**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра систем автоматизированного проектирования**

Курсовая работа

«Оптимизация маршрутов с использованием алгоритма A\* (A-star)»

**по дисциплине «Алгоритмы и структуры данных»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3353 |  | Шинкарь К.Д. |
| Преподаватель |  | Пестерев Д.О. |

Санкт-Петербург

2024

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

|  |  |  |
| --- | --- | --- |
| Студентка Шинкарь К.Д. | | |
| Группа 3353 | | |
| Тема работы: Оптимизация маршрутов с использованием алгоритма A\* (A-star) | | |
| Исходные данные:  Язык программирования: Python.  Среда разработки (IDE): Pycharm | | |
| Содержание пояснительной записки:  «Оглавление», «Введение», «Теоретическая часть», «Практическая часть», «Заключение», «Коды программ» | | |
| Предполагаемый объем пояснительной записки:  Не менее 20 страниц. | | |
| Дата выдачи задания: 03.12.2024 | | |
| Дата сдачи: | | |
| Дата защиты реферата: | | |
| Студентка |  | Шинкарь К.Д. |
| Преподаватель |  | Пестерев Д.О. |

**АННОТАЦИЯ**

В курсовой работе рассматривается алгоритм A\*, используемый для оптимизации маршрутов и поиска кратчайшего пути в графе. Алгоритм A\* сочетает в себе принципы поиска в ширину и эвристические методы, что позволяет эффективно находить оптимальные маршруты даже в сложных условиях. В работе уделено внимание теории оптимизации маршрутов, ключевым функциям алгоритма A\* (g, h и f), а также рассматривается манхэттенская эвристика, наиболее подходящая для работы с лабиринтами. Принцип работы алгоритма A\* проиллюстрирован на примерах, а также рассмотрено его применение в реальных задачах. Практическая часть включает реализацию алгоритма с использованием библиотек Python, таких как pygame, heapq и random, для визуализации и тестирования поиска путей в лабиринте.

**SUMMARY**

The coursework explores the A\* algorithm, which is used for route optimization and finding the shortest path in a graph. The A\* algorithm combines the principles of breadth-first search and heuristic methods, allowing it to efficiently find optimal routes even in complex conditions. The work focuses on the theory of route optimization, the key functions of the A\* algorithm (g, h, and f), and the Manhattan heuristic, which is particularly suitable for working with mazes. The principles of the A\* algorithm are illustrated with examples, and its application to real-world tasks is discussed. The practical part includes the implementation of the algorithm using Python libraries such as pygame, heapq, and random for visualizing and testing pathfinding in a maze.

Оглавление

[**Введение** 5](#_Toc184950587)

[**Теоретическая часть** 6](#_Toc184950588)

[**Оптимизация маршрутов** 6](#_Toc184950589)

[**Алгоритм A\*** 6](#_Toc184950590)

[**Эвристика** 6](#_Toc184950591)

[**Типы эвристик** 7](#_Toc184950592)

[**Манхэттенская эвристика** 7](#_Toc184950593)

[**Принцип работы алгоритма A\*** 8](#_Toc184950594)

[**Практическая часть** 10](#_Toc184950595)

[**Используемые библиотеки** 10](#_Toc184950596)

[**Используемые типы данных** 11](#_Toc184950597)

[**Результат работы программы:** 12](#_Toc184950598)

[**Заключение** 14](#_Toc184950599)

[**Код программы** 15](#_Toc184950600)

[**Ссылка на репозиторий** 20](#_Toc184950601)

# **Введение**

Оптимизация маршрутов играет ключевую роль во многих областях жизни людей, поскольку позволяет находить наилучшие пути для перемещения тех или иных объектов. Правильно построенные маршруты помогают существенно экономить время и ресурсы.

Для эффективной оптимизации маршрутов необходимы специальные алгоритмы, и одним из самых популярных является алгоритм A\*. Этот алгоритм сочетает в себе преимущества поиска в ширину и эвристических методов, что помогает находить кратчайший путь с наименьшими затратами времени. Он позволяет строить маршруты с учётом препятствий и меняющихся условий, что делает его незаменимым инструментом для решения задач, связанных с оптимизацией перемещений.

В отличие от простых методов поиска, A\* не просто перебирает все возможные варианты, а оценивает каждый из них, учитывая, как уже пройденный путь, так и предполагаемое расстояние до цели. Это делает его особенно эффективным в случаях, когда нужно найти оптимальный маршрут среди множества возможных вариантов.

