**Министерство науки и высшего образования Российской Федерации**

**Федеральное государственное автономное образовательное учреждение**

**высшего образования**

**«КАЗАНСКИЙ (ПРИВОЛЖСКИЙ) ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»**

Институт вычислительной математики и информационных технологий

Кафедра системного анализа и информационных технологий

Направление подготовки: 02.03.02 – Фундаментальная информатика

и информационные технологии

Профиль: Системный анализ и информационные технологии

**КУРСОВАЯ РАБОТА**

Система управления движения беспилотного транспорта для решения задачи нахождения оптимального маршрута в неизвестной местности

Студент 3 курса

группы 09-732

«\_\_\_» \_\_\_\_\_\_\_ 2020 г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Коробейников К.В.

Научный руководитель

Ассистент

«\_\_\_» \_\_\_\_\_\_\_ 2020 г. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Багавеев В.А.

Казань-2020

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 3](#_Toc41153731)

[ГЛАВА 1. ПРОБЛЕМЫ БЕСПИЛОТНОГО ТРАНСПОРТА И СПОСОБЫ ИХ РЕШЕНИЯ 5](#_Toc41153732)

[1.1. Классический подход. 5](#_Toc41153733)

[1.1.1. Алгоритмы на графах 5](#_Toc41153734)

[1.1.2. Оптимизационные методы 9](#_Toc41153735)

[1.1.3. Стохастические алгоритмы 12](#_Toc41153736)

[1.1.4. Специализированные алгоритмы 13](#_Toc41153737)

[1.2. Нейросетевой подход 14](#_Toc41153738)

[ГЛАВА 2. ПРОБЛЕМЫ БЕСПИЛОТНОГО ТРАНСПОРТА И СЛОЖНОСТЬ ИХ РЕШЕНИЯ 15](#_Toc41153739)

[2.1. Задача построения лабиринта и его определение 15](#_Toc41153740)

[ГЛАВА 3. РЕАЛИЗАЦИЯ АЛГОРИТМА НАХОЖДЕНИЯ КРАТЧАЙШЕГО ПУТИ В ЛАБИРИНТЕ 20](#_Toc41153741)

[3.1. Описание алгоритма и его свойства 20](#_Toc41153742)

[3.2. Визуализация 25](#_Toc41153743)

[ЗАКЛЮЧЕНИЕ 28](#_Toc41153744)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 30](#_Toc41153745)

[ПРИЛОЖЕНИЕ 31](#_Toc41153746)

# **ВВЕДЕНИЕ**

**Беспилотное автомобильное средство** (**БАС**) — транспортное средство, оборудованное системой автоматического управления, которое может передвигаться без участия человека. БАС - это обычный автомобиль, оснащенный многочисленными и разнообразными электронными устройствами. БАС разделяют по степени автоматизации. Согласно классификации международного общества автомобильных инженеров (SAE International) существует 6 уровней систем интеллектуальной помощи водителю или ADAS (Advanced Driver Assistance System). Важно заметить ,что на данный момент лучшие системы остановились на 3 уровне ADAS.

Для решения задач беспилотного транспорта используются стек технологий которые позволяют выполнять основные задачи БАС:

1. Ориентация в пространстве. Одним из наиболее важных компонентов систем компьютерного зрения, связанных с пространственной ориентацией, является модуль SLAM. Именно он выполняет глобальную оптимизацию измерений. Фактически основная цель SLAM - минимизация ошибок при определении смещения камеры - благодаря этому траектория движения робота (камеры) выравнивается с пространством просмотра, на основе которого формируется карта мира. В нашей работе мы не будем особо фокусироваться на алгоритмах, которые используются для визуализации и обработки сигналов для построения карты среды. В работе будем учитывать что эта задача нами решена.
2. Построение маршрута. В данной работе речь пойдёт о применении задачи нахождении выхода в лабиринте для решения задачи оптимального маршрута при неизвестной местности. Также будет рассмотрена проблема построения лабиринта для симуляций различных вариантов решений.

**Цель данной работы –** разработать и рассмотреть различные алгоритмы, способные генерировать лабиринты и решить задачу нахождения оптимального пути в неизвестной местности.

**Задачи работы** :

‒ Изучить литературу по методам;

‒ Изучить существующие решения проблемы планирования пути;

‒ Рассмотреть модификации алгоритмов;

‒ Реализовать методы построение и решения.

# **ГЛАВА 1. ПРОБЛЕМЫ БЕСПИЛОТНОГО ТРАНСПОРТА И СПОСОБЫ ИХ РЕШЕНИЯ**

## Классический подход.

Классический подход разделен на 4 части :

1. Локализация;
2. Распознавание;
3. Планирование;
4. Управление.

Модуль локализации отвечает за то, чтобы машина понимала, где она находится. Модуль распознавания — за то, что находится вокруг машины. Модуль планирования обладает информацией о том, что находится вокруг, и зная, куда хочется приехать, строит маршрут. Модуль управления говорит, как же ехать по маршруту, чтобы приехать, выполнить эту траекторию.

БАС обладает рядом довольно существенных ограничений. Это не шар в пространстве, который может катиться в любую сторону и мгновенно останавливаться и замедляться. У автомобиля есть текущее направление, угол поворота колес, и он не может просто оказаться на два метра левее от текущего местоположения, это очень сложно. Он может ехать прямо вперед, поворачивая на какой-то угол, но тем не менее, перемещение очень сильно ограничено. И траектория, которую мы строим, подвержена ограничениям, которые следуют из кинематики. Например не можем мгновенно разогнаться и даже мгновенно увеличить свое ускорение.

### **Алгоритмы на графах**

Вот стоит машина, мы можем понять, где она стоит, но в реальности графа никакого нет, вершин ребер на дороге не нарисовано. Нам этот граф нужно как-то придумать самим. Первое, что делаем: разбиваем все пространство на клетки, рассмотрим маленькие клеточки и скажем, что есть вся наша поверхность земли, разбитая на клетки 25 на 25 или 50 на 50 см. Потом соединим ближайшие клетки ребрами и будем искать на них путь. Это будет довольно далеко от того пути, который БАС может проехать, но какое-то приближение это даст. И у нас будут такие вершины в двумерном пространстве.

Мы можем усложнять наше пространство, добавляя туда текущий угол поворота машины. У нас уже будут клетки не просто x и y, но и текущая ориентация машины в направлениях север, юг, запад и восток, тоже как-то дискретизированных. Кроме направления можем учитывать: текущую скорость машины, текущее ускорение, текущее тангенциальное ускорение, нормальное. Все это важно учитывать. Но чем больше мы усложняем наше пространство, тем более сложным становится наш граф.

Алгоритм разбиения пространства на клетки:

1. Разбиваем все пространство на клетки как на рисунке 1.



Рисунок 1 - Разбитие и построение линии продвижения

1. Построим граф из примитивов движения как на рисунке 2.

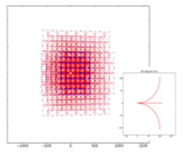


Рисунок 2 - Добавляем примитивы движения

1. Ограничить количество вершин в каждой клетке как показано на рисунке 3.



Рисунок 3 - Ограничение вершин нашего маршрута

**Построение окружающей среды БАС**

Помимо деления пространства на клетки, мы можем сказать, что мы можем двигаться небольшими шагами в форме примитивов. Например, переместитесь на 50 см вперед или на 50 см вперед и поверните налево или на 50 см вперед и поверните направо. И с такими примитивами мы можем соединить все наше пространство. У нас нет явных ячеек, но если эти маленькие шаги достаточно хорошо скоординированы друг с другом, мы получаем регулярную сетку, которая хорошо и аккуратно покрывает пространство.

Предположим, мы рассматриваем примитивы, которые плохо сочетаются друг с другом, например, скажем, что поворот влево происходит под углом 89 градусов, а поворот вправо - под углом 90 градусов. Тогда у нас уже есть такое же количество вершин, оно будет занимать гораздо меньшую площадь пространства, так как они плохо связаны друг с другом, а плотность точек будет очень высокой, и мы не сможем охватить пространство далеко,  
Мы можем комбинировать наши подходы и рассматривать примитивы движения под разными углами и разными движениями. Например, сначала вперед, затем снова вперед и поверните налево на 40 градусов и так далее.

