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 Смертин Д. А.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc133771930)

[Задачи 3](#_Toc133771931)

[1.Теоретическая часть 4](#_Toc133771932)

[Алгоритм Дейкстры 4](#_Toc133771933)

[Алгоритм А\* 5](#_Toc133771934)

[**Использование приоритетных очередей** 6](#_Toc133771935)

[2. Реализация алгоритма 7](#_Toc133771936)

[Как реализовать алгоритм Дейкстры 7](#_Toc133771937)

[Как реализовать алгоритм «A Star» 7](#_Toc133771938)

[Пример работы 9](#_Toc133771939)

[Заключение 10](#_Toc133771940)

[Список литературы 11](#_Toc133771941)

[Приложение 1 13](#_Toc133771942)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании.

Алгоритм Дейкстры используется для поиска кратчайшего пути между узлами графа. В реальных приложениях он используется для автоматического поиска направлений между физическими местоположениями, поскольку направления, которые мы получаем на Картах Google, являются примером алгоритма Дейкстры.

Алгоритм Дейкстры помогает нам найти кратчайший путь, где стоимость каждого пути не одинакова. Мы сначала имеем дело с ближайшими узлами, так что для поиска кратчайшего пути требуется очень небольшое количество шагов.

Алгоритм А\* является продвинутой формой алгоритма BFS (поиск по ширине), который сначала ищет более короткий путь, чем более длинные пути. Это полное, а также оптимальное решение для решения задач траектории и сетки.

**Цель работы:** реализовать алгоритмы обхода графа: Дейкстры и А\* для задачи поиска маршрута 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.

# 1.Теоретическая часть

## Алгоритм Дейкстры

Алгоритм Дейкстры— алгоритм поиска пути на графах, изобретённый нидерландским ученым Э. Дейкстрой в 1959 году. Находит кратчайшее расстояние от одной из вершин графа до всех остальных или до заданной конечной. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях, например, его использует протокол OSPF для устранения кольцевых маршрутов.

**Эффективность**

В области нахождения пути от конкретной вершины графа до всех его вершин алгоритм Дейкстры является лучшим на данный момент. Данный алгоритм также отлично справляется с поиском пути только между 2 вершинами на графе, в котором невозможно составить эвристическую функцию, находящую примерное расстояние между двумя его вершинами (текущей и конечной).

Алгоритм Дейкстры в python очень удобен, когда мы хотим найти кратчайшее расстояние между источником и целью. Он может работать как для направленных, так и для неориентированных графов. Ограничение этого алгоритма заключается в том, что он может давать или не давать правильный результат для отрицательных чисел. В Google Maps для поиска кратчайшего маршрута между одним источником и другим мы используем алгоритм Дейкстры.

В противном случае, алгоритм Дейкстры по всем параметрам уступает своей модификации – алгоритму A\*. Причиной различий служит то, что алгоритм Дейкстры будет проверять узлы графа равномерно в порядке удаления от начального, а A\* отдаёт предпочтения тем узлам, которые по результатам эвристических расчётов ближе к конечному узлу, а значит, с большей вероятностью, будут принадлежать конечному пути. Таким образом, за счёт использования не затратной эвристической функции, A\* будет проверять не больше, а на практике – значительно меньше узлов графа, чем алгоритм Дейкстры, а значит, будет работать быстрее. [3]

**Для чего нужен алгоритм Дейкстры:**

* автоматическое построение маршрута на онлайн-карте;
* поиск системой бронирования наиболее быстрых или дешевых билетов, в том числе с возможными пересадками;
* моделирование движения робота, который перемещается по местности;
* разработка поведения неигровых персонажей, создание игрового ИИ в GameDev;
* автоматическая обработка транспортных потоков;
* маршрутизация движения данных в компьютерной сети;
* расчет движения тока по электрическим цепям. [2]

## 1.2. Алгоритм А\*

Алгоритм A\* в Python или вообще в основном представляет собой задачу искусственного интеллекта, используемую для поиска пути (из точки A в точку B) и обхода графа. Этот алгоритм является гибким и может быть использован в широком диапазоне контекстов. Алгоритм поиска A\* использует эвристическую стоимость пути, стоимость начальной точки и конечную точку. Этот алгоритм был впервые опубликован Питером Хартом, Нильсом Нильссоном и Бертрамом Рафаэлем в 1968 году. [10]

А\* - алгоритм поиска пути использует эвристику для нахождения кратчайшего пути в графе. Каждый узел  имеет стоимость f(n), которая рассчитывается как f(n)=g(n) + h(n). В этой формуле g(n) - это фактическая стоимость узла с начала пути, а h(n) - его эвристическая стоимость для достижения цели. A\* является допустимым, что означает оно всегда находит решение с оптимальным значением.[8]

**Эффективность**

Алгоритм A\* на данный момент является оптимальным способом поиска пути между двумя точками в тех случаях, когда существует сравнительно простой эвристический метод оценки расстояния между элементами области поиска. Если такого метода не существует, A\* идентичен либо алгоритму Дейкстры в вариации для двух точек, либо волновому алгоритму в зависимости от вида области поиска.

