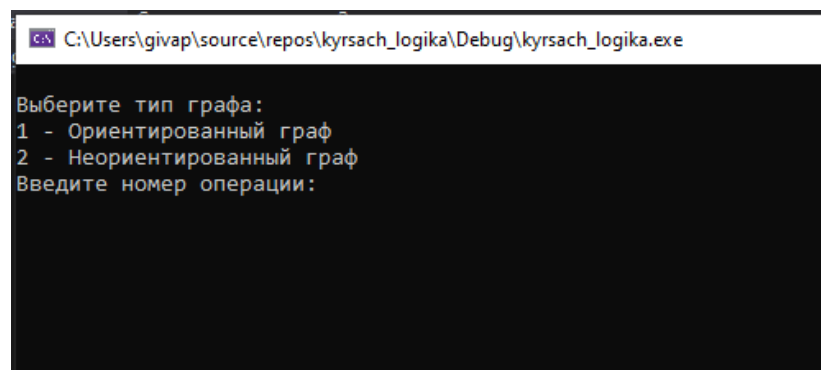


Рисунок 10 – Тестирование работы меню

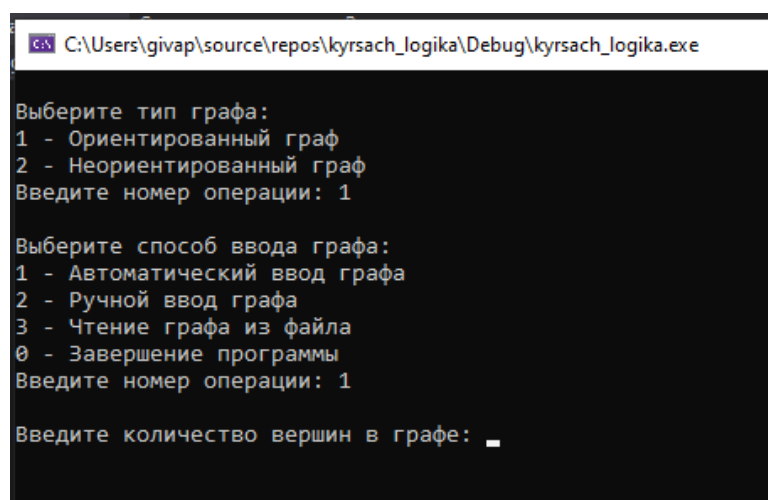


Рисунок 11 – Тестирование выбора функции

```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe
Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 1

Введите количество вершин в графе: 5
Граф в виде списка ребер:
Вершина 0 --> Вершина 2 (вес: -1)
Вершина 0 --> Вершина 3 (вес: 5)
Вершина 0 --> Вершина 1 (вес: 7)
Вершина 1 --> Вершина 4 (вес: 5)
Вершина 1 --> Вершина 2 (вес: 3)
Вершина 2 --> Вершина 4 (вес: 10)
Вершина 2 --> Вершина 3 (вес: -1)
Вершина 3 --> Вершина 4 (вес: 15)
Вершина    Расстояние от источника
0           0
1           7
2          -1
3          -2
4           9
```

Рисунок 12 – Создание ориентированного графа

```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe
Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 2

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 1

Введите количество вершин в графе: 4
Граф в виде списка ребер:
Вершина 0 --> Вершина 1 (вес: 7)
Вершина 1 --> Вершина 2 (вес: 9)
Вершина 1 --> Вершина 0 (вес: 7)
Вершина 2 --> Вершина 3 (вес: 3)
Вершина 2 --> Вершина 1 (вес: 9)
Вершина 3 --> Вершина 2 (вес: 3)
Вершина    Расстояние от источника
0           0
1           7
2          16
3          19
```

Рисунок 13 – Создание неориентированного графа

```

C:\Users\givap\source\repos\kysrascn_logika\Debug\kysrascn_logika.exe
Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 1

Введите количество вершин в графе: 4
Граф в виде списка ребер:
Вершина 0 --> Вершина 3 (вес: 9)
Вершина 0 --> Вершина 1 (вес: 11)
Вершина 1 --> Вершина 2 (вес: 10)
Вершина 2 --> Вершина 3 (вес: -1)
Вершина    Расстояние от источника
0           0
1           11
2           21
3           9
Хотите сохранить результат в файл? (yes/no): no

```

Рисунок 14 – Генерация графа случайным образом

```

C:\Users\givap\source\repos\kysrascn_logika\Debug\kysrascn_logika.exe
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 2

Введите количество вершин в графе: 4
Введите рёбра в формате "источник конечная вершина вес" (например, 0 1 5).
Введите "-1 -1 -1", чтобы завершить ввод.
Введите ребро: 0 1 -1
Введите ребро: 0 3 2
Введите ребро: 1 2 5
Введите ребро: 1 3 2
Введите ребро: 2 3 4
Введите ребро: -1 -1 -1
Граф в виде списка ребер:
Вершина 0 --> Вершина 3 (вес: 2)
Вершина 0 --> Вершина 1 (вес: -1)
Вершина 1 --> Вершина 3 (вес: 2)
Вершина 1 --> Вершина 2 (вес: 5)
Вершина 2 --> Вершина 3 (вес: 4)
Вершина    Расстояние от источника
0           0
1           -1
2           4
3           1