Мы также разделим наше пространство на ячейки и скажем, что в одной ячейке может быть не более 1, 2, ..., k вершин.

И вот здесь у нас есть хорошее решение – алгоритм А\*, рисунок 4.

* Алгоритм Дейкстры обходит вершины графа, доставая их из очереди с приоритетом по расстоянию от стартовой вершины – f(x);
* Алгоритм A\* достает из очереди вершины по возрастанию f(x) + h(x), где h(x) – эвристическая функция оценки расстояния от вершины до цели;
* Необходимое условие для h(x): значение функции оценки не должно превышать действительное расстояние до цели.

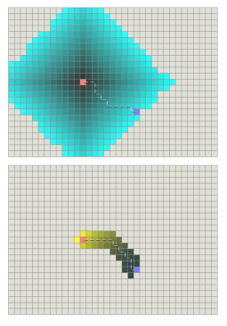


Рисунок 4 - Алгоритм А\*

Преимущества и недостатки А\*

**Преимущества :**

* Находит заведомо кратчайший путь в дискретизированном пространстве

**Недостатки :**

* В пространствах малой размерности путь почти всегда не является кинематически выполнимым;
* В пространствах большой размерности время работы драматически ухудшается.

Алгоритм А\* позволяет найти заведомо оптимальный путь, который ведет нас к цели, огибая препятствия. Но если у нас пространство малой размерности, учитываем только x, y, ориентацию машины и то, что она не может мгновенно разогнаться, — значит, все эти ограничения будут учтены. Если в наш граф добавлять большое количество параметров, то он будет находиться в пространстве очень высокой размерности. У нас будет 7-, 8-, 10-мерное пространство, мы будем успевать рассматривать небольшие кусочки этого пространства и не сможем построить достаточно далекий маршрут из-за очень высокой вариативность параметров. В каких-то ситуациях А\* сложно применять, а где-то он достаточно неплохо себя показывает. Следовательно данный подход будем использовать для решения нашей задачи.

### **Оптимизационные методы**

Определим нашу траекторию

Определим оптимальную траекторию как минимизирующую интеграл :

**Постановка задачи** : рассмотрим траекторию нашего положения БАС во времени, x и y, зависящие от времени t, определим, в какой точке мы хотим оказаться в момент времени t. Можем определить угол касательной через арктангенс от производных, и сказать, что оптимальной в этом случае будет траектория, которая минимизирует функционал, являющийся интегралом по времени вперед от какой-то функции от траектории. Функция от траектории здесь каким-либо образом нас штрафует за резкие повороты, резкие разгоны, нахождение близко к препятствиям и т. д. Тогда, если мы просуммируем вдоль нашей траектории все эти штрафы, которые мы сами себе придумаем, и попытаемся это минимизировать стандартным математическим аппаратом, никак не связанным с автомобилями в целом и беспилотными автомобилями в частности, то мы решим задачу в каком-то общем виде.

Наш БАС— объект довольно сложной формы, и препятствия, которые мы объезжаем, это тоже объекты сложных форм. Следовательно тут нужно производить какое-то упрощение. Например, машина — просто пять окружностей, которые ездят в разные стороны.

Для удобства вычисления расстояния представим машину в виде набора окружностей как на рисунке 5.

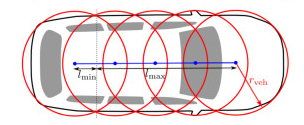


Рисунок 5 - Пример построения окружностей машины

А для плавной функции расстояния до препятствий введем поле градиентов как показано на рисунке 6.

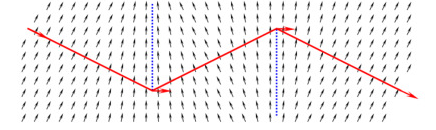


Рисунок 6 - Поле градиентов

От окружностей очень легко считать расстояния до чего угодно и очень легко проверять окружность на пересечения с остальными геометрическими примитивами.

Наше эвклидово расстояние до невыпуклых многоугольников не обладает необходимыми нам свойствами и плохо дифференцируемо в местах, где эта не выпуклость возникает. Поэтому можем построить такое псевдо расстояние по градиентному полю до нашей ломаной, она здесь обозначена красным и представляет собой препятствие.

**Преимущества :**

* Пространство управления непрерывно

**Недостатки:**

* Сходится к локальным минимумам;
* Все ограничения должны быть дифференцируемыми функциями.

К преимуществам таких методов отнесем то, что получается хорошая траектория и пространство управления непрерывно. По нему можно ехать, все ограничения в той или иной степени соблюдаются. Но к сожалению, большинство оптимизационных методов или почти все так или иначе страдают от локальных минимумов, где их попытки что-либо оптимизировать застревают и не находят достаточно хорошего решения. Очень сложно все формулировать в виде математических дифференцируемых функций.  
Можно применить алгоритмы, которые работают некоторым случайным образом, но зато позволяют нам построить какой-то приближенный маршрут достаточно быстро и удобно. Например, мы будем строить наш граф, являющийся деревом, итеративным образом. Вначале ничего делить на клетки не будем, никаких примитивов строить не будем. Возьмем симулятор нашей машины, который умеет симулировать движение, максимально похожее на то, как машина едет по-настоящему. И возьмем стартовую точку.

После этого итеративно выберем случайную точку в пространстве. И найдем в текущем построенном дереве ближайший к ней узел. Построим ребро в сторону этой точки с помощью симуляции проезда в эту сторону.  
Получается, каждый раз едем от нашего построенного дерева в случайную сторону и можем вершины в этом дереве делать в пространстве любой размерности, то есть в каждой вершине учитывать текущее направление машины, текущую скорость, ускорение, все углы, которые важны. А потом, когда возьмем новую точку, то возьмем и ближайшую к этой точке вершину.

Проедем с помощью такой симуляции, 5 метров в сторону этой точки. Затем возьмем другую точку и проедем в ее сторону.  
Не ищем оптимальные способы объехать препятствие. Просто едим в разные стороны, но каждый раз делаем это из наиболее исследуемого участка нашего пространства к той, неизведанной стороне. Данный способ не особо подходит под нашу задачу из-за того что все ограничения должны быть дифференцируемыми.

### **Стохастические алгоритмы**

Будем строить дерево итеративным образом. На каждой итерации:

* Выберем случайную точку в пространства;
* Найдем в текущем дереве ближайший к ней узел;
* Построим ребро в сторону симуляции проезда нескольких метров.

Пример построения показан на рисунке 7.

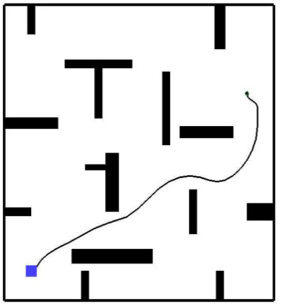


Рисунок 7 - Построение модели стохастическими методами

Не оптимально, но быстро. Синий квадрат в углу — наша стартовая позиция. Позже получаем какой-то путь к цели, который выглядит, не совсем оптимально и элегантно, но зато получен достаточно быстрым способом.   
Основной подход лежит на Марковских цепях.

**Преимущества:**

* Высокая скорость работы в пространствах большой размерности;
* Пути можно подавать практически напрямую в управляющий блок.

**Недостатки :**

* Никаких гарантий на оптимальность решения;
* Найденные траектории могут быть слишком извилистыми;
* Сходится к локальным минимумам;
* Все ограничения должны быть дифференцируемыми функциями.

Получается, преимущества заключаются в том, что работаем очень быстро в многомерных пространствах. Поскольку пути получаем с помощью честной симуляции, то практически напрямую можем передавать их в управляющий блок. Но у нас нет гарантий, что точки будут выбираться достаточно хорошо. Решения могут получаться довольно кривые и не всегда оптимальные. Данный метод тоже не подходит так как очень часто решения получаются не оптимальными.

