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 Фильчугов В.Е.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Содержание

[Введение 3](#_Toc136562735)

[Задачи 3](#_Toc136562736)

[Теоретическая часть 4](#_Toc136562737)

[Реализация алгоритма 6](#_Toc136562738)

[Пример работы 8](#_Toc136562739)

[Заключение 10](#_Toc136562740)

[Список литературы 11](#_Toc136562741)

[Приложение 1 12](#_Toc136562742)

[Листинг программы 12](#_Toc136562743)

# Введение

Алгоритмы обхода графа являются одной из важнейших задач в программировании. Существует множество алгоритмов обхода графа, при своей работе все они отличаются скоростью, точностью и многими другими параметрами. Одним из таких является “Жадный алгоритм”, его работа заключается в быстром принятии решения на основе локальной оптимизации т.е. выбирает лучшее решение на основе текущих данных, следовательно в дальнейшей перспективе результат данного алгоритма может и не оказаться наилучшем и оптимальным т.к. он не учитывает возможные последствия выбора определенного варианта, а ограничивается только мгновенными выгодами.

**Цель работы**: реализовать алгоритмы обхода графа: алгоритма 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;
* Написать программу, которая использует выбранный алгоритм для нахождения маршрута и сохраняет результаты в файл
* Протестировать программу на различных наборах исходных данных и убедиться в ее правильной работе.

# Теоретическая часть

Графы и обход графов

Разберемся, что такое графы и для чего они нужны.

Графы являются одним из основных инструментов в алгоритмах. Граф представляет собой математическую модель, которая может быть использована в решение множества различных задач, таких - как поиск пути, сортировка и т.д. Примеры графов(рис1)

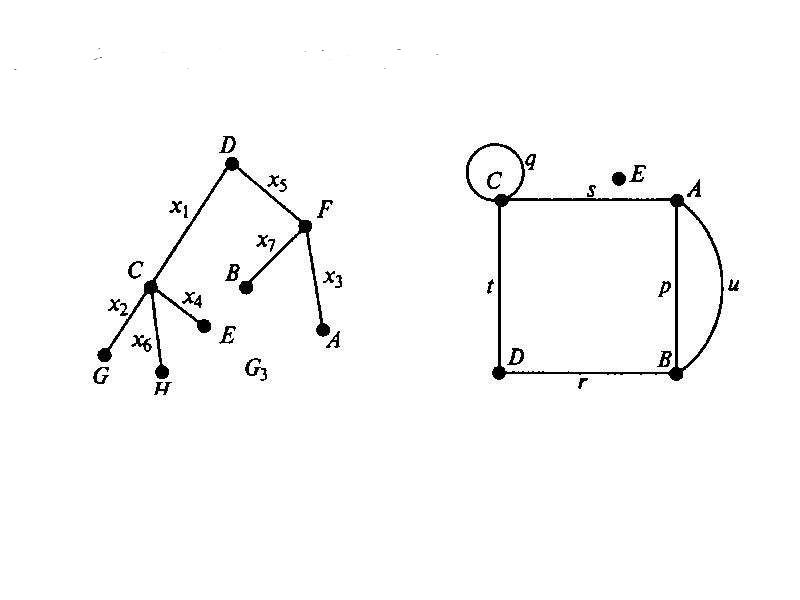


Рис1

Существует несколько типов графов, таких как ориентированные и неориентированные, связные и несвязные, взвешенные и невзвешенные. Каждый тип графа имеет свои особенности и используется для решения различных задач. Графы применяются в различных областях, включая транспортную логистику для оптимизации маршрутов доставки, социальные сети для анализа взаимосвязей между пользователями, биоинформатику для анализа геномов и многое другое. Изучение графов и их алгоритмов является важной частью обучения в области компьютерных наук и математики.

Обход графов – это переход от одной вершины к другой в поисках свойств связей этих вершин. Обход может происходить с помощью различных алгоритмов, основные из них это поиск в глубину (Depth-First Search, DFS) и поиск в ширину (Breadth-First Search, BFS).

