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 Вершинин.Д.А.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г

Оглавление

**[Введение 3](#_Toc136109114)**

[**Теоретическая часть 5**](#_Toc136109115)

[**Реализация алгоритма 12**](#_Toc136109116)

[**Пример работы кода 14**](#_Toc136109117)

[**Вывод 15**](#_Toc136109118)

[**Список литературы 16**](#_Toc136109119)

[**Приложение 1 17**](#_Toc136109120)

[**Листинг программы 17**](#_Toc136109121)

# Введение

В настоящее время информатика является одной из самых важных наук, которая изучает общие свойства информации, методы ее создания, обработки, хранения, передачи и поиска. В рамках данной курсовой работы мы будем изучать алгоритмы поиска пути и структурное программирование на языке Python. Алгоритмы поиска пути используются в различных областях, таких как компьютерные игры, робототехника, геоинформационные системы и другие. Структурное программирование является важной техникой программирования, которая позволяет создавать более эффективный и понятный код. В данной работе мы рассмотрим основные алгоритмы поиска пути, такие как алгоритм Дейкстры, алгоритм A\* и алгоритм поиска в глубину. Мы также изучим основы структурного программирования на языке Python, такие как функции, модули, классы и объекты.

Цель работы -  изучение алгоритмов поиска пути и структурного программирования на языке Python. Код, реализующий алгоритмы поиска пути и демонстрирующий применение структурного программирования на языке Python.

Задачи:

* Изучение основных алгоритмов поиска пути, таких как алгоритм Дейкстры, алгоритм A\* и алгоритм поиска в глубину.
* Реализация алгоритмов поиска пути на языке Python.
* Применение структурного программирования на языке Python при реализации алгоритмов.
* Реализация кода, демонстрирующего работу заданных алгоритмов поиска пути, выход из лабиринта.
* Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.

# Теоретическая часть

1. Алгоритм A\*

Это модификация алгоритма Дейкстры, оптимизированная для поиска пути к единственной конечной точке. Он используется для нахождения кратчайшего пути между двумя точками на графе. Алгоритм A\* использует эвристику для оценки расстояния от текущей вершины до конечной точки, что позволяет ему искать путь более эффективно, чем алгоритм Дейкстры. Основные задачи, которые решает алгоритм A\*:

* Нахождение кратчайшего пути между двумя точками на графе.
* Оптимизация алгоритма Дейкстры для поиска пути
* Использование эвристики для оценки расстояния от текущей вершины до конечной точки, что позволяет ускорить поиск пути.

Разберем пример. У нас есть граф:

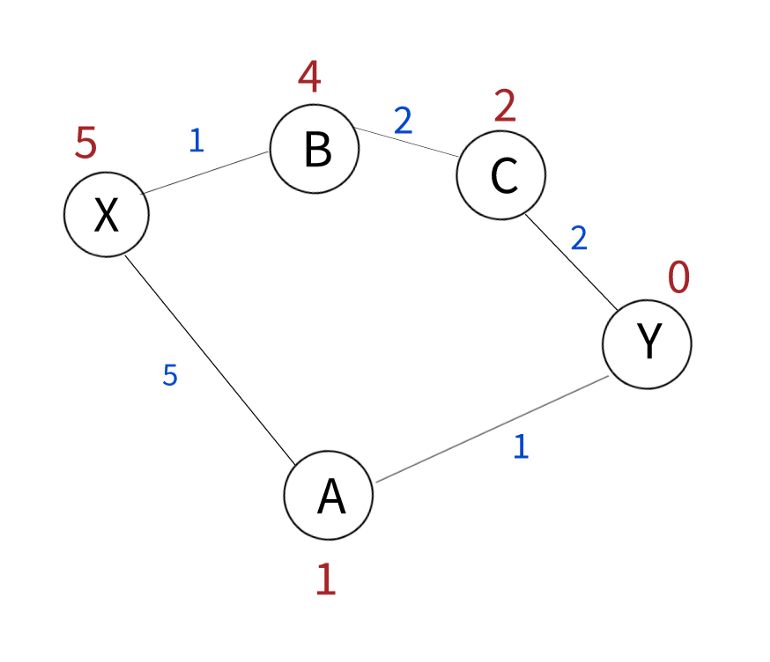


Рисунок 1. Простой граф алгоритма A\*

Допустим, мы хотим попасть из точки X в точку Y. Так как вершина Х не меняет своего положения, мы можем отбросить g(n) — ее значение равно 0. Эвристическое значение этой вершины выделено красным шрифтом — 5.

В подобных задачах эвристическое значение —  стоимость достижения рассматриваемой вершины из начальной.