```

Рисунок 15 – Тестирование ввода графа вручную

```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe
Выберите тип графа:
1 - Ориентированный граф
2 - Неориентированный граф
Введите номер операции: 1

Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 3

Введите имя файла, для чтения из него графа: graph.txt
Граф в виде списка ребер:
Вершина 0 --> Вершина 2 (вес: 4)
Вершина 0 --> Вершина 1 (вес: 1)
Вершина 1 --> Вершина 3 (вес: 6)
Вершина 1 --> Вершина 2 (вес: 2)
Вершина 2 --> Вершина 3 (вес: 3)
Вершина    Расстояние от источника
0           0
1           1
2           3
3           6
Хотите сохранить результат в файл? (yes/no): no
```

Рисунок 16 – Тестирование считывания графа из файла

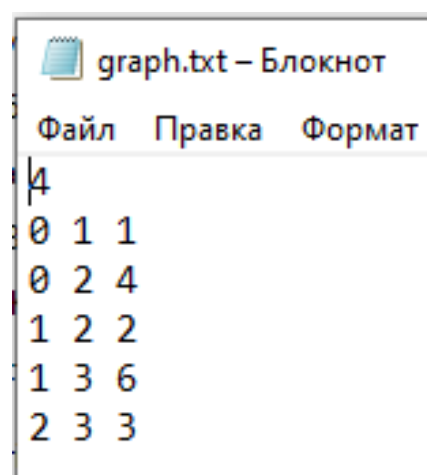


Рисунок 17 – Граф занесенный в файл


```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe
Введите номер операции: 1
Выберите способ ввода графа:
1 - Автоматический ввод графа
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 1
Введите количество вершин в графе: 5
Граф в виде списка ребер:
Вершина 0 --> Вершина 4 (вес: 1)
Вершина 0 --> Вершина 1 (вес: 1)
Вершина 1 --> Вершина 3 (вес: 14)
Вершина 1 --> Вершина 4 (вес: 13)
Вершина 1 --> Вершина 2 (вес: 14)
Вершина 2 --> Вершина 3 (вес: 11)
Вершина 3 --> Вершина 4 (вес: 9)
Вершина    Расстояние от источника
0           0
1           1
2          15
3          15
4           1
Хотите сохранить результат в файл? (yes/no): yes
Введите название файла: result.txt
Результат успешно сохранен в файл result.txt
```

Рисунок 18 – Тестирование сохранения в файл результата

```
result.txt – Блокнот
Файл  Правка  Формат  Вид  Справка
Результаты алгоритма Форда-Беллмана:
Вершина    Расстояние от источника
0           0
1           1
2          15
3          15
4           1
Список рёбер графа:
Ребро: 0 -> 1 вес = 1
Ребро: 0 -> 4 вес = 1
Ребро: 1 -> 4 вес = 13
Ребро: 1 -> 2 вес = 14
Ребро: 1 -> 3 вес = 14
Ребро: 2 -> 3 вес = 11
Ребро: 3 -> 4 вес = 9
```

Рисунок 19 – Сохраненный результат

В результате тестирования было выявлено, что программа успешно проверяет данные на соответствие необходимым требованиям.

6 Ручной расчёт программы

Проведем проверку программы посредством ручных вычислений на примере графа, представленного на рисунке 20.

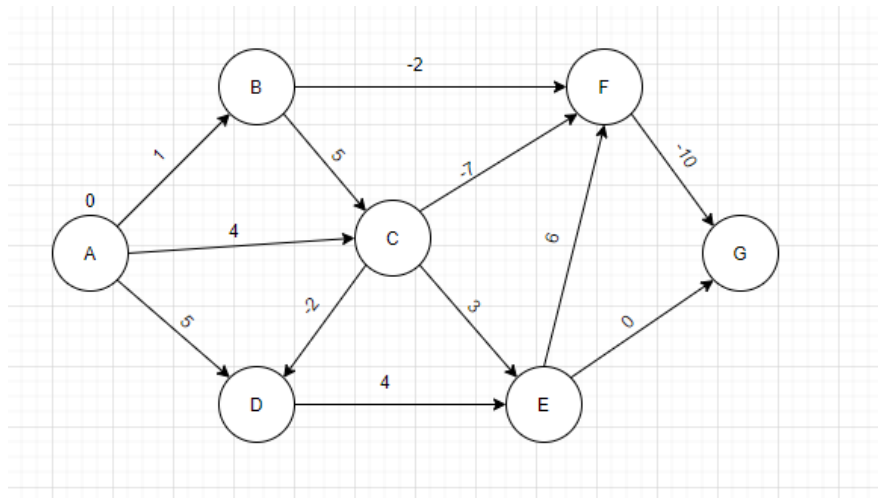


Рисунок 20 – Пример графа для ручного расчета

Дан взвешенный ориентированный граф $G(V, E)$, в котором возможны отрицательные рёбра. Необходимо от вершины s до всех остальных вершин найти пути кратчайшие пути.

На первом шаге расстояние до начальной вершины, то есть до вершины A определяется равным нулю, а до всех остальных вершин равным бесконечности.