Алгоритм Дейкстры

Алгоритм Дейкстры - это алгоритм поиска кратчайшего пути в графе с неотрицательными весами ребер. Он основан на идее постепенного расширения дерева кратчайших путей из начальной точки до всех остальных вершин. Теория алгоритма Дейкстры основана на понятии "метки вершины". Начальная вершина помечается нулевым расстоянием, а все остальные - бесконечностью. Затем алгоритм выбирает вершину с наименьшей меткой, добавляет ее в дерево кратчайших путей и обновляет метки всех смежных с ней вершин. Если новая метка короче, чем старая, она обновляется. Алгоритм продолжает выполняться, пока все вершины не будут помечены. В конечном итоге, расстояние от начальной вершины до любой другой вершины будет равно значению ее метки.(рис2)

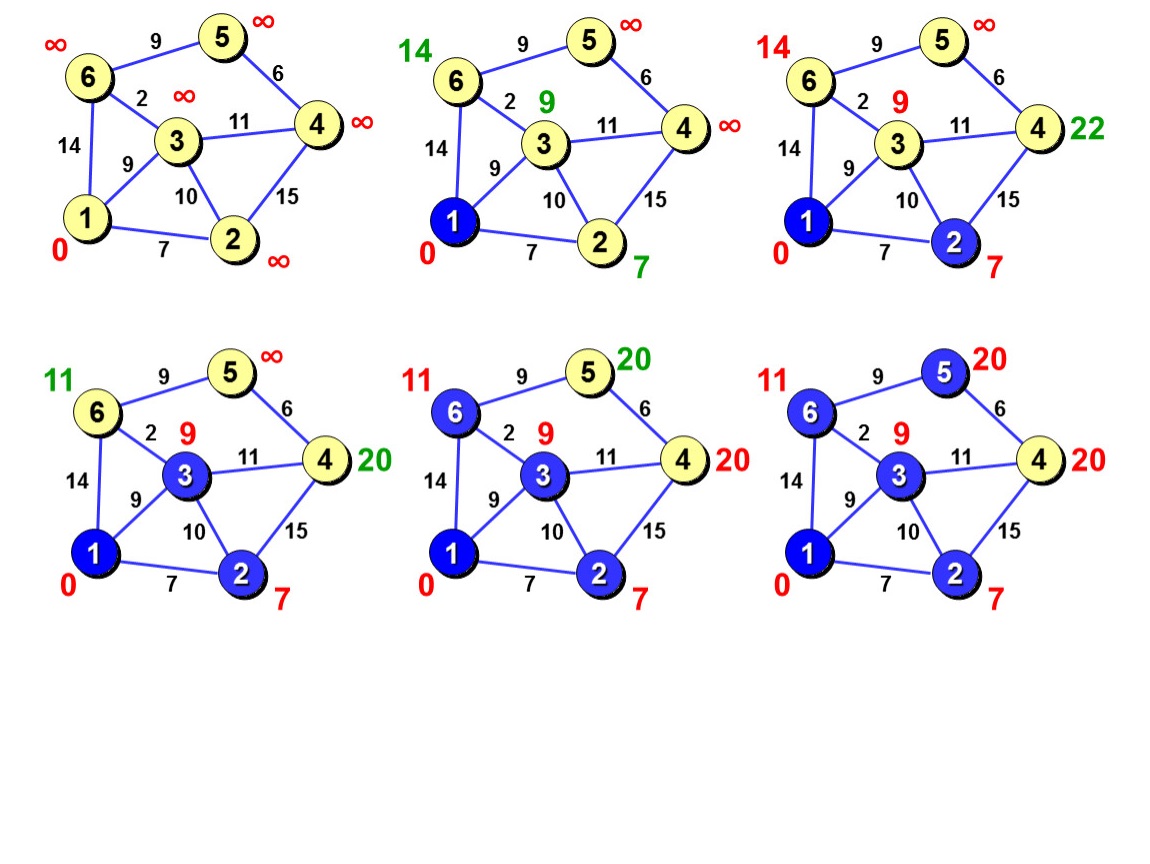


Рис 2

Алгоритм Дейкстры имеет полиномиальную сложность времени O(|E|+|V|log|V|), где |E| - количество ребер, а |V| - количество вершин в графе. Однако, это возможно только при использовании эффективных структур данных, таких как куча (heap), для выбора вершины с наименьшей меткой.

Важно отметить, что алгоритм Дейкстры работает только для графов с неотрицательными весами. Если граф содержит ребра с отрицательными весами, необходимо использовать другой алгоритм, например, алгоритм Беллмана-Форда.

