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

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовой работе

по курсу «Логика и основы алгоритмизации

в инженерных задачах»

на тему «Реализация алгоритма поиска пути в лабиринте»

Выполнил:
студент группы 24ВВВ3
Тарасов А.В.

Принял:
к.т.н. Юрова О.В.

25.12.25
отлично

Пенза 2025

Содержание

ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Факультет Вычислительной техники

Кафедра "Вычислительная техника"

"УТВЕРЖДАЮ"

Зав. кафедрой ВТ _____

«____»_____20__

ЗАДАНИЕ

на курсовое проектирование по курсу

Логика и Основы алгоритмизации в инженерных задачах

Студенту Тарасову Алексею Владимировичу Группа 14ВВВ3

Тема_проекта Алгоритм поиска пути в лабиринте

Исходные данные (технические требования) на проектирование

Разработка алгоритмов и программного обеспечения в соответствии с данным заданием курсового проекта.

Пояснительная записка должна содержать:

1) Постановку задачи;
2) Теоретическую часть;
3) Описание алгоритма построения задачи;
4) Пример ручного расчета задачи и вычисления;
5) Описание самой программы;
6) Тесты;
7) Список литературы;
8) Листинг программы;
9) Результаты работы программы.

Объем работы по курсу

1. Расчетная часть

Ручной расчет работы алгоритма.

2. Графическая часть

Схема алгоритма в формате блок-схем.

3. Экспериментальная часть

Тестирование программы;
Результаты работы программы на тестовых данных.

Срок выполнения проекта по разделам

1 Исследование теоретической части курсового
2 Разработка алгоритмов программы
3 Разработка программы
4 Тестирование и завершение разработки программы
5 Оформление пояснительной записки
6
7
8

Дата выдачи задания "11" сентября 2025

Дата защиты проекта "____"_____

Руководитель Юлова Вика Викторовна

Задание получил "11" сентября 2025 г.

Студент Тарасов Алексей Владимирович

Реферат

Отчет содержит 51стр, 12 рисунков, 2 таблицы.

ГРАФ, ТЕОРИЯ ГРАФОВ, ДОСТИЖИМОСТЬ, ПОИСК В ГЛУБИНУ, ПОИСК В ШИРИНУ.

Цель исследования – разработка программы, способная генерировать полный лабиринт при помощи алгоритма, а также самостоятельно искать пути от входа и до выхода при помощи алгоритмов DFS и BFS.

В работе рассмотрены правила поиска в глубину и в ширину на основе которых происходит поиск путей в лабиринте.

Введение

Алгоритм поискав глубину (англ. depth-firstsearch, DFS) позволяет построить обход ориентированного или неориентированного графа, при котором посещаются все вершины, доступные из начальной вершины.

Отличие поиска в глубину от поиска в ширину заключается в том, что (в случае неориентированного графа) результатом алгоритма поиска в глубину является некоторый маршрут, следуя которому можно обойти последовательно все вершины графа, доступные из начальной вершины. Этим он принципиально отличается от поиска в ширину, где одновременно обрабатывается множество вершин, в поиске в глубину в каждый момент исполнения алгоритма обрабатывается только одна вершина.

С другой стороны, поиск в глубину не находит кратчайших путей, зато он применим в ситуациях, когда граф неизвестен целиком, а исследуется каким-то автоматизированным устройством.

В качестве среды разработки мною была выбрана среда MicrosoftVisualStudio2010, 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 создания лабиринта, а также поиска путей при помощи алгоритма DFSи BFS.

1. Постановка задачи

Требуется разработать программу, которая реализует алгоритмы поиска пути в лабиринте с использованием методов поиска в ширину и поиска в глубину. Исходный лабиринт в программе должен генерироваться случайным образом, причём при создании лабиринта должны быть предусмотрены граничные условия для обеспечения проходимости. Программа должна работать так, чтобы пользователь вводил параметры для генерации лабиринта или загружал существующий лабиринт из файла. После обработки этих данных на экран должен выводиться лабиринт с обозначенными стартовой и конечной точками, а также найденный путь между ними. Необходимо предусмотреть различные исходы поиска, чтобы программа не выдавала ошибок и работала корректно при любых входных данных.

Устройство ввода – клавиатура. Программа должна поддерживать интерактивный выбор алгоритма поиска, BFS или DFS, отображать время выполнения алгоритма и предоставлять возможность сохранения и загрузки лабиринтов.

2. Теоретическая часть задания

Лабиринт представляет собой особый вид графа, где вершины соответствуют клеткам лабиринта, а рёбра обозначают возможность перемещения между соседними клетками. Лабиринт задается матрицей, элементы которой указывают тип клетки: стена, проход, старт и выход. При представлении лабиринта матрицей информация о возможных перемещениях хранится в квадратной структуре, где наличие прохода из одной клетки в соседнюю определяется отсутствием стены между ними.

Для поиска пути в лабиринте применяются алгоритмы обхода графов: Поиск в ширину (BFS - Breadth-First Search) и поиск в глубину (DFS - Depth-First Search). BFS - алгоритм, который исследует все соседние вершины на текущей глубине перед переходом к вершинам на следующем уровне. Использует структуру данных "очередь" (FIFO - First In First Out). Данный алгоритм гарантированно находит кратчайший путь в невзвешенном графе. DFS - алгоритм, который идет вглубь по одному пути до тех пор, пока не достигнет тупика, после чего возвращается назад и исследует альтернативные пути. Использует структуру данных "стек" (LIFO - Last In First Out). Этот алгоритм может найти путь быстрее в некоторых случаях, но не гарантирует его оптимальность.

Матричное представление лабиринта позволяет эффективно проверять соседние клетки и определять возможные направления движения (вверх, вниз, влево, вправо). Каждая клетка лабиринта имеет координаты (i, j) в матрице размера M×N, где M - количество строк, N - количество столбцов.

3. Описание алгоритма программы

Для программной реализации алгоритмов поиска пути в лабиринте понадобятся следующие структуры данных: матрица лабиринта (char) - для хранения информации о типах клеток, двумерный массив visited (bool) - для отметки посещенных вершин, массив parent (Point**) - для восстановления найденного пути, а также структуры данных очередь (Queue) и стек (Stack) для реализации алгоритмов BFS и DFS соответственно.

Имеется лабиринт L размером M×N. Каждая клетка лабиринта изначально помечена как не посещенная, т. е. элементам массива visited присваивается значение false. В качестве стартовой точки выбирается клетка start с координатами (row, col) и помечается как посещенная: visited[row][col] = true.

Алгоритм BFS использует принцип FIFO (First In First Out), что обеспечивает поиск вширь и гарантирует нахождение кратчайшего пути в невзвешенном графе. Работает по следующему принципу: инициализация - Создается очередь “Queue” и в нее помещается стартовая клетка start; Основной цикл - Пока очередь не пуста извлекается текущая клетка current из очереди, если current равна целевой клетке end, путь найден, иначе проверяются четыре соседние клетки (вверх, вниз, влево, вправо), а для каждой допустимой соседней клетки (не стена и не посещена) она помечается как посещенная, ее родитель устанавливается как current, и клетка добавляется в очередь; восстановление пути - при нахождении целевой клетки путь восстанавливается от end к start через массив parent, отмечая клетки пути символом '*'.

Алгоритм DFS использует принцип LIFO (Last In First Out), что приводит к поиску вглубь и может быть эффективнее в лабиринтах с длинными тупиковыми ответвлениями. Работает подобным образом: инициализация - создается стек “Stack” и в него помещается стартовая клетка start; основной цикл - пока стек не пуст извлекается текущая клетка current из стека, если current равна целевой клетке end, путь найден, иначе проверяются четыре соседние клетки, для каждой допустимой соседней клетки она помечается как посещенная, ее родитель устанавливается как current, и клетка добавляется в стек; восстановление пути - при нахождении целевой клетки путь восстанавливается от end к start через массив parent, отмечая клетки пути символом '*'.

Перед запуском алгоритмов программа не выполняет предварительную проверку связности графа лабиринта. Вместо этого алгоритмы самостоятельно определяют достижимость конечной точки. Если после полного обхода всех достижимых клеток из стартовой точки целевая клетка не была найдена, алгоритмы возвращают false, что интерпретируется как отсутствие пути между start и end.

Для анализа эффективности реализовано измерение времени выполнения каждого алгоритма. BFS всегда находит кратчайший путь, но может использовать больше памяти из-за хранения всех клеток текущего уровня в очереди. DFS может быть быстрее в лабиринтах с прямыми путями к цели, но найденный путь не обязательно будет кратчайшим.

```
Intmain():
1. ВЫВЕСТИ "=== ГЕНЕРАТОР СЛУЧАЙНЫХ ЛАБИРИНТОВ ==="
2. ВЫБРАТЬ вариант:
   ЕСЛИ 1: СОЗДАТЬ новый лабиринт
   ЕСЛИ 2: ЗАГРУЗИТЬ лабиринт из файла
3. ПОЛУЧИТЬ параметры rows, cols
4. ЕСЛИ вариант == 1:
5.   СОЗДАТЬ лабиринт maze[rows][cols]
6.   ВЫЗВАТЬ generateMaze(maze, rows, cols, start, end)
7. ИНАЧЕ ЕСЛИ вариант == 2:
8.   ВВЕСТИ имя файла filename
9.   ВЫЗВАТЬ loadMazeFromFile(maze, rows, cols, start, end, filename)
10. ВЫВЕСТИ параметрылабиринта
11. ВЫЗВАТЬ printMaze(maze, rows, cols)
12. ПОКА ИСТИНА делать:
13.   ВЫВЕСТИ меню
14.   ВВЕСТИ choice
15.   ЕСЛИ choice == 1:
16.     ВЫЗВАТЬ clearSolution(maze, rows, cols, start, end)
17.     ВЫБРАТЬ алгоритм:
        ЕСЛИ 1: алгоритм = BFS
        ЕСЛИ 2: алгоритм = DFS
18.     start_time = текущее_время()
19.     ЕСЛИ алгоритм == BFS:
20.       result = ВЫЗВАТЬ bfsFindPath(maze, rows, cols, start, end)
21.     ИНАЧЕ:
22.       result = ВЫЗВАТЬ dfsFindPath(maze, rows, cols, start, end)
23.     end_time = текущее_время()
24.     time = end_time - start_time
25.     ЕСЛИ result == ИСТИНА:
26.       ВЫВЕСТИ "Путь найден за " + time + " секунд"
27.       ВЫЗВАТЬ printMaze(maze, rows, cols)
28.     ИНАЧЕ:
29.       ВЫВЕСТИ "Путь не найден"
30.   ИНАЧЕ ЕСЛИ choice == 2:
```

