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: «Алгоритмы поиска пути и структурное

программирование»

Выполнил студент 1 курса гр. ИС-33 Айрапетян Г.С.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

**Содержание**

[Введение 3](#_Toc136038086)

[Задачи 3](#_Toc136038087)

[1. Теоретическая часть 4](#_Toc136038088)

[1. Алгоритмы поиска пути 4](#_Toc136038089)

[1.1. Поиск в ширину 4](#_Toc136038090)

[1.2. Алгоритм A\* 5](#_Toc136038091)

[2. Основы структурного программирования 6](#_Toc136038092)

[2. Реализация алгоритма 7](#_Toc136038093)

[а) Реализация поиска в ширину 7](#_Toc136038094)

[б) Реализация алгоритма A\* 8](#_Toc136038095)

[Пример работы алгоритма 9](#_Toc136038096)

[1) Поиск в ширину 9](#_Toc136038097)

[2) Поиск по алгоритму A\* 9](#_Toc136038098)

[Заключение 10](#_Toc136038099)

[Список литературы 11](#_Toc136038100)

[Приложение 1 12](#_Toc136038101)

[Листинг программы 12](#_Toc136038102)

# **Введение**

В текущей работе будут рассмотрены две важные области программирования: алгоритмы поиска пути и структурное программирование. Рассмотрим последовательно каждую тему: алгоритмы поиска пути - незаменимый инструмент в любой области, где требуется нахождение оптимальных решений. А структурное программирование - подход, направленный на повышение читаемости, сопровождаемости и тестируемости кода. В рамках работы будут изучены основные алгоритмы поиска пути, такие как поиск в ширину и А\*, а также принципы структурного программирования. В результате будет создана программа на языке Python, которая будет использовать оба алгоритма поиска пути и соответствующие принципы структурного программирования.

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

# **Задачи**

* Изучить алгоритмы построения маршрута в графе, включая ширину и А\*;
* Познакомиться с основами структурного программирования;
* Создать программу на языке программирования Python, которая в использовании рассмотренных алгоритмов (поиск в ширину и A\*) обойдет лабиринт и найдет оптимальный путь;
* Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# **1. Теоретическая часть**

## **1. Алгоритмы поиска пути**

### **1.1. Поиск в ширину**

**Поиск в ширину** (breadth-first search) представляет собой один из простейших алгоритмов для обхода графа и является основой для многих важных алгоритмов для работы с графами. Например, алгоритм Прима (Prim) поиска минимального остовного дерева или алгоритм Дейкстры (Dijkstra) поиска кратчайшего пути из одной вершины используют идеи, сходные с идеями, используемыми при поиске в ширину [1, c. 613].

Поиск в ширину имеет такое название потому, что в процессе обхода мы идем вширь, т.е. перед тем, как приступить к поиску вершин на расстоянии k + 1, выполняется обход всех вершин на расстоянии k [1, c. 614].

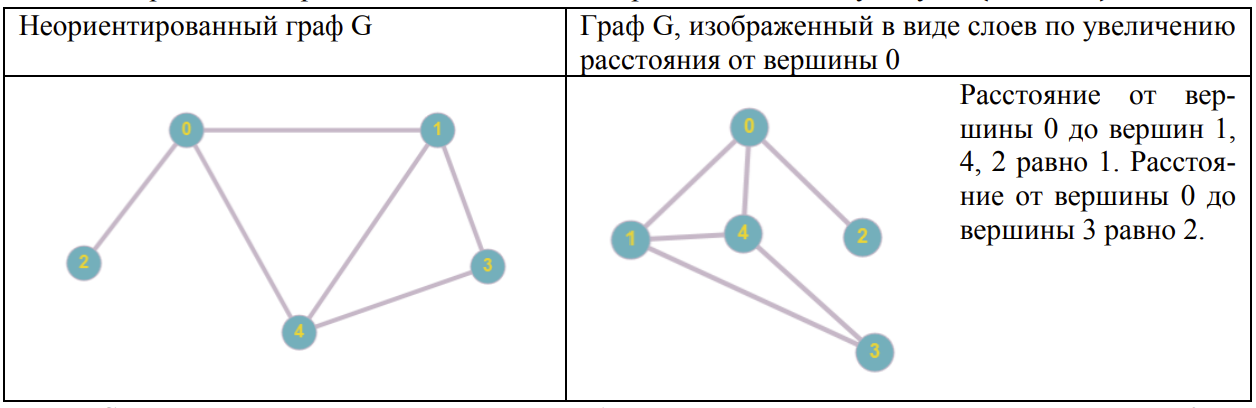
В результате поиска в ширину находится путь кратчайшей длины в невзвешенном графе (ребра которого не имеют весов), то есть путь, содержащий наименьшее число рёбер. Сложность работы алгоритма такая же, как и алгоритма обхода в глубину – 𝑂 (|𝑉| + |𝐸|) [2]. Пример функционирования алгоритма поиска и в ширину показан на рисунке 1.