Алгоритм A\*

Алгоритм А\* - это по сути усовершенствованный алгоритм Дэйкстры, который благодаря эвристической функции достигает большей производительности.

A\*(A star или A звезда) – алгоритм поиска, который находит маршрут с наименьшей стоимостью от одной вершины, которая является начальной, до другой являющейся конечной. Порядок в котором наш алгоритм будет обходить все вершины определяется с помощью эврестической функции(F(x)) = расстояние(h(x)) + стоимость(g(x)).

F(x) - данная функция оценивает минимальную стоимость перехода от вершины к вершине

Эвристическая оценка h(x) определяет оценку расстояния от текущего узла до целевого узла. Она должна быть "допустимой", т.е. не переоцененной, чтобы обеспечить правильные результаты работы алгоритма. Оценки, которые не удовлетворяют этому требованию, могут привести к неправильному выбору оптимального пути.

Функция g(x) обозначает стоимость (расстояние или время) самого дешевого пути от начальной вершины до вершины x. С помощью суммирование ребер она вычисляет их стоимости в пути от начальной вершины до конечной.

Принцип работы данного алгоритма:

При посещении одной вершины, алгоритм исследует все соседние и только после переходить к следующей. A star делит вершины на 3 типа:

1. Это уже посещенные вершины, к которым известен путь
2. Это неизвестные вершины, которые еще не были посещены
3. Это исследованные вершины, те к которым известен самый кротчайшей путь

## Реализация алгоритма

Реализация алгоритма Дейкстры

Он работает следующим образом:

1. Создается словарь distances, в котором для каждой точки лабиринта находится значение расстояния от стартовой точки до этой точки. На данный момент известно расстояние только до стартовой точки, поэтому distances = {start: 0}.

2. Создается очередь heap, в которую помещается кортеж (cost, current, path), где cost - стоимость пути от стартовой точки до текущей точки, current - текущая точка, path - путь от стартовой точки до текущей точки.

3. До тех пор, пока очередь не пуста, извлекается кортеж с наименьшей стоимостью из очереди. Если текущая точка - целевая, то возвращается найденный путь. Если текущая точка уже была посещена и стоимость пути от стартовой до текущей точки больше значения в словаре distances, то текущая точка пропускается и переходим к следующей итерации цикла.

4. Если текущая точка не была посещена, то добавляем ее в словарь distances, присваивая ей значение cost + 1 (стоимость перехода от текущей точки к соседней точке). Затем проходимся по всем соседним точкам и добавляем их в очередь heap с соответствующими значениями стоимости пути и пути до этой точки.

5. Если соседняя точка уже была посещена, пропускаем ее, иначе добавляем ее в очередь heap. Если стоимость пути до соседней точки меньше, чем значение в словаре distances, то обновляем словарь distances и путь до этой точки.

6. Если в результате обхода лабиринта не был найден путь до целевой точки, то функция возвращает None.

Реализация алгоритма A\*

1. Находится эвристическая оценка расстояния между текущей точкой и конечной точкой с помощью функции heuristic(current, end)

2. Создается приоритетная очередь heap, в которую добавляется кортеж (f\_score[neighbor], neighbor), где f\_score[neighbor] - суммарный путь от начальной точки до текущей + эвристическая оценка пути до конечной точки, neighbor - текущая точка

3. Происходит извлечение точки с наименьшей f\_score из heap, и проверяется, является ли данная точка конечной. Если это так, то происходит возврат пути от конечной точки до начальной с помощью словаря came\_from

4. Добавление текущей точки в множество visited

5. Для каждого соседнего узла neighbor, происходит вычисление временной оценки пути, проходящего через этот узел tentative\_g\_score, как сумма пути до текущей точки g\_score[current] и расстояния до соседа (равного 1 в данном случае)

6. Если neighbor уже находится в visited и tentative\_g\_score >= g\_score.get(neighbor, float('inf')), то происходит переход к следующему соседу

7. Если tentative\_g\_score меньше, чем текущий g\_score[neighbor] или neighbor еще не находится в visited, то для соседа neighbor обновляются значения словарей came\_from, g\_score, f\_score, а также сосед добавляется в heap

8. Наконец, если цель не удалось достичь, то функция возвращает пустой список

Таким образом, алгоритм A\* использует эвристическую оценку расстояния до конечной точки для улучшения скорости работы поиска пути.