### **Специализированные алгоритмы**

Алгоритм:

* Следуем вдоль известного пути;
* Достаточно строить гладкую траекторию.

Пример движения БАСа показан ниже на рисунке 8.

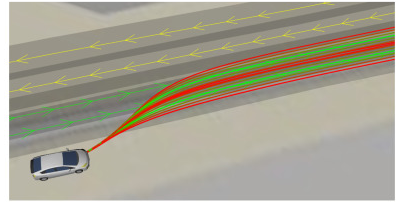


Рисунок 8 - Специализированные алгоритмы движения по дороге

Нам не всегда нужно строить деревья для решения задач парковки или выполнения сложных маневров. Они нужны, но когда мы едем по полосе движения, нам достаточно построить более или менее плавный путь, который идет к центру этой полосы движения или с некоторым смещением влево-вправо. Это гораздо проще, чем искать абстрактный путь в графе. Просто возьмите текущее положение автомобиля, посмотрите на путь, по которому мы хотели бы идти, и плавно сверните на этот путь, выруливайте на него. Но езда в городе связана с тем, что нужно соблюдать правила дорожного движения. Мы не знаем, в каких условиях мы можем оказаться, поэтому такой подход тоже не вариант.

## Нейросетевой подход

Нейросетевой подход в последнее время показывает неплохие результаты относительно классического подхода. Давайте просто картинку с камер подавать в нейросеть, предварительно ее обучив на каких-нибудь человеческих перемещениях. Обучим, в каких ситуациях куда нужно крутить руль, тормозить, газовать, возьмем много-много данных, сделаем большую нейросеть, и по задумке она должна хорошо себя вести после этого.  
В этом направлении ведутся большие работы, но на практике кажется, что нужно все-таки слишком много данных, нужна слишком большая нейросеть, чтобы за человеком все успешно повторять в различных ситуациях. Поэтому данный подход в данной работе не будет рассматриваться.

# **ГЛАВА 2. ПРОБЛЕМЫ БЕСПИЛОТНОГО ТРАНСПОРТА И СЛОЖНОСТЬ ИХ РЕШЕНИЯ**

## Задача построения лабиринта и его определение

Вопрос о лабиринтах интересовал в свое время многих. В самом деле, возможно ли построить или даже начертить "безвыходный" лабиринт, то есть такой, в котором найти путь к центру и выход было бы только делом случая.

Пример построения и прохождения показан ниже на рисунке 9.

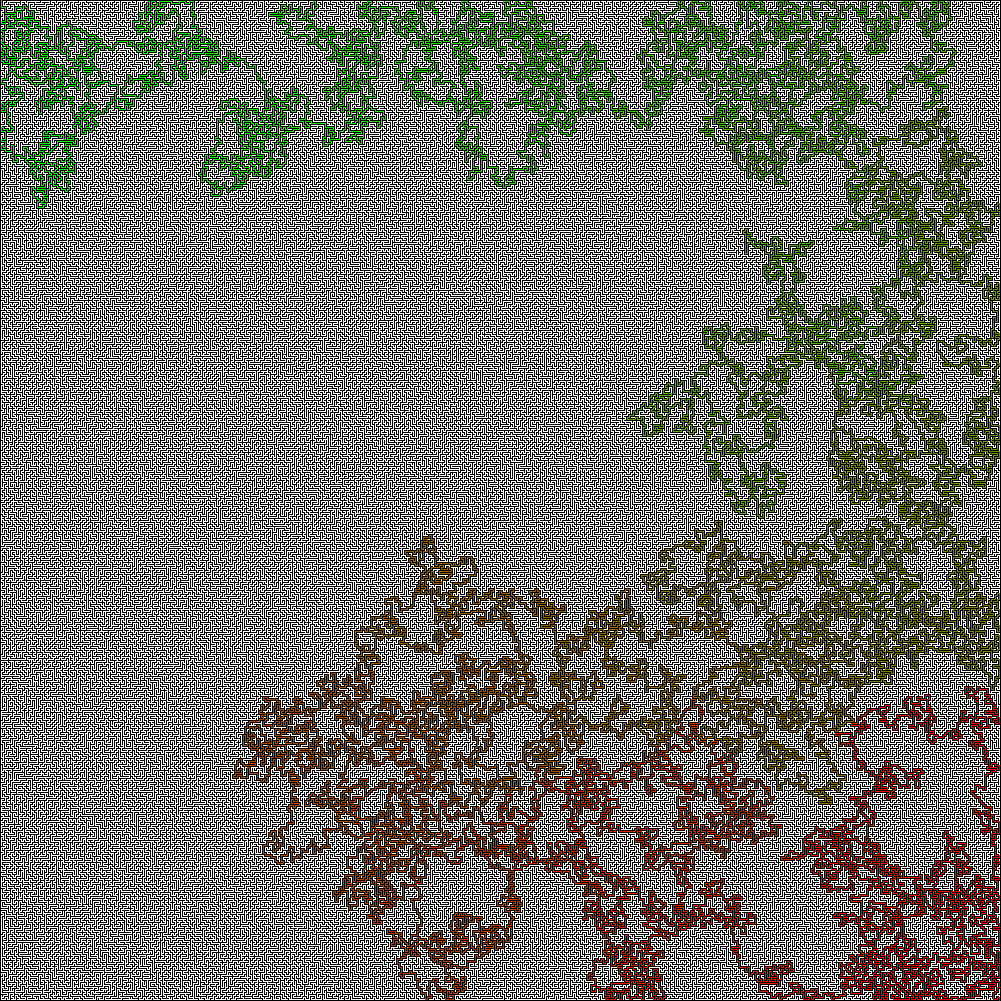


Рисунок 9 - Случайно сгенерированный лабиринт

Результаты произведенных в этом отношении изысканий привели исследователей к заключению, что **безвыходных лабиринтов не существует**.

Необходимо отметить что лабиринты бывают разные и в нашей задаче мы будем использовать 2D лабиринты с единым выходом и входом так как их можно подвезти к общему понятию неизвестной местности и вот причины:

Первый выход который мы найдем является для нас самым оптимальным в неизвестной местности

Лабиринты подразумевают в себе тупики и необходимость возврата для поиска выхода, что можно назвать частным случаем нахождения в неизвестном месте

Мы не будем рассматривать многомерные лабиринты так как в частном случае мы предполагаем что наше средство передвигается по земле

**Классификация лабиринтов**

Лабиринты в целом (а значит, и алгоритмы для их создания) можно разбить по различным классификациям: размерности, гиперразмерности, топологии, тесселяции, маршрутизации, текстуре и приоритету. Лабиринт может использовать по одному элементу из каждого класса в любом сочетании.В нашем случае главное размерность и маршрутизация

**Размерность:** класс размерности по сути определяет, сколько измерений в пространстве заполняет лабиринт. Существуют следующие типы:

**Двухмерные:** большинство лабиринтов, как бумажных, так и реальных, имеют эту размерность, то есть мы всегда можем отобразить план лабиринта на листе бумаги и двигаться по нему, не пересекая никаких других коридоров лабиринта.

**Трёхмерные:** трёхмерный лабиринт имеет несколько уровней; в нём (по крайней мере, в ортогональной версии) проходы могут кроме четырёх сторон света опускаться вниз и подниматься вверх. 3D-лабиринт часто визуализируют как массив из 2D-уровней с пометками лестниц «вверх» и «вниз»

**Алгоритм двоичного дерева**

Суть его в том, чтобы проложить путь в случайном направлении от каждой ячейки поля: мы обрабатываем только 1 ячейку за единицу времени, поэтому мы можем генерировать лабиринты бесконечного размера, сохраняя только конечный результат (лабиринт) без необходимости сохранять какие-либо вторичную информацию.

Этот метод генерации имеет два побочных эффекта:

1. Лабиринты имеют сильные диагональные смещения и отсутствие тупиков в своем направлении.
2. Два пустых коридора по бокам лабиринта. Когда алгоритм «копает» до конца строки / столбца, у него нет другого выбора, кроме как продолжить путь в одном направлении, создавая пустые «границы».