Из вершины Х есть два пути.

Если мы перейдем в вершину А, g(n) будет равна 5 (стоимость пути), так как мы перемещаемся в новую вершину. Значение h(n) теперь равно 1. Значение f(n) в точке А будет равно 5+1 = 6. Теперь найдем значение f(n) каждой точки:

X— A => g(A) + f(A) = 5 + 1 = 6,

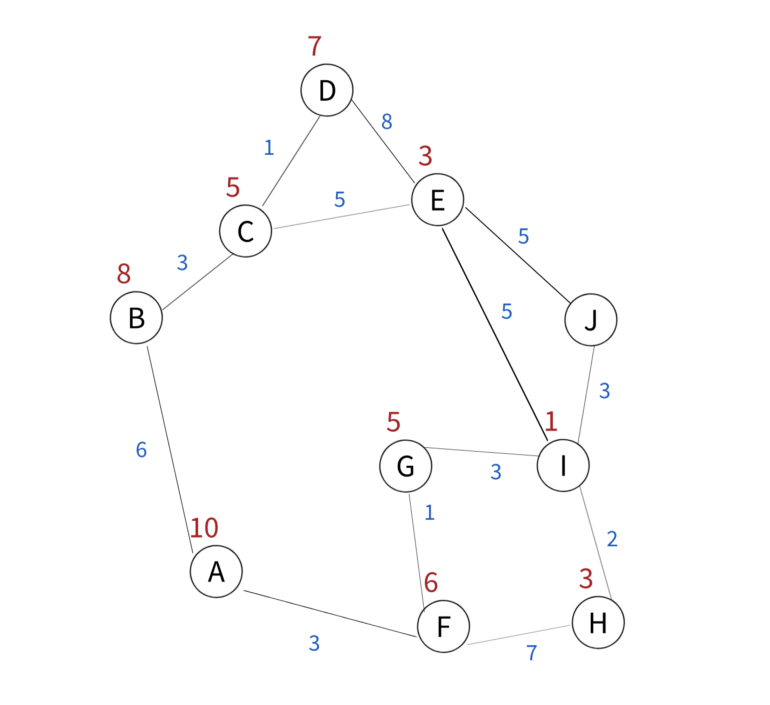
A — Y=> g(Y) + f(Y) = 6+ 0= 6,

X— B => g(B) + f(B) = 1+ 4= 5,

B — C => g(C) + f(C) = 3+ 2= 5,

C — Y=> g(Y) + f(Y) = 5 + 0= 5,

Как видно из наших вычислений, кратчайший путь — X-B-C-Y. Его стоимость равна 5, в то время как X-A-Y — 6. С этим примером разобрались, рассмотрим следующий:



*Рисунок 2. Усложненный граф алгоритма A\**

Допустим, мы хотим найти кратчайший путь от вершины А до вершины J. Из А есть только два пути — B и F. Вычислим стоимость: f(B) = 8 + 6 = 14 и f(F) = 3+6 =9. Следовательно, нам нужно перейти в вершину F — алгоритм продолжит работу отсюда.

Из точки F есть два пути — G и H. Снова вычислим стоимость:  f(G) = 4 +5 = 9 and f(H) = 10 + 3 = 13. Значит, мы переходим в точку G. Следуя пути I—J, получаем следующее:  f(I) = 7 + 1 = 8 и f(J) = 10. Так как все значения, следующие за вершиной F, меньше f(B), возвращаться к вершине B не имеет смысла.

Но допустим другой вариант. Предположим, что f(I) больше f(B) при прохождении через F и G (f(I) > 14). В этом случае алгоритм A\* прекратит дальнейшую работу и переместится в вершину В. Но, так как f(C) > f(I), работа алгоритма продолжается именно в вершине I.

1. Поиск в ширину

Поиск в ширину (BFS) - это алгоритм обхода графа, который исследует вершины в порядке их удаления от исходной вершины, где расстояние - это минимальная длина пути от исходной вершины до текущей вершины. Основные шаги алгоритма:

* Выбрать начальную вершину и пометить ее как посещенную.
* Добавить начальную вершину в очередь.
* Пока очередь не пуста, извлечь первую вершину из очереди и посетить ее соседей.
* Если соседняя вершина не была посещена, пометить ее как посещенную и добавить в очередь.
* Повторять шаги 3-4, пока не будут посещены все вершины, достижимые из начальной вершины.

