Федеральное государственное бюджетное образовательное учреждение высшего образования «Нижегородский государственный архитектурно-строительный университет» (ННГАСУ)

*Факультет инженерно-экологических систем и сооружений*

*Кафедра информационных систем и технологий*

КУРСОВАЯ РАБОТА

по дисциплине: «Язык программирования Python»

На тему: «Реализация алгоритмов поиска пути в лабиринте»

Выполнил студент 1 курса гр. ИС-34 Шестых Д.О.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc135937714)

[Задачи 3](#_Toc135937715)

[Глава 1. Теоретическая часть 4](#_Toc135937716)

[1.1 Алгоритм Дейкстры 4](#_Toc135937717)

[1.2 Алгоритм А\* 4](#_Toc135937718)

[2. Реализация алгоритма 6](#_Toc135937719)

[2.1 Реализация алгоритма Дейкстры 6](#_Toc135937720)

[2.2 Реализация алгоритма А\* 7](#_Toc135937721)

[Пример работы 9](#_Toc135937722)

[Заключение 10](#_Toc135937723)

[Список литературы 11](#_Toc135937724)

[Приложение 1 12](#_Toc135937725)

[Листинг программы 12](#_Toc135937726)

[Приложение 2 15](#_Toc135937727)

[Представление лабиринта 15](#_Toc135937728)

# Введение

Цель данной курсовой работы - реализация алгоритма поиска пути в лабиринте. Поиск пути широко используется в робототехнике, игровой индустрии и что не менее важно в разработке искусственного интеллекта.

В данной работе мы рассмотрим два алгоритма: алгоритм Дейкстры и алгоритм А\*. Алгоритмы будут выполнены на языке программирования Python.

**Цель работы**: реализация алгоритмов (Дейкстра и А\*) для поиска пути в лабиринте.

# Задачи

* Изучить алгоритмы построения маршрута в графе;
* Выделить особенности реализации, необходимые в конкретной задаче поиска маршрута;
* Подготовить исходные данные: лабиринт, координаты точек для посещения при обходе;
* Реализовать алгоритмы с заданными параметрами;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# Глава 1. Теоретическая часть

## Алгоритм Дейкстры

Алгоритм назван в честь Дейкстры, голландского программиста Эдсгера Дейкстры, который открыл его в 1956 году и опубликовал в 1959. Он берет в качестве вводных граф и стартовый узел графа и возвращает два массива.

Алгоритм использует минимальную очередь с приоритетом, чтобы отслеживать, какой узел мы посетим следующим.

Алгоритм Дейкстры работает и с ориентированными, и с неориентированными графами. Цикличность не имеет значения, так как цикл лишь прибавит длину пути, поэтому кратчайшего пути не получится.

Корректное выполнение алгоритма зависит от нахождения путей с увеличивающейся длиной. Если у нас будут отрицательные веса, мы не сможем уверенно сказать, что все пути, которые мы рассчитаем в будущем, не окажутся короче того, который мы уже вычислили, когда вынимали узел из очереди с приоритетом. В двух словах: если у вас в графе отрицательные веса, алгоритм Дейкстры вам не подходит[1].

## 1.2 Алгоритм А\*

Алгоритм Дейкстры можно рассматривать как частный случай более общего алгоритма, называемого поиском по первому наилучшему совпадению, который к тому же может оказаться более эффективным.

Обобщением алгоритма Дейкстры является алгоритм A\*, разработанный Хартом, Нильсоном и Рафаэлем в 1968 году. A\* широко применим, и его чаще всего используют для нахождения пути в компьютерных играх. Так как алгоритм Дейкстры можно использовать для нахождения кратчайшего маршрута в дорожной сети между городами, его стали применять в сетевых протоколах маршрутизации, чья задача состоит в том, чтобы перемещать пакеты данных между узлами сети[2].

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[3].