Кстати, название не просто совпадает со структурой данных. Результатом его работы является случайное двоичное дерево, в котором от каждой ячейки (вершины) есть ровно 1 путь к корню (родительской вершине) и, соответственно, ровно 1 путь к любой другой ячейке. В результате любая ячейка имеет не более 3 соединений со своими соседями.

**Формальный алгоритм (для северо-восточного смещения)**

Выбрать начальную клетку;

Выбрать случайное направление для прокладывания пути. Если соседняя клетка в этом направлении выходит за границы поля, прокопать клетку в единственно возможном направлении;

Перейти к следующей клетке;

Повторять 2-3 до тех пор, пока не будут обработаны все клетки;

**Преимущества:**

* Простая реализация;
* Быстрый;
* Возможность генерировать бесконечные лабиринты.

**Недостатки**:

* Низкая сложность картинки;
* Сильный диагональный сдвиг;
* Отсутствие тупиков при смене;
* Однородность генерируемых лабиринтов.

**Алгоритм Sidewinder**

Алгоритм с непереводимым именем Sidewinder в своей работе очень похож на алгоритм двоичного дерева, в отличие от того, что он не имеет характерного диагонального сдвига, мы не рассматриваем один пустой коридор и ячейку отдельно, а в наборах. Лабиринты получаются с преимущественно вертикальным или горизонтальным смещением (в зависимости от реализации), без тупиков в их направлении. По сравнению с более примитивным аналогом смещение не так заметно и больше похоже на «спираль», которая плавно заменяет вертикальные и горизонтальные коридоры.

Что касается побочных эффектов, Sidewinder создает только один пустой коридор на одной стороне вместо двух. Начиная создавать наборы из первого ряда поля, у нас нет возможности вырыть путь, так как мы находимся в крайнем вертикальном положении, и попытка подняться выше приведет к выходу за пределы поля. , Если мы организуем наборы, не идя вертикально, то мы создадим несколько областей, изолированных друг от друга.

Формальный алгоритм (для стандартного смещения):

1. Выберите начальную строку;
2. Выберите начальную ячейку строки и сделайте ее текущей;
3. Инициализировать пустой набор;
4. Добавьте текущую ячейку в набор;
5. Решите, проложить ли путь вправо;
6. Если проложено, перейдите к новой ячейке и сделайте ее текущей;

Повторите шаги 3-6;

1. Если вы не проложили дорогу, выберите случайную ячейку из набора и проложите путь оттуда. Идем к следующему ряду и повторяем 2-7;
2. Продолжайте, пока каждая строка не будет обработана.

**Преимущества:**

* Возможность генерировать бесконечные лабиринты;
* Только 1 пустой коридор;
* Более сложная картина, в отличие от алгоритма двоичного дерева.

**Недостатки:**

* Более запутанная реализация;
* Отсутствие тупиков при смене;
* Сильное вертикальное смещение.

Из всего многообразия вариантов построения лабиринтов мы воспользуемся самыми простыми так как наши лабиринты входят в круг задач решаемых этими алгоритмами.

# **ГЛАВА 3. РЕАЛИЗАЦИЯ АЛГОРИТМА НАХОЖДЕНИЯ КРАТЧАЙШЕГО ПУТИ В ЛАБИРИНТЕ**

## 3.1. Описание алгоритма и его свойства

Есть алгоритм Дейкстры, который чуть более известен, чем алгоритм А\*, и который обходит все вершины в графе по возрастанию расстояния от стартовой вершины. Достает их из какой-то очереди с приоритетом в порядке увеличения расстояния.

Чтобы производить поиск более эффективно, мы будем рассматривать вершины не только по увеличению расстояния от старта, а еще и прибавим к этому расстоянию некоторую эвристическую оценку того, сколько нам осталось ехать до конца. Логика очень понятна: мы не просто говорим, что мы сейчас стоим в какой-то вершине и поэтому рассматриваем следующую по очереди вершину на каком-то удалении от старта. Мы еще к этому параметру добавляем какую-то функцию, позволяющую нам оценить более перспективные вершины, от которых, скорее всего, до финиша ехать недалеко. Мы не знаем, сколько на самом деле ехать от каждой вершины до финиша, потому что если бы мы знали, нам и путь не надо было бы искать. Но мы можем это как-то оценить и получить картинку на нижнем рисунке. Мы будем оценивать вершины, которые как-то идут в сторону нашей цели, и рассмотрим существенно меньшее количество вершин, чем в алгоритмеДейкстры.

Планирование пути – Эвристики для A\*

* Расстояние до цели напрямик;
* Расстояние с учетом кинематических ограничений, но без учета препятствий;
* Расстояние с учетом, но без кинематических ограничений;
* Максимум допустимых эвристик есть допустимая эвристика.

В качестве эвристических функций можно использовать разные подходы. В этом и заключается сложность применения алгоритма А\*. Чем более качественные эвристики подбираем, тем лучше алгоритм будет идти в сторону цели, тем меньше вершин он рассмотрит и тем быстрее он будет работать.

Рассмотрим расстояние до цели напрямик. Очевидно, что не можем приехать к конечной точке быстрее, чем если поедем к ней напрямик.  
Более сложная и более эффективная эвристика — расстояние с учетом наших кинематических ограничений, но без учета препятствий. Предположим, в мире ничего нет, только мы и цель, но машина не может мгновенно перемещаться влево или вправо, ездит по своим законам, по имеющейся физической модели машины. Поскольку никаких препятствий нет, мы можем заранее просчитать расстояние до цели из различных мест, в которых мы можем находиться, и использовать это в качестве эвристической функции.

Можем поступить наоборот: сказать, что препятствие есть, но, допустим, машина умеет двигаться вперед, назад, влево, вправо, как угодно, быстро разгоняться и быстро тормозить. Построим наше расстояние с учетом препятствий, которые мы видим. Снова получится оценка, находящаяся ниже, чем реальное расстояние, которое нам осталось проехать.

И наконец, из всех эвристик мы можем рассмотреть максимум. Поскольку каждая из них не превышает нашего действительного расстояния, максимум из них тоже не будет превышать действительного расстояния и алгоритм будет работать неплохо.

Как и алгоритм поиска в ширину, A\* является полным в том смысле, что он всегда находит решение, если таковое существует.

Если эвристическая функция h допустима, то есть никогда не переоценивает действительную минимальную стоимость достижения цели, то A\* сам является допустимым (или оптимальным), также при условии, что мы не отсекаем пройденные вершины. Если же мы это делаем, то для оптимальности алгоритма требуется, чтобы h(x) была ещё и монотонной, или преемственной эвристикой. Свойство монотонности означает, что если существуют пути A—B—C и A—C (не обязательно через B), то оценка стоимости пути от A до C должна быть меньше либо равна сумме оценок путей A—B и B—C. (Монотонность также известна как неравенство треугольника: одна сторона треугольника не может быть длиннее, чем сумма двух других сторон.) Математически, для всех путей x, y (где y — потомок x) выполняется:

g ( x ) + h ( x ) ≤ g ( y ) + h ( y ) . {\displaystyle g(x)+h(x)\leq g(y)+h(y).} A\* также оптимально эффективен для заданной эвристики h. Это значит, что любой другой алгоритм исследует не меньше узлов, чем A\* (за исключением случаев, когда существует несколько частных решений с одинаковой эвристикой, точно соответствующей стоимости оптимального пути).

В то время как A\* оптимален для «случайно» заданных графов, нет гарантии, что он сделает свою работу лучше, чем более простые, но и более информированные относительно проблемной области алгоритмы. Например, в неком лабиринте может потребоваться сначала идти по направлению от выхода, и только потом повернуть назад. В этом случае обследование вначале тех вершин, которые расположены ближе к выходу (по прямой дистанции), будет потерей времени.

**Особые случаи**

В общем говоря, поиск в глубину и поиск в ширину являются двумя частными случаями алгоритма A\*. Для поиска в глубину возьмём глобальную переменную-счётчик С, инициализировав её неким большим значением. Всякий раз при раскрытии вершины будем присваивать значение счётчика смежным вершинам, уменьшая его на единицу после каждого присваивания. Таким образом, чем раньше будет открыта вершина, тем большее значение h(x) она получит, а значит, будет просмотрена в последнюю очередь. Если положить h(x) = 0 для всех вершин, то мы получим ещё один специальный случай — алгоритм Дейкстры.