31. ВЫЗВАТЬ generateMaze(maze, rows, cols, start, end)
32. ВЫЗВАТЬ printMaze(maze, rows, cols)
33. ИНАЧЕ ЕСЛИ choice == 3:
34. ОСВОБОДИТЬ maze
35. ВВЕСТИ имя файла filename
36. result = ВЫЗВАТЬ loadMazeFromFile(maze, rows, cols, start, end, filename)
37. ЕСЛИ result == ИСТИНА:
38. ВЫЗВАТЬ printMaze(maze, rows, cols)
39. ИНАЧЕ ЕСЛИ choice == 4:
40. ВЫЗВАТЬ clearSolution(maze, rows, cols, start, end)
41. ВЫЗВАТЬ printMaze(maze, rows, cols)
42. ИНАЧЕ ЕСЛИ choice == 5:
43. ВЫВЕСТИ "Сохранить лабиринт в файл"
44. ВВЕСТИ имяфайла filename
45. ВЫЗВАТЬ saveMazeToFile(maze, rows, cols, filename)
46. ИНАЧЕ ЕСЛИ choice == 6:
47. ВЫВЕСТИ "Выход из программы"
48. ВЫХОД
49. КОНЕЦ ВЫБОРА
50. КОНЕЦ ЦИКЛА

VoidgenerateMaze:
1. ДЛЯ i = 0 ПОКА i< rows ДЕЛАТЬ i = i + 1
2. ДЛЯ j = 0 ПОКА j<cols ДЕЛАТЬ j = j + 1
3. ЕСЛИ i == 0 ИЛИ i == rows-1 ИЛИ j == 0 ИЛИ j == cols-1:
4. maze[i][j] = '#'
5. ИНАЧЕ:
6. maze[i][j] = ' '
7. КОНЕЦ УСЛОВИЯ
8. КОНЕЦ ЦИКЛА
9. КОНЕЦ ЦИКЛА
10. ДЛЯ i = 1 ПОКА i< rows-1 ДЕЛАТЬ i = i + 1
11. ДЛЯ j = 1 ПОКА j<cols-1 ДЕЛАТЬ j = j + 1
12. ЕСЛИ i % 2 == 0 И j % 2 == 0:

```
13.       maze[i][j] = '#'
14.     КОНЕЦ УСЛОВИЯ
15.   КОНЕЦ ЦИКЛА
16. КОНЕЦ ЦИКЛА
17. ДЛЯi = 2 ПОКАi<rows-1 ДЕЛАТЬi = i + 2
18.   ДЛЯ j = 2 ПОКА j<cols-1 ДЕЛАТЬ j = j + 2
19.     direction = случайное_число(0, 3)
20.     wallRow = i, wallCol = j
21.     ВЫБРАТЬ direction:
        ЕСЛИ 0: wallRow = i - 1
        ЕСЛИ 1: wallRow = i + 1
        ЕСЛИ 2: wallCol = j - 1
        ЕСЛИ 3: wallCol = j + 1
22.     ЕСЛИ wallRow> 0 И wallRow< rows-1 И wallCol> 0 И wallCol< cols-1:
23.       maze[wallRow][wallCol] = '#'
24.     КОНЕЦ УСЛОВИЯ
25.   КОНЕЦ ЦИКЛА
26. КОНЕЦ ЦИКЛА
27. ПОКА ИСТИНА:
28.   start.row = случайное_число(1, rows-2)
29.   start.col = случайное_число(1, cols-2)
30.   ЕСЛИ maze[start.row][start.col] != '#':
31.     ПРЕРВАТЬ
32. КОНЕЦ ЦИКЛА
33. maze[start.row][start.col] = 'O'
34. ПОКА ИСТИНА:
35.   end.row = случайное_число(1, rows-2)
36.   end.col = случайное_число(1, cols-2)
37.   ЕСЛИ maze[end.row][end.col] != '#' И (расстояние(start, end) >
(rows+cols)/4):
38.     ПРЕРВАТЬ
39. КОНЕЦ ЦИКЛА
40. maze[end.row][end.col] = 'X'
```

Bool bfsFindPath:

1. СОЗДАТЬ visited[rows][cols] = ЛОЖЬ
2. СОЗДАТЬparent[rows][cols] = { -1, -1 }
3. СОЗДАТЬочередьq
4. visited[start.row][start.col] = ИСТИНА
5. q.добавить(start)
6. dr = {-1, 1, 0, 0}
7. dc = {0, 0, -1, 1}
8. ПОКА q НЕ пуста:
9. current = q.извлечь()
10. ЕСЛИ current == end:
11. p = end
12. ПОКАp != start:
13. ЕСЛИ maze[p.row][p.col] != 'O' И maze[p.row][p.col] != 'X':
14. maze[p.row][p.col] = '*'
15. p = parent[p.row][p.col]
16. ВЕРНУТЬ ИСТИНА
17. КОНЕЦ УСЛОВИЯ
18. ДЛЯi = 0 ПОКАi< 4ДЕЛАТЬi = i + 1
19. newRow = current.row + dr[i]
20. newCol = current.col + dc[i]
21. ЕСЛИnewRow>= 0 ИnewRow< rows ИnewCol>= 0 ИnewCol< cols:
22. ЕСЛИ maze[newRow][newCol] != '#' И visited[newRow][newCol] == ЛОЖЬ:
23. visited[newRow][newCol] = ИСТИНА
24. parent[newRow][newCol] = current
25. СОЗДАТЬnewPoint(newRow, newCol)
26. q.добавить(newPoint)
27. КОНЕЦ УСЛОВИЯ
28. КОНЕЦ УСЛОВИЯ
29. КОНЕЦ ЦИКЛА
30. КОНЕЦ ЦИКЛА
31. ВЕРНУТЬ ЛОЖЬ

```
BooldfsFindPath:
1. СОЗДАТЬvisited[rows][cols] = ЛОЖЬ
2. СОЗДАТЬparent[rows][cols] = { -1, -1 }
3. СОЗДАТЬстек s
4. visited[start.row][start.col] = ИСТИНА
5. s.положить(start)
6. dr = {-1, 1, 0, 0}
7. dc = {0, 0, -1, 1}
8. ПОКА s НЕ пуст:
9.   current = s.взять()
10.  ЕСЛИ current == end:
11.    p = end
12.    ПОКАp != start:
13.      ЕСЛИ maze[p.row][p.col] != 'O' И maze[p.row][p.col] != 'X':
14.        maze[p.row][p.col] = '*'
15.      p = parent[p.row][p.col]
16.    ВЕРНУТЬ ИСТИНА
17.  КОНЕЦ УСЛОВИЯ
18.  ДЛЯi = 0 ПОКАi< 4ДЕЛАТЬi = i + 1
19.    newRow = current.row + dr[i]
20.    newCol = current.col + dc[i]
21.    ЕСЛИnewRow>= 0 InewRow< rows InewCol>= 0 InewCol< cols:
22.      ЕСЛИ maze[newRow][newCol] != '#' И visited[newRow][newCol] ==
ЛОЖЬ:
23.        visited[newRow][newCol] = ИСТИНА
24.        parent[newRow][newCol] = current
25.        СОЗДАТЬnewPoint(newRow, newCol)
26.        s.положить(newPoint)
27.      КОНЕЦ УСЛОВИЯ
28.    КОНЕЦ УСЛОВИЯ
29.  КОНЕЦ ЦИКЛА
30. КОНЕЦ ЦИКЛА
31. ВЕРНУТЬ ЛОЖЬ
```

BoolloadMazeFromFile:
1. ОТКРЫТЬ файл filename для чтения
2. ЕСЛИ файл НЕ открыт: ВЕРНУТЬ ЛОЖЬ
3. ПРОЧИТАТЬ rows, cols из файла
4. ЕСЛИ rows< 3 ИЛИ rows> 100 ИЛИ cols< 3 ИЛИ cols> 100:
5. ВЫВЕСТИ "Неверные размеры"
6. ЗАКРЫТЬ файл
7. ВЕРНУТЬ ЛОЖЬ
8. КОНЕЦ УСЛОВИЯ
9. СОЗДАТЬ maze[rows][cols]
10. startFound = ЛОЖЬ
11. endFound = ЛОЖЬ
12. ДЛЯ i = 0 ПОКА i< rows ДЕЛАТЬ i = i + 1
13. ПРОЧИТАТЬ строку line из файла
14. ЕСЛИ длина(line) != cols:
15. ВЫВЕСТИ "Ошибка формата"
16. ОСВОБОДИТЬ maze
17. ЗАКРЫТЬ файл
18. ВЕРНУТЬ ЛОЖЬ
19. КОНЕЦ УСЛОВИЯ
20. ДЛЯ j = 0 ПОКА j<cols ДЕЛАТЬ j = j + 1
21. maze[i][j] = line[j]
22. ЕСЛИ line[j] == 'O':
23. start.row = i
24. start.col = j
25. startFound = ИСТИНА
26. ИНАЧЕ ЕСЛИ line[j] == 'X':
27. end.row = i
28. end.col = j
29. endFound = ИСТИНА
30. КОНЕЦ УСЛОВИЯ
31. КОНЕЦ ЦИКЛА
32. КОНЕЦ ЦИКЛА
33. ЗАКРЫТЬ файл
34. ЕСЛИ startFound == ЛОЖЬ ИЛИ endFound == ЛОЖЬ:

35. ВЫВЕСТИ "Не найдены O и/или X"
36. ОСВОБОДИТЬ maze
37. ВЕРНУТЬ ЛОЖЬ
38. КОНЕЦ УСЛОВИЯ
39. ВЕРНУТЬ ИСТИНА

VoidprintMaze:
1. ВЫВЕСТИ " "
2. ДЛЯ j = 0 ПОКА j<cols ДЕЛАТЬ j = j + 1
3. ВЫВЕСТИ форматированный номер столбца
4. КОНЕЦ ЦИКЛА
5. ПЕРЕЙТИ на новую строку
6. ДЛЯi = 0 ПОКАi<rowsДЕЛАТЬi = i + 1
7. ВЫВЕСТИ форматированный номер строки
8. ДЛЯ j = 0 ПОКА j<cols ДЕЛАТЬ j = j + 1
9. ВЫВЕСТИ maze[i][j] + " "
10. КОНЕЦ ЦИКЛА
11. ПЕРЕЙТИ на новую строку
12. КОНЕЦ ЦИКЛА
13. ПЕРЕЙТИ на новую строку

Полный код программы можно увидеть в Приложении А.

4. Описание программы

Для написания данной программы использован язык программирования Си. Язык программирования Си - универсальный язык программирования, который завоевал особую популярность у программистов, благодаря сочетанию возможностей языков программирования высокого и низкого уровней.

Проект был создан в виде консольного приложения Win32 (Visual C++). Данная программа является многомодульной, поскольку состоит из нескольких функций: main, generateMaze, bfsFindPath, fsFindPath, loadMazeFromFile, printMaze.

Работа программы начинается с запроса генерации лабиринта. Если пользователь выбрал сгенерировать новый лабиринт, то на экран выводится запрос на указание размера лабиринта. Как только размерность была задана в диапазоне от 7 и до 50 по вертикали и горизонтали, программа выводит случайно сгенерированный лабиринт. При выборе второго варианта – загрузка из файла – пользователь должен будет ввести название, при выполнении этого запроса ранее сохранённый лабиринт вновь откроется в консольной программе.

Далее, при выборе одного из вариантов, в консоли будет изображён лабиринт, а чуть ниже него будет выведено меню. В зависимости от указанного номера, пользователь может сделать следующее: 1 – программа найдёт путь(алгоритм bfs или dfs выбирается в этой функции); 2 – программа сгенерирует новый лабиринт; 3 – программа загрузит лабиринт из файла(потребуется ввести его название); 4 – пользователь изменит размерность лабиринта и программа сгенерирует новый лабиринт на

основании новых параметров; 5 – если путь был ранее был найден в лабиринте, программа его сотрёт;

6 – позволяет пользователю сохранить сгенерированный лабиринт, позволяя ввести название файла; 7 – выход из консольной программы.

Ниже можно увидеть оформление начального запроса и дальнейшие действия с ним.

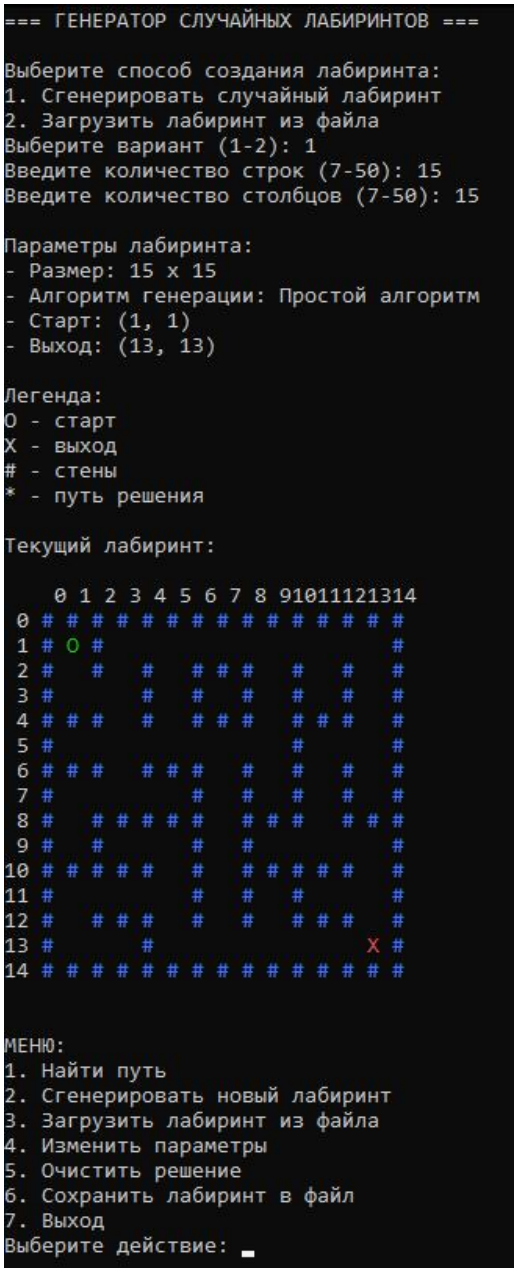

Рисунок 1 – Генерация лабиринта при выборе пользователем первого варианта

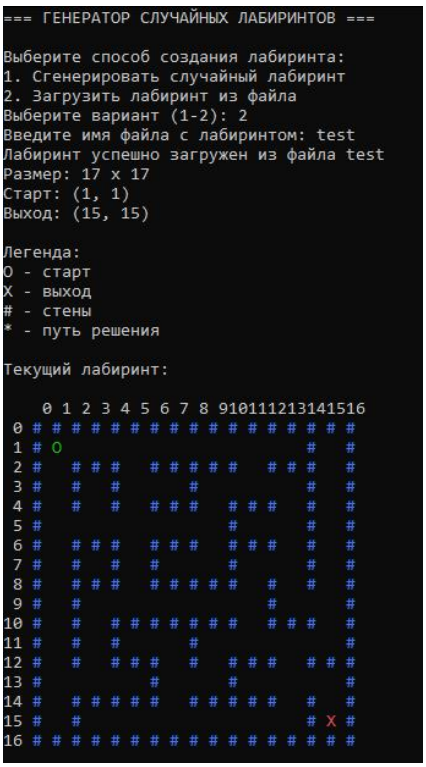

Рисунок 2 – Вывод лабиринта из файла

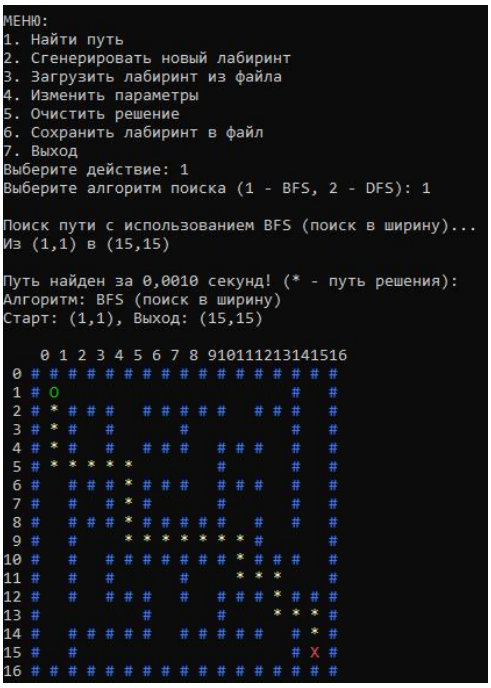

Рисунок 3 – вывод на экран меню, а также выбор первого пункта меню

5. Тестирование

Среда разработки MicrosoftVisualStudio 2019 предоставляет все средства, необходимые при разработке и отладке многомодульной программы.

Тестирование проводилось на рабочем порядке, в процессе разработки, после завершения написания программы. В ходе тестирования было выявлено и исправлено множество проблем, связанных с сохранением данных, выводом лабиринта на экран, алгоритмом программы, взаимодействием функций.

Таблица 1 – План тестирования программы

Описание теста	Ожидаемый результат
случайная генерация лабиринта	Вывод на экран сообщения о задании размерности лабиринта
Выбор пункта “найти путь”	Вывод лабиринта с найденным путём
Выбор пункта “сгенерировать новый лабиринт”	Вывод на экран сообщения о задании размерности лабиринта
Выбор пункта “загрузить лабиринт из файла”	Вывод на экран сообщения о вводе названия файла, вывод лабиринта на экран
Выбор пункта “изменить параметры”	Вывод на экран сообщения о задании размерности лабиринта, вывод лабиринта на экран

Описание теста	Ожидаемый результат
Выбор пункта “очистить решение”	Удаление всех символов пути “*”, не изменение других аспектов лабиринта
Выбор пункта “сохранить лабиринт в файл”	Вывод на экран сообщения о вводе названия файла для сохранения, появление файла в папке

Ниже продемонстрирован результат тестирования программы при взаимодействии пользователя с различными функциями программы.

Для начала тестирования запустим программу и зададим размер лабиринта 20x13 (Рисунок 4).