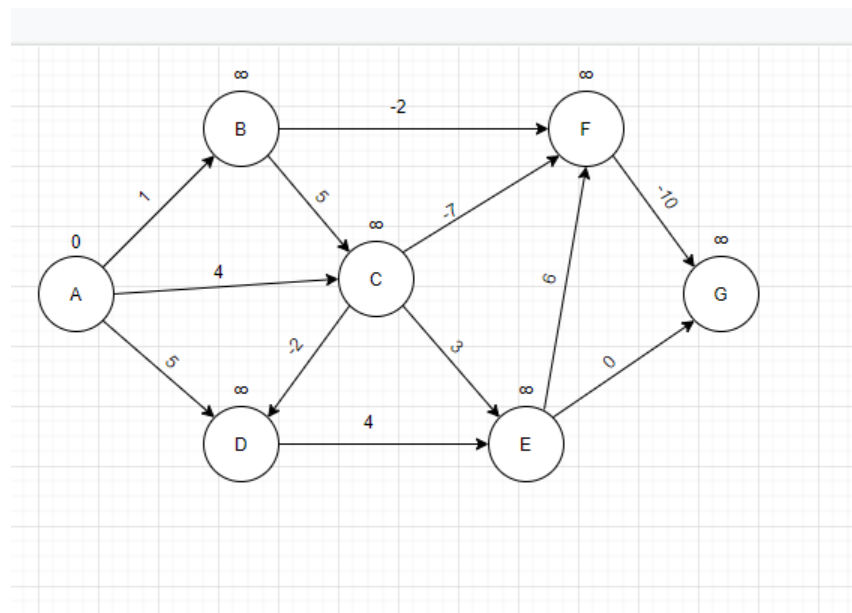


Рисунок 21 – Установление кратчайших путей

Далее определяются кратчайшие пути до вершин В, С и D, так как пока еще недоступны другие пути, расстояния будут равны 1, 4 и 5 соответственно.

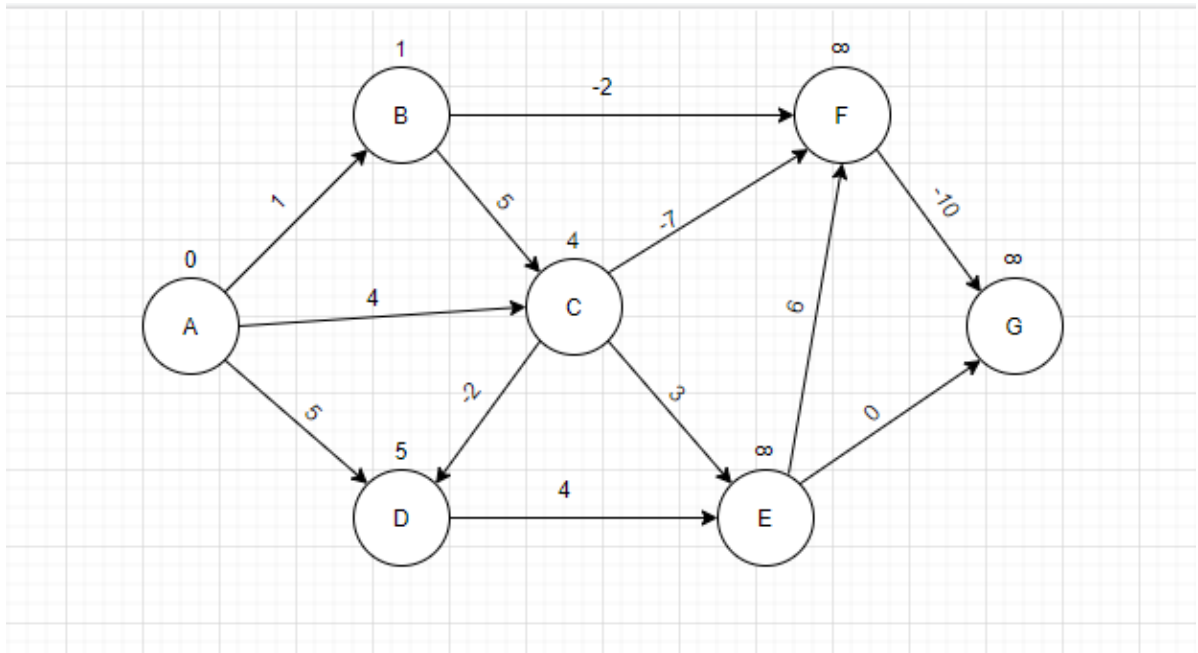


Рисунок 22 – Обновление кратчайших путей (Часть 1)

Рассмотрим вершину В. Пройдя в эту вершину, изменится длина пути А-В-F с бесконечности до -1. Длина пути А-В-С (6) больше, чем длина пути А-С (4), поэтому длина пути в вершину С будет равна также 4.