**Тонкости реализации**

Существует несколько особенностей реализации и приёмов, которые могут значительно повлиять на эффективность алгоритма. Первое, на что не мешает обратить внимание — это то, как очередь с приоритетом обрабатывает связи между вершинами. Если вершины добавляются в неё так, что очередь работает по принципу LIFO, то в случае вершин с одинаковой оценкой A\* «пойдёт» в глубину. Если же при добавлении вершин реализуется принцип FIFO, то для вершин с одинаковой оценкой алгоритм, напротив, будет реализовывать поиск в ширину. В некоторых случаях это обстоятельство может оказывать существенное значение на производительность.

В случае, если по окончании работы от алгоритма требуется маршрут, вместе с каждой вершиной обычно хранят ссылку на родительский узел. Эти ссылки позволяют реконструировать оптимальный маршрут. Если так, тогда важно, чтобы одна и та же вершина не встречалась в очереди дважды (имея при этом свой маршрут и свою оценку стоимости). Обычно для решения этой проблемы при добавлении вершины проверяют, нет ли записи о ней в очереди. Если она есть, то запись обновляют так, чтобы она соответствовала минимальной стоимости из двух. Для поиска вершины в сортирующем дереве многие стандартные алгоритмы требуют времени O(n). Если усовершенствовать дерево с помощью хеш-таблицы, то можно уменьшить это время.

**Почему A\* допустим и оптимален?**

Алгоритм A\* и допустим, и обходит при этом минимальное количество вершин, благодаря тому, что он работает с «оптимистичной» оценкой пути через вершину. Оптимистичной в том смысле, что, если он пойдёт через эту вершину, у алгоритма «есть шанс», что реальная стоимость результата будет равна этой оценке, но никак не меньше. Но, поскольку A\* является информированным алгоритмом, такое равенство может быть вполне возможным.

Когда A\* завершает поиск, он, согласно определению, нашёл путь, истинная стоимость которого меньше, чем оценка стоимости любого пути через любой открытый узел. Но поскольку эти оценки являются оптимистичными, соответствующие узлы можно без сомнений отбросить. Иначе говоря, A\* никогда не упустит возможности минимизировать длину пути, и потому является допустимым.

Предположим теперь, что некий алгоритм B вернул в качестве результата путь, длина которого больше оценки стоимости пути через некоторую вершину. Следовательно на основании эвристической информации, для алгоритма B нельзя исключить возможность, что этот путь имел и меньшую реальную длину, чем результат. Соответственно, пока алгоритм B просмотрел меньше вершин, чем A\*, он не будет допустимым. Итак, A\* проходит наименьшее количество вершин графа среди допустимых алгоритмов, использующих такую же точную (или менее точную) эвристику.

**Оценка сложности**

Временна́я сложность алгоритма A\* зависит от эвристики. В худшем случае, число вершин, исследуемых алгоритмом, растёт экспоненциально по сравнению с длиной оптимального пути, но сложность становится полиномиальной, когда ошибка h(x) не растет быстрее, чем логарифм от оптимальной эвристики.

Однако ещё бо́льшую проблему, чем временна́я сложность, представляют собой потребляемые алгоритмом ресурсы памяти. В худшем случае ему приходится помнить экспоненциальное количество узлов. Для борьбы с этим было предложено несколько вариаций алгоритма, таких как алгоритм A\* с итеративным углублением (iterative deeping A\*, IDA\*), упрощённый MA\* (simplified MA\*, SMA\*) и рекурсивный поиск по первому наилучшему совпадению (recursive best-first search, RBFS).

Это может быть описано с помощью следующих шагов:

1. Выберите начальную ячейку, отметьте ее как посещенную и поместите в стек
2. Пока стек не пустой, извлеките ячейку из стека и сделайте ее текущей ячейкой
3. Если в текущей ячейке есть соседи, которые не были посещены
4. Вставьте текущую ячейку в стек
5. Выберите одного из не посещенных соседей
6. Удалить стену между текущей ячейкой и выбранной ячейкой
7. Отметьте выбранную ячейку как посещенную и поместите ее в стек

## 3.2. Визуализация

Для визуализации используется 2 библиотеки python – **Pygame** и встроенная библиотека **Tkinter**

Первое решение – Tkinter(рисунки 10 и 11)

Второе решение – Pygame(рисунки 12 и 13)

**Tkinter** – это кроссплатформенная библиотека для разработки графического интерфейса на языке Python (начиная с Python 3.0 переименована в tkinter). Tkinter расшифровывается как Tk interface, и является интерфейсом к Tcl/Tk.

**Pygame** — это фреймворк языка Python для программирования игр. Он создан поверх SDL.Важно отметить что из pygame используем часть - pygame.display.set\_caption ("Python Maze Generator") .

Она построена на основе игры Atari и использовалась для генерации уровней игры.