# 2. Реализация алгоритма

## 2.1 Реализация алгоритма Дейкстры

Дан некий граф для реализации алгоритма. Надо найти кратчайший путь к вершине 5:

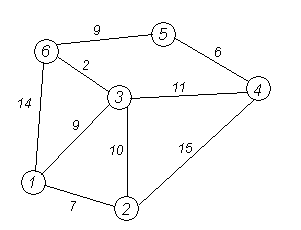


Рисунок 1. Граф

1. Создаем массив для хранения кратчайших путей.
2. Определяем множество непросмотренных вершин. Первоначально в пути содержатся все вершины графа, кроме начальной.
3. На каждом шаге из этого множества выбирается та из вершин, расстояние до которой от начальной меньше, чем для других оставшихся вершин.
4. Кратчайшие расстояния от начальной точки до соответствующей вершины находиться на изображении графа. Добавьте найденную вершину в множество посещенных вершин.
5. Далее пробуем с помощью ребер уменьшить длину пути до оставшихся непросмотренных вершин.
6. Если это удается, то массив корректируется.
7. Повторяем шаги 5 и 6.
8. После посещения всех вершин находим кратчайший путь[1].

## 2.2 Реализация алгоритма А\*

1. Создайте пустой словарь для хранения рассчитанных расстояний от начальной вершины до всех остальных вершин и инициализируйте его бесконечными значениями. Установите расстояние от начальной вершины до нее самой равным 0.
2. Создайте пустой словарь для хранения родительских вершин.
3. Создайте пустое множество для хранения посещенных вершин.
4. Создайте список открытых вершин и добавьте в него начальную вершину.
5. Начните цикл, который будет выполняться до тех пор, пока все вершины не будут посещены или не будет найден путь до конечной вершины.
6. Из открытых вершин выберите вершину с наименьшей стоимостью, которая равна сумме расстояния от начальной вершины до текущей вершины и эвристической оценки расстояния от текущей вершины до конечной вершины.
7. Если текущая вершина является конечной, то путь найден. В противном случае добавьте текущую вершину в множество посещенных.
8. Обойдите все соседние вершины текущей вершины и обновите их расстояние, если оно больше, чем расстояние от начальной вершины до текущей вершины плюс вес ребра между текущей вершиной и соседней вершиной. Если расстояние было обновлено, то добавьте соседнюю вершину в список открытых вершин и установите ее родительскую вершину.
9. Повторяйте шаги 6-8, пока все вершины не будут посещены или не будет найден путь до конечной вершины.

Если путь был найден, то по родительским вершинам можно восстановить путь от начальной вершины до конечной. Верните список вершин, составляющих путь [4].

# Пример работы

Пример можно увидеть в результате 3 лабораторной работы.Можно увидеть двумерный массив, состоящих из различных символов, где “#” - стена, “.” – путь от входа до ключа, “;” – места, где путь от входа до ключа и от ключа до входа пересекаются, “,” – путь от ключа до выхода, все остальное – элементы, куда можно двигаться. (приложение 2).

# Заключение

В ходе проделанной работы была создана программа, которая при помощи двух алгоритмов пути возвращает результат пути по координатам от точки начала до ключа и от ключа до конца лабиринта.

A\* алгоритм является разновидностью алгоритма Дейкстры, который использует оценку расстояния до целевой вершины для оптимизации процесса поиска. Таким образом, A\* может находить более оптимальные пути, чем алгоритм Дейкстры, за более короткое время.

Таким образом, если у вас есть информация о местоположении целевой вершины и оценка расстояния до нее, то A\* алгоритм будет эффективнее, чем алгоритм Дейкстры. Однако, если такой информации нет, то алгоритм Дейкстры будет более подходящим выбором.

В результате было разработана эффективная реализация алгоритмов, которые могут применяться не только в лабиринте, а еще и в графо подобных системах, где нужно найти оптимальный путь от точки А до точки Б.

# Список литературы

1. Дольников, В. Л. Основные алгоритмы на графах : текст лекций / В. Л. Дольников, О. П. Якимова; Яросл. гос. ун-т им. П. Г. Демидова. – Ярославль : ЯрГУ, 2011. – 80 с.
2. A\* [Электронный ресурс]: Википедия. Свободная энциклопедия. – Режим доступа: [https://ru.wikipedia.org/wiki/A\*](https://ru.wikipedia.org/wiki/A*) (дата обращения: 20.05.2023).
3. Луридас, Панос. Алгоритмы для начинающих: теория и практика для разработчика / Панос Луридас ; [пер. с англ. Е.М. Егоровой]. – Москва : Эксмо, 2018. – 608 с.
4. Введение в алгоритм A\* - Текст: электронный // Хабр: интернет-портал. – URL: <https://habr.com/ru/articles/331192/> (дата обращения 22.05.2023).