# Пример работы

Построенный путь от входа, до ключа **‘ . '** (рис) и от ключа, до выхода **' , '**, пересечения путей обозначены **‘ ; ’** (рис - рис)

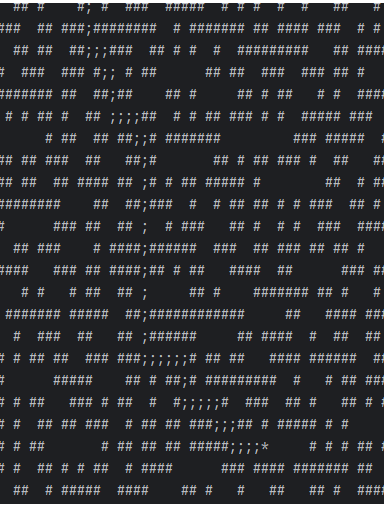


Рис3

****

Рис 4

# Заключение

В ходе проделанной работы была создана программ а в которой было реализовано решение задачи поиска пути в лабиринте с использованием алгоритмов Дейкстры и А\*. Было также создано обновленное изображение лабиринта с отмеченными найденными путями и ключевыми точками.

Решение задачи с помощью алгоритма Дейкстры наглядно демонстрирует принцип работы этого алгоритма – он ищет кратчайший путь от одной точки до другой, полностью перебирая все возможные варианты пути.

Алгоритм А\* применяется для поиска кратчайшего пути в лабиринте с учетом препятствий и оценки расстояния до цели. Он был использован для поиска пути от ключа до выхода.

Таким образом, данная программа демонстрирует работу двух разных алгоритмов поиска пути с использованием различных подходов и оценок, что делает ее полезной для изучения принципов работы алгоритмов и их применения в реальной жизни.

# Список литературы

1. [Dijkstra E. W.](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D0%B0,_%D0%AD%D0%B4%D1%81%D0%B3%D0%B5%D1%80_%D0%92%D0%B8%D0%B1%D0%B5) [A note on two problems in connexion with graphs](https://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf) (англ.) // [Numerische Mathematik](https://en.wikipedia.org/wiki/Numerische_Mathematik" \o "en:Numerische Mathematik) / [F. Brezzi](https://en.wikipedia.org/wiki/Franco_Brezzi) — [Springer](https://ru.wikipedia.org/wiki/Springer_Science%2BBusiness_Media" \o "Springer Science+Business Media)

[2. Science+Business Media](https://ru.wikipedia.org/wiki/Springer_Science%2BBusiness_Media" \o "Springer Science+Business Media), 1959. — Vol. 1, Iss. 1. — P. 269—271. — ISSN [0029-599X](https://www.worldcat.org/issn/0029-599X); [0945-3245](https://www.worldcat.org/issn/0945-3245) — [doi:10.1007/BF01386390](https://dx.doi.org/10.1007/BF01386390)

3. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд

4. Штайн. Алгоритмы: построение и анализ = Introduction to Algorithms. — 2-е изд. — М.: [«Вильямс»](https://ru.wikipedia.org/w/index.php?title=%D0%92%D0%B8%D0%BB%D1%8C%D1%8F%D0%BC%D1%81_(%D0%B8%D0%B7%D0%B4%D0%B0%D1%82%D0%B5%D0%BB%D1%8C%D1%81%D1%82%D0%B2%D0%BE)&action=edit&redlink=1), 2006. — С. 1296. — [ISBN 0-07-013151-1](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/0070131511).

5. [Левитин А. В.](https://www.wikidata.org/wiki/Q21694518) Глава 9. Жадные методы: Алгоритм Дейкстры // [Алгоритмы. Введение в разработку и анализ](https://www.wikidata.org/wiki/Q21694522) — М.: [Вильямс](https://www.wikidata.org/wiki/Q21694521), 2006. — С. 189—195. — 576 с. — [ISBN 978-5-8459-0987-9](https://ru.wikipedia.org/wiki/%D0%A1%D0%BB%D1%83%D0%B6%D0%B5%D0%B1%D0%BD%D0%B0%D1%8F:%D0%98%D1%81%D1%82%D0%BE%D1%87%D0%BD%D0%B8%D0%BA%D0%B8_%D0%BA%D0%BD%D0%B8%D0%B3/9785845909879)

6. Hart P. E., Nilsson, N. J., Raphael, B. A Formal Basis for the Heuristic Determination of Minimum Cost Paths // [IEEE](https://ru.wikipedia.org/wiki/IEEE) Transactions on Systems Science and Cybernetics SSC4. — 1968. — № 2. — С. 100 — 107.