# **Теоретическая часть**

## **Оптимизация маршрутов**

**Оптимизация маршрутов** — это процесс поиска наилучшего пути для перемещения между заданными точками с учётом различных условий и ограничений. Целью является минимизация времени, расстояния или затрат на путь, а также эффективное использование ресурсов.

## **Алгоритм A\***

**Алгоритм A\*** — один из самых эффективных алгоритмов поиска кратчайшего пути на графе. Он широко применяется в системах навигации, компьютерных играх и робототехнике. A\* использует эвристические оценки для ускорения поиска, что позволяет ему находить оптимальный маршрут быстрее, чем многие другие алгоритмы.

Алгоритм A\* использует три ключевые функции для оценки стоимости пути:

**g(n)** — стоимость пути от начальной точки до текущей вершины n. Она показывает, сколько уже "потрачено", чтобы достичь текущей вершины.

**h(n)** — эвристическая оценка стоимости от текущей вершины n до целевой вершины. Она показывает, сколько "ещё предстоит потратить", чтобы достичь цели. Эвристика должна быть быстрой в вычислении и давать приближение к реальному расстоянию.

**f(n)** — полная стоимость пути через вершину n. Она определяется как:

f(n) = g(n) + h(n)

## **Эвристика**

Эвристика в контексте алгоритма A\* — это функция, которая оценивает, насколько "близки" два состояния. В задаче поиска пути эвристика помогает определить, как далеко находится текущая позиция от цели.

**Типы эвристик**

Существует несколько типов эвристик, которые могут быть использованы в алгоритме A\*. Вот наиболее популярные:

1. Манхэттенская эвристика (Manhattan Distance)
2. Евклидова эвристика (Euclidean Distance)
3. Диагональная эвристика (Diagonal Distance)

Каждый тип эвристики подходит для разных типов задач. В контексте лабиринта, где движения ограничены только горизонтальными и вертикальными перемещениями, манхэттенская эвристика подойдет лучше всего.

**Манхэттенская эвристика**

Манхэттенское расстояние — это сумма абсолютных разностей по осям X и Y между двумя точками:

h(n) = ∣x1 − x2∣ + ∣y1 − y2∣

x1, y1​ — координаты текущего узла.

x2, y2 — координаты целевого узла.

Эвристика вычисляет, сколько шагов по горизонтали и вертикали нужно сделать, чтобы добраться до целевой точки, не учитывая препятствия.

Эвристика позволяет A\* быть более эффективным, чем обычный алгоритм поиска в ширину.

Используя эвристику, A\* выбирает те узлы, которые кажутся более перспективными для достижения цели, и игнорирует менее обещающие пути, что значительно ускоряет выполнение алгоритма.

## **Принцип работы алгоритма A\***

* 1. Начинаем с начальной вершины (точка старта).
  2. Устанавливаем g = 0 для начальной вершины и вычисляем f для неё.
  3. Добавляем начальную вершину в **открытый список** — это список вершин, которые нужно проверить.
  4. Создаём **закрытый список** для уже проверенных вершин.

**Открытый список** — это список вершин, которые **ещё предстоит обработать**. В этом списке находятся вершины, которые были найдены алгоритмом, но для которых ещё не были рассмотрены все возможные пути.

На каждом шаге из открытого списка выбирается вершина с **наименьшим значением** f(n).

Выбранная вершина извлекается из открытого списка и обрабатывается.

Если у вершины есть соседи, которые ещё не обработаны, они добавляются в открытый список.

**Закрытый список** — это список вершин, которые **уже были обработаны**. В этом списке находятся вершины, для которых алгоритм уже нашёл наилучший путь и которые больше не нужно проверять.

После того как вершина обрабатывается и все её соседи добавлены в открытый список, эта вершина переносится из открытого списка в закрытый.

Если алгоритм натыкается на вершину, которая уже есть в закрытом списке, её можно пропустить, так как она уже была обработана.

**Принцип поиска пути**:

Пока открытый список не пуст, выполняем следующие шаги:

1. **Выбираем вершину** из открытого списка с минимальным значением f.
2. **Е**сли эта вершина является целевой, поиск завершён, и мы восстанавливаем путь, пройдя назад от цели к началу через родителей вершин.
3. **Переносим текущую вершину** из открытого списка в закрытый список.
4. **Рассматриваем всех соседей** текущей вершины:

Если сосед уже в закрытом списке, пропускаем его.