# Приложение 1

## Листинг программы

def av\_path(coordsXY, maze):  
 LenMazeY = len(maze[0])  
 LenMazeX = len(maze)  
 coordsX = coordsXY[0]  
 coordsY = coordsXY[1]  
 possibleWays = []  
  
 if (coordsX - 1) >= 0 and maze[coordsX - 1][coordsY] == " ":  
 coord\_for\_append = (coordsX - 1, coordsY)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsX + 1) < LenMazeX and maze[coordsX + 1][coordsY] == " ":  
 coord\_for\_append = (coordsX + 1, coordsY)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsY - 1) >= 0 and maze[coordsX][coordsY - 1] == " ":  
 coord\_for\_append = (coordsX, coordsY - 1)  
 possibleWays.append(coord\_for\_append)  
  
 if (coordsY + 1) < LenMazeY and maze[coordsX][coordsY + 1] == " ":  
 coord\_for\_append = (coordsX, coordsY + 1)  
 possibleWays.append(coord\_for\_append)  
  
 return possibleWays  
  
  
  
  
def Astar(maze, start, end):  
 open\_list = [start]  
 closed\_list = []  
 came\_from = {}  
 g\_score = {start: 0}  
 f\_score = {start: heuristic(start, end)}  
  
 while open\_list:  
 current = min(open\_list, key=lambda x: f\_score[x])  
 if current == end:  
 path = [end]  
 while current in came\_from:  
 current = came\_from[current]  
 path.append(current)  
 path.reverse()  
 return path  
  
 open\_list.remove(current)  
 closed\_list.append(current)  
  
 for neighbor in av\_path(current, maze):  
 if neighbor in closed\_list:  
 continue  
 tentative\_g\_score = g\_score[current] + 1  
 if neighbor not in open\_list or tentative\_g\_score < g\_score[neighbor]:  
 came\_from[neighbor] = current  
 g\_score[neighbor] = tentative\_g\_score  
 f\_score[neighbor] = tentative\_g\_score + heuristic(neighbor, end)  
 if neighbor not in open\_list:  
 open\_list.append(neighbor)  
 return open\_list  
  
def dijkstra(maze, start, end):  
 open\_list = [start]  
 closed\_list = []  
 came\_from = {}  
 g\_score = {start: 0}  
  
 while open\_list:  
 current = min(open\_list, key=lambda x: g\_score[x])  
 if current == end:  
 path = [end]  
 while current in came\_from:  
 current = came\_from[current]  
 path.append(current)  
 path.reverse()  
 return path  
  
 open\_list.remove(current)  
 closed\_list.append(current)  
  
 for neighbor in av\_path(current, maze):  
 if neighbor in closed\_list:  
 continue  
 tentative\_g\_score = g\_score[current] + 1  
 if neighbor not in open\_list or tentative\_g\_score < g\_score[neighbor]:  
 came\_from[neighbor] = current  
 g\_score[neighbor] = tentative\_g\_score  
 if neighbor not in open\_list:  
 open\_list.append(neighbor)  
 return open\_list  
  
  
def heuristic(current, end):  
 return abs(current[0] - end[0]) + abs(current[1] - end[1])  
  
with open('maze-for-u.txt', 'r') as f:  
 maze = [list(line.strip()) for line in f.readlines()]  
  
  
for Y in range(len(maze[0])):  
 if maze[0][Y] == " ":  
 start = (0, Y)  
 break  
  
for Y in range(len(maze[0])):  
 if maze[len(maze) - 1][Y] == " ":  
 end = (len(maze) - 1, Y)  
 break  
  
  
for i in range(len(maze)):  
 for j in range(len(maze[0])):  
 if maze[i][j] == "\*":  
 key = (i, j)  
 maze[i][j] = " "  
 break  
  
  
pathToKey = dijkstra(maze, start, key)  
pathToExit = Astar(maze, key, end)  
  
for coords in pathToKey:  
 x, y = coords  
 maze[x][y] = "."  
  
for coords in pathToExit:  
 x, y = coords  
 if maze[x][y] == ".":  
 maze[x][y] = ";"  
 else:  
 maze[x][y] = ","  
  
with open('maze-for-me-done.txt', 'w') as f:  
 for line in maze:  
 f.write("".join(line) + "\n")

# Приложение 2

## Представление лабиринта