Рисунок 1. Пример работы алгоритма поиска в ширину

Суть алгоритма заключается в том, чтобы просматривать сначала стартовую вершину 0, затем те вершины, которые удалены от нее на расстояние 1 и так далее слоями: затем просматриваем вершины, которые удалены на расстояние d, далее d+1. Для этого в алгоритме используется очередь Q, в которую сначала заносится стартовая вершина 0. Затем повторяем следующие итерации: пока очередь не пуста, достаем из ее головы очередную вершину и просматриваем всех соседей этой вершины, и если какие-то из них еще не помещены в очередь, то помещаем их в конец очереди. При таком обходе, когда очередь станет пуста, все вершины будут просмотрены, причем в порядке увеличения расстояния от стартовой вершины. Длины кратчайших путей считаются в процессе алгоритма при помощи массива расстояний d [] и массива предков вершин p [] [2].

Одним из важных применений BFS является поиск кратчайшего пути в невзвешенном графе. Также алгоритм может быть применен для проверки связности графа, поиска циклов и решения других задач, связанных с обходом графа в ширину [3].   
 Однако следует отметить, что BFS требует дополнительной памяти для хранения списка посещенных вершин и очереди. Кроме того, время выполнения BFS зависит от размера графа и количества ребер. В худшем случае алгоритм имеет временную сложность O (|V| + |E|), где |V| - количество вершин, а |E| - количество ребер графа [3].

### **1.2. Алгоритм A\***

**Алгоритм A\*** (A-star) является эффективным алгоритмом поиска кратчайшего пути в графе с использованием эвристической функции для оценки стоимости перехода от текущей вершины к целевой вершине**.**

A\* похож на алгоритм Дейкстры в том смысле, что его можно использовать для поиска кратчайшего пути. A\* похож на жадный поиск наилучшего первого поиска в том, что он может использовать эвристику для управления собой.

Секрет его успеха заключается в том, что он сочетает в себе части информации, которые использует алгоритм Дейкстры (предпочтение вершин, близких к начальной точке), и информацию, которую использует жадный поиск наилучшего первого поиска (предпочтение вершин, близких к цели). В стандартной терминологии, используемой при разговоре об A\*, g(n) представляет собой *точную стоимость* пути от начальной точки до любой вершины n и h(n) представляет собой эвристическую *оценку стоимости* от вершины n до цели. На приведенных выше диаграммах желтый (h) представляет вершины, удаленные от цели, а бирюзовый (g) представляет вершины, далекие от начальной точки. A\* уравновешивает их по мере продвижения от начальной точки к цели. Каждый раз в основном цикле проверяется вершина n с наименьшим значением f(n) = g(n) + h(n) [4].

## **2. Основы структурного программирования**

Структурное программирование является одним из фундаментальных подходов к разработке программного обеспечения. Оно предоставляет методологические инструменты и принципы для построения программных систем с понятной и легко поддерживаемой структурой. В данном докладе мы рассмотрим основные принципы структурного программирования и его влияние на развитие современных языков программирования [5].

Основные принципы структурного программирования:

* Использование последовательных блоков кода;
* Применение условных операторов и циклов;
* Избегание использования операторов безусловного перехода;
* Разделение программы на функции или подпрограммы;
* Принцип единственного входа и выхода [5].

Преимущества структурного программирования:

* Упрощение понимания и отладки кода;
* Облегчение поддержки и модификации программы;
* Повышение читаемости и структурированности кода;
* Снижение вероятности ошибок при разработке программного обеспечения [5].

# **2. Реализация алгоритма**

Данный код на Python представляет собой реализацию алгоритма поиска пути в лабиринте с использованием двух методов: проходки в ширину и алгоритма A\*.

1. Импортируются необходимые модули:

* **deque** из **collections** для работы с очередью;
* **sqrt** из **math** для вычисления квадратного корня;
* **PriorityQueue** из **queue** для использования приоритетной очереди.

2. Создаем функцию **get\_lab**, которая открывает файл с заданным именем, считывает из него лабиринт и возвращает его в виде матрицы.

3. Создаем функцию **get\_valid\_neighbors**, которая получает координаты текущей клетки лабиринта и возвращает список ее соседей (клеток, куда можно перейти из текущей). Соседи определяются по четырем направлениям: снизу, слева, справа и сверху. Функция проверяет, что координаты соседа находятся в пределах лабиринта и что клетка не является стеной.