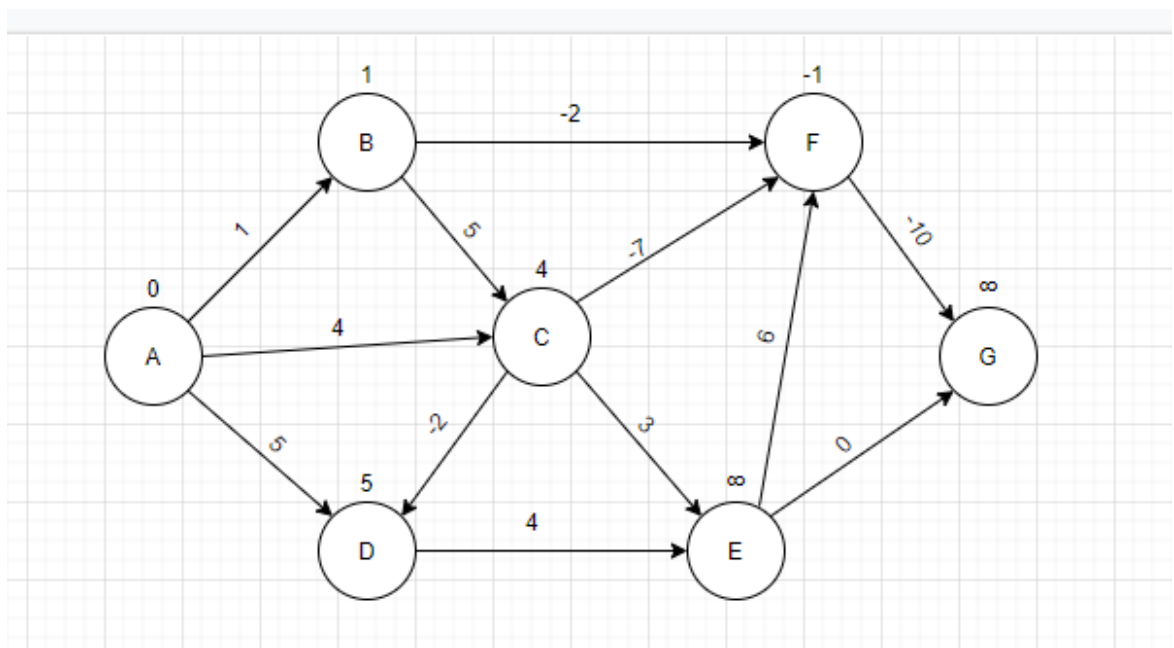


Рисунок 23 – Обновление кратчайших путей (Часть 2)

Далее рассмотрим вершину С. Пройдя в эту вершину изменится длина пути в вершину F, так как длина пути A-C-F (-3), что меньше длины пути A-B-F (-1). Длина пути в вершину D изменится, потому что длина пути A-C-D (2) меньше длины пути A-D (5), длина пути в вершину E изменится с бесконечности до длины пути A-C-E (7).

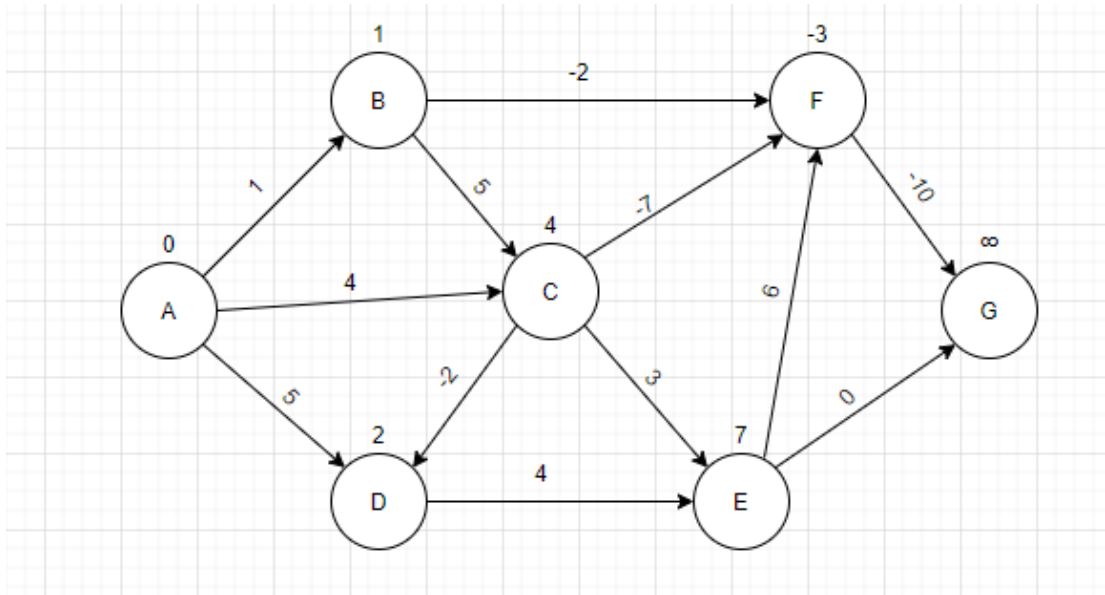


Рисунок 24 – Обновление кратчайших путей (Часть 3)

Проходя через вершину D, длина пути в вершину E изменится, так как длина пути A-C-D-E (6) короче длины пути A-C-E (7).