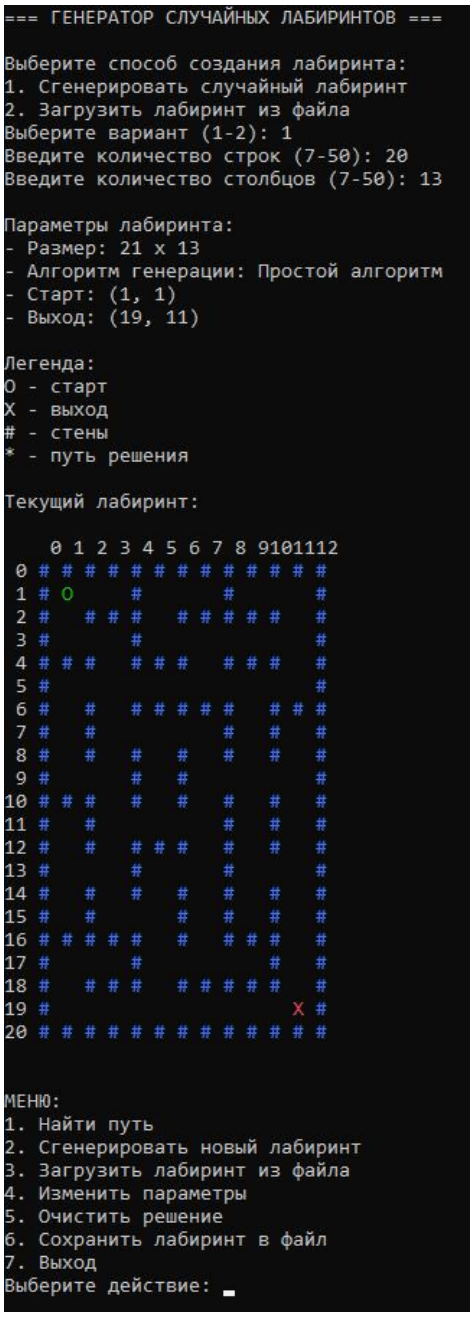

Рисунок 4 – случайная генерация лабиринта при размерности 20x13

После создания лабиринта, в консоль введем «1» для выбора алгоритма поиска пути и выберем, последовательно, алгоритмы BFS и DFS. Как мы видим, в обоих случаях программа вывела нам кратчайшие пути в лабиринте, которые она нашла за 0,002 и 0,005 секунд (Рисунок 5 и 6).

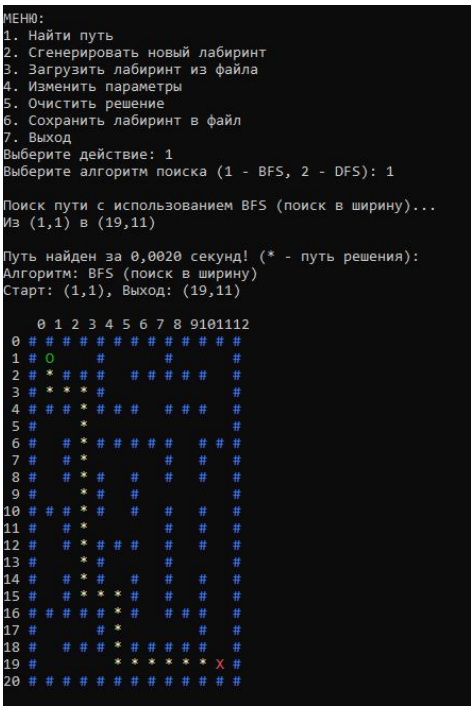

Рисунок 5 – тестирование поиска пути при выборе алгоритма bfs

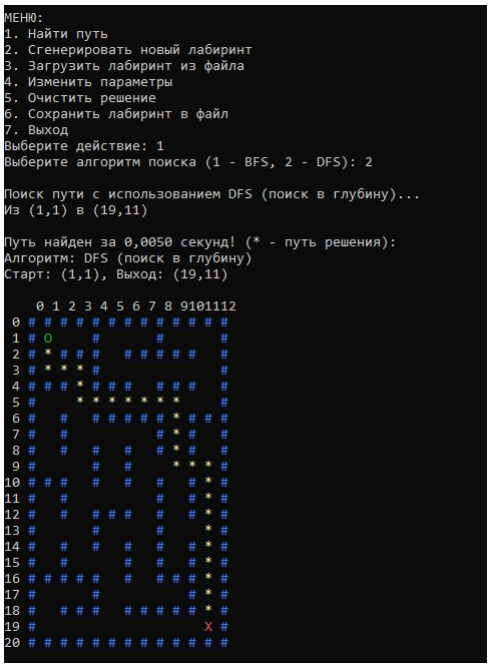

Рисунок 6 – тестирование поиска пути при выборе алгоритма dfs

После нахождения кратчайшего пути с помощью алгоритмов BFS и DFS, сгенерируем новый лабиринт с размерами 9x21, введя в консоль «2» (Рисунок 7).

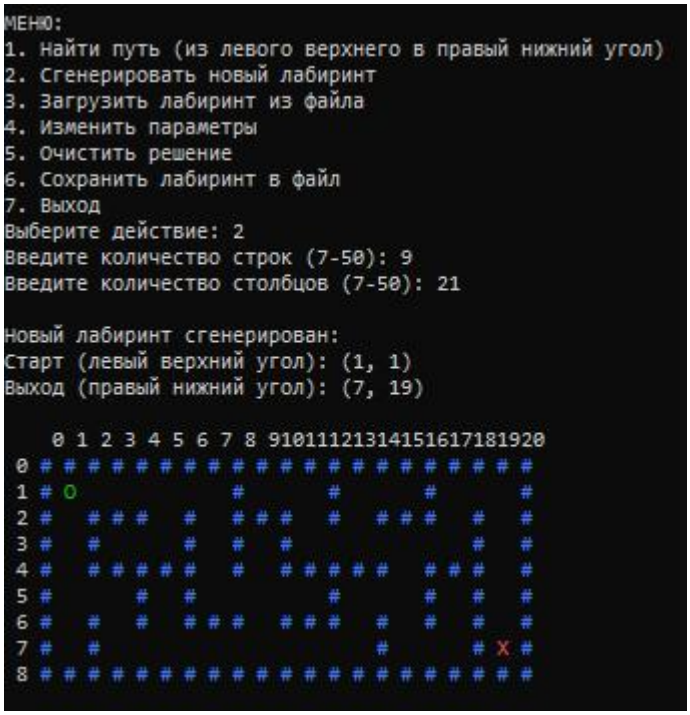

Рисунок 7 – тестирование второго пункта меню, генерация нового лабиринта, были заданы значения 9x21

Для тестирования третьего пункта меню, заранее создадим лабиринт и назовем его «test», после чего, введя в консоль «3» - загрузим его в нашу программу (Рисунок 8).

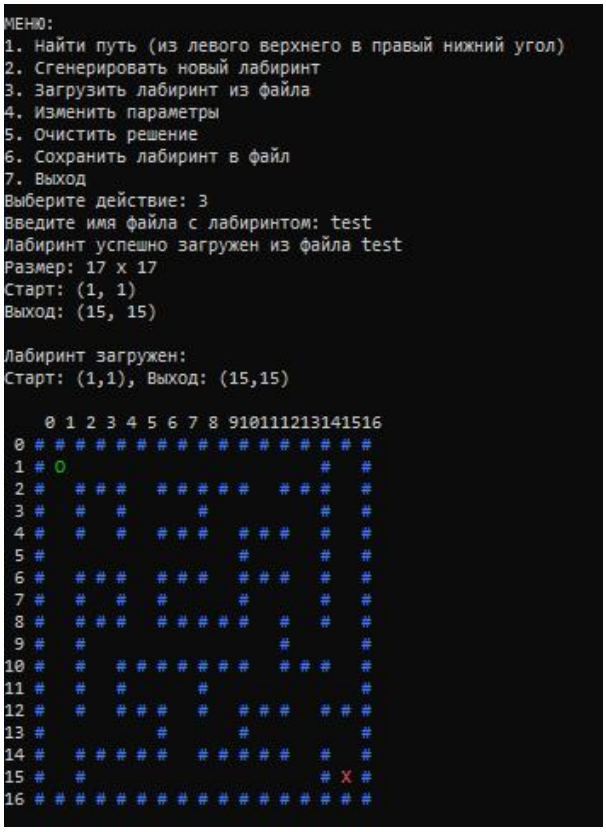

Рисунок 8 – тестирование третьего пункта меню, загрузка лабиринта из файла

Теперь проведем тестирование четвертого пункта меню «Изменить параметры», задав новые размеры лабиринту 30x30 (Рисунок 9).

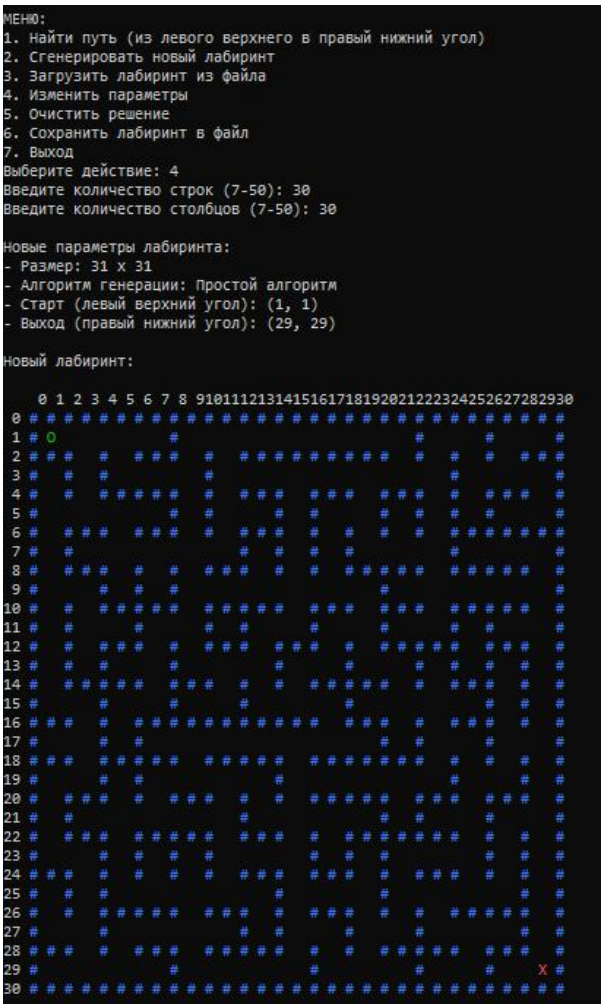

Рисунок 9 – тестирование четвёртого пункта меню, изменение параметров, была поставлена размерность 30x30

Протестируем пятый пункт «Очистить решение», путем ввода «5» в консоль. После ввода номера пункта, программа покажет лабиринт без кратчайшего пути, который вывели нам алгоритмы BFS и DFS (Рисунок 10).