7. Hart P. E., Nilsson, N. J., Raphael, B. Correction to «A Formal Basis for the Heuristic Determination of Minimum Cost Paths» // SIGART Newsletter. — 1972. — Т. 37. — С. 28 — 29.

8. Дольников, В. Л. Основные алгоритмы на графах: текст лекций / В. Л. Дольников, О. П. Якимова. - Ярославль: ЯрГУ, 2011. – 80 с. – URL: <http://www.lib.uniyar.ac.ru/edocs/iuni/20110210.pdf>(дата обращения 26.04.2023) – Текст: электронный

# Приложение 1

## Листинг программы

import heapq

# функция для чтения txt файла

def readmaze(file\_name):

with open(file\_name) as f:

maze = [list(line.strip()) for line in f]

return maze

# вход и выход

def door(maze):

for X in range(len(maze[0])):

if maze[0][X] == " ":

start = (0, X)

break

for X in range(len(maze[0])):

if maze[len(maze) - 1][X] == " ":

end = (len(maze) - 1, X)

break

return start, end

# поиск ключа

def find\_key\_position(maze):

for i, g in enumerate(maze):

if "\*" in g:

return(i, g.index("\*"))

# Поиск пути от входа до ключа с помощью алгоритма Дейкстры

def Dijkstra\_algorithm(maze, start, end):

distances = {start: 0}

queue = [(0, start, [start])]

while queue:

(cost, current, path) = heapq.heappop(queue)

if current == end:

return path

if current not in distances or cost > distances[current]:

continue

for neighbor in available\_paths(current, maze):

if neighbor not in distances or cost + 1 < distances[neighbor]:

distances[neighbor] = cost + 1

heapq.heappush(queue, (cost + 1, neighbor, path + [neighbor]))

return None

# функция для поиска доступных путей из текущей точки

def available\_paths(current, maze):

neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]

paths = []

for neighbor in neighbors:

x = current[0] + neighbor[0]

y = current[1] + neighbor[1]

if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != "#":

paths.append((x, y))

return paths

# нахождения пути от ключа до выхода с помощью алгоритма А\*

def A\_star(maze, start, end):

heap = [(0, start)]

heapq.heapify(heap)

visited = set()

came\_from = {}

g\_score = {start: 0}

f\_score = {start: heuristic(start, end)}

while heap:

current = heapq.heappop(heap)[1]

if current == end:

path = [end]

while current in came\_from:

current = came\_from[current]

path.append(current)

return path[::-1]

visited.add(current)

for neighbor in available\_paths(current, maze):

if neighbor == start or neighbor == key:

continue

tentative\_g\_score = g\_score[current] + 1

if neighbor in visited and tentative\_g\_score >= g\_score.get(neighbor, float('inf')):

continue

if tentative\_g\_score < g\_score.get(neighbor, float('inf')):

came\_from[neighbor] = current

g\_score[neighbor] = tentative\_g\_score

f\_score[neighbor] = g\_score[neighbor] + heuristic(neighbor, end)

if neighbor not in visited:

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

return []

# функция для определения эвристической оценки расстояния между двумя точками

def heuristic(current, end):

return abs(current[0] - end[0]) + abs(current[1] - end[1])

# создание нового лабиринта с отмеченными путями

def new\_maze(maze, path, marker):

for i, j in path:

if maze[i][j] == " ":

if marker == "." and (i, j) in path2:

maze[i][j] = ";"

else:

maze[i][j] = marker

return maze

# функция для записи обновленного лабиринта в txt файл

def new\_file(maze, file\_name):

with open(file\_name, "w") as f:

for g in maze:

f.write("".join(g)+"\n")

maze = readmaze("maze-for-u.txt")

start, end = door(maze)

key = find\_key\_position(maze)

path1 = Dijkstra\_algorithm(maze, start, key)

path2 = A\_star(maze, key, end)

maze = new\_maze(maze, path1, ".")

maze = new\_maze(maze, path2, ",")

new\_file(maze, "maze-for-me-done.txt")