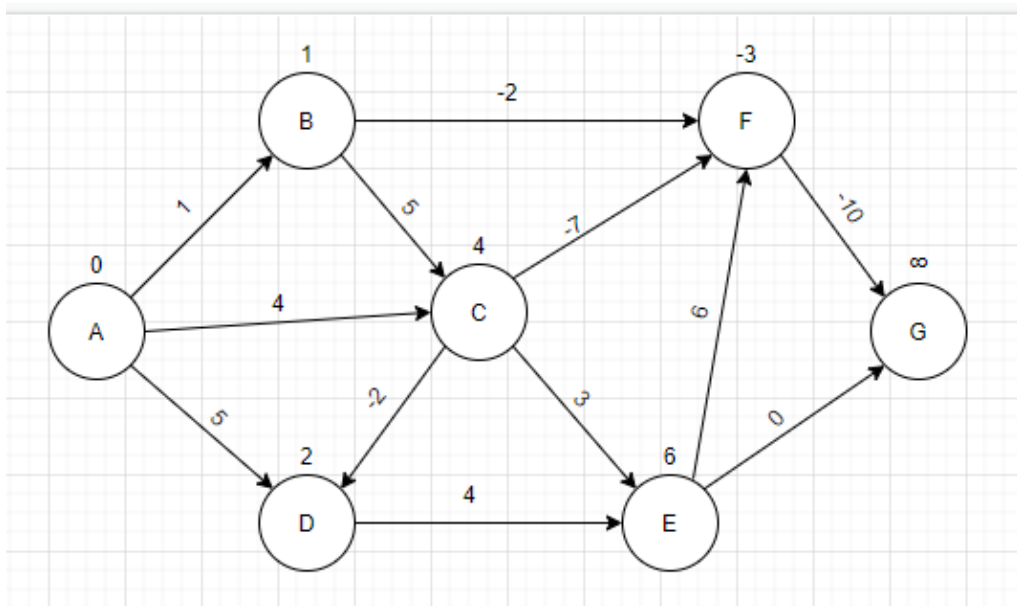


Рисунок 25 – Обновление кратчайших путей (Часть 4)

Рассматривая вершину E, длина пути в вершину G изменится с бесконечности до 6, а длина пути в вершину F не изменится, потому что длина пути A-C-D-E-F (12) больше длины пути A-C-F (-3).

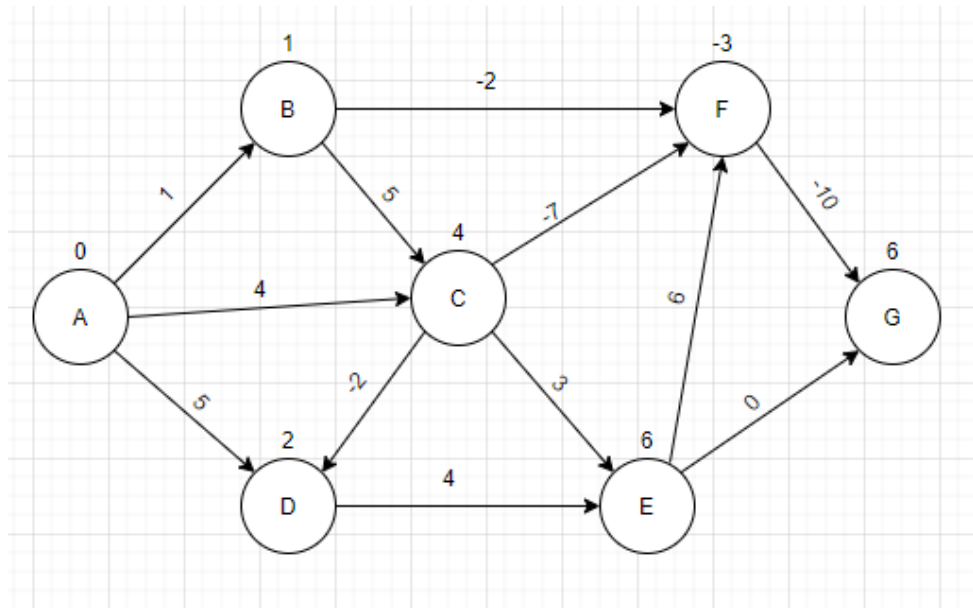


Рисунок 26 – Обновление кратчайших путей (Часть 5)

Последним действием в данном графе будет являться построение путей из вершины F. Длина пути в вершину G изменится, так как длина пути A-C-F-G (-13) меньше длины пути A-C-D-E-G (6).