Применение алгоритма BFS:

* Поиск кратчайшего пути между двумя вершинами в невзвешенном графе.
* Поиск всех вершин, достижимых из заданной вершины.
* Поиск компонент связности в графе.
* Поиск циклов в графе.
* Поиск решения задач/игр за наименьшее число ходов.
* Реализация алгоритма BFS:
* Создать массив меток для всех вершин графа.
* Изначально все метки равны false, кроме метки начальной вершины, которая равна true.
* Создать очередь и добавить в нее начальную вершину.
* Пока очередь не пуста, извлекать из нее вершину и посещать ее соседей.
* Если соседняя вершина не была посещена, пометить ее как посещенную и добавить в очередь.

Преимущества алгоритма BFS:

* Простота реализации.
* Гарантировано находит кратчайший путь в невзвешенном графе.

Недостатки алгоритма BFS:

* Неэффективен для поиска пути в больших графах.
* Не работает для графов с отрицательными весами ребер.

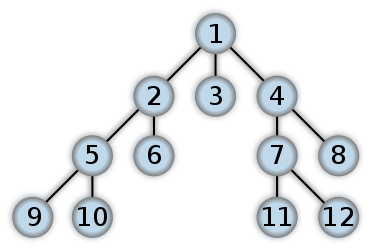


Рисунок 3 - Пример работы алгоритма поиска в ширину

Начиная с узла 1, мы посетим все соседние узлы графа. Сначала мы посетим узел 2, а затем узлы 3 и 4. После посещения узла 4 мы вернемся назад и перейдем к узлам 5, 6.Далее мы посетим узлы 7 и 8. Из 5 мы посетим узлы 9 и 10. Из 7 мы посетим узлы 11 и 12. Каждый посещенный узел помещается в список посещенных. Поскольку очередь пуста, мы завершили обход в ширину графика. Алгоритм BFS будет продолжаться до тех пор, пока не будут посещены все узлы графа.

1. Структурное программирование

Это методология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры из блоков, которые могут быть выполнены последовательно, условно или повторяющимися циклами. Основные принципы структурного программирования включают в себя:

* Разделение программы на логические блоки.
* Использование только трех типов управляющих структур: последовательность, ветвление и цикл.
* Избегание использования операторов безусловного перехода.
* Избегание использования глобальных переменных.

Структурное программирование было разработано в ответ на проблемы, связанные с использованием безусловных переходов и глобальных переменных в программировании. Оно позволяет создавать более надежные и легко поддерживаемые программы. Однако, структурное программирование имеет свои ограничения и не всегда может быть применено в современном программировании. Например, оно не подходит для разработки больших и сложных систем, где требуется более гибкий подход к управлению программным кодом. Структурное программирование также связано с другими методологиями разработки программного обеспечения, такими как объектно-ориентированное программирование (ООП). ООП представляет программу в виде набора объектов, которые могут взаимодействовать друг с другом. ООП позволяет создавать более гибкие и расширяемые программы, чем структурное программирование. Однако, ООП также имеет свои ограничения и не всегда может быть применено в ситуациях, где требуется максимальная производительность и эффективность.

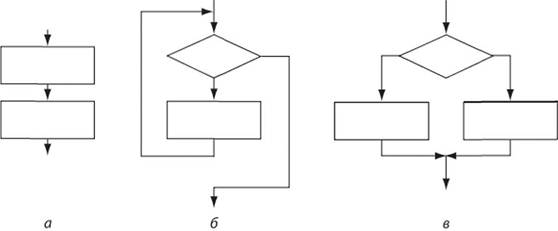


Рисунок 4 - Схемы структурного программирования.

А)Конструкция "следование" (или последовательность) описывает порядок выполнения команд в программе. Программа выполняется пошагово, каждая команда исполняется после предыдущей.

Б)“Цикл” позволяет выполнять повторяющиеся команды множество раз, пока выполняется определенное условие.

Г) "Ветвление" позволяет изменять поток выполнения программы в зависимости от определенных условий.

# Реализация алгоритма

Для начала импортируем модули **sqrt** из **math** и **PriorityQueue** из **queue**.

Определяем функцию **read\_maze**, которая на вход получает текстовый файл с лабиринтом, а возвращает его в виде матрицы **maze**.

Далее, реализуем функцию **get\_neighbors**, которая получает в себя матрицу лабиринта и координаты точки, возвращает список в котором хранятся координаты соседних ячеек, в которые можно перейти.

