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 Мокеичева Е.В.

Проверил Морозов Н.С.

Нижний Новгород – 2023 г.

Оглавление

[Введение 3](#_Toc135774690)

[**1.** **Теоретическая часть** 4](#_Toc135774691)

[**Алгоритмы поиска пути** 4](#_Toc135774692)

[Поиск в ширину (Breadth-First Search(BFS)) 4](#_Toc135774693)

[Алгоритм А\* (А Star или А «со звездочкой») 9](#_Toc135774694)

[**Структурное программирование** 12](#_Toc135774695)

[**2. Реализация алгоритма** 14](#_Toc135774696)

[**Пример работы алгоритма** 18](#_Toc135774697)

[**Заключение** 19](#_Toc135774698)

[**Список литературы** 20](#_Toc135774699)

[**Приложения** 22](#_Toc135774700)

[Приложение 1 22](#_Toc135774701)

[Листинг программы 22](#_Toc135774702)

# Введение

«Задача поиска кратчайшего пути» в последнее время получила широ­кое распространение, благодаря своему применению для решения множества задач, и программирование не исключение. Алгоритмы обхода графа явля­ются одной из важнейших задач в программировании. Существует множе­ство алгоритмов поиска пути, каждый из которых имеет свои преимущества и недостатки. В данной курсовой работе мы рассмотрим несколько из них: алгоритм A\* и алгоритм поиска в ширину.

Также нами будет рассмотрен принцип структурного программирова­ния, который предполагает последовательное разложение общей функции обработки данных на простые функциональные элементы.

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

**Задачи**

1. Изучить теорию алгоритмов обхода графа: в ширину и А\*;
2. Изучить, что такое структурное программирование;
3. Написать программу на языке Python для поиска маршрута в лабиринте с помощью алгоритма обхода в ширину и А\*.
4. Сохранить результаты обходов лабиринта и получившиеся маршруты в файл.
5. Разобрать работу написанного кода, объяснить принцип работы алгоритмов и структуру программы.
6. **Теоретическая часть**

**Алгоритмы поиска пути**

**Алгоритм** – последовательность чётко определенных действий, выполнение которых ведёт к решению задачи. Алгоритм, записанный на языке машины, есть программа решения задачи. Алгоритм – это совокупность действий, приводящих к достижению результата за конечное число шагов [2].

### Поиск в ширину (Breadth-First Search(BFS))

Поиск в ширину был формально предложен Э. Ф. Муром в контексте поиска пути в лабиринте. Ли независимо открыл тот же алгоритм в контексте разводки проводников на печатных платах [5].

Поиск в ширину позволяет вычислить кратчайшие расстояния выделенной вершины ориентированного графа до всех остальных вершин, и/или построить корневое направленное дерево, расстояния в котором совпадают с расстояниями в исходном графе. Кроме того, поиск в ширину позволяет решать задачу проверки достижимости (существуют ли пути между вершиной источником и остальными вершинами графа) [6].

**Алгоритм BFS**

Стандартная реализация ВFS помещает каждую вершину графа в одну из двух категорий:

1. Посещенные.
2. Не посещенные.

*Цель алгоритма*- пометить каждую вершину, как посещенную, избегая циклов.

Алгоритм работает следующим образом:

1. Начните с размещения любой вершины графа в конце очереди.
2. Возьмите передний элемент очереди и добавьте его в список посещенных.
3. Создайте список смежных узлов этой вершины. Добавьте те, которых нет в списке посещенных, в конец очереди.
4. Продолжайте повторять шаги 2 и 3, пока очередь не опустеет.

Граф может иметь две разные несвязанные части, поэтому, чтобы убедиться, что мы покрываем каждую вершину, мы также можем запустить алгоритм BFS на каждом узле [7].

**Описание алгоритма**

На вход алгоритма подаётся невзвешенный граф и номер стартовой вершины *s*. Граф может быть как ориентированным, так и неориентированным — для алгоритма это не важно.

Основную идею алгоритма можно понимать как процесс «поджигания» графа: на нулевом шаге мы поджигаем вершину *s*, а на каждом следующем шаге огонь с каждой уже горящей вершины перекидывается на всех её соседей, в конечном счёте поджигая весь граф.

Если моделировать этот процесс, то за каждую итерацию алгоритма будет происходить расширение «кольца огня» в ширину на единицу. Номер шага, на котором вершина *v* начинает гореть, в точности равен длине её минимального пути из вершины *s*.

Моделировать это можно следующим образом. Создадим очередь, в которую будут помещаться горящие вершины, а также заведём булевый массив, в котором для каждой вершины будем отмечать, горит она или нет — или иными словами, была ли она уже посещена. Изначально в очередь помещается только вершина *s*, которая сразу помечается горящей.

Затем алгоритм представляет собой такой цикл: пока очередь не пуста, достать из её головы одну вершину *v*, просмотреть все рёбра, исходящие из этой вершины, и если какие-то из смежных вершин *u* ещё не горят, поджечь их и поместить в конец очереди.

В итоге, когда очередь опустеет, мы по одному разу обойдём все достижимые из *s* вершины, причём до каждой дойдём кратчайшим путём. Длины кратчайших путей можно посчитать, если завести для них отдельный массив *d* и при добавлении в очередь пересчитывать по правилу *du*​=*dv*​+1. Также можно компактно сохранить дополнительную информацию для восстановления самих путей, заведя массив «предков», в котором для каждой вершины хранится номер вершины из которой мы в неё попали [4].

**Пример работы BFS**

Давайте посмотрим, как алгоритм «поиска в ширину» работает на примере. Мы используем неориентированный граф с 5 вершинами (Рисунок 1).