## **а) Реализация поиска в ширину**

1. Создаем функцию **search\_in\_width**, которая реализует алгоритм поиска пути в лабиринте с помощью поиска в ширину.
2. Функция **search\_in\_width** получает на вход матрицу лабиринта **maze**, координаты начальной **start** и конечной **end** клеток.
3. Алгоритм проходит по всем соседям текущей клетки и добавляет их в очередь поиска. Каждый раз, когда клетка извлекается из очереди, ее соседи добавляются в очередь, если они еще не были посещены. Для каждой клетки в словарь **parents** записывается родитель (клетка, из которой пришли в текущую клетку). Если алгоритм находит конечную клетку, он завершает работу.
4. Функция **get\_heuristic** вычисляет эвристическую оценку расстояния от текущей клетки до конечной. В данной реализации используется эвристика "манхэттенское расстояние", которое вычисляется как сумма модулей разностей координат по осям x и y.
5. Функция **get\_path\_1** получает на вход конечную клетку **node**, начальную клетку **start** и пустой список **path**. Функция проходит по родителям от конечной клетки до начальной и добавляет каждого родителя в список **path**.

## **б) Реализация алгоритма A\***

1. Функция **search\_in\_A** реализует алгоритм A\*. Функция работает аналогично функции **search\_in\_width**, но вместо простой очереди используется приоритетная очередь, которая сортирует элементы по их приоритету. Приоритет каждого элемента вычисляется как сумма длины пути от начальной клетки до текущей и эвристической оценки расстояния от текущей клетки до конечной.
2. Функция **main** вызывает все описанные выше функции для поиска путей в лабиринте и записи результатов в файлы **res3.txt** и **res4.txt**. Сначала вызывается функция **search\_in\_width** для поиска пути с помощью проходки в ширину, затем вызывается функция **get\_path\_1** для получения пути к выходу.
3. Затем матрица лабиринта изменяется, чтобы на ней был отмечен найденный путь, и результат записывается в файл **res3.txt**.
4. Далее вызывается функция **search\_in\_A** для поиска пути с помощью алгоритма A\*, затем вызывается функция **get\_path\_1** для получения пути к выходу.
5. Матрица лабиринта изменяется, чтобы на ней был отмечен найденный путь, и результат записывается в файл **res4.txt**.

**Пример работы алгоритма**

## **1) Поиск в ширину**

Файл «res3.txt», получившийся в результате работы алгоритма, представляет собой лабиринт, где «#» - стены, «.» - траектория пути алгоритма BFS (поиска в ширину)

1. Матрица лабиринта изменяется, чтобы на ней был отмечен найденный путь, и результат записывается в файл res4.txt.21:37

## **2) Поиск по алгоритму A\***

Файл «res4.txt», получившийся в результате работы алгоритма, представляет собой лабиринт, где «#» - стены, «,» - траектория пути алгоритма А\*, «\*» - ключ.

# **Заключение**

В процессе выполнения работы были изучены основы алгоритмов обхода графа в ширину и А\*, а также структурного программирования.

Было выявлено, что поиск в ширину (BFS) является эффективным и широко используемым алгоритмом для обхода и анализа графов, особенно в случаях, когда необходимо найти кратчайший путь или выполнить операции на всех вершинах в одинаковом порядке удаленности от стартовой вершины, а алгоритм A\* является оптимальным в том смысле, что он всегда находит кратчайший путь, если такой путь существует. Однако его производительность может зависеть от выбора эвристической функции и структуры графа.

Также была создана программа на Python, которая реализует алгоритмы обхода графа в ширину и А\* для поиска маршрута в лабиринте, которая на выходе создает два текстовых документа с обходом лабиринта по двум рассмотренным алгоритмам.

# **Список литературы**

1. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. В24 Алгоритмы: построение и анализ, 2-е издание.: Пер. с англ. — М.: Издательский дом “Вильямс”, 2011. — 1296 с.: ил. — Парал. тит. Англ
2. Лекция 7. Графы: способы их хранения и обхода (в ширину и в глубину). Проверка графа на двудольность, поиск циклов и топологическая сортировка графа / [Электронный ресурс]. URL: <https://informatics.msk.ru/pluginfile.php/252386/mod_resource/content/0/Zanyatie7_Grafy.pdf> (Дата обращения: 10.05.2023)
3. Басараб М.А., Домрачева А.Б., Купляков В.М. Алгоритмы решения задачи быстрого поиска пути на географических картах. – URL: <https://cyberleninka.ru/article/n/algoritmy-resheniya-zadachi-bystrogo-poiska-puti-na-geograficheskih-kartah/viewer>
4. Amit Patel - "Amit's A\* Pages" / [Электронный ресурс]. URL: <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html#the-a-star-algorithm> (Дата обращения: 11. 05.2023)
5. Структурное программирование / [Электронный ресурс.]. URL: <http://www.pmbk.ru/lister/035/99/index.shtml>
6. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб.: Питер. 2017. – 288 с.: ил. – (Серия «Библиотека программиста»).
7. Хайнеман, Джордж, Пояяис, Гэри, Сеяков, Стэнли. Алгоритмы. Справочник с примерами на С, C++, Java и Python, 2-е изд.: Пер. с англ. — СпБ.: 2017. — 432 с.: ил. — Парал. тит. англ.
8. Robert Sedgewick and Kevin Wayne. «Algorithms» FOURTH EDITION, Princeton University.