Рисунок 10 – тестирование пятого пункта меню, очищение решения, для демонстрации был построен путь заранее

Теперь проверим, как программа сохраняет готовый лабиринт, выбрв шестой пункт меню и введя название файла «return» (Рисунок 11).

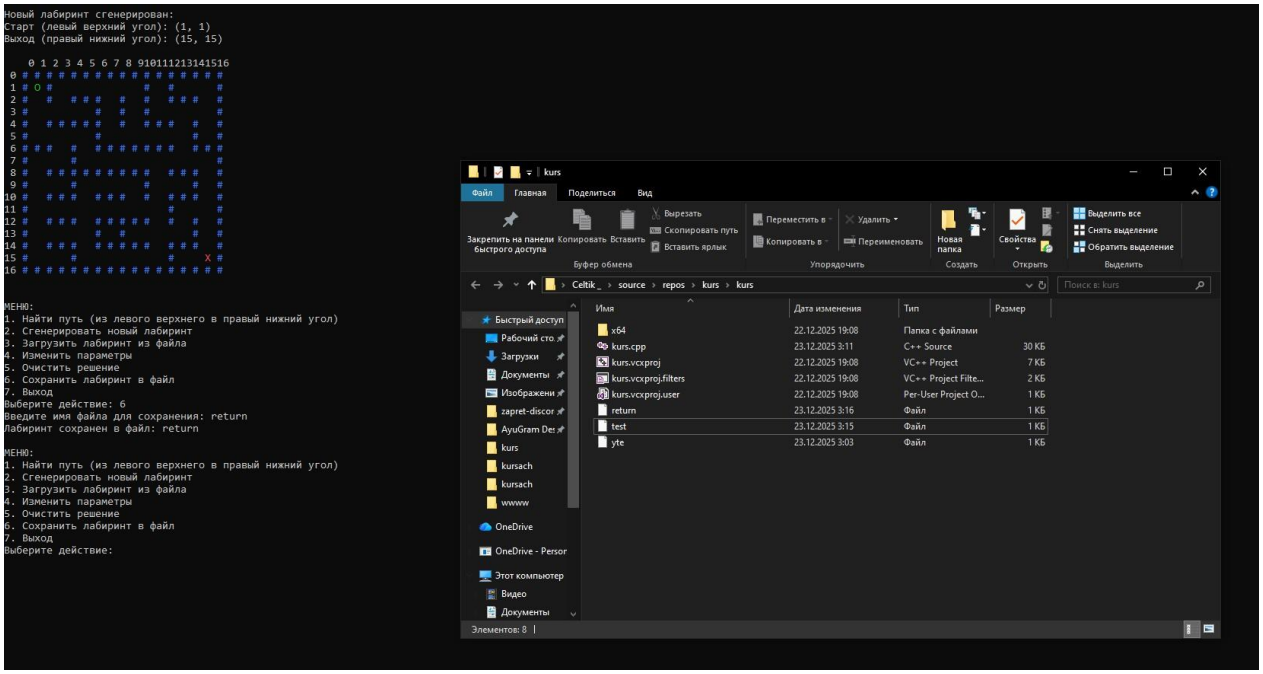

Рисунок 11 – тестирование шестого пункта меню, сохранение в файл, также на рисунке видна папка, в которую произошла запись

Таблица 2 – Описание поведения программы при тестировании

Описание теста	Полученный результат
Запуск программы	Верно
Выбор случайной генерации пользователем	Верно
Выбор загрузки лабиринта из файла	Верно
Выбор пункта “найти путь”	Верно
Выбор пункта “сгенерировать новый лабиринт”	Верно
Выбор пункта “загрузить лабиринт из файла”	Верно
Выбор пункта “изменить параметры”	Верно
Выбор пункта “очистить решение”	Верно
Выбор пункта “сохранить лабиринт в файл”	Верно

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

6. Ручной расчёт задачи

Проведём проверку работы алгоритмов поиска пути (DFS и BFS) на примере лабиринта размером 21×21. За основу была взята схема, указанная на рисунке после расчёта.

Алгоритм DFS: начинаем обход из стартовой позиции S(17,5) в соседние свободные клетки. Проверяем, если есть проход из текущей позиции в соседние, то двигаемся дальше. В нашем случае сначала идём вниз в позицию (18,5). Аналогично проверяем и в позиции (18,5). Из (18,5) идём вниз в (19,5). Из (19,5) идём влево в (19,4). Далее из (19,4) идём вверх в (18,4). Из (18,4) идём влево в (18,3), затем вверх в (17,3) и влево в (17,2). Из позиции (17,2) идём вверх в (16,2) и продолжаем движение вверх по свободному вертикальному проходу через (15,2), (14,2), (13,2), (12,2), (11,2), (10,2), (9,2), (8,2), (7,2), (6,2), (5,2) до (4,2). Из (4,2) идём вверх в (3,2).Из позиции (3,2) двигаемся вправо по горизонтальному коридору через (3,3), (3,4), (3,5), (3,6), (3,7), (3,8), (3,9), (3,10), (3,11), (3,12) до (3,13). Из (3,13) идём вниз в (4,13), затем вниз в (5,13) и вниз в (6,13). Из (6,13) идём вниз в (7,13), где находится выход X.

Таким образом, алгоритм DFS нашёл путь от старта до выхода. Полный путь состоит из 37 шагов: (17,5) → (18,5) → (19,5) → (19,4) → (18,4) → (18,3) → (17,3) → (17,2) → (16,2) → (15,2) → (14,2) → (13,2) → (12,2) → (11,2) → (10,2) → (9,2) → (8,2) → (7,2) → (6,2) → (5,2) → (4,2) → (3,2) → (3,3) → (3,4) → (3,5) → (3,6) → (3,7) → (3,8) → (3,9) → (3,10) → (3,11) → (3,12) → (3,13) → (4,13) → (5,13) → (6,13) → (7,13).

Алгоритм BFS: начинаем обход также из стартовой позиции O(17,5). BFS использует очередь и обрабатывает вершины по принципу "первым пришёл — первым вышел". Сначала посещаем все соседние доступные клетки на расстоянии 1 шага, затем на расстоянии 2 шагов, пока не достигнем выхода. Обрабатываем (17,5), добавляем в очередь соседей: (18,5).Обрабатываем (18,5), добавляем (19,5) и (18,4). Обрабатываем (19,5),добавляем (19,4). Обрабатываем (18,4), добавляем (18,3). Обрабатываем (19,4), Все соседи уже посещены или являются стенами, обрабатываем (18,3), добавляем (17,3). Обрабатываем (17,3), добавляем (17,2). Обрабатываем (17,2), добавляем (16,2).

Далее BFS продолжает распространяться вверх по левой части лабиринта, затем вправо через центральные проходы, пока не достигнет выхода X(7,13). Так как BFS на каждом шаге исследует все возможные направления одновременно, он гарантированно найдёт кратчайший путь.

Пример кратчайшего пути, найденного BFS (длина 39 шагов): (17,5) → (18,5) → (18,4) → (18,3) → (17,3) → (17,2) → (16,2) → (15,2) → (14,2) → (13,2) → (12,2) → (11,2) → (10,2) → (9,2) → (8,2) → (7,2) → (6,2) → (5,2) → (4,2) → (3,2) → (3,3) → (3,4) → (3,5) → (3,6) → (3,7) → (3,8) → (3,9) → (3,10) → (3,11) → (3,12) → (3,13) → (4,13) → (5,13) → (6,13) → (7,13).

DFS нашёл путь длиной 37 шагов, двигаясь вглубь по одному направлению до тупика, затем возвращаясь и исследуя другие ветви. BFS нашёл путь длиной 39 шагов, исследуя все соседние клетки на каждом уровне, что гарантирует кратчайший путь по количеству шагов. Из этого делаем вывод, что оба алгоритма успешно находят путь в лабиринте, но DFS может найти путь быстрее по времени выполнения в некоторых случаях,

но не гарантирует кратчайший путь; BFS всегда находит кратчайший путь, но требует больше памяти для хранения очереди.

Таким образом, можно сделать вывод, что программа работает верно.