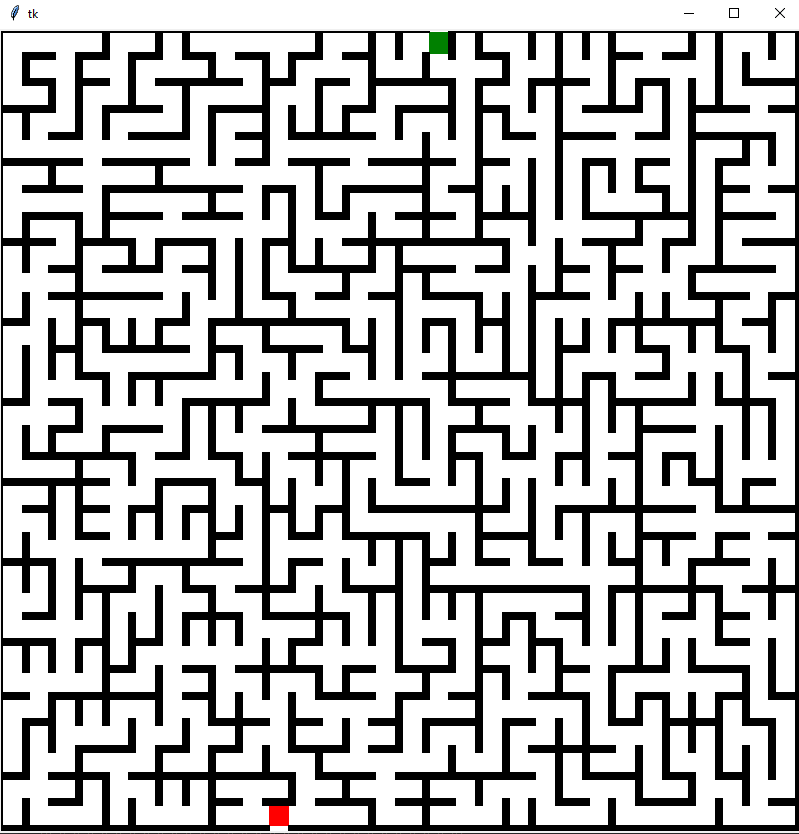


Рисунок 10 - Начальное состояние лабиринта

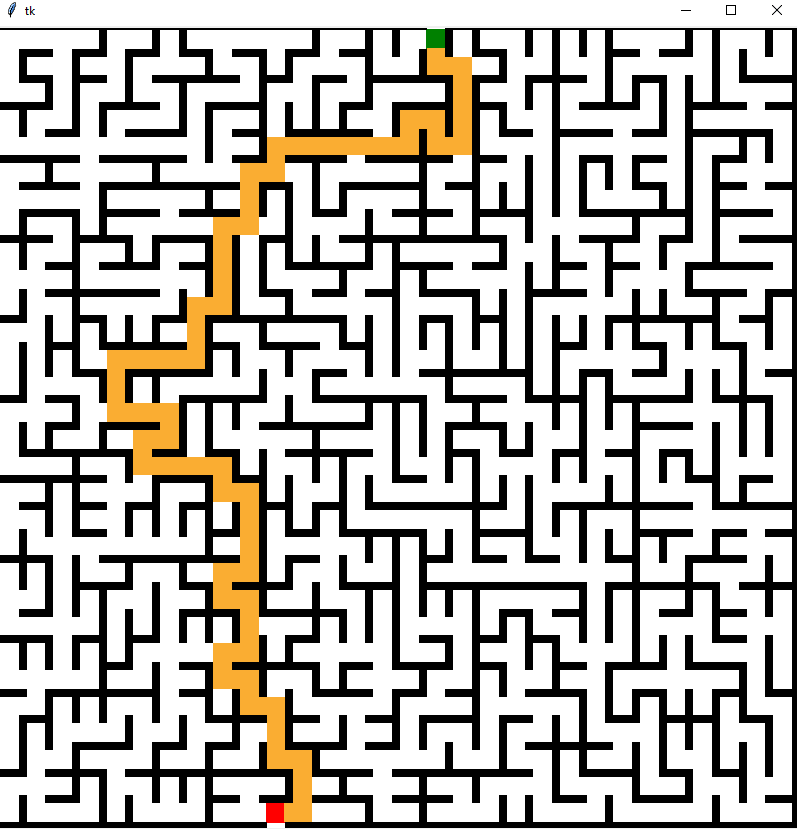


Рисунок 11 - Итоговое состояние лабиринта

Второй вариант построения и решения лабиринта

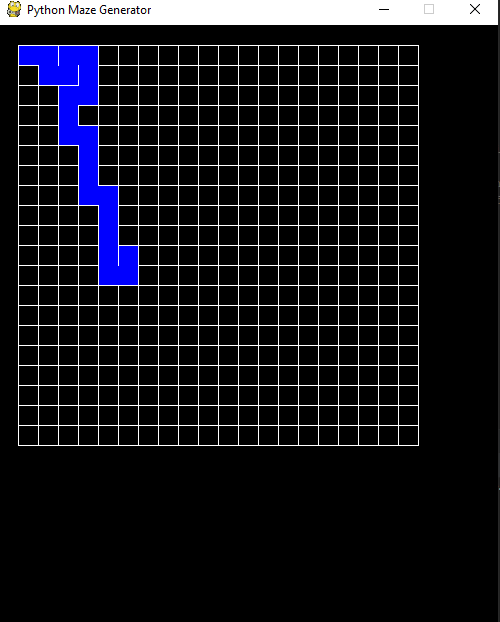


Рисунок 12 - Поклеточный поиск кратчайшего пути

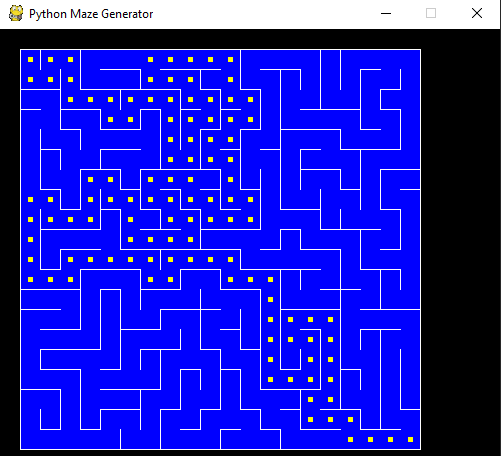


Рисунок 13 - Итоговый результат

# **ЗАКЛЮЧЕНИЕ**

В данной работе была реализован алгоритм А\*, а также рассмотрена возможность его применения для задачи поиска оптимального пути в неизвестной местности . После построения и генерации лабиринта, алгоритм А\* решает задачу ,при этом имеется возможность управлять генерацией лабиринта с помощью указания дополнительных условий генератора, например: указание размерности лабиринта.

По сравнению с результатами других похожих работ, в которых рассматривалась такая же задача, здесь были перечислены и другие варианты решения этой задачи. Данная модификация позволила шире взглянуть на процесс условной генерации, благодаря чему стало возможным смешивание условий для получение лучших результатов. Поставленную цель работы можно считать выполненной, также были выполнены все поставленные задачи в данной работе.

Однако полученный результат всё ещё не доходит до качества реальных алгоритмов обработки местности и прокладывания пути. Для повышения оптимальности алгоритма возможно сочетание с другими способами решения.

Таким образом, по данной теме возможны дальнейшие исследования с целью создания новых и более совершенных методов решения задач для БАС.

За время выполнения курсовой работы были реализованы следующие компетенции:

| Шифр компетенции | Расшифровка приобретаемой компетенции | Расшифровка освоения компетенции |
| --- | --- | --- |
| ОПК-2 | Способность применять в профессиональной деятельности современные языки программирования и языки баз данных, методологии системной инженерии, системы автоматизации проектирования, электронные библиотеки и коллекции, сетевые технологии, библиотеки и пакеты программ, современные профессиональные стандарты информационных технологий | Освоил Python 3 и встроенные в оболочку библиотеки такие как :  -Tkinter  -Pygame  -Numpy |
| ОПК-3 | Способность к разработке алгоритмических и программных решений в области системного и прикладного программирования, математических, информационных и имитационных моделей, созданию информационных ресурсов, глобальных сетей, образовательного контента, прикладных баз данных, тестов и средств тестирования систем и средств на соответствие стандартам и исходным требованиям | Освоил различные подходы генераций лабиринтов и различные способы решения  Были проведены тесты на разные виды лабиринтов и возможности их обхода |
| ПК-1 | Способность собирать, обрабатывать и интерпретировать данные современных научных исследований, необходимые для формирования выводов по соответствующим научным исследованиям | Было рассмотрено огромное количество различных ресурсов по разработке алгоритмов |
| ПК-2 | Способность понимать, совершенствовать и применять современный математический аппарат, фундаментальные концепции и системные методологии, международные и профессиональные стандарты в области информационных технологий | Были рассмотрены различные математические подходы решения задачи такие как градиентные и стохастические способы |
| ПК-3 | Способность использовать современные инструментальные и вычислительные средства | Использовалась последняя версия языка Python – 3.8 |

# **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ**

1. Джон Пол Мюллер, Лука Массарон. [Текст] Алгоритмы для чайников . – 2017. — №98. –С. 85 – 94. (дата обращения 09.05.2020)
2. Deep Learning Makes Driverless Cars Better at Spotting Pedestrians (англ.). IEEE Spectrum [Электронный ресурс] : Technology, Engineering, and Science News (дата обращения 10.05.2020)
3. Davies, Alex. Everyone Wants a Level 5 Self-Driving Car [Электронный ресурс] —Here’s What That Means (англ.), WIRED. (дата обращения 10.05.2020)
4. Pycharm [Электронный ресурс] – Режим доступа: https://ru.wikipedia.org/wiki/PyCharm (Дата обращения 03.05.2020)
5. Джордж Хайнеман, Гари Поллис, Стэнли Селков [Текст] – Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2017 –С. 144 – 187. (дата обращения 10.05.2020)
6. Wilson, David Bruce (May 22–24, 1996). "Generating random spanning trees more quickly than the cover time". Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing. Symposium on Theory of Computing. Philadelphia: ACM. [Электронный ресурс] pp. 296–303. (дата обращения 10.05.2020)

# 

# **ПРИЛОЖЕНИЕ**

Основной пример построения лабиринта и его решения

Название файла – **Main.py**

import numpy as np  
import random  
from tkinter import \*  
  
lab = None  
ver\_walls = None  
hor\_walls = None  
size\_win = 800  
closed = 0  
opened = 1  
path = 2  
  
def contains\_only\_zeros(matrix):  
 for y in range(0, matrix.shape[0]):  
 for x in range(0, matrix.shape[0]):  
 if matrix.item(x, y) != 0:  
 return False  
 return True  
  
def open\_entrees(size):  
 global ver\_walls, hor\_walls  
  
 if random.random() < 0.5:  
 # Open top-bot  
 cell\_top = random.randint(0, size - 1)  
 cell\_bot = random.randint(0, size - 1)  
 hor\_walls.itemset((0, cell\_top), opened)  
 hor\_walls.itemset((size, cell\_bot), opened)  
 else:  
   
 cell\_left = random.randint(0, size - 1)  
 cell\_right = random.randint(0, size - 1)  
 ver\_walls.itemset((cell\_left, 0), opened)  
 ver\_walls.itemset((cell\_right, size), opened)  
  