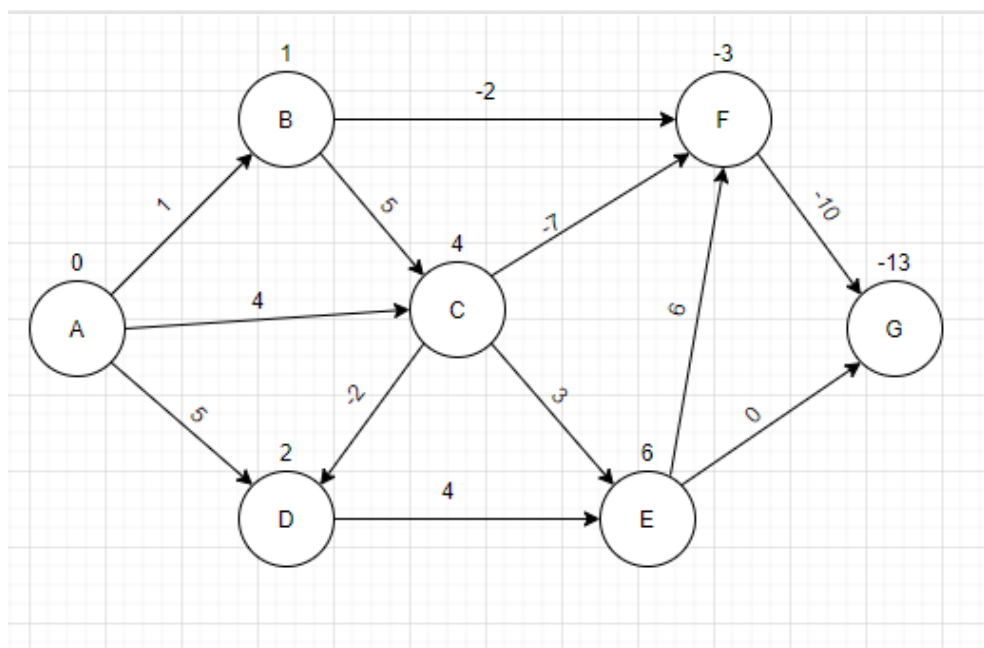


Рисунок 27 – Результат ручного расчета

Так как из вершины G нет доступных путей, граф, представленный на рисунке 27, будет являться результатом.

Таким образом можно сделать вывод, что программа работает верно.

```
C:\Users\givap\source\repos\kyrsach_logika\Debug\kyrsach_logika.exe
2 - Ручной ввод графа
3 - Чтение графа из файла
0 - Завершение программы
Введите номер операции: 2

Введите количество вершин в графе: 7
Введите рёбра в формате "источник конечная вершина вес" (например, 0 1 5).
Введите "-1 -1 -1", чтобы завершить ввод.
Введите ребро: 0 1 1
Введите ребро: 0 2 4
Введите ребро: 0 3 5
Введите ребро: 1 2 5
Введите ребро: 1 5 -2
Введите ребро: 2 3 -2
Введите ребро: 2 4 3
Введите ребро: 2 5 -7
Введите ребро: 3 4 4
Введите ребро: 4 5 6
Введите ребро: 4 6 0
Введите ребро: 5 6 -10
Введите ребро: -1 -1 -1
Граф в виде списка ребер:
Вершина 0 --> Вершина 2 (вес: 4)
Вершина 0 --> Вершина 3 (вес: 5)
Вершина 0 --> Вершина 1 (вес: 1)
Вершина 1 --> Вершина 5 (вес: -2)
Вершина 1 --> Вершина 2 (вес: 5)
Вершина 2 --> Вершина 3 (вес: -2)
Вершина 2 --> Вершина 4 (вес: 3)
Вершина 2 --> Вершина 5 (вес: -7)
Вершина 3 --> Вершина 4 (вес: 4)
Вершина 4 --> Вершина 6 (вес: 0)
Вершина 4 --> Вершина 5 (вес: 6)
Вершина 5 --> Вершина 6 (вес: -10)
Вершина    Расстояние от источника
0           0
1           1
2           4
3           2
4           6
5          -3
6          -13
Введите имя файла, для записи в него последующего результата: _
```

Рисунок 28 – Проверка результата в программе

Заключение

Таким образом, в процессе создания данного проекта разработана программа, реализующая алгоритм Форда – Беллмана для поиска кратчайших путей в Microsoft Visual Studio 2019.

При выполнении данной курсовой работы были получены навыки разработки программ и освоены приемы генерации графов, а также сохранения результатов в файл. Приобретены навыки по осуществлению алгоритма поиска кратчайших путей. Углублены знания языка программирования Си.

Так как программа работает в консольном режиме, не добавляющем к сложности языка сложность программного оконного интерфейса, пользовательский интерфейс достаточно примитивен, что является недостатком программы.

Приложение имеет небольшой, но достаточный для выполнения алгоритма Форда – Беллмана функционал.

Список литературы

1. Керниган Б. Ритчи Д. Язык программирования С. 1985 г.
2. Теория графов: [сайт] – URL: https://foxford.ru/wiki/informatika/teoriya-grafov?utm_referrer=https%3A%2F%2Fyandex.ru%2F (дата обращения: 01.11.2024).
3. Графы и программирование: [сайт] – URL: <https://habr.com/ru/articles/734642/> (дата обращения: 05.11.2024).
4. Работа с файлами в языке С: [сайт] – URL: <https://prog-cpp.ru/c-files/> (дата обращения: 12.11.2024).
5. Харви Дейтел, Пол Дейтел. Как программировать на С/С++. 2009 г.
6. Структуры в С: [сайт] – URL: <https://metanit.com/c/tutorial/6.1.php> (дата обращения: 20.11.2024).
7. Определение и описание функций в С: [сайт] – URL: <https://metanit.com/c/tutorial/4.1.php> (дата обращения: 22.11.2024).

Приложение А.