Вычисляем g для соседа (стоимость пути от начальной вершины до этого соседа).

Вычисляем f = g + h

Если соседа ещё нет в открытом списке, добавляем его туда и сохраняем текущую вершину как "родителя" соседа.

Если сосед уже в открытом списке, но новый путь короче предыдущего (g меньше), обновляем информацию о родителе и стоимости.

1. Если целевая вершина найдена, восстанавливаем путь от цели к началу, следуя по "родителям" вершин.
2. Если открытый список стал пустым до достижения цели, значит пути не существует.

# **Практическая часть**

## **Используемые библиотеки**

**pygame** — это библиотека для создания игр и мультимедийных приложений на Python. Она предоставляет инструменты для работы с графикой, звуком, событиями, анимацией и пользовательским вводом. Она понадобилась для работы с окнами, графикой, отслеживанием кликов.

**random** — стандартный модуль Python для генерации случайных чисел и случайного выбора элементов из последовательностей. Полезен в играх, моделировании, тестировании и алгоритмах. Пригодился для генерации случайных лабиринтов.

**heapq** — это модуль для работы с **кучами**, которые являются структурами данных для приоритетных очередей. В Python реализована **минимальная куча**, где корневой элемент — наименьший.

В коде программы библиотека heapq используется для реализации очереди с приоритетом в алгоритме **A**\*, пример ниже.

**Добавление узла в очередь с приоритетом:**

heapq.heappush(open\_set, (0, start))

Функция heapq.heappush() добавляет узел в очередь open\_set. Узел представляется в виде кортежа (f\_score, node), где f\_score — оценка стоимости пути, а node — координаты узла. Это позволяет хранить узлы в порядке возрастания значений f\_score.

**Извлечение узла с минимальным значением f\_score:**

current = heapq.heappop(open\_set)[1]

Функция heapq.heappop() извлекает элемент с наименьшим f\_score из очереди open\_set.

Поскольку heappop возвращает кортеж (f\_score, node), используется [1] для получения только узла current.

**Добавление соседнего узла при обновлении оценок:**

heapq.heappush(open\_set, (f\_score[neighbor], neighbor))

Если соседний узел имеет улучшенный (меньший) g\_score, он добавляется в очередь с новым значением f\_score.

## **Используемые типы данных**

int:

Целочисленный тип данных.

Целые числа в программе используются для хранения размеров, координат и индексов.

tuple (кортежи):

Неизменяемый тип данных.

Используется для хранения упорядоченной последовательности элементов (координаты точек старта, цели и соседних клеток.).

list (списки):

Упорядоченный набор значений, в котором некоторое значение может встречаться более одного раза

В программе списки используются для хранения лабиринта, пути и узлов.

set (множества):

Хранят ограниченное число значений определённого типа без определённого порядка.

Множества используются для хранения уникальных элементов, например, посещённых узлов.

dict (словарь):

Представляет собой коллекцию пар «ключ-значение».

Словари используются для хранения путей, стоимости узлов и функций.

bool:

Используется для логических условий.