A\* в Python-это мощный и полезный алгоритм со всем потенциалом. Однако он хорош лишь настолько, насколько хороша его эвристическая функция, которая весьма изменчива, учитывая природу проблемы. Он нашел свое применение в программных системах машинного обучения и поисковой оптимизации для разработки игр. [12]

**Приоритетная очередь в Python**

Очередь с приоритетом – это особый тип очереди в структуре данных. Как следует из названия, она сортирует элементы и удаляет их в соответствии с приоритетами. В отличие от обычной, она извлекает элемент с наивысшим приоритетом вместо следующего элемента. Приоритет отдельных элементов определяется порядком их ключей.[6]

Priority Queue возвращает элемент в порядке приоритета. Поэтому, когда мы добавляем элемент в очередь, мы также предоставляем ему приоритет. Затем мы получаем элементы, они возвращаются в порядке приоритета. Сначала возвращаются товары с более низким приоритетом.[4]

**Использование приоритетных очередей**

Очередь с приоритетом и куча, как реализация такой очереди, полезны для программ, которые включают в себя поиск некоторого экстремального элемента. Очередь с приоритетом может понадобиться для эффективного решения следующих задач:

* Получить три самых популярных поста в блоге по данным о посещениях.
* Найти скорейший способ добраться из одной точки в другую.
* Спрогнозировать на основе частоты прибытия автобусов, какой из них прибудет первым.[5]

# 2. Реализация алгоритма

## Как реализовать алгоритм Дейкстры

Каждой вершине сопоставим метку — минимальное известное расстояние от этой вершины до А. Алгоритм работает пошагово — на каждом шаге он «посещает» одну вершину и пытается уменьшать метки. Работа алгоритма завершается, когда все вершины посещены. Метка самой вершины А полагается равной 0, метки остальных вершин — бесконечности. Это отражает то, что расстояния от А до других вершин пока неизвестны. Все вершины графа помечаются как непосещённые. Если доступные вершины посещены, алгоритм завершается.

В противном случае, выбирается одна из непосещённых вершин. Для каждого соседа данной вершины, кроме отмеченных как посещённые, рассмотрим новую длину пути, равную сумме значений пути от начала до вершины (метки вершины) и длины ребра, соединяющего её с этим соседом. Если полученное значение длины меньше значения метки соседа, заменим значение метки полученным значением длины. Рассмотрев всех соседей, пометим вершину как посещенную и повторим шаг алгоритма для другой вершины.[3]

## Как реализовать алгоритм «A Star»

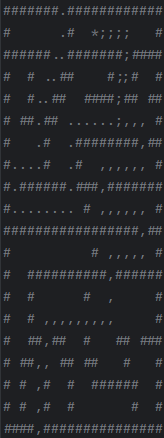
Посетив одну конкретную вершину, алгоритм «A Star» перед переходом к следующей исследует все соседние вершины. Все вершины алгоритм разделяет на 3 категории:

1. Неизвестные вершины. Это те, которые не были еще посещены и пока что даже не найдены. Получается, что и путь к ним пока остается загадкой. Таким образом, изначально все вершины, кроме стартовой, будут в этой категории.
2. Известные вершины. Это те вершины, о которых уже известно алгоритму и уже даже известен путь к ним. Такие вершины сохраняются в «списке алгоритма» и становятся в очередь для их посещения и исследования. Из этого списка исследуются те вершины, которые считаются наиболее перспективными.
3. Исследованные вершины. В эту категорию попадают те вершины, которые уже были посещены алгоритмом «A Star». К этим вершинам известен самый короткий путь, поэтому они попадают в «закрытый список» — этот список нужен для того, чтобы исключить многократное исследование одних и тех же вершин.

Когда одна из вершин становится полностью исследованной, она попадает в категорию «исследованные вершины», а все ее соседи попадают в категорию «известные вершины» и становятся годными для исследования. На каждой уже исследованной вершине устанавливается указатель до той уже исследованной вершины, к которой у нее будет кратчайший путь.