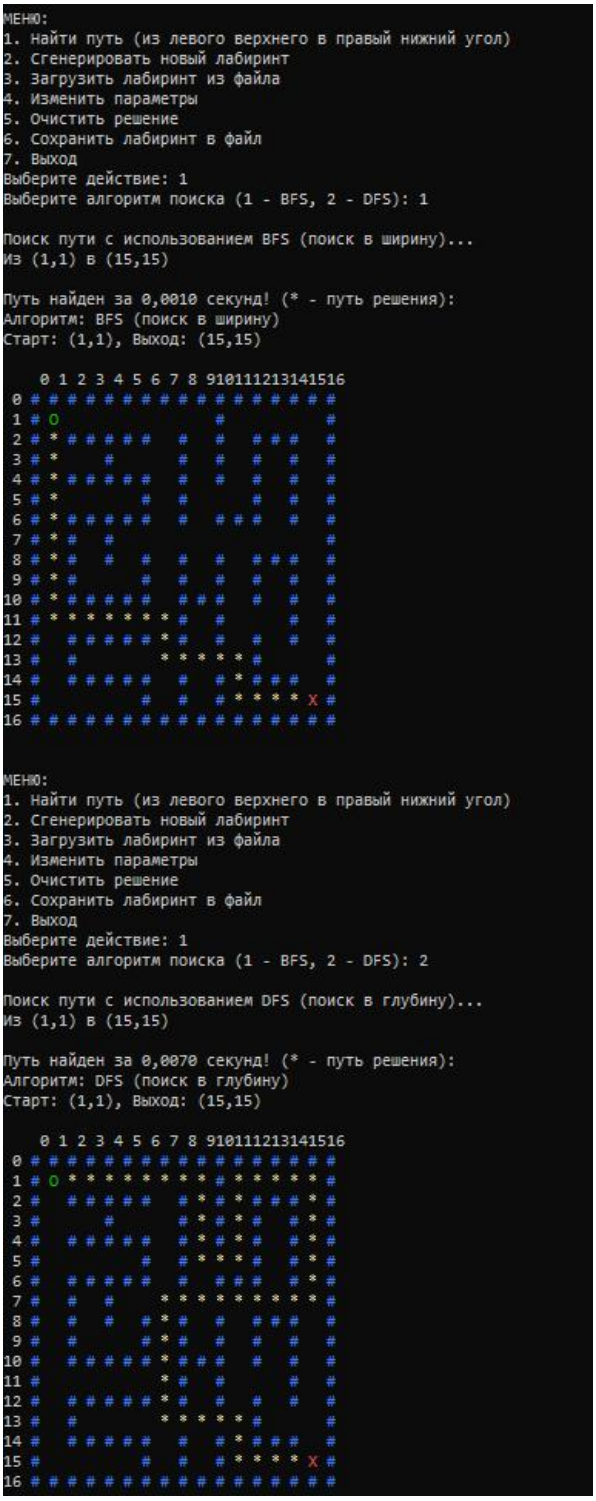

Рисунок 12 – Тестирование работы программы

Заключение

Таким образом, в процессе создания данного проекта разработана программа, реализующая алгоритм поиска в глубину для поиска компонент сильной связности орграфа в MicrosoftVisualStudio 2019.

При выполнении данной курсовой работы были получены навыки разработки программ и освоены приемы создания матриц смежностей, основанных на теории орграфов. Приобретены навыки по осуществлению алгоритма поиска в глубину. Углублены знания языка программирования Си.

Недостатком разработанной программы является примитивный пользовательский интерфейс. Потому что программа работает в консольном режиме, не добавляющем к сложности языка сложность программного оконного интерфейса.

Программа имеет небольшой, но достаточный для использования функционал возможностей.

Список литературы

1. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ. – М.: Вильямс, 2022. – 1328 с.

2. Седжвик Р., Уэйн К. Алгоритмы на C++. – М.: Вильямс, 2019. – 848 с.

3. Рассел С., Норвиг П. Искусственный интеллект: современный подход. – М.: Вильямс, 2021. – 1408 с.

4. Скиена С. С. Алгоритмы: руководство по разработке. – М.: Диалектика, 2020. – 720 с.

5. Ахо А., Хопкрофт Д., Ульман Д. Структуры данных и алгоритмы. – М.: Вильямс, 2016. – 400 с.

6. Халимов Т., Меньшиков Ф. Олимпиадное программирование. – М.: ДМК Пресс, 2022. – 320 с.

7. Макдауэлл Г. Л. Карьера программиста. – СПб.: Питер, 2022. – 720 с.

ПриложениеА.

Листинг программы.

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

#define WALL '#'
#define PATH ' '
#define START 'O'
#define END 'X'
#define SOLUTION '*'

typedef struct
    { int  row;
    int col;
} Point;

typedef struct
    { Point*
    data; int
    front; int
    rear;
    int capacity;
} Queue;

typedef   struct
    {     Point*
    data;    int
    top;
    int capacity;
} Stack;

Queue* createQueue(int capacity) {
    Queue* q = (Queue*)malloc(sizeof(Queue));
    q->data = (Point*)malloc(capacity * sizeof(Point));
    q->front = 0;
    q->rear = 0;
    q->capacity = capacity;
    return q;
}

void freeQueue(Queue* q)
    { free(q->data);
    free(q);
}

bool isEmptyQueue(Queue* q)
    { return q->front == q->rear;
```

```
}
```

```
void enqueue(Queue* q, Point p)
    { if (q->rear >= q->capacity)
    {
        q->capacity *= 2;
        q->data = (Point*)realloc(q->data, q->capacity * sizeof(Point));
    }
    q->data[q->rear] = p;
    q->rear++;
}

Point dequeue(Queue* q) {
    Point p = q->data[q->front];
    q->front++;
    if (q->front == q->rear)
        { q->front = q->rear =
        0;
    }
    return p;
}

Stack* createStack(int capacity) {
    Stack* s = (Stack*)malloc(sizeof(Stack));
    s->data = (Point*)malloc(capacity * sizeof(Point));
    s->top = -1;
    s->capacity = capacity;
    return s;
}

void freeStack(Stack* s)
    { free(s->data);
    free(s);
}

bool isEmptyStack(Stack* s)
    { return s->top == -1;
}

void push(Stack* s, Point p) {
    if (s->top >= s->capacity - 1)
        { s->capacity *= 2;
        s->data = (Point*)realloc(s->data, s->capacity * sizeof(Point));
    }
    s->top++;
    s->data[s->top] = p;
}

Point pop(Stack* s) {
    Point p = s->data[s->top];
    s->top--;
    return p;
}

char** createMaze(int rows, int cols) {
    char** maze = (char**)malloc(rows * sizeof(char*));
    for (int i = 0; i < rows; i++) {
        maze[i] = (char*)malloc(cols * sizeof(char));
    }
    return maze;
}
```

```
void freeMaze(char** maze, int rows)
    { for (int i = 0; i < rows; i++) {
```

```
        free(maze[i]);
    }
    free(maze);
}

bool loadMazeFromFile(char*** maze, int* rows, int* cols, Point* start, Point* end, const
char* filename) {
    FILE* file = fopen(filename, "r");
    if (!file) {
        printf("Ошибка: не удалось открыть файл %s\n", filename);
        return false;
    }

    if (fscanf(file, "%d %d", rows, cols) != 2)
        { printf("Ошибка: неверный формат файла!\n");
        fclose(file);
        return false;
    }

    fgetc(file);

    if (*rows < 3 || *rows > 100 || *cols < 3 || *cols > 100) {
        printf("Ошибка: недопустимые размеры лабиринта (%d x %d)!\n", *rows, *cols);
        fclose(file);
        return false;
    }

    *maze = createMaze(*rows, *cols);

    bool startFound = false;
    bool endFound = false;

    for (int i = 0; i < *rows; i++)
        { char line[256];
        if (fgets(line, sizeof(line), file) == NULL)
            { printf("Ошибка: не удалось прочитать строку %d!\n",
            i); freeMaze(*maze, *rows);
            fclose(file);
            return false;
        }

        line[strcspn(line, "\n")] = 0;

        if (strlen(line) != *cols) {
            printf("Ошибка: длина строки %d не соответствует объявленной ширине!\n", i);
            freeMaze(*maze, *rows);
            fclose(file);
            return false;
        }

        for (int j = 0; j < *cols; j++)
            { (*maze)[i][j] = line[j];

            if (line[j] == START)
                { start->row = i;
                start->col = j;
                startFound = true;
            }
            else if (line[j] == END)
                { end->row = i;
```

```
                end->col = j;
                endFound = true;
            }
            else if (line[j] != WALL && line[j] != PATH && line[j] != SOLUTION)
                { printf("Предупреждение: недопустимый символ '%c' в позиции (%d,%d)\n",
                    line[j], i, j);
                (*maze)[i][j] = PATH;
            }
        }
    }

    fclose(file);

    if (!startFound)
    {
        if ((*maze)[0][0] == PATH)
        {
            (*maze)[0][0] = START;
            start->row = 0;
            start->col = 0;
            startFound = true;
            printf("Старт утановлен в левый верхний угол (0,0)\n");
        }
        else
        {
            bool foundStart = false;
            for (int i = 0; i < *rows && !foundStart; i++) {
                for (int j = 0; j < *cols && !foundStart; j++)
                    { if ((*maze)[i][j] == PATH) {
                        (*maze)[i][j] = START;
                        start->row = i;
                        start->col = j;
                        startFound = true;
                        foundStart = true;
                        printf("Старт установлен в (%d,%d)\n", i, j);
                    }
                }
            }
        }
    }

    if (!endFound) {
        // Если в файле нет конца, ставим его в правый нижний угол
        int lastRow = *rows - 1;
        int lastCol = *cols - 1;
        if ((*maze)[lastRow][lastCol] == PATH)
            { (*maze)[lastRow][lastCol] = END;
            end->row = lastRow;
            end->col = lastCol;
            endFound = true;
            printf("Автоматически установлен выход в правый нижний угол (%d,%d)\n",
lastRow, lastCol);
        }
        else {
            // Если правый нижний угол - стена, ищем ближайшую свободную клетку
            bool foundEnd = false;
            for (int i = *rows - 1; i >= 0 && !foundEnd; i--) {
                for (int j = *cols - 1; j >= 0 && !foundEnd; j--) {
                    if ((*maze)[i][j] == PATH && (i != start->row || j != start->col))
                        { (*maze)[i][j] = END;
```