# **Результат работы программы:**

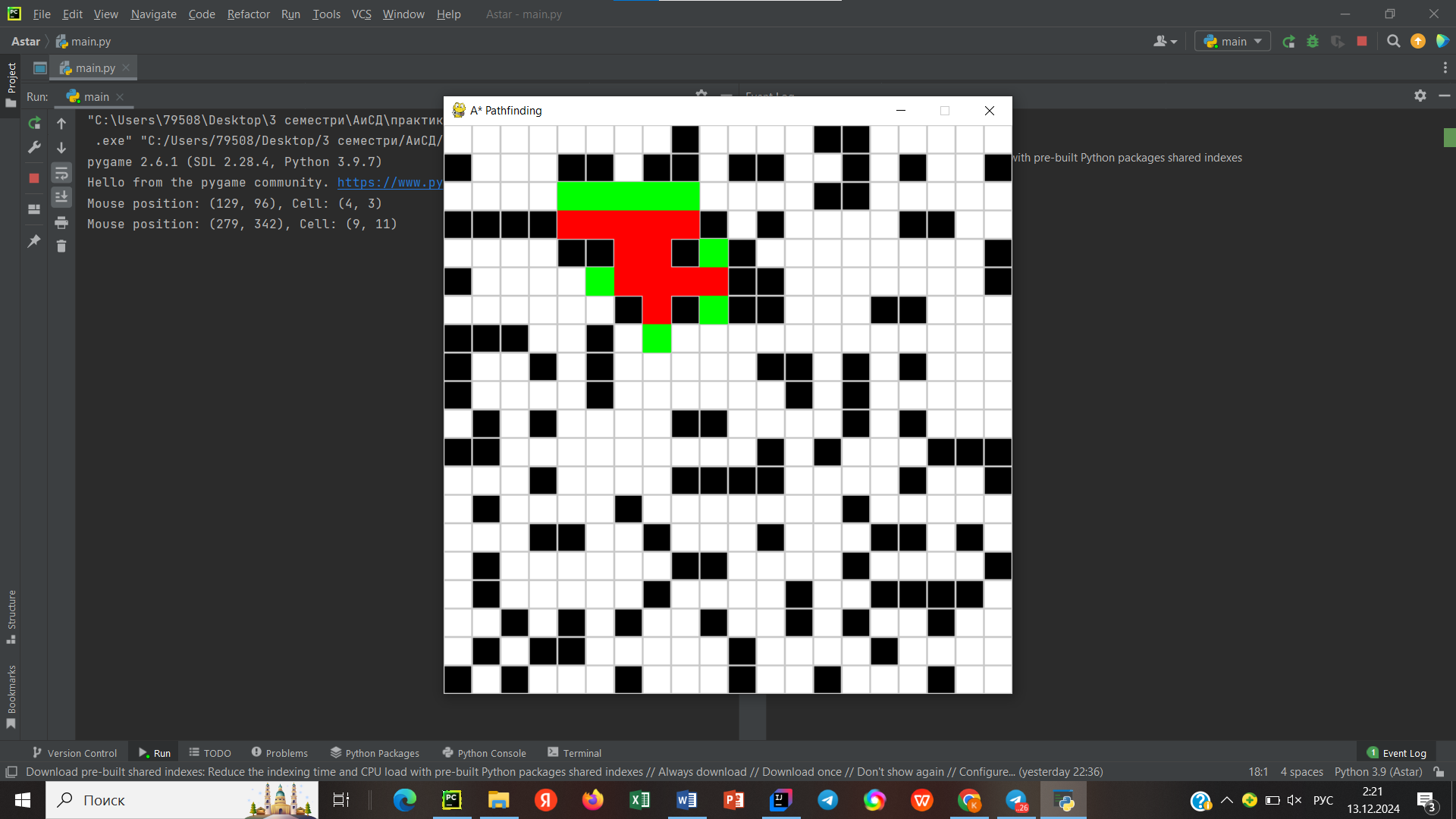
Процесс поиска наименьшего пути:

Белые клетки – пустые поля.

Черные клетки – препятствия.

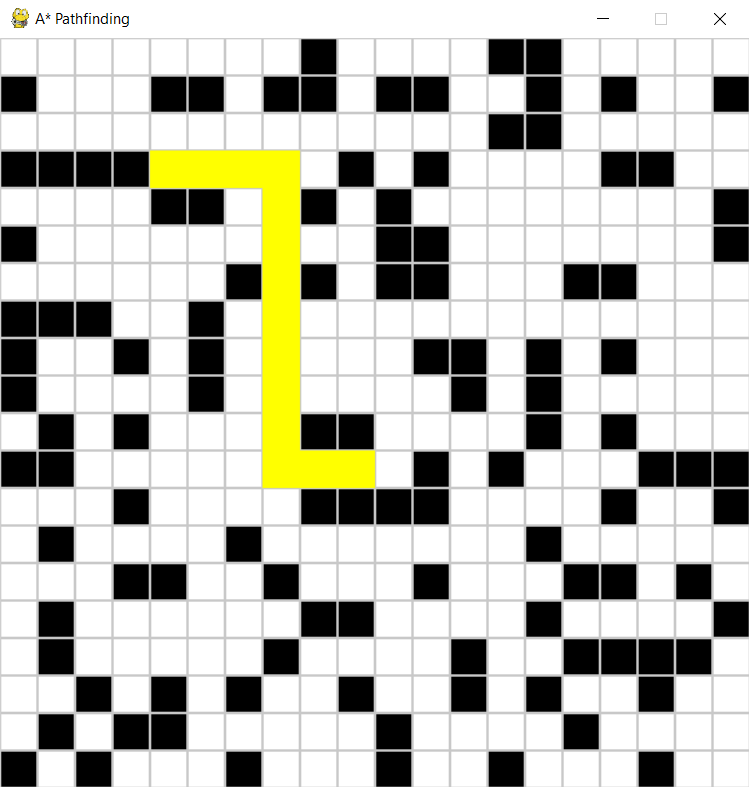
Красные клетки – вершины в закрытом списке.