Алгоритм «A Star» завершает свою работу только в том случае, если конечная вершина переносится в категорию «исследованные вершины». В этом случае уже будет весь список исследованных вершин, а на каждой из них будет стоять указатель с кратчайшим путем. Поэтому несложно будет по указателям отследить кратчайший путь от конечной вершины до начальной.[7]

# Пример работы



# Заключение

В ходе проделанной работы были изучены алгоритмы построения маршрута в графе, в частности алгоритмы Дейкстры и А\*. Подготовили лабиринт, реализовали алгоритмы с заданными параметрами. Для построения маршрута в лабиринте, мы использовали алгоритм Дейкстры от начала лабиринта до ключа «\*», и алгоритмом А\* находили путь от ключа «\*» до выхода. Результат сохранили в отдельный файл «maze-for-me-done», в котором сам маршрут построен точками (.) до ключа, а от него запятыми (,) до выхода.

# Список литературы

1. 10 графовых алгоритмов. [Электронный ресурс].URL: <https://medium.com/nuances-of-programming/10-графовых-алгоритмов-наглядное-объяснение-53226d8e6ba0>. (дата обращения: 25.05.2023)
2. Алгоритм Дейкстры – поиск кратчайшего пути в графе. [Электронный ресурс].URL: <https://blog.skillfactory.ru/glossary/algoritm-dejkstry/> . ( дата обращения: 25.05.2023)
3. Алгоритм Дейкстры. [Электронный ресурс].URL:<https://studfile.net/preview/3239584/page:2/> ( дата обращения: 25.05.2023)
4. Модуль в Queue в Python. [Электронный ресурс]. URL: <https://pythonim.ru/osnovy/ochered-queue-v-python>. (дата обращения: 25.05.2023)
5. Обрабатываем в Python очереди с приоритетом. [Электронный ресурс]. URL:<https://proglib.io/p/robot-v-labirinte-obrabatyvaem-ocheredi-s-prioritetom-v-python-2020-07-07>. (дата обращения: 25.05.2023)
6. Очередь в Python. [Электронный ресурс].URL: <https://pythonpip.ru/examples/ochered-python>.(дата обращения: 25.05.2023)
7. Поисковый алгоритм A Star: что это и как эффективно его использовать? [Электронный ресурс].URL: <https://codernet.ru/articles/web/poiskovyij_algoritm_a_star_chto_eto_i_kak_effektivno_ego_ispolzovat/>. ( дата обращения: 25.05.2023)
8. Реализация алгоритма A\*. [Электронный ресурс]. URL: <https://habr.com/ru/articles/331220//>. ( дата обращения: 25.05.2023)
9. Реализация алгоритма Дейкстры в Python. [Электронный ресурс].URL: <https://pythobyte.com/dijkstras-algorithm-python-00092/>. ( дата обращения: 25.05.2023)
10. Руководство инсайдера по алгоритму A\* в Python. [Электронный ресурс].URL: <https://pythobyte.com/a-star-algorithm-python-93427/>. ( дата обращения: 25.05.2023)
11. A Byte of Python (Russian), Версия 2.02, Swaroop C. H. (Перевод: Владимир Смоляр). [Электронный ресурс]. URL: <https://wombat.org.ua/AByteOfPython/AByteofPythonRussian-2.02.pdf> [c. 123-125]. (дата обращения: 25.05.2023)
12. Алгоритм А\* История создания. [Электронный ресурс]. URL:<https://studfile.net/preview/3239584/page:4/#11>. (дата обращения: 25.05.2023)

# Приложение 1

**Листинг программы**

def available\_paths(point, maze):  
 x, y = point  
 x\_length = len(maze)  
 y\_length = len(maze[0])  
 awailable\_ways = []  
  
 if (x - 1) >= 0 and maze[x - 1][y] == " ":  
 waypoint = (x - 1, y)  
 awailable\_ways.append(waypoint)  
  
 if (y + 1) < y\_length and maze[x][y + 1] == " ":  
 waypoint = (x, y + 1)  
 awailable\_ways.append(waypoint)  
  
 if (x + 1) < x\_length and maze[x + 1][y] == " ":  
 waypoint = (x + 1, y)  
 awailable\_ways.append(waypoint)  
  
 if (y - 1) >= 0 and maze[x][y - 1] == " ":  
 waypoint = (x, y - 1)  
 awailable\_ways.append(waypoint)  
 return awailable\_ways  
  
  
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 available\_paths(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])  
  
  
def a\_star(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 available\_paths(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 replace\_path(maze, path, symbol):  
 for x, y in path:  
 if maze[x][y] != ' ':  
 maze[x][y] = ';'  
 else:  
 maze[x][y] = symbol  
 return maze  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 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  
  
 path\_to\_key = dijkstra(maze, start, key)  
 path\_to\_end = a\_star(maze, key, end)  
  
 result = replace\_path(replace\_path(maze, path\_to\_key, '.'), path\_to\_end, ',')  
 x\_key, y\_key = key  
 result[x\_key][y\_key] = '\*'  
  
 with open('maze-for-me-done.txt', 'w') as f:  
 for line in result:  
 f.write("".join(line) + "\n")