```
                        end->row = i;
                        end->col = j;
                        endFound = true;
                        foundEnd = true;
                        printf("Автоматически установлен выход в (%d,%d)\n", i, j);
                    }
                }
            }
        }
    }

    if (!startFound || !endFound) {
        printf("Ошибка: в лабиринте не найдены старт (S) и/или выход (E)!\n");
        if (!startFound) printf("- Старт (S) ненайден\n");
        if (!endFound) printf("- Выход (E) ненайден\n");
        freeMaze(*maze, *rows);
        return false;
    }

    printf("Лабиринт успешно загружен из файла %s\n", filename);
    printf("Размер: %d x %d\n", *rows, *cols);
    printf("Старт: (%d, %d)\n", start->row, start->col);
    printf("Выход: (%d, %d)\n", end->row, end->col);

    return true;
}

void generateMaze(char** maze, int rows, int cols, Point* start, Point* end)
    { for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (i == 0 || i == rows - 1 || j == 0 || j == cols - 1)
                { maze[i][j] = WALL;
            }
            else {
                maze[i][j] = PATH;
            }
        }
    }

    for (int i = 1; i < rows - 1; i++) {
        for (int j = 1; j < cols - 1; j++)
            { if (i % 2 == 0 && j % 2 == 0) {
                maze[i][j] = WALL;
            }
        }
    }

    for (int i = 2; i < rows - 1; i += 2) {
        for (int j = 2; j < cols - 1; j += 2)
            { int direction = rand() % 4;
            int wallRow = i, wallCol = j;

            switch (direction) {
            case 0: wallRow = i - 1; break;
            case 1: wallRow = i + 1; break;
            case 2: wallCol = j - 1; break;
            case 3: wallCol = j + 1; break;
            }

            if (wallRow > 0 && wallRow < rows - 1 &&
```

```
            wallCol> 0 && wallCol < cols - 1)
            { maze[wallRow][wallCol] = WALL;
        }
    }
}

// Старт всегда в левом верхнем углу (если это не стена)
start->row = 1;
start->col = 1;
if (maze[start->row][start->col] == WALL)
    { maze[start->row][start->col] = START;
}
else {
    maze[start->row][start->col] = START;
}

end->row = rows - 2;
end->col = cols - 2;
if (maze[end->row][end->col] == WALL)
    { maze[end->row][end->col] = END;
}
else {
    maze[end->row][end->col] = END;
}

// Гарантируем, что старт и конец не находятся в одной клетке
if (start->row == end->row && start->col == end->col)
    { if (end->row > 1) {
        end->row--;
        maze[end->row][end->col] = END;
    }
    else if (end->col > 1)
        { end->col--;
        maze[end->row][end->col] = END;
    }
}

int deadEndPaths = 0;
for (int i = 1; i < rows - 1; i++) {
    for (int j = 1; j < cols - 1; j++)
        { if (maze[i][j] == PATH) {
            int walls = 0;
            if (maze[i - 1][j] == WALL) walls++;
            if (maze[i + 1][j] == WALL) walls++;
            if (maze[i][j - 1] == WALL) walls++;
            if (maze[i][j + 1] == WALL) walls++;

            if (walls >= 3)
                { deadEndPaths++;
            }
        }
    }
}

if (deadEndPaths < 5) {
    for (int i = 0; i < 10; i++) {
        int r = 1 + rand() % (rows - 2);
        int c = 1 + rand() % (cols - 2);

        if (maze[r][c] == WALL) {
```

```
                int neighborPaths = 0;
                if (maze[r - 1][c] != WALL) neighborPaths++;
                if (maze[r + 1][c] != WALL) neighborPaths++;
                if (maze[r][c - 1] != WALL) neighborPaths++;
                if (maze[r][c + 1] != WALL) neighborPaths++;

                if (neighborPaths >= 2)
                    { maze[r][c] = PATH;
                }
            }
        }
    }
}

bool isValid(int row, int col, int rows, int cols, char** maze, bool** visited)
    { return (row >= 0 && row < rows && col >= 0 && col < cols &&
        maze[row][col] != WALL && !visited[row][col]);
}

void printMaze(char** maze, int rows, int cols)
    { printf("\n ");
    for (int j = 0; j < cols; j++)
        { printf("%2d", j % 100);
    }
    printf("\n");

    for (int i = 0; i < rows; i++)
        { printf("%2d ", i % 100);
        for (int j = 0; j < cols; j++)
            { if (maze[i][j] == START) {
                printf("\033[1;32m%c \033[0m", START); // Зеленый для старта
            }
            else if (maze[i][j] == END) {
                printf("\033[1;31m%c \033[0m", END); // Красный для конца
            }
            else if (maze[i][j] == SOLUTION) {
                printf("\033[1;33m%c \033[0m", SOLUTION); // Желтый для пути
            }
            else if (maze[i][j] == WALL) {
                printf("\033[1;34m%c \033[0m", WALL); // Синий для стен
            }
            else {
                printf("%c ", maze[i][j]);
            }
        }
        printf("\n");
    }
    printf("\n");
}

bool bfsFindPath(char** maze, int rows, int cols, Point start, Point end)
    { bool** visited = (bool**)malloc(rows * sizeof(bool*));
    Point** parent = (Point**)malloc(rows * sizeof(Point*));

    for (int i = 0; i < rows; i++) {
        visited[i] = (bool*)malloc(cols * sizeof(bool));
        parent[i] = (Point*)malloc(cols * sizeof(Point));
        for (int j = 0; j < cols; j++) {
            visited[i][j] = false;
            parent[i][j].row = -1;
```

```
            parent[i][j].col = -1;
        }
    }

    Queue* q = createQueue(rows * cols);

    visited[start.row][start.col] = true;
    enqueue(q, start);

    int dr[] = { -1, 1, 0, 0 };
    int dc[] = { 0, 0, -1, 1 };

    bool found = false;

    while (!isEmptyQueue(q))
        { Point current =
        dequeue(q);

        if (current.row == end.row && current.col == end.col)
            { Point p = end;
            while (p.row != start.row || p.col != start.col) {
                if (maze[p.row][p.col] != START && maze[p.row][p.col] != END)
                    { maze[p.row][p.col] = SOLUTION;
                }
                p = parent[p.row][p.col];
            }
            found = true;
            break;
        }

        for (int i = 0; i < 4; i++) {
            int newRow = current.row + dr[i];
            int newCol = current.col + dc[i];

            if (isValid(newRow, newCol, rows, cols, maze, visited))
                { visited[newRow][newCol] = true;
                parent[newRow][newCol] = current;

                Point newPoint;
                newPoint.row = newRow;
                newPoint.col = newCol;
                enqueue(q, newPoint);
            }
        }
    }

    for (int i = 0; i < rows; i++)
        { free(visited[i]);
        free(parent[i]);
    }
    free(visited);
    free(parent);
    freeQueue(q);

    return found;
}

bool dfsFindPath(char** maze, int rows, int cols, Point start, Point end)
    { bool** visited = (bool**)malloc(rows * sizeof(bool*));
    Point** parent = (Point**)malloc(rows * sizeof(Point*));
```

```
for (int i = 0; i < rows; i++) {
    visited[i] = (bool*)malloc(cols * sizeof(bool));
    parent[i] = (Point*)malloc(cols * sizeof(Point));
    for (int j = 0; j < cols; j++) {
        visited[i][j] = false;
        parent[i][j].row = -1;
        parent[i][j].col = -1;
    }
}

Stack* s = createStack(rows * cols);

visited[start.row][start.col] = true;
push(s, start);

int dr[] = { -1, 1, 0, 0 };
int dc[] = { 0, 0, -1, 1 };

bool found = false;

while (!isEmptyStack(s))
    { Point current = pop(s);

    if (current.row == end.row && current.col == end.col)
        { Point p = end;
        while (p.row != start.row || p.col != start.col) {
            if (maze[p.row][p.col] != START && maze[p.row][p.col] != END)
                { maze[p.row][p.col] = SOLUTION;
            }
            p = parent[p.row][p.col];
        }
        found = true;
        break;
    }

    for (int i = 0; i < 4; i++) {
        int newRow = current.row + dr[i];
        int newCol = current.col + dc[i];

        if (isValid(newRow, newCol, rows, cols, maze, visited))
            { visited[newRow][newCol] = true;
            parent[newRow][newCol] = current;

            Point newPoint;
            newPoint.row = newRow;
            newPoint.col = newCol;
            push(s, newPoint);
        }
    }
}

for (int i = 0; i < rows; i++)
    { free(visited[i]);
    free(parent[i]);
}
free(visited);
free(parent);
freeStack(s);

return found;
```

```
}

void clearSolution(char** maze, int rows, int cols, Point start, Point end)
    { for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++)
            { if (maze[i][j] == SOLUTION)
            {
                maze[i][j] = PATH;
            }
        }
    }
    maze[start.row][start.col] = START;
    maze[end.row][end.col] = END;
}

int getIntInput(const char* prompt, int min, int max)
    { int value;
    char input[100];
    int success = 0;

    while (!success)
        { printf("%s", prompt);

        if (fgets(input, sizeof(input), stdin) == NULL) {
            printf("Ошибка ввода!\n");
            continue;
        }

        if (input[0] == '\n') {
            printf("Ошибка: введите число!\n");
            continue;
        }

        char* endptr;
        value = strtol(input, &endptr, 10);

        if (*endptr != '\n' && *endptr != '\0') {
            while (*endptr != '\0' && isspace(*endptr))
                { endptr++;
            }
            if (*endptr != '\0') {
                printf("Ошибка: введите целое число!\n");
                continue;
            }
        }

        if (value < min || value > max) {
            printf("Значение должно быть от %d до %d.\n", min, max);
            continue;
        }

        success = 1;
    }

    return value;
}

int getMenuChoice()
    { int choice;
    char input[100];
```

```
int success = 0;
```

```
    while (!success) {
        printf("Выберите действие: ");

        if (fgets(input, sizeof(input), stdin) == NULL) {
            printf("Ошибка ввода!\n");
            continue;
        }

        if (input[0] == '\n') {
            printf("Ошибка: введите число!\n");
            continue;
        }

        char* endptr;
        choice = strtol(input, &endptr, 10);

        if (*endptr != '\n' && *endptr != '\0') {
            while (*endptr != '\0' && isspace(*endptr))
                { endptr++;
            }
            if (*endptr != '\0') {
                printf("Ошибка: введите целое число!\n");
                continue;
            }
        }

        success = 1;
    }

    return choice;
}

int getAlgorithmChoice()
    { int choice;
    char input[100];
    int success = 0;

    while (!success) {
        printf("Выберите алгоритм поиска (1 - BFS, 2 - DFS): ");

        if (fgets(input, sizeof(input), stdin) == NULL) {
            printf("Ошибка ввода!\n");
            continue;
        }

        if (input[0] == '\n') {
            printf("Ошибка: введите число!\n");
            continue;
        }

        char* endptr;
        choice = strtol(input, &endptr, 10);

        if (*endptr != '\n' && *endptr != '\0') {
            while (*endptr != '\0' && isspace(*endptr))
                { endptr++;
            }
            if (*endptr != '\0') {
                printf("Ошибка: введите целое число!\n");
```