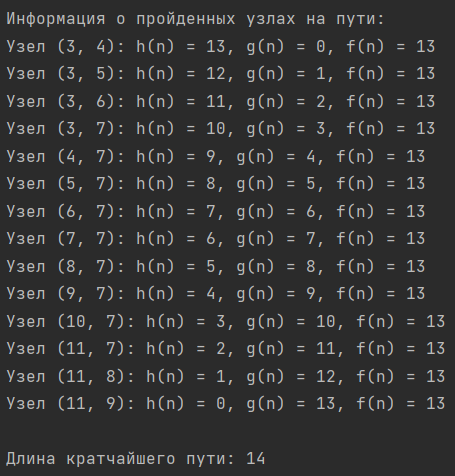
Зеленые клетки – вершины в открытом списке.



Результат работы программы:

Кратчайший путь выглядит следующим образом:





# **Заключение**

В ходе выполнения курсовой работы был реализован алгоритм A\* на Python c графическим выводом процесса поиска самого короткого пути от начальной точки до конечной. Предварительная оценка не всегда совпадает с итоговой из-за особенностей расчета эвристической функции и расположения препятствий в лабиринте, но алгоритм работает быстро и точно, что подтверждает его эффективность.

# **Код программы**

import pygame  
import random  
import heapq  
  
# Константы  
WIDTH, HEIGHT = 600, 600  
GRID\_SIZE = 20  
CELL\_SIZE = WIDTH // GRID\_SIZE  
  
# Цвета  
WHITE = (255, 255, 255)  
BLACK = (0, 0, 0)  
RED = (255, 0, 0)  
GREEN = (0, 255, 0)  
BLUE = (0, 0, 255)  
YELLOW = (255, 255, 0)  
GRAY = (200, 200, 200)  
  
# Инициализация Pygame  
pygame.init()  
screen = pygame.display.set\_mode((WIDTH, HEIGHT))  
pygame.display.set\_caption("A\* Pathfinding")  
clock = pygame.time.Clock()  
  
# Функция для генерации лабиринта  
def generate\_maze(grid\_size):  
 maze = [[1 if random.random() < 0.3 else 0 for \_ in range(grid\_size)] for \_ in range(grid\_size)]  
 maze[0][0] = maze[grid\_size - 1][grid\_size - 1] = 0 # Гарантируем начало и конец  
 return maze  
  
# Визуализация лабиринта  
def draw\_maze(maze, start=None, goal=None, path=None, open\_set=None, closed\_set=None):  
 for row in range(GRID\_SIZE):  
 for col in range(GRID\_SIZE):  
 # Цвет клетки  
 color = WHITE if maze[row][col] == 0 else BLACK  
 pygame.draw.rect(screen, color, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))  
 pygame.draw.rect(screen, GRAY, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE), 1)  
  
 # Отрисовка начальной и конечной точки  
 if start:  
 pygame.draw.rect(screen, GREEN, (start[1] \* CELL\_SIZE, start[0] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)) # start[0] - row, start[1] - col  
 if goal:  
 pygame.draw.rect(screen, RED, (goal[1] \* CELL\_SIZE, goal[0] \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE)) # goal[0] - row, goal[1] - col  
  
 # Отрисовка открытых и закрытых узлов  
 if open\_set:  
 for row, col in open\_set:  
 pygame.draw.rect(screen, GREEN, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))  
  
 if closed\_set:  
 for row, col in closed\_set:  
 pygame.draw.rect(screen, RED, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))  
  
 # Отрисовка пути  
 if path:  
 for row, col in path:  
 pygame.draw.rect(screen, YELLOW, (col \* CELL\_SIZE, row \* CELL\_SIZE, CELL\_SIZE, CELL\_SIZE))  
  
 pygame.display.flip()  
  