 Поиск в ширину, задаем функцю **def width\_search(maz**e) которая принимает в себя **maze**,далее задем начальную и конечную точки, **start** и **end**, соответственно. Создание очереди поиска **queue** и множества посещенных точек **[(start, [start])]**. Затем, пока очередь не пуста, извлекается первый элемент, с помощью метода **pop(0**), Затем этот элемент распаковывается в две переменные **current** и **path**. Переменная **current** содержит текущую точку, а переменная **path** содержит путь к этой точке в виде списка координат точек.  Далее проходятся все соседи текущей точки, и если они еще не были посещены, добавляются в очередь поиска. Алгоритм продолжает работу до тех пор, пока не будет найден путь к выходу или очередь не опустеет. В итоге, если путь к выходу найден, возвращается путь в виде списка координат точек, иначе возвращается None.

Данная функция **get\_heuristic(cell, end)** вычисляет эвристическую оценку расстояния от текущей ячейки **cell** до конечной ячейки **end** на двумерной плоскости. Она использует формулу расстояния между двумя точками на плоскости, которая вычисляется как квадратный корень из суммы квадратов разностей координат по осям **x** и **y**.

Функция **way\_to\_exit(maze)** реализует алгоритм поиска кратчайшего пути в лабиринте вместо простой очереди используется приоритетная очередь, которая сортирует элементы по их приоритету. В начале функции определяется начальная и конечная точки лабиринта, затем создается двумерный массив той же формы, что и maze, и заполняется значениями -1. Значение -1 означает, что ячейка еще не была посещена. Затем начальная точка помечается значением 0, и алгоритм распространяется от нее по лабиринту. Каждая ячейка, которая еще не была посещена и имеет соседа с меткой n, помечается значением n+1. Этот процесс продолжается до тех пор, пока не достигнет конечной точки или не перестанет распространяться. Если алгоритм достиг конечной точки, то функция строит путь от конечной точки к начальной, следуя по ячейкам с метками, уменьшающимися на 1. В итоге функция возвращает путь в виде списка координат точек, если путь найден, иначе возвращает None.

Функция **main()** является точкой входа в программу. Она вызывает функции для решения заданной задачи для поиска путей в лабиринте и записи результатов в файлы **"maze-for-me-done(width\_search).txt"** и **"maze-for-me-done(A).txt"**. Функция **main()** не принимает аргументов и не возвращает значений.

# Пример работы кода

**Поиск в ширину:**

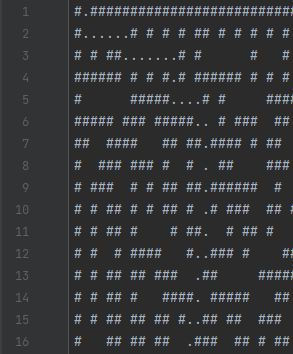
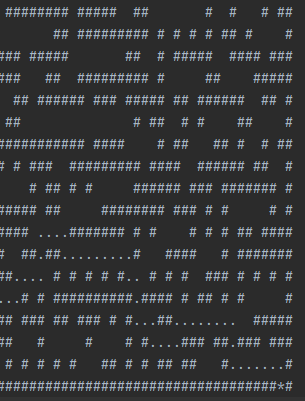
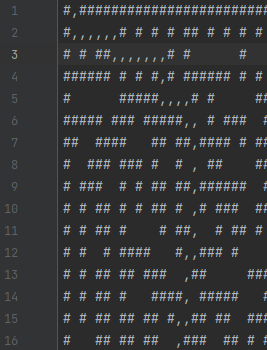
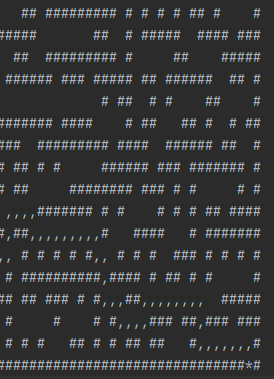
 

Рисунок 5 - Вход в лабиринт (поиск в ширину) Рисунок 6 - Выход из лабиринта (поиск в ширину)

После работы кода мы получаем текстовый файл «maze-for-me-done(width\_search).txt» в котором отражен выход из данного нам лабиринта, путь проведен символом «.»

**Алгоритм A\*:**

** **

*Рисунок 7 - Вход в лабиринт (алгоритм A\*) Рисунок 8 - Выход из лабиринта (алгоритм A\*)*

После работы кода мы получаем текстовый файл «maze-for-me-done(A).txt» в котором отражен выход из данного нам лабиринта, путь проведен символом «,»

# Вывод

В заключении можно отметить, что поиск кратчайшего пути в лабиринте является важной задачей, которая может быть решена с помощью различных алгоритмов, таких как волновой алгоритм, алгоритм A\*, алгоритмы поиска в глубину и ширину, алгоритм Дейкстры, алгоритм Уоршелла-Флойда и алгоритм Левита. Каждый из этих алгоритмов имеет свои особенности и применяется в зависимости от конкретной задачи и требований к ее решению. Решение задачи поиска кратчайшего пути в лабиринте может быть полезно в различных областях, таких как игры, робототехника, логистика и многие другие.