```
                continue;
            }
        }

        if (choice < 1 || choice > 2)
            { printf("Выберите 1 или 2!\n");
            continue;
        }

        success = 1;
    }

    return choice;
}

void saveMazeToFile(char** maze, int rows, int cols, const char* filename)
    { FILE* file = fopen(filename, "w");
    if (!file) {
        printf("Ошибка при сохранении файла!\n");
        return;
    }

    fprintf(file, "%d %d\n", rows, cols);
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++)
            { fprintf(file, "%c", maze[i][j]);
        }
        fprintf(file, "\n");
    }
    fclose(file);
    printf("Лабиринт сохранен в файл: %s\n", filename);
}

bool getFilename(char* filename, int max_len, const char* prompt)
    { printf("%s", prompt);

    if (fgets(filename, max_len, stdin) == NULL) {
        printf("Ошибка ввода!\n");
        return false;
    }

    filename[strcspn(filename, "\n")] = 0;

    if (strlen(filename) == 0) {
        printf("Ошибка: имя файла не может быть пустым!\n");
        return false;
    }

    return true;
}

int main() {
    setlocale(LC_ALL, "");
    srand(time(NULL));

    int rows = 0, cols = 0;
    char** maze = NULL;
    Point start, end;
    bool mazeLoaded = false;
```

```
    printf("=== ГЕНЕРАТОР СЛУЧАЙНЫХ ЛАБИРИНТОВ ===\n\n");

    printf("Выберите способ создания лабиринта:\n");
    printf("1. Сгенерировать случайный лабиринт\n");
    printf("2. Загрузить лабиринт из файла\n");

    int initialChoice = getIntInput("Выберите вариант (1-2): ", 1, 2);

    if (initialChoice == 1) {
        rows = getIntInput("Введите количество строк (7-50): ", 7, 50);
        if (rows % 2 == 0) rows++;

        cols = getIntInput("Введите количество столбцов (7-50): ", 7, 50);
        if (cols % 2 == 0) cols++;

        maze = createMaze(rows, cols);
        generateMaze(maze, rows, cols, &start, &end);
        mazeLoaded = true;

        printf("\nПараметры лабиринта:\n");
        printf("- Размер: %d x %d\n", rows, cols);
        printf("- Алгоритм генерации: Простой алгоритм\n");
        printf("- Старт: (%d, %d)\n", start.row, start.col);
        printf("- Выход: (%d, %d)\n", end.row, end.col);
    }
    else {
        char filename[256];
        while (!mazeLoaded) {
            if (!getFilename(filename, sizeof(filename), "Введите имя файла с лабиринтом:
")) {
                continue;
            }

            mazeLoaded = loadMazeFromFile(&maze, &rows, &cols, &start, &end, filename);
        }
    }

    printf("\nЛегенда:\n");
    printf("O - старт\n");
    printf("X - выход\n");
    printf("# - стены\n");
    printf("* - путь решения\n");
    printf("\nТекущий лабиринт:\n");
    printMaze(maze, rows, cols);

    int choice;
    do {
        printf("\nМЕНЮ:\n");
        printf("1. Найти путь\n");
        printf("2. Сгенерировать новый лабиринт\n");
        printf("3. Загрузить лабиринт из файла\n");
        printf("4. Изменить параметры\n"); printf("5.
        Очистить решение\n");
        printf("6. Сохранить лабиринт в файл\n");
        printf("7. Выход\n");

        choice = getMenuChoice();

        switch (choice) {
        case 1: {
```

```
            if (!mazeLoaded) {
                printf("Сначала загрузите или сгенерируйте лабиринт!\n");
                break;
            }

            int algorithm = getAlgorithmChoice();
            clearSolution(maze, rows, cols, start, end);

            bool found;
            clock_t start_time = clock();

            if (algorithm == 1)
            {
                printf("\nПоиск пути с использованием BFS (поиск в ширину)...\n");
                printf("Из (%d,%d) в (%d,%d)\n", start.row, start.col, end.row, end.col);
                found = bfsFindPath(maze, rows, cols, start, end);
            }
            else {
                printf("\nПоиск пути с использованием DFS (поиск в глубину)...\n");
                printf("Из (%d,%d) в (%d,%d)\n", start.row, start.col, end.row, end.col);
                found = dfsFindPath(maze, rows, cols, start, end);
            }

            clock_t end_time = clock();
            double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;

            if (found) {
                printf("\nПуть найден за %.4f секунд! (* - путь решения):\n",
time_taken);
                printf("Алгоритм: %s\n", algorithm == 1 ? "BFS (поиск в ширину)" : "DFS
(поиск в глубину)");
                printf("Старт: (%d,%d), Выход: (%d,%d)\n", start.row, start.col, end.row,
end.col);
                printMaze(maze, rows, cols);
            }
            else {
                printf("\nПуть не найден за %.4f секунд!\n", time_taken);
                printf("Старт: (%d,%d), Выход: (%d,%d)\n", start.row, start.col, end.row,
end.col);
            }
            break;
        }

        case 2:
            if (maze) freeMaze(maze, rows);

            rows = getIntInput("Введите количество строк (7-50): ", 7, 50);
            if (rows % 2 == 0) rows++;

            cols = getIntInput("Введите количество столбцов (7-50): ", 7, 50);
            if (cols % 2 == 0) cols++;

            maze = createMaze(rows, cols);
            generateMaze(maze, rows, cols, &start, &end);
            mazeLoaded = true;

            printf("\nНовый лабиринт сгенерирован:\n");
            printf("Старт: (%d, %d)\n", start.row, start.col);
            printf("Выход: (%d, %d)\n", end.row, end.col);
            printMaze(maze, rows, cols);
```

```
            break;

        case 3:
            if (maze) freeMaze(maze, rows);

            char filename[256];
            if (!getFilename(filename, sizeof(filename), "Введите имя файла с лабиринтом:
")) {
                mazeLoaded = false;
                break;
            }

            mazeLoaded = loadMazeFromFile(&maze, &rows, &cols, &start, &end, filename);
            if (mazeLoaded) {
                printf("\nЛабиринт загружен:\n");
                printf("Старт: (%d,%d), Выход: (%d,%d)\n", start.row, start.col, end.row,
end.col);
                printMaze(maze, rows, cols);
            }
            else {
                printf("\nНе удалось загрузить лабиринт.\n");
            }
            break;

        case 4:
            if (!mazeLoaded) {
                printf("Сначала загрузите или сгенерируйте лабиринт!\n");
                break;
            }

            if (initialChoice == 2) {
                 printf("Изменение параметров доступно только для сгенерированных
лабиринтов!\n");
                printf("Для изменения загруженного лабиринта используйте пункт меню
3.\n");
                break;
            }

            if (maze) freeMaze(maze, rows);

            rows = getIntInput("Введите количество строк (7-50): ", 7, 50);
            if (rows % 2 == 0) rows++;

            cols = getIntInput("Введите количество столбцов (7-50): ", 7, 50);
            if (cols % 2 == 0) cols++;

            maze = createMaze(rows, cols);
            generateMaze(maze, rows, cols, &start, &end);

            printf("\nНовые параметры лабиринта:\n");
            printf("- Размер: %d x %d\n", rows, cols);
            printf("- Алгоритм генерации: Простой алгоритм\n");
            printf("- Старт: (%d, %d)\n", start.row, start.col);
            printf("- Выход: (%d, %d)\n", end.row, end.col);
            printf("\nНовый лабиринт:\n");
            printMaze(maze, rows, cols);
            break;

        case 5:
            if (!mazeLoaded) {
```

```
                printf("Сначала загрузите или сгенерируйте лабиринт!\n");
                break;
            }

            clearSolution(maze, rows, cols, start, end);
            printf("\nРешение очищено:\n");
            printf("Старт: (%d,%d), Выход: (%d,%d)\n", start.row, start.col, end.row,
end.col);
            printMaze(maze, rows, cols);
            break;

        case 6:
            if (!mazeLoaded) {
                printf("Сначала загрузите или сгенерируйте лабиринт!\n");
                break;
            }

            char saveFilename[256];
            if (getFilename(saveFilename, sizeof(saveFilename), "Введите имя файла для
сохранения: ")) {
                saveMazeToFile(maze, rows, cols, saveFilename);
            }
            break;

        case 7:
            printf("Выход из программы.\n");
            break;

        default:
            printf("Неверный выбор. Попробуйте снова.\n");
        }
    } while (choice != 7);

    if (mazeLoaded)
        { freeMaze(maze, rows);
    }

    return 0;
}
```