# Алгоритм A\*  
def a\_star(maze, start, goal):  
 def heuristic(a, b):  
 return abs(a[0] - b[0]) + abs(a[1] - b[1]) # Манхэттенская эвристика  
  
 open\_set = []  
 heapq.heappush(open\_set, (0, start))  
 came\_from = {}  
 g\_score = {start: 0}  
 f\_score = {start: heuristic(start, goal)}  
  
 open\_set\_hash = {start}  
 closed\_set = set()  
  
 while open\_set:  
 current = heapq.heappop(open\_set)[1]  
 open\_set\_hash.remove(current)  
  
 if current == goal:  
 path = []  
 node\_info = [] # Список для хранения значений h(n), g(n), f(n) для узлов на пути  
 while current in came\_from:  
 path.append(current)  
 node\_info.append((current, heuristic(current, goal), g\_score[current], f\_score[current]))  
 current = came\_from[current]  
 path.append(start)  
 node\_info.append((start, heuristic(start, goal), g\_score[start], f\_score[start]))  
 path.reverse()  
 node\_info.reverse()  
  
 # Вывод значений h(n), g(n), f(n) для узлов на пути  
 print("\nИнформация о пройденных узлах на пути:")  
 for idx, (node, h, g, f) in enumerate(node\_info):  
 print(f"Узел {node}: h(n) = {h}, g(n) = {g}, f(n) = {f}")  
 # Выводим узлы каждые 5 шагов или начальный и конечный узлы  
 #if idx % 5 == 0 or idx == len(node\_info) - 1:  
 # print(f"\*\*\* Узел {node}: h(n) = {h}, g(n) = {g}, f(n) = {f} \*\*\*")  
  
 print(f"\nДлина кратчайшего пути: {len(path)}")  
 return path, closed\_set  
  
 closed\_set.add(current)  
  
 for dx, dy in [(-1, 0), (1, 0), (0, -1), (0, 1)]:  
 neighbor = (current[0] + dx, current[1] + dy)  
 if 0 <= neighbor[0] < GRID\_SIZE and 0 <= neighbor[1] < GRID\_SIZE and maze[neighbor[0]][neighbor[1]] == 0:  
 if neighbor in closed\_set:  
 continue  
  
 tentative\_g\_score = g\_score[current] + 1  
  
 if neighbor not in open\_set\_hash or tentative\_g\_score < g\_score.get(neighbor, float('inf')):  
 came\_from[neighbor] = current  
 g\_score[neighbor] = tentative\_g\_score  
 f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, goal)  
 if neighbor not in open\_set\_hash:  
 heapq.heappush(open\_set, (f\_score[neighbor], neighbor))  
 open\_set\_hash.add(neighbor)  
  
 draw\_maze(maze, None, None, None, open\_set\_hash, closed\_set)  
 clock.tick(10)  
  
 return None, closed\_set  
  
# Главная программа  
def main():  
 maze = generate\_maze(GRID\_SIZE)  
 start = (0, 0)  
 goal = (GRID\_SIZE - 1, GRID\_SIZE - 1)  
 path = None # Путь, который будет отображаться  
  
 running = True  
 while running:  
 screen.fill(WHITE)  
 draw\_maze(maze, start, goal, path)  
  
 for event in pygame.event.get():  
 if event.type == pygame.QUIT:  
 running = False  
  
 # Ввод начальной и конечной точки  
 if event.type == pygame.MOUSEBUTTONDOWN:  
 x, y = pygame.mouse.get\_pos()  
 col, row = x // CELL\_SIZE, y // CELL\_SIZE # Вычисляем индексы клетки  
  
 print(f"Mouse position: ({x}, {y}), Cell: ({col}, {row})") # Отладочный вывод  
  
 if event.button == 1: # Левая кнопка для установки начальной точки  
 start = (row, col) # Начальная точка  
 elif event.button == 3: # Правая кнопка для установки конечной точки  
 goal = (row, col) # Конечная точка  
 elif event.button == 2: # Средняя кнопка для переключения стен  
 maze[row][col] = 1 - maze[row][col] # Переключаем стену  
 print(  
 f"Cell ({row}, {col}) changed to {'wall' if maze[row][col] == 1 else 'empty'}") # Проверка обновления  
  
 # Запуск A\* при нажатии Enter  
 if event.type == pygame.KEYDOWN and event.key == pygame.K\_RETURN:  
 path, closed\_set = a\_star(maze, start, goal)  
 draw\_maze(maze, start, goal, path, None, closed\_set)  
  
 pygame.display.flip()  
 clock.tick(30)  
  
 pygame.quit()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 main()

# **Ссылка на репозиторий**

<https://github.com/ksushinia/A-star-Algorithm>