Листинг программы.

```
#define _CRT_SECURE_NO_WARNINGS
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <locale.h>
#include <time.h>
#include <stdbool.h>

#define MAX_VERTICES 100 // определить максимум вершин
#define INF INT_MAX // определить значение бесконечности для инициализации

// структура ребра
typedef struct {
    int source; // начальная вершина ребра
    int destination; // конечная вершина ребра
    int weight; // вес
} Edge;

// Функция для чтения графа из файла
int readGraphFromFile(const char* filename, Edge edges[], int* verticesCount, bool
isDirected) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        printf("Ошибка при открытии файла %s!\n", filename);
        return -1;
    }

    int edgesCount = 0;
    fscanf(file, "%d", verticesCount); // Чтение количества вершин

    while (fscanf(file, "%d %d %d", &edges[edgesCount].source,
&edges[edgesCount].destination,
&edges[edgesCount].weight) == 3) {
        edgesCount++;
        if (!isDirected) {
            edges[edgesCount].source = edges[edgesCount - 1].destination;
            edges[edgesCount].destination = edges[edgesCount - 1].source;
            edges[edgesCount].weight = edges[edgesCount - 1].weight;
            edgesCount++;
        }
    }
    fclose(file);
    return edgesCount;
}

int compareEdges(const void* a, const void* b) {
    return ((Edge*)a)->source - ((Edge*)b)->source;
}

// функция для вывода графа
void printGraph(Edge edges[], int edgesCount) {
    printf("Граф в виде списка ребер:\n");
    qsort(edges, edgesCount, sizeof(Edge), compareEdges);
    for (int i = 0; i < edgesCount; i++) {
        printf("Вершина %d --> Вершина %d (вес: %d)\n", edges[i].source,
edges[i].destination, edges[i].weight);
    }
}

// функция для сохранения графа в файл
```

```

void printGraphInFile(Edge edges[], int edgesCount, FILE* file) {
    fprintf(file, "Список рёбер графа:\n");
    qsort(edges, edgesCount, sizeof(Edge), compareEdges);
    for (int i = 0; i < edgesCount; i++) {
        fprintf(file, "Ребро: %d -> %d вес = %d\n", edges[i].source, edges[i].destination,
edges[i].weight);
    }
}

// алгоритм Форда-Беллмана
void bellmanFord(Edge edges[], int vertices, int edgesCount, int source) {
    int* distance = (int*)malloc(vertices * sizeof(int)); // массив для хранения расстояний
    от исходной вершины до всех остальных

    // инициализация расстояния от источника до всех вершин как бесконечность
    for (int i = 0; i < vertices; ++i)
        distance[i] = INF;

    distance[source] = 0; // дистанция из источника до самого себя = 0

    // расслабление ребер vertices - 1 раз
    for (int i = 0; i < vertices - 1; ++i) {
        for (int j = 0; j < edgesCount; ++j) {
            // если текущая вершина не бесконечна и это ребро можно расслабить
            if (distance[edges[j].source] != INF &&
                distance[edges[j].destination] > distance[edges[j].source] +
edges[j].weight) {
                // обновление расстояния до целевой вершины
                distance[edges[j].destination] = distance[edges[j].source] +
edges[j].weight;
            }
        }
    }

    // проверка на наличие циклов с отрицательным весом
    for (int i = 0; i < edgesCount; ++i) {
        // если можно улучшить расстояние - значит есть отрицательный цикл
        if (distance[edges[i].source] != INF &&
            distance[edges[i].destination] > distance[edges[i].source] + edges[i].weight) {
            printf("Присутствует отрицательный цикл\n");
            free(distance);
            return; // выход из функции, если цикл есть
        }
    }

    // выводятся кратчайшие расстояния от источника до всех вершин
    printf("Вершина    Расстояние от источника\n");
    for (int i = 0; i < vertices; ++i)
        // вывод номера вершины и расстояние до неё (-1 если она не достижима)
        printf("%d \t\t %d\n", i, (distance[i] == INF) ? -1 : distance[i]);

    // Сохранение результатов в файл
    char filename[100];
    printf("Хотите сохранить результат в файл? (yes/no): ");
    char response[4];
    scanf("%s", response);

    if (strcmp(response, "yes") == 0) {
        printf("Введите название файла: ");
        scanf("%s", filename);
        FILE* file = fopen(filename, "w");
        if (file == NULL) {
            printf("Ошибка при открытии файла.\n");
        }
        else {
            fprintf(file, "Результаты алгоритма Форда-Беллмана:\n");
            fprintf(file, "Вершина    Расстояние от источника\n");
            for (int i = 0; i < vertices; ++i) {

```

```

        fprintf(file, "%d \t\t %d\n", i, (distance[i] == INF) ? -1 : distance[i]);
    }
    printGraphInFile(edges, edgesCount, file);
    fclose(file);
    printf("Результат успешно сохранен в файл %s\n", filename);
}
}
free(distance);
}

// функция для проверки существует ли ребро между вершинами или нет
bool edgeExists(Edge* edges, int edgesCount, int source, int dest) {
    for (int i = 0; i < edgesCount; i++) {
        if ((edges[i].source == source && edges[i].destination == dest) ||
            (edges[i].source == dest && edges[i].destination == source)) {
            return true; // если путь уже существует
        }
    }
    return false;
}

// Функция для обеспечения связности графа без создания дублирующих путей
void ensurePathToVertex(Edge* edges, int* edgesCount, int source, int destination, bool
isDirected) {
    if (source >= destination) return; // проверка на валидность

    if (!edgeExists(edges, *edgesCount, source, destination)) {
        edges[*edgesCount].source = source;
        edges[*edgesCount].destination = destination;
        edges[*edgesCount].weight = (rand() % 21) - 2; // вес рёбер
        (*edgesCount)++;
        if (!isDirected) {
            edges[*edgesCount].source = destination;
            edges[*edgesCount].destination = source;
            edges[*edgesCount].weight = edges[*edgesCount - 1].weight;
            (*edgesCount)++;
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    int vertices;
    char op;
    char filenameForRead[100];
    FILE* filenameForRead2;
    bool isDirected;

    while (1) {
        printf("\nВыберите тип графа:\n");
        printf("1 - Ориентированный граф\n");
        printf("2 - Неориентированный граф\n");
        printf("Введите номер операции: ");
        scanf(" %c", &op);
        if (op == '1') {
            isDirected = true;
        }
        else if (op == '2') {
            isDirected = false;
        }
        else {
            printf("Некорректный ввод, попробуйте снова.\n");
            continue;
        }

        printf("\nВыберите способ ввода графа:\n");
        printf("1 - Автоматический ввод графа\n");
        printf("2 - Ручной ввод графа\n");
    }
}