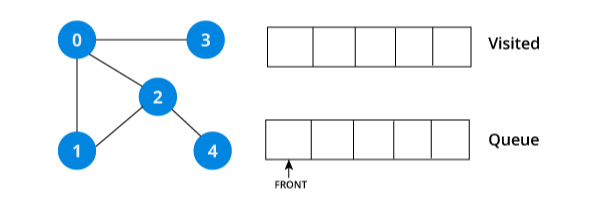


Рисунок 1. Неориентированный граф без посещений

Мы начнем с вершины 0, алгоритм BFS начинается с помещения его в список посещенных и размещения всех смежных вершин в стеке (Рисунок 2).

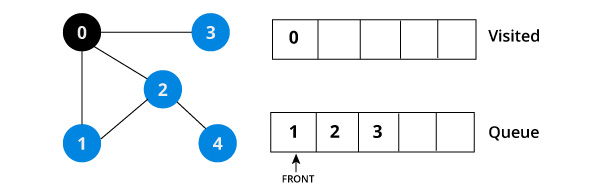
****

Рисунок 2. Посещение вершины 0

Затем мы посещаем элемент в начале очереди, то есть 1, и переходим к соседним узлам (Рисунок 3).

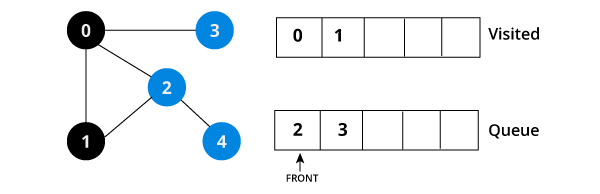


Рисунок 3. Посещение вершины 1

Так как 0 уже был посещен, мы посещаем 2 (Рисунок 4). У вершины 2 есть соседняя не посещенная вершина 4, поэтому мы добавляем ее в конец очереди.

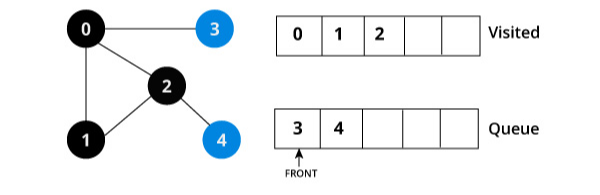


Рисунок 4. Посещение вершины 2

Посещаем 3, которая находится в начале очереди (Рисунок 5). В очереди остается только 4, поскольку единственный соседний узел с 3, то есть 0, уже посещен.

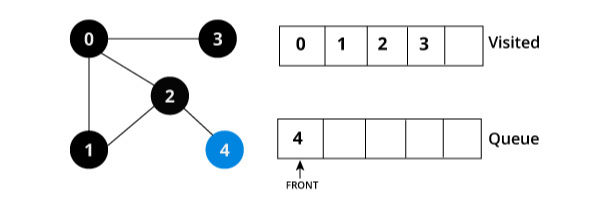


Рисунок 5. Посещение вершины 3

Мы посещаем вершину 4 (Рисунок 6). Поскольку очередь пуста, мы завершили обход в ширину графика [7].

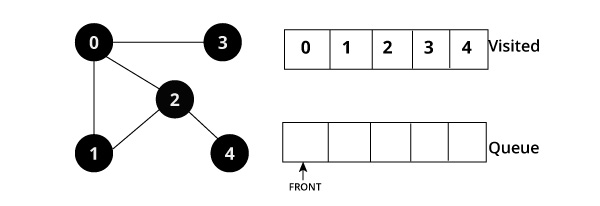


Рисунок 6. Посещение вершины 4

**Для чего нужен BFS**

1. Для решения задач поиска оптимального пути. Классической задачей считается автоматизированный поиск выхода из лабиринта.
2. Для решения задач, связанных непосредственно с теорией графов, например для поиска компонент связности. Эти задачи в свою очередь решаются в Data Science, теории сетей и электронике.
3. Для задач искусственного интеллекта, связанных с поиском решения с минимальным количеством ходов. В таком случае состояния «умной машины» представляются как вершины, а переходы между ними — как ребра.
4. Для оптимизации памяти при обходе графа в некоторых ситуациях, например для некоторых специфических структур.
5. Для работы с информацией в определенных структурах данных, таких как деревья. Их тоже можно обходить с помощью алгоритма BFS, потому что они — подвид графов [11].

### Алгоритм А\* (А Star или А «со звездочкой»)

Впервые описан в 1968 году Питером Хартом, Нильсом Нильсоном и Бертрамом Рафаэлем. Данный алгоритм является расширением алгоритма Дейкстры, ускорение работы достигается за счет эвристики — при рассмотрении каждой отдельной вершины переход делается в ту соседнюю вершину, предположительный путь из которой до искомой вершины самый короткий. При этом существует множество различных методов подсчета длины предполагаемого пути из вершины. Результатом работы также будет кратчайший путь [1].

Алгоритм «A Star» характеризуется тремя важными свойствами:

* оптимальностью — это означает, что алгоритм гарантирует получение лучшего из возможных результатов;
* полнотой — это означает, что алгоритм «A Star» всегда найдет решение, если оно существует;
* эффективностью — на сегодняшний день нет других алгоритмов, которые смогут найти кратчайший путь быстрее, чем «A Star», применяя эвристическую функцию.

Алгоритм поиска «A Star» несет в себе следующую идею: изначально он всегда посещает вершины, которые, скорее всего, ведут по кратчайшему пути к цели. Такие вершины он определяет по формуле:

F(x) = G(x) + H(x) , где:

F(x) — это функция для вершины; чем меньше функция, тем «ближе» вершина стоит в очереди для посещения; данная функция оценивает минимальную стоимость перехода от вершины к вершине;

G(x) — это стоимость пути от первоначальной вершины и до любой другой;