# Список литературы

1. Структурное программирование / [Электронный ресурс.]. URL: http://www.pmbk.ru/lister/035/99/index.shtml
2. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб.: Питер. 2017. – 288 с.: ил. – (Серия «Библиотека программиста»).
3. Рассел, С., Норвиг, П. (2009). Artificial Intelligence: A Modern Approach (3-е изд.). Москва: ДМК Пресс.
4. Сайт технологий программирования: электронный сайт.

URL: https://tproger.ru/

1. Агапов И., «Обход графа: поиск в глубину и поиск в ширину простыми словами на примере JavaScript» [Электронный ресурс]. URL: https://habr.com/ru/articles/504374/
2. Стакановец А.В. АЛГОРИТМЫ НА ГРАФАХ / Станковец А.В. – Текст: электронный // MODERN SCIENCE. — 2020. — №10-2. — С.532-536. — EDN: pdmxpx
3. Кратчайший путь в лабиринте — Алгоритм Ли / [Электронный ресурс.].

URL: https://www.techiedelight.com/ru/lee-algorithm-shortest-path-in-a-maze/

Дата обращения: (27.05.2023)

## Приложение

## Листинг программы

from math import sqrt  
from queue import PriorityQueue  
  
def read\_maze(filename):  
 with open(filename) as file:  
 maze = [[char for char in line.strip()] for line in file]  
 return maze  
  
*#Соседние точки*def get\_neighbors(maze, point: tuple[int, int]):  
 x, y = point  
 neighbors = [(x - 1, y), (x + 1, y), (x, y - 1), (x, y + 1)]  
 neighbors\_current = []  
 for neighbor in neighbors:  
 x, y = neighbor  
 if 0 <= x < len(maze) and 0 <= y < len(maze[0]) and maze[x][y] != "#":  
 neighbors\_current.append(neighbor)  
 return neighbors\_current  
  
*#Поиск в ширину*def width\_search(maze):  
 start = (0, 1)  
 end = (len(maze) - 1, len(maze[0]) - 2)  
 queue = [(start, [start])]  
 visited = set()  
 while queue:  
 current, path = queue.pop(0)  
 if current == end:  
 return path  
 visited.add(current)  
 for neighbor in get\_neighbors(maze, current):  
 if neighbor not in visited:  
 queue.append((neighbor, path + [neighbor]))  
 return None  
  
*#Эвристическое расстояние*def get\_heuristic(cell, end):  
 return sqrt((cell[0] - end[0]) \*\* 2 + (cell[1] - end[1]) \*\* 2)  
  
*#A\**def way\_to\_exit(maze):  
 start\_point = (0, 1)  
 end\_point = (len(maze) - 1, len(maze[0]) - 2)  
 queue = PriorityQueue()  
 queue.put((0, start\_point, [start\_point]))  
 visited\_points = set()  
 while not queue.empty():  
 p, current\_point, path = queue.get()  
 if current\_point == end\_point:  
 return p, path  
 visited\_points.add(current\_point)  
 for i in get\_neighbors(maze, current\_point):  
 if i not in visited\_points:  
 new\_way = path + [i]  
 priority = len(new\_way) + get\_heuristic(i, end\_point)  
 queue.put((priority, i, new\_way))  
 return None  
  
def main():  
 *#Width\_search\_alg* filename = "maze-for-u.txt"  
 maze = read\_maze(filename)  
 way\_1 = width\_search(maze)  
 end = (len(maze) - 1, len(maze[0]) - 2)  
  
 for place in way\_1:  
 maze[place[0]][place[1]] = "."  
 maze[end[0]][end[1]] = "\*"  
 result\_width = ""  
 for j in maze:  
 result\_width += "".join(j) + "\n"  
  
 with open("maze-for-me-done(width\_search).txt", "w") as f:  
 f.write(result\_width)  
  
 *#A\*\_alg* way\_2 = way\_to\_exit(maze)  
 way\_2\_ = way\_2[1]  
  
 for place in way\_2\_:  
 maze[place[0]][place[1]] = ","  
 maze[end[0]][end[1]] = "\*"  
 result\_A = ""  
 for i in maze:  
 result\_A += "".join(i) + "\n"  
  
 with open("maze-for-me-done(A).txt", "w") as z:  
 z.write(result\_A)  
main()