  
# Инициализируем листы  
def init\_matrix(size):  
 global lab, hor\_walls, ver\_walls  
  
 # Init values of lab and the other two walls matrix  
 lab = np.fromfunction(lambda i, j: j + size \* i, (size, size), dtype=int)  
 ver\_walls = np.zeros((size, size + 1), dtype=np.int)  
 hor\_walls = np.zeros((size + 1, size), dtype=np.int)  
  
 open\_entrees(size)  
  
def walls\_splitting():  
 global lab  
   
 res = ([], [])  
  
 max\_size = lab.shape[0]  
  
 for j in range(0, max\_size):  
 for i in range(1, max\_size):  
 if lab.item(j, i) != lab.item(j, i - 1):  
 couple = (i, j)  
 res[0].append(couple)  
  
 for j in range(1, max\_size):  
 for i in range(0, max\_size):  
 if lab.item(j, i) != lab.item(j - 1, i):  
 couple = (i, j)  
 res[1].append(couple)  
  
 return res  
  
def set\_path\_min\_value(c1, c2):  
 global lab  
  
 value = min(lab.item(c1[1], c1[0]), lab.item(c2[1], c2[0]))  
 old = max(lab.item(c1[1], c1[0]), lab.item(c2[1], c2[0]))  
  
 for j in range(0, lab.shape[1]):  
 for i in range(0, lab.shape[0]):  
 if lab.item(j, i) == old:  
 lab.itemset((j, i), value)  
  
  
def open\_next\_wall():  
 global lab, hor\_walls, ver\_walls  
  
 walls = walls\_splitting()  
 vertical\_walls = walls[0]  
 horizontal\_walls = walls[1]  
  
 if (random.random() < 0.5 and vertical\_walls) or not horizontal\_walls:  
 number = random.randint(0, len(vertical\_walls) - 1)  
  
 x = vertical\_walls[number][0]  
 y = vertical\_walls[number][1]  
 x2 = x - 1  
 y2 = y  
 ver\_walls.itemset((y, x), opened)  
 else:  
 number = random.randint(0, len(horizontal\_walls) - 1)  
  
 x = horizontal\_walls[number][0]  
 y = horizontal\_walls[number][1]  
 x2 = x  
 y2 = y - 1  
 hor\_walls.itemset((y, x), opened)  
  
 c1 = (x, y)  
 c2 = (x2, y2)  
  
 set\_path\_min\_value(c1, c2)  
  
  
  
def draw\_lab():  
 global lab, ver\_walls, hor\_walls, size\_win  
  
 w = Tk()  
  
 width\_l = size\_win / 100  
 size\_mat = lab.shape[0]  
 size\_line = size\_win / size\_mat  
  
 canvas = Canvas(w, width=size\_win, height=size\_win, background='white')  
  
 # We draw the path if exists  
 for j in range(0, size\_mat):  
 for i in range(0, size\_mat):  
 if lab.item(i, j) == path:  
 # Path  
 canvas.create\_rectangle(j\*size\_line+(width\_l/2)-1, i\*size\_line+(width\_l/2)-1,  
 (j+1)\*size\_line-(width\_l/2), (i+1)\*size\_line-(width\_l/2),  
 fill='#FAAD32', outline="white")  
 # We complete vertical white spaces  
 if i > 0 and lab.item(i-1, j) == path:  
 canvas.create\_line(j \* size\_line + (width\_l / 2), i \* size\_line,  
 j \* size\_line + size\_line - (width\_l / 2), i \* size\_line,  
 width=width\_l+2, fill="#FAAD32")  
 # We complete horizontal white spaces  
 if j > 0 and lab.item(i, j-1) == path:  
 canvas.create\_line(j \* size\_line, i \* size\_line + (width\_l / 2), j \* size\_line,  
 i \* size\_line + size\_line - (width\_l / 2),  
 width=width\_l+2, fill="#FAAD32")  
 # Start in green  
 if (i, j) == get\_start():  
 canvas.create\_rectangle(j \* size\_line + (width\_l / 2)-2, i \* size\_line + (width\_l / 2)-2,  
 (j + 1) \* size\_line - (width\_l / 2)+2,  
 (i + 1) \* size\_line - (width\_l / 2)+2,  
 fill='green', outline="white")  
 # End in red  
 elif (i, j) == get\_exit():  
 canvas.create\_rectangle(j \* size\_line + (width\_l / 2)-2, i \* size\_line + (width\_l / 2)-2,  
 (j + 1) \* size\_line - (width\_l / 2 + 1)+2,  
 (i + 1) \* size\_line - (width\_l / 2 + 1)+2,  
 fill='red', outline="white")  
  
 # We draw vertical lines  
 for j in range(0, size\_mat + 1):  
 for i in range(0, size\_mat):  
 if ver\_walls.item(i, j) == closed:  
 canvas.create\_line(j \* size\_line, i \* size\_line - (width\_l / 2), j \* size\_line,  
 i \* size\_line + size\_line + (width\_l / 2),  
 width=width\_l)  
  
 # We draw horizontal lines  
 for j in range(0, size\_mat):  
 for i in range(0, size\_mat + 1):  
 if hor\_walls.item(i, j) == closed:  
 canvas.create\_line(j \* size\_line - (width\_l / 2), i \* size\_line,  
 j \* size\_line + size\_line + (width\_l / 2), i \* size\_line,  
 width=width\_l)  
  
 canvas.pack()  
 w.mainloop()  
  
  
def get\_start():  
 global lab, ver\_walls, hor\_walls  
  
 size\_mat = lab.shape[0]  
  
 for y in range(0, size\_mat):  
 if hor\_walls.item(0, y) == opened:  
 return 0, y  
  
 for x in range(0, size\_mat):  
 if ver\_walls.item(x, 0) == opened:  
 return x, 0  
  
 return None  
  
  
def get\_exit():  
 global lab, ver\_walls, hor\_walls  
  
 size\_mat = lab.shape[0]  
  
 for y in range(0, size\_mat):  
 if hor\_walls.item(size\_mat, y) == opened:  
 return size\_mat-1, y  
  
 for x in range(0, size\_mat):  
 if ver\_walls.item(x, size\_mat) == opened:  
 return x, size\_mat-1  
  
 return None  
  
  
def is\_exit(x, y):  
 global lab, ver\_walls, hor\_walls  
  
 size\_mat = lab.shape[0]  
  
 if x == size\_mat - 1 and hor\_walls.item(x + 1, y) == opened:  
 return True  
 elif y == size\_mat - 1 and ver\_walls.item(x, y + 1) == opened:  
 return True  
  
 return False  
  
  
def contains\_exit(x, y, di):  
 global lab, ver\_walls, hor\_walls  
  
 size\_mat = lab.shape[0]  
  
 if x < 0 or y < 0:  
 return False  
 elif x >= size\_mat or y >= size\_mat:  
 return False  
  
 elif is\_exit(x, y):  
 return True  
  
 elif di == "Top":  
 if x - 1 < 0:  
 return False  
 if hor\_walls.item(x, y) == opened and contains\_exit(x - 1, y, "Top"):  
 return True  
 if hor\_walls.item(x, y) == opened and contains\_exit(x - 1, y, "Left"):  
 return True  
 if hor\_walls.item(x, y) == opened and contains\_exit(x - 1, y, "Right"):  
 return True  
  
 elif di == "Bot":  
 if x + 1 >= size\_mat:  
 return False  
 if hor\_walls.item(x + 1, y) == opened and contains\_exit(x + 1, y, "Bot"):  
 return True  
 if hor\_walls.item(x + 1, y) == opened and contains\_exit(x + 1, y, "Left"):  
 return True  
 if hor\_walls.item(x + 1, y) == opened and contains\_exit(x + 1, y, "Right"):  
 return True  
  
 elif di == "Left":  
 if y - 1 < 0:  
 return False  
 if ver\_walls.item(x, y) == opened and contains\_exit(x, y - 1, "Top"):  
 return True  
 if ver\_walls.item(x, y) == opened and contains\_exit(x, y - 1, "Bot"):  
 return True  
 if ver\_walls.item(x, y) == opened and contains\_exit(x, y - 1, "Left"):  
 return True  
  
 elif di == "Right":  
 if y + 1 >= size\_mat:  
 return False  
 if ver\_walls.item(x, y + 1) == opened and contains\_exit(x, y + 1, "Top"):  
 return True  
 if ver\_walls.item(x, y + 1) == opened and contains\_exit(x, y + 1, "Bot"):  
 return True  
 if ver\_walls.item(x, y + 1) == opened and contains\_exit(x, y + 1, "Right"):  
 return True  
  
 return False  
  
  
def get\_exit\_path(x, y):  
 global lab, hor\_walls, ver\_walls  
  
 lab.itemset(x, y, path)  
  
