|  |  |
| --- | --- |
|  | |
| МИНОБРНАУКИ РОССИИ | |
| Федеральное государственное бюджетное образовательное учреждение  высшего профессионального образования  **"Московский государственный информационных технологий,**  **радиотехники и электроники"**  **МИРЭА** | |
| Институт информационных технологий (ИТ) | |
| Кафедра информатики и информационных систем (ИИС) | |
| **КУРСОВАЯ РАБОТА** | |
| **по дисциплине** | |
| **«Объектно-ориентированное программирование»** | |
| **Тема курсовой работы: «Разработка программного обеспечения, реализующее создание и прохождение лабиринта при помощи алгоритма Рекурсивного обхода».** | |
| Студент группы ИСБОп-01-13 | *Беляева О.И.* |
| Руководитель курсовой работы | *Хлебников А. А.* |
| Рецензент | *Хлебников А. А.* |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работа представлена к защите | «\_\_»\_\_\_\_\_\_\_2015 г. |  |
|  |  |  |
| «Допущен к защите» | «\_\_»\_\_\_\_\_\_\_2015 г. |  |

Москва 2015

*Руководитель и рецензент:* Хлебников Андрей Александрович

*Студент:* Беляева Ольга Ивановна

*Тема:* «Разработка программного обеспечения, реализующее создание и прохождение лабиринта при помощи алгоритма Рекурсивного обхода»

Основной задачей данной курсовой работы является исследование вопросов проектирования и разработки программного обеспечения на языке программирования Java, изучение документации для работы с открытыми программными интерфейсами, написание программного кода, тестирование продукта.

Было создано соответствующее теме работы программное обеспечение и разработана документация работы, включающая в себя: утвержденное задание на курсовую работу; расчетно-пояснительную записку (РПЗ); рецензия (первичная) руководителя работы с итоговой отметкой о допуске работы к защите.

The main objective of this course work is to study the design and software development in the programming language Java, the study of the documentation for open APIs, writing code, testing the product.

It creates the appropriate subject of the work-developed software and documentation work, including approved task for course work; settlement and explanatory note (RPGs); Reviews (primary) of the head with a final mark for admission to the work of the defense.

**Ключевые слова:** Java, JDK, IntelliJ IDEA, Git, Backtracking.

Координаты автора:

МИРЭА: 119454, Москва, пр-т Вернадского, д. 78

Кафедра Информатики и Информационных систем (ИИС)

**Реферат**

Данная курсовая работа содержит 20 страниц, 3 иллюстрации и 6 использованных источников.

Целью данной курсовой работы является разработка программного обеспечения на языке программирования Java.

Основными задачами явились: изучение документации Java, изучение алгоритмов создания и прохождения лабиринта, сторонних библиотек, изучение среды разработки, создание программных модулей и классов, написание программного кода, реализующего необходимый функционал.

The purpose of this course is to develop software in the programming language Java.

The main objectives were to study the documentation Java, the study of algorithms creation and passage of the maze, third-party libraries, studying the development environment, the creation of program modules and classes, writing code that implements the required functionality.

**Оглавление**

[Введение 4](#_Toc438543099)

[1.1. Перебор с возвратами 5](#_Toc438543100)

[1.2. Варианты задач для перебора с возвратами 6](#_Toc438543101)

[1.3. Общая схема рекурсивного перебора 6](#_Toc438543102)

[1.4. Общая схема перебора с возвратами 10](#_Toc438543103)

[1.5. Построение кратчайшего пути 12](#_Toc438543104)

[Заключение 18](#_Toc438543107)

[Список использованной литературы 18](#_Toc438543108)

**Введение**

Целью представленной работы является разработка приложения “Поиск кратчайшего пути в лабиринте при помощи алгоритма рекурсивного обхода”, которое создает лабиринт и находит кратчайший путь его прохождения и отображает его.

Курсовая работы выполнена на языке программирования Java (JDK 1.8), с применением целого ряда сторонних библиотек и модулей. В качестве среды разработки использован программный продукт IntelliJ IDEA. Для контроля версий файлов применяется система контроля версий Git.

В результате выполнения курсовой работы ожидается получение готовой к работе программы, соответствующей теме работы.

* 1. **Перебор с возвратами**

Метод перебора с возвратами позволяет решать практически бесчисленное множество задач, для многих из которых не известны другие алгоритмы. Несмотря на такое большое многообразие переборных задач, в основе их решения есть нечто общее, позволяющее применить данный метод. Таким образом перебор можно считать практически универсальным методом переборных решения задач.

Для изложения дальнейшего материала введем несколько простых определений:

* **Ход** - отдельная часть решения, мини-задача, из решения которых складывается решение полной задачи.
* **Вариант (хода)** - возможный ход.
* Подпрограмма, которая перебирает всевозможные варианты ходов для поиска решения, складывающегося из этих ходов, называется подпрограммой **перебора с возвратами (backtracking)**.

Эти определения носят несколько размытый характер, так как фактически не существует устоявшейся терминологии, к тому же, переборные задачи очень разнообразны, так что для одной задачи такая терминология вполне уместна, для другой можно будет найти и лучше. Определение переборной подпрограммы следует понимать следующим образом: для всех первых ходов, будут перебраны всевозможные вторые ходы, затем для вторых ходов будут перебраны всевозможные третьи ходы и так далее, пока не будут перебраны все варианты ходов. Таким образом, ищется первый подходящий вариант, затем для этого варианта хода второй и так далее, пока для очередного хода не будут перебраны все варианты. Можно сравнить это с построением некой цепочки ходов. Поскольку для каждого хода есть несколько вариантов следующих ходов, образуется своеобразное дерево рекурсивных вызовов перебора с возвратами. Отсюда следует и оценка времени выполнения переборных программ - KN, где K - количество вариантов следующих ходов для данного хода, n - количество ходов, то есть время выполнения программы растет экспоненциально от количества различных вариантов для данного варианта. Это значит, что чисто переборные решения не эффективны, и их стоит применять только, если не было найдено подходящего полиномиального алгоритма, время работы которого пропорционально Nk, где k - некоторая константа.

* 1. **Варианты задач для перебора с возвратами**

Все задачи, для решения которых можно применить перебор с возвратами, можно условно разделить на несколько больших групп:

* Поиск одного решения
* Поиск всех решений
* Поиск оптимального решения

Для решения некоторых задач требуется найти одно решения, для этого можно просто перебрать все варианты и решением будет первый подходящий вариант. В задачах поиска всех решений дело обстоит несколько иначе: необходимо перебрать все варианты и выбрать все подходящие. Третий класс задач похож на предыдущий: нам также надо будет найти все решения, и затем выбрать из них оптимальное. Обычно выбор оптимального решения происходит в процессе самого перебора.

* 1. **Общая схема рекурсивного перебора**

Есть такие комбинаторные задачи, для решения которых приходится организовывать полный перебор возможных вариантов. Перебор с возвратом (backtrack) - это общий метод упорядоченного перебора. Перебор с возвратом особенно удобен для решения задач, требующих проверки потенциально большого, но конечного числа решений. В настоящем разделе описывается общая схема рекурсивного перебора с возвратом.

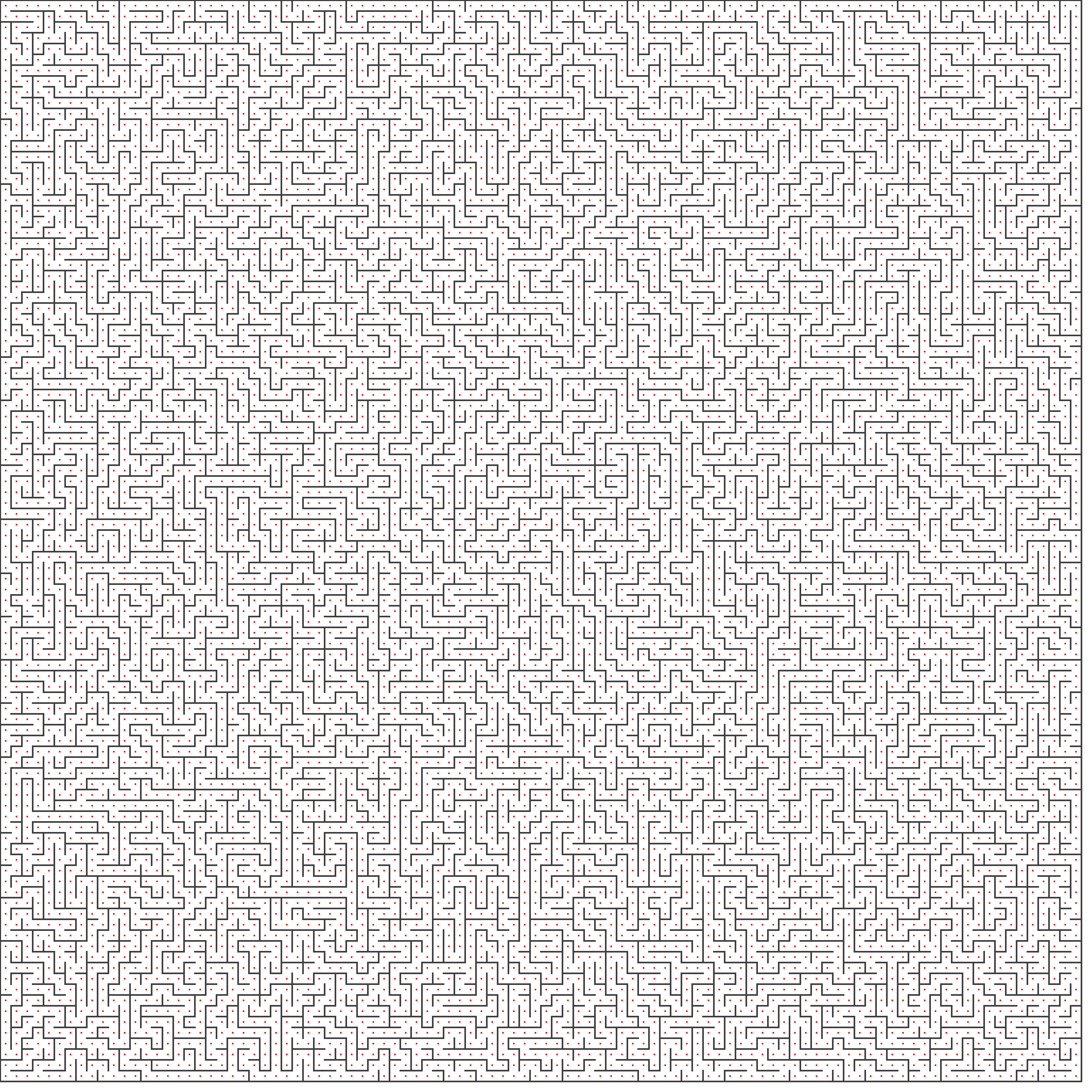
В самом общем случае мы полагаем, что решение задачи состоит из вектора конечной длины, удовлетворяющего заданным условиям и ограничениям, или из множества таких векторов. Для каждого элемента задано упорядоченное множество его возможных значений. В одних задачах размерность решения (число) может быть известна заранее, а в других – заранее не определена.

Метод перебора с возвратом основан на том, что при поиске решения многократно делается попытка расширить текущее частичное решение, то есть его продолжить. Если расширение невозможно, то происходит возврат к предыдущему более короткому частичному решению, и делается попытка его расширить другим возможным способом.

В качестве исходного частичного решения мы выбираем пустой вектор. На основе заданных ограничений выясняем, какие элементы из являются кандидатами в; обозначим это подмножество через. В качестве выбираем первый элемент из и получаем частичное решение. В общем случае по частичному решению на основе ограничений задачи строится, из которого выбираются кандидаты для расширения частичного решения до. Если, то есть частичное решение не может быть расширено, мы возвращаемся и выбираем новый элемент. Если новый элемент выбрать нельзя, мы возвращаемся еще дальше и выбираем новый элемент и т.д.

Проиллюстрируем идею перебора с возвратом с помощью задачи прохода через лабиринт. Пример лабиринта приведен на рис. 1.

Рис. 1 Построение лабиринта.



Наш лабиринт представляет собой связную конечную область на плоскости, состоящую из клеток. Наша цель – попасть из некоторой заданной клетки в другую заданную клетку, последовательно перемещаясь по клеткам. В каждый момент времени можно переместиться в одну из соседних клеток по горизонтали или вертикали. Трудность задачи состоит в существовании преград, запрещающих некоторые перемещения.

Мы можем искать путь в лабиринте, руководствуясь двумя правилами:

В каждой клетке выбирать еще не исследованный путь.

Если из исследуемой в данный момент клетки не ведут неисследованные пути, то нужно вернуться на одну клетку назад по последнему пройденному пути, по которому мы пришли в клетку.

Первое правило говорит о том, как расширить исследуемый путь, если это возможно, второе правило – как выходить из тупика. В этом и состоит сущность перебора с возвратом: продолжать расширение исследуемого решения до тех пор, пока это возможно, и когда решение нельзя расширить, возвращаться по нему и пытаться сделать другой выбор на самом близком шаге, где имеется такая возможность.

Рассмотрим подробнее, как можно решить задачу прохода через лабиринт, изображенный на рис. 1, применяя перебор с возвратом.

Обозначим через множество направлений возможных перемещений в клетке: - север, - юг, - запад, - восток.

Под решением будем понимать вектор, представляющий собой последовательность элементов из множества. Каждая координата вектора определяет направление перемещения для соответствующей текущей клетки; очевидно, первая координата определяет направление движения в начальной клетке лабиринта, а последняя – направление движения на заключительном шаге, приводящем в конечную клетку. Заметим, что размерность вектора не определена заранее, т.к. мы не знаем, сколько шагов надо сделать, чтобы пройти лабиринт.

На i-м шаге будем определять множество возможных перемещений для текущей клетки следующим образом: включим элемент в том и только том случае, если соседняя клетка на юге не отделена стенкой и не была использована ранее при предыдущих посещениях клетки. Например, так как клетка, расположенная на востоке от начальной клетки, не отделена стенкой, и мы ее не посещали ранее. Других возможных перемещений для начальной клетки нет. Договоримся, что, если множество содержит более одного элемента, сохраним в нем тот же порядок элементов, что и в множестве.

Процесс получения решения для нашей задачи можно изобразить графически, построив дерево поиска с возвратом. Построение дерева начинается с начальной вершины (корня дерева), которой соответствует пустое частичное решение. Ребра дерева помечаются элементами множества и соответствуют выбору расширения частичного решения. В дереве поиска отмечена вершина, соответствующая решению. Само решение складывается из последовательности символов, приписанных ребрам пути при движении от корня к отмеченной вершине.

Мы привели пример задачи, в которой требуется найти любое решение, если оно есть. В этом случае алгоритм останавливается, как только найдет первое решение. Если бы надо было перечислить все решения, мы продолжили бы поиск и перебрали бы все возможные пути в лабиринте. В этом случае перебор закончился бы в начальной вершине дерева поиска, при этом все варианты продолжения для корня дерева были использованы для поиска. Может оказаться, что решения в задаче нет, т.е. нельзя попасть из начальной клетки в конечную. Для этого случая также пришлось бы исследовать все возможные варианты путей в лабиринте.

На рис. 2 приведен общий вид дерева при использовании метода перебора с возвратом.

Процесс решения задачи перебором с возвратом подразделяется на отдельные подзадачи, которые удобно описывать с использованием рекурсии.

При использовании рекурсии отпадает необходимость непосредственно организовывать возвраты и отслеживать правильность их выполнения. Они становятся встроенной частью механизма выполнения рекурсивных вызовов.

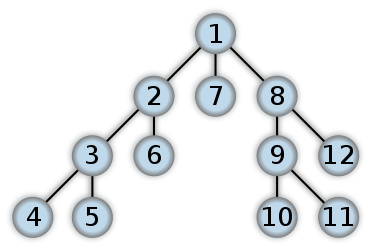


Рис 2. Дерево, поиск в глубину.

* 1. **Общая схема перебора с возвратами**

Создание **общей схемы перебора с возвратами** позволит при решении задач сосредоточиться именно на самой задаче и на ее особенностях, а не на написании переборной логике, что и будет представлять из себя схема перебора. Для начала приведу переборную схему разработаннную Никлаусом Виртом:

|  |
| --- |
| { поиск одного решения }  procedure backtracking(k: integer); { k - номер хода }  begin  { инициализация выбора варианта }  repeat  { выбор варинта }  if { вариант подходит } then  begin  { запись варианта }  if { решение не найдено } then  begin  backtracking(k+1); { рекурсивный вызов }  if { решение не найдено } then  { стирание варианта }  end  else  { вывод решения }  end;  until { вариантов нет } or { решение найдено }  end;  begin  { запись первого варианта }  backtracking(1);  end. |

Давайте ее обсудим: какие она имеет достоинства и недостатки. К достоинствам следует отнести общность, простоту и логичность. Но она имеет и недостатки. Во-первых, надо самому делать запись первого варианта, неплохо бы, чтобы это делала сама процедура. Также в ней использует цикл repeat: можно допустить ошибку в формировании уловия выхода из цикла, особенно, если не знаешь законы де Моргана, к тому же иногда проще использовать цикл for, а если вариантов мало, можно обойтись вообще без циклов. Попытаемся устранить выше приведенные недостатки. Для разработки общей схемы перебора с возвратами воспользуемся процедурой из задачи о лабиринте, просто следует ее обобщить:

|  |
| --- |
| { поиск одного решения }  procedure backtracking(k: integer); { k - номер хода }  begin  { запись варианта }  if { решение найдено } then  { вывод решения }  else  { перебор всех вариантов }  if { вариант подходит } then  backtracking(k+1); { рекурсивный вызов }  { стирание варианта }  end;  begin  backtracking(1);  end. |

Разумеется, данная схема также не идеальная, но она устраняет указанные выше недостатки схемы Вирта. Также можно соответственно сделать схемы для других классов переборных задач. Сначала схема для поиска всех решений:

|  |
| --- |
| { поиск всех решений }  procedure backtracking(k: integer); { k - номер хода }  begin  { запись варианта }  if { решение найдено } then  { запись решения }  else  { перебор всех вариантов }  if { вариант подходит } then  backtracking(k+1); { рекурсивный вызов }  { стирание варианта }  end; |

Вместо записи решения его можно выводить в выходной файл, либо обрабатывать иным образом в зависимости от условия задачи. Эту схему можно изменить, что находились не все решения, а только одно оптимальное. Под оптимальностью решения обычно понимают, что для данного решения некоторая функция принимает либо максимальное, либо минимальное значение.

* 1. **Построение кратчайшего пути**

Каким образом должен работать алгоритм, который найдет нам выход из лабиринта? Алгоритм должен каким-то образом ходить по лабиринту, проходить по всем клеткам, которые доступны из заданной, и как только текущая клетка окажется выходом, нужно будет напечатать матрицу лабиринта. Следует отметить, что при построении пути не следует бывать в одной и той же клетки более одного раза, так как таких путей с циклами существует бесконечное множество.

Алгоритм изначально рассматривает первую клетку лабиринта (верхнюю левую), и у всех клеток, из которых можно попасть в эту первую клетку мы устанавливаем первую клетку как цель, таким образом, мы знаем, для того чтобы попасть в итоге в первую клетку нам нужно ид-ти туда-то, для этих первых соседей клетки путь к первой уже найден, далее рассматриваем соседей наших соседей, указываем им необходимые цели, и таким образом у нас для всех клеток находится путь прохождения к первой клетке, на перву клетку А ссылаются клетки Б и В, значит если мы находимся в Б и В, мы знаем, что надо идти в А чтобы попасть в А. на Б ссылается Г, Д, Е, тогда, когда мы находимся Г, Д, или Е, чтобы прийти в А нам надо идти в Б, и так далее, пока не заполним весь массив. А в конце пробегаемся по цепочке ссылок из последней клетки лабиринта. Проход лабиринта представлен на рисунке 3.

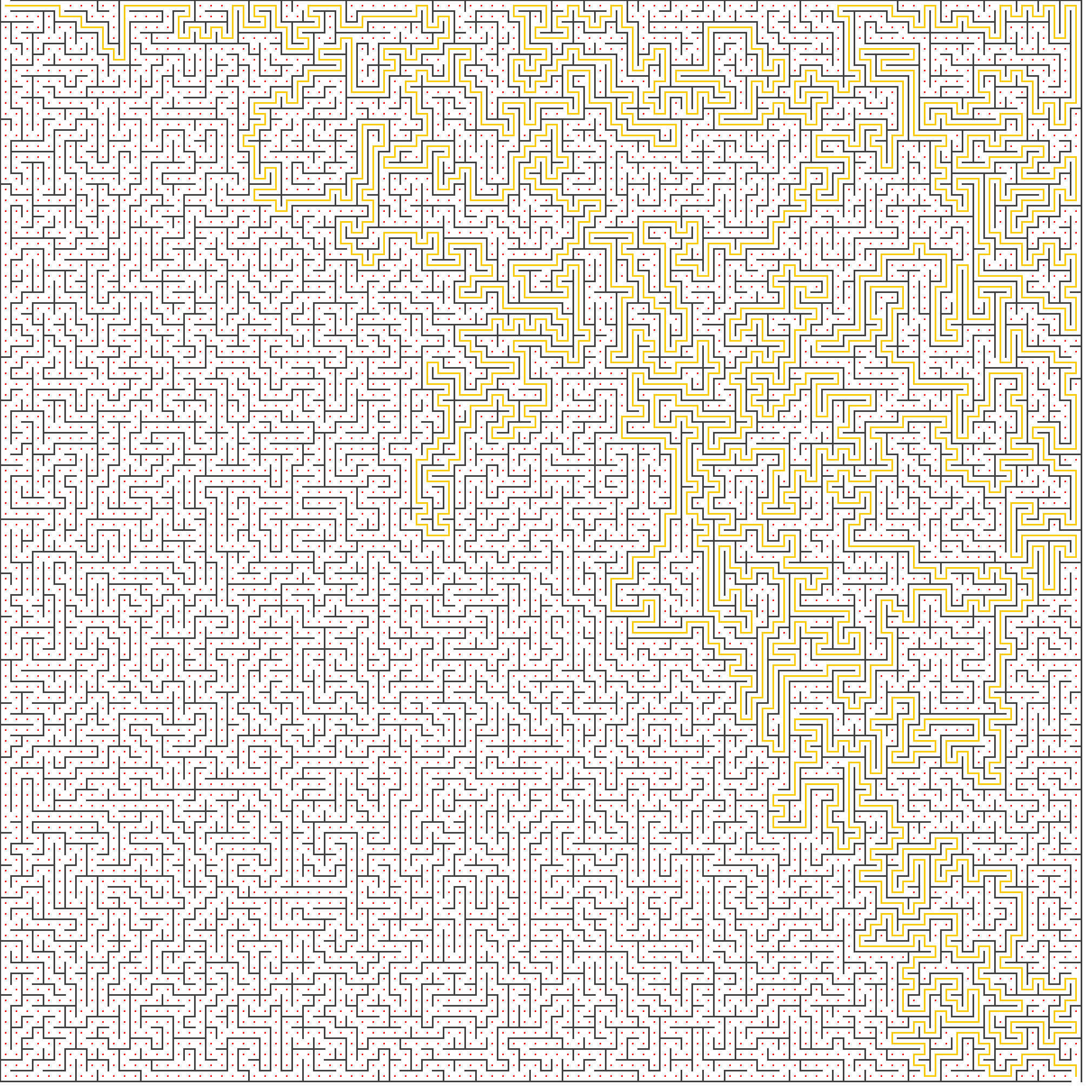


Рис. 3. Пример работы программы.

* 1. **Часть рабочего кода**

1. **public** **final** **class** MazeCreatorExtension **extends** AbstractMazeExtension {
2. Cell[][] cells;
3. LinkedList<Cell> nonCheckCells = **new** LinkedList<>();
4. LinkedList<Cell> stack = **new** LinkedList<>();
5. LinkedList<Cell> neighbors;
6. Random random;
7. @Override
8. **public** Maze generateMaze(**int** rows, **int** cols) {
9. nonCheckCells = **new** LinkedList<>();
10. cells = **new** Cell[rows][cols];
11. stack = **new** LinkedList<>();
12. random = **new** Random();
14. **for** (**int** i = 0; i < rows; i++) {
15. **for** (**int** j = 0; j < cols; j++) {
16. cells[i][j] = **new** Cell(i, j);
17. nonCheckCells.add(cells[i][j]);
18. }
19. }
20. **while** (nonCheckCells.size() != 0) {
21. **if** (stack.size() == 0) {
22. addToStack(nonCheckCells.get(random.nextInt(nonCheckCells.size())));
23. } **else** {
24. **while** (getNeighbors(stack.getLast(), rows, cols).size() == 0)
25. stack.removeLast();
26. }
27. **while** ((neighbors = getNeighbors(stack.getLast(), rows, cols)).size() != 0) {
28. addToStack(neighbors.get(random.nextInt(neighbors.size())));
29. }
30. }
32. Maze maze = **new** Maze(rows, cols);
33. **for** (**int** x = 0; x < rows; x++) {
34. **for** (**int** y = 0; y < cols; y++) {
35. maze.data[x][y] = cells[x][y].getValue();
36. }
37. }
38. **return** maze;
39. }
41. @Override
42. **public** Point[] findPath(Maze maze) {
43. cells[0][0].UP = **false**;
44. maze.data[0][0] = cells[0][0].getValue();
45. cells[maze.rows-1][maze.cols-1].DOWN = **false**;
46. maze.data[maze.rows -1][maze.cols1] = cells[maze.rows - 1][maze.cols - 1].getValue();
47. LinkedList<Point> path = **new** LinkedList<>();
48. LinkedList<Cell> cellsInWork = **new** LinkedList<>();
49. Cell firstCell = cells[0][0];
50. cellsInWork.add(firstCell);
51. Cell workCell;
52. **while** (cellsInWork.size() != 0){
53. workCell = cellsInWork.removeFirst();
54. **for** (Cell cell : workCell.capabilities){
55. cell.capabilities.remove(workCell);
56. cell.target = workCell;
57. cellsInWork.add(cell);
58. }
59. }
60. workCell = cells[maze.rows - 1][maze.cols - 1];
61. **while** (workCell.target != **null**){
62. path.add(**new** Point(workCell.x, workCell.y));
63. workCell = workCell.target;
64. }
65. path.add(**new** Point(firstCell.x, firstCell.y));
67. Point[] points = **new** Point[path.size()];
68. path.toArray(points);
69. **return** points;
70. }
72. **void** addToStack(Cell cell) {
73. **if** (stack.size() != 0)
74. setCorridor(cell, stack.getLast());
76. stack.add(cell);
77. cell.check();
78. nonCheckCells.remove(cell);
79. }
81. **void** setCorridor(Cell firstCell, Cell secondCell) {
82. **switch** (firstCell.x - secondCell.x) {
83. **case** 0:
84. **break**;
85. **case** 1:
86. firstCell.LEFT = **false**;
87. secondCell.RIGHT = **false**;
88. **break**;
89. **case** -1:
90. firstCell.RIGHT = **false**;
91. secondCell.LEFT = **false**;
92. **break**;
93. }
94. **switch** (firstCell.y - secondCell.y) {
95. **case** 1:
96. firstCell.UP = **false**;
97. secondCell.DOWN = **false**;
98. **break**;
99. **case** -1:
100. firstCell.DOWN = **false**;
101. secondCell.UP = **false**;
102. **break**;
103. }
104. **if** (!firstCell.capabilities.contains(secondCell))
105. firstCell.capabilities.add(secondCell);
106. **if**(!secondCell.capabilities.contains(firstCell))
107. secondCell.capabilities.add(firstCell);
109. }
111. LinkedList<Cell> getNeighbors(Cell cell, **int** rows, **int** cols) {
112. LinkedList<Cell> Neighbors = **new** LinkedList<>();
113. **if** (cell.x != 0)
114. **if** (!cells[cell.x - 1][cell.y].isCheck)
115. Neighbors.add(cells[cell.x - 1][cell.y]);
116. **if** (cell.y != 0)
117. **if** (!cells[cell.x][cell.y - 1].isCheck)
118. Neighbors.add(cells[cell.x][cell.y - 1]);
119. **if** (cell.x != rows - 1)
120. **if** (!cells[cell.x + 1][cell.y].isCheck)
121. Neighbors.add(cells[cell.x + 1][cell.y]);
122. **if** (cell.y != cols - 1)
123. **if** (!cells[cell.x][cell.y + 1].isCheck)
124. Neighbors.add(cells[cell.x][cell.y + 1]);
126. **return** Neighbors;
127. }
129. final **class** Cell {
131. final **int** x;
132. final **int** y;
133. **boolean** isCheck;
134. **boolean** UP;
135. **boolean** LEFT;
136. **boolean** DOWN;
137. **boolean** RIGHT;
138. final List<Cell> capabilities;
139. Cell target;

142. **public** Cell(**int** x, **int** y) {
143. **this**.x = x;
144. **this**.y = y;
145. isCheck = **false**;
146. UP = **true**;
147. LEFT = **true**;
148. DOWN = **true**;
149. RIGHT = **true**;
150. capabilities = **new** LinkedList<>();
151. }
152. **public** **char** getValue() {
153. **char** val = 0;
154. **if** (UP) val |= SQUARE\_UP;
155. **if** (LEFT) val |= SQUARE\_LEFT;
156. **if** (DOWN) val |= SQUARE\_DOWN;
157. **if** (RIGHT) val |= SQUARE\_RIGHT;
158. **return** val;
159. }
161. **public** **void** check() {
162. isCheck = **true**;
163. }
164. }
165. }

**Заключение**

Выполненная программная разработка соответствует заданию на курсовую работу. Результатом работы создано приложение, которое находит в нём кратчайший путь. Была изучена документация, JDK 1.8, ряда используемых сторонних библиотек и протоколов. Были разработаны все необходимые классы. Проведена сборка программы и проверка её работоспособности.

**Список использованной литературы**

1. Электронная энциклопедия Википедия // Backtracking. URL: [https://en.wikipedia.org/wiki/Backtracking](https://en.wikipedia.org/wiki/Backtracking%20)
2. Oracle Documentation // The Java Tutorials. URL: <https://docs.oracle.com/javase/tutorial/>
3. TutorialsPoint // LinkedList Tutorial. URL: [http://www.tutorialspoint.com/java/java\_linkedlist\_class.htm](http://www.tutorialspoint.com/java/java_linkedlist_class.htm%20)
4. Think Labyrinth // Labyrinth algrithm. URL: [http://www.astrolog.org/labyrnth/algrithm.htm](http://www.astrolog.org/labyrnth/algrithm.htm%20)
5. Jamisbuck // Mazes. URL: [http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking](http://weblog.jamisbuck.org/2010/12/27/maze-generation-recursive-backtracking%20)
6. Oracle // Java SE Development Kit 8 Documentation. URL: <http://www.oracle.com/technetwork/java/javase/documentation/jdk8-doc-downloads-2133158.html>