```

```

printf("3 - Чтение графа из файла\n");
printf("0 - Завершение программы\n");
printf("Введите номер операции: ");
scanf(" %c", &op);
switch (op) {
case '1':
{
    srand(time(NULL));

    printf("\nВведите количество вершин в графе: ");
    scanf("%d", &vertices);

    if (vertices <= 0) {
        printf("Введено некорректное количество вершин! Попробуйте снова\n");
        break;
    }

    // определение макс количества рёбер
    int maxEdgesCount = isDirected ? vertices * (vertices - 1) : vertices *
(vertices - 1) / 2;
    Edge* graph = (Edge*)malloc(maxEdgesCount * sizeof(Edge));
    if (graph == NULL) {
        printf("Ошибка при выделении памяти.\n");
        return 1;
    }

    int edgesCount = 0;

    // создание минимального остоного дерева, добавляя рёбра между
последовательными вершинами
    for (int i = 0; i < vertices - 1; ++i) {
        graph[edgesCount].source = i;
        graph[edgesCount].destination = i + 1;
        graph[edgesCount].weight = (rand() % 21) - 5; // вес
        edgesCount++;
        if (!isDirected) {
            graph[edgesCount].source = i + 1;
            graph[edgesCount].destination = i;
            graph[edgesCount].weight = graph[edgesCount - 1].weight;
            edgesCount++;
        }
    }

    // добавление доп рёбер
    for (int i = vertices; i < maxEdgesCount; ++i) {
        int source = rand() % vertices;
        int destination = rand() % vertices;
        ensurePathToVertex(graph, &edgesCount, source, destination,
isDirected);
    }

    printGraph(graph, edgesCount); // вызов функции вывода
    bellmanFord(graph, vertices, edgesCount, 0); // вызов функции выполнения
алгоритма

    free(graph); // освобождение памяти
    break;
}
case '2': { // Ручной ввод графа
    printf("\nВведите количество вершин в графе: ");
    scanf("%d", &vertices);

    if (vertices <= 0) {
        printf("Введено некорректное количество вершин! Попробуйте снова\n");
        break;
    }

    // макс количество рёбер

```

```

(vertices - 1) / 2;

int maxEdgesCount = isDirected ? vertices * (vertices - 1) : vertices *
Edge* graph = (Edge*)malloc(maxEdgesCount * sizeof(Edge));
if (graph == NULL) {
    printf("Ошибка при выделении памяти.\n");
    return 1;
}

int edgesCount = 0;
int source, destination, weight;

printf("Введите рёбра в формате \"источник конечная вершина вес\" (например,
0 1 5).\n");
printf("Введите \"-1 -1 -1\", чтобы завершить ввод.\n");

while (1) {
    printf("Введите ребро: ");
    scanf("%d %d %d", &source, &destination, &weight);

    // Проверка на завершение
    if (source == -1 && destination == -1 && weight == -1) {
        break;
    }

    // Проверка на валидность введённого ребра
    if (source < 0 || source >= vertices || destination < 0 || destination
    >= vertices) {
        printf("Некорректные вершины. Попробуйте снова.\n");
        continue;
    }

    // Запись в граф
    graph[edgesCount].source = source;
    graph[edgesCount].destination = destination;
    graph[edgesCount].weight = weight;
    edgesCount++;
    if (!isDirected) {
        graph[edgesCount].source = destination;
        graph[edgesCount].destination = source;
        graph[edgesCount].weight = weight;
        edgesCount++;
    }
}

printGraph(graph, edgesCount);
bellmanFord(graph, vertices, edgesCount, 0);

free(graph);
break;
}
case '3':
{
    printf("\nВведите имя файла, для чтения из него графа: ");
    scanf("%s", filenameForRead);
    filenameForRead2 = fopen(filenameForRead, "r");
    if (filenameForRead2 == NULL) {
        printf("Не удалось открыть файл, так как файл пустой или имеет не
соответствующий вид! Повторите ввод\n");
        break;
    }
    else {
        Edge graph[MAX_VERTICES];
        int edgesCount = readGraphFromFile(filenameForRead, graph, &vertices,
isDirected);

        if (edgesCount < 0) {
            return 1;
        }
        printGraph(graph, edgesCount);
    }
}
}

```

```

        bellmanFord(graph, vertices, edgesCount, 0);
        break;
    }
    fclose(filenameForRead2);
}
case '0':
{
    return 0;
    break;
}
default: {
    printf("Некорректный ввод, попробуйте снова.\n");
    break;
}
}
}
return 0;
}

```