 # ввыходим здесь  
 if is\_exit(x, y):  
 return [(x, y)]  
  
 # We are in the labyrinth  
 if contains\_exit(x, y, "Top"):  
 next\_path = get\_exit\_path(x - 1, y)  
 elif contains\_exit(x, y, "Bot"):  
 next\_path = get\_exit\_path(x + 1, y)  
 elif contains\_exit(x, y, "Left"):  
 next\_path = get\_exit\_path(x, y - 1)  
 else:  
 next\_path = get\_exit\_path(x, y + 1)  
  
 return [(x, y)] + next\_path  
  
  
#   
def main():  
 global lab, hor\_walls, ver\_walls  
  
   
 size = 30  
  
 init\_matrix(size)  
  
 i = 0  
 while not (contains\_only\_zeros(lab)):  
 print(i)  
 open\_next\_wall()  
 i += 1  
  
 print(hor\_walls)  
 print()  
 print(ver\_walls)  
 print()  
 print(lab)  
  
 draw\_lab()  
  
 start = get\_start()  
 get\_exit\_path(start[0], start[1])  
  
 draw\_lab()  
  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()

Второй вариант решения этой задачи с визуализацией и использованием бибилиотеки – pygame

Название файла – **Anothervar.py**

import pygame

import time  
import random  
  
# set up pygame window  
WIDTH = 500  
HEIGHT = 600  
FPS = 30  
  
# Define colours  
WHITE = (255, 255, 255)  
GREEN = (0, 255, 0,)  
BLUE = (0, 0, 255)  
YELLOW = (255 ,255 ,0)  
  
# initalise Pygame  
pygame.init()  
pygame.mixer.init()  
screen = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption("Python Maze Generator")  
clock = pygame.time.Clock()  
  
# setup maze variables  
x = 0 # x axis  
y = 0 # y axis  
w = 20 # width of cell  
grid = []  
visited = []  
stack = []  
solution = {}  
  
  
# build the grid  
def build\_grid(x, y, w):  
 for i in range(1,21):  
 x = 20 # set x coordinate to start position  
 y = y + 20 # start a new row  
 for j in range(1, 21):  
 pygame.draw.line(screen, WHITE, [x, y], [x + w, y]) # top of cell  
 pygame.draw.line(screen, WHITE, [x + w, y], [x + w, y + w]) # right of cell  
 pygame.draw.line(screen, WHITE, [x + w, y + w], [x, y + w]) # bottom of cell  
 pygame.draw.line(screen, WHITE, [x, y + w], [x, y]) # left of cell  
 grid.append((x,y)) # add cell to grid list  
 x = x + 20 # move cell to new position  
def push\_up(x, y):  
 pygame.draw.rect(screen, BLUE, (x + 1, y - w + 1, 19, 39), 0) # draw a rectangle twice the width of the cell  
 pygame.display.update() # to animate the wall being removed  
  
  
def push\_down(x, y):  
 pygame.draw.rect(screen, BLUE, (x + 1, y + 1, 19, 39), 0)  
 pygame.display.update()  
  
  
def push\_left(x, y):  
 pygame.draw.rect(screen, BLUE, (x - w +1, y +1, 39, 19), 0)  
 pygame.display.update()  
  
  
def push\_right(x, y):  
 pygame.draw.rect(screen, BLUE, (x +1, y +1, 39, 19), 0)  
 pygame.display.update()  
  
  
def single\_cell( x, y):  
 pygame.draw.rect(screen, GREEN, (x +1, y +1, 18, 18), 0) # draw a single width cell  
 pygame.display.update()  
  
  
def backtracking\_cell(x, y):  
 pygame.draw.rect(screen, BLUE, (x +1, y +1, 18, 18), 0) # used to re-colour the path after single\_cell  
 pygame.display.update() # has visited cell  
  
  
def solution\_cell(x,y):  
 pygame.draw.rect(screen, YELLOW, (x+8, y+8, 5, 5), 0) # used to show the solution  
 pygame.display.update() # has visited cell  
  
  
def carve\_out\_maze(x,y):  
 single\_cell(x, y) # starting positing of maze  
 stack.append((x,y)) # place starting cell into stack  
 visited.append((x,y)) # add starting cell to visited list  
 while len(stack) > 0: # loop until stack is empty  
 time.sleep(.07) # slow program now a bit  
 cell = [] # define cell list  
 if (x + w, y) not in visited and (x + w, y) in grid: # right cell available?  
 cell.append("right") # if yes add to cell list  
  
 if (x - w, y) not in visited and (x - w, y) in grid: # left cell available?  
 cell.append("left")  
  
 if (x , y + w) not in visited and (x , y + w) in grid: # down cell available?  
 cell.append("down")  
  
 if (x, y - w) not in visited and (x , y - w) in grid: # up cell available?  
 cell.append("up")  
  
 if len(cell) > 0: # check to see if cell list is empty  
 cell\_chosen = (random.choice(cell)) # select one of the cell randomly  
  
 if cell\_chosen == "right": # if this cell has been chosen  
 push\_right(x, y) # call push\_right function  
 solution[(x + w, y)] = x, y # solution = dictionary key = new cell, other = current cell  
 x = x + w # make this cell the current cell  
 visited.append((x, y)) # add to visited list  
 stack.append((x, y)) # place current cell on to stack  
  
 elif cell\_chosen == "left":  
 push\_left(x, y)  
 solution[(x - w, y)] = x, y  
 x = x - w  
 visited.append((x, y))  
 stack.append((x, y))  
  
 elif cell\_chosen == "down":  
 push\_down(x, y)  
 solution[(x , y + w)] = x, y  
 y = y + w  
 visited.append((x, y))  
 stack.append((x, y))  
  
 elif cell\_chosen == "up":  
 push\_up(x, y)  
 solution[(x , y - w)] = x, y  
 y = y - w  
 visited.append((x, y))  
 stack.append((x, y))  
 else:  
 x, y = stack.pop() # if no cells are available pop one from the stack  
 single\_cell(x, y) # use single\_cell function to show backtracking image  
 time.sleep(.05) # slow program down a bit  
 backtracking\_cell(x, y) # change colour to green to identify backtracking path  
  
  
def plot\_route\_back(x,y):  
 solution\_cell(x, y) # solution list contains all the coordinates to route back to start  
 while (x, y) != (20,20): # loop until cell position == start position  
position == start position  
 x, y = solution[x, y] # "key value" now becomes the new key  
 solution\_cell(x, y) # animate route back  
 time.sleep(.1)  
  
  
x, y = 20, 20 # starting position of grid  
build\_grid(40, 0, 20) # 1st argument = x value, 2nd argument = y value, 3rd argument = width of cell  
carve\_out\_maze(x,y) # call build the maze function  
plot\_route\_back(400, 400) # call the plot solution function  
  
  
# ##### pygame loop #######  
running = True  
while running:  
 # keep running at the at the right speed  
 clock.tick(FPS)  
 # process input (events)  
 for event in pygame.event.get():  
 # check for closing the window  
 if event.type == pygame.QUIT:  
running = False