# **Приложение 1**

## **Листинг программы**

rom collections import deque  
from math import sqrt  
from queue import PriorityQueue  
  
parents = {}  
  
  
def get\_lab(filename):  
 with open(filename) as file:  
 maze = file.read().split('\n')  
 for i in range(len(maze) - 1):  
 maze[i] = [char for char in maze[i]]  
 return maze  
  
  
# поиск соседей  
def get\_valid\_neighbors(coord, maze):  
 neighbors = [(coord[0] + 1, coord[1]), (coord[0], coord[1] - 1), (coord[0], coord[1] + 1),  
 (coord[0] - 1, coord[1])] # Снизу, слева, справа, сверху  
 valid\_neighbors = []  
 for el in neighbors:  
 row, col = el  
 if row >= 0 and row < len(maze) and col >= 0 and col < len(maze[0]) and maze[row][col] != '#':  
 valid\_neighbors.append(el)  
 return valid\_neighbors  
  
  
# Проходка вширь для поиска пути (родитель записывается в словарь)  
def search\_in\_width(maze, start, end):  
 search\_deque = deque()  
 search\_deque.append(start)  
 searched = set()  
 while search\_deque:  
 coord = search\_deque.popleft()  
 if coord == end:  
 print("Success!")  
 return  
 searched.add(coord)  
 neighbors = get\_valid\_neighbors(coord, maze)  
 for n in neighbors:  
 if n not in searched:  
 parents[n] = coord # запись родителя  
 search\_deque.append(n)  
 return  
  
  
parents2 = {}  
  
  
def get\_heuristic(cell, end):  
 return sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2)  
  
  
# проход по родителям для поиска пути к выходу  
def get\_path\_1(node, start, path):  
 while node != start:  
 path.append(parents[node])  
 node = parents[node]  
  
  
def search\_in\_A(maze, start, end):  
 path\_new = []  
 queue = PriorityQueue()  
 queue.put((0, start))  
 searched = set()  
 while not queue.empty():  
 p, coord = queue.get()  
 if coord == end:  
 print("Success for A\*")  
 return p  
 searched.add(coord)  
 neighbors = get\_valid\_neighbors(coord, maze)  
 for n in neighbors:  
 if n not in searched:  
 parents2[n] = coord # запись родителя  
 if coord == start:  
 get\_path\_1(n, start, path\_new)  
 else:  
 path\_new.append(n)  
 priority = len(path\_new) + get\_heuristic(n, end)  
 queue.put((priority, n))  
 return  
  
  
def main():  
 filename = 'maze-for-u.txt'  
  
 # преобразование лабиринта в матрицу  
 maze = get\_lab(filename)  
  
 start = (0, 1)  
 end = (len(maze) - 2, len(maze[0]) - 2)  
 path = []  
  
 # поиск путей  
 search\_in\_width(maze, start, end)  
 # получение пути к выходу  
 get\_path\_1(end, start, path)  
  
 # заполнение матрицы лабиринта  
 for place in path:  
 maze[place[0]][place[1]] = "."  
 maze[end[0]][end[1]] = "\*"  
 result1 = ""  
 # преобразование матрицы в строку  
 for line in maze:  
 result1 += "".join(line) + "\n"  
 # запись лабиринта в файл  
 with open("res3.txt", "w") as f:  
 f.write(result1)  
  
 # для А\*  
 path2 = []  
 search\_in\_A(maze, start, end)  
 get\_path\_1(end, start, path2)  
  
 for place in path2:  
 maze[place[0]][place[1]] = ","  
 maze[end[0]][end[1]] = "\*"  
 result2 = ""  
 for line in maze:  
 result2 += "".join(line) + "\n"  
  
 with open("res4.txt", "w") as f:  
 f.write(result2)  
  
  
if \_\_name\_\_ == '\_\_main\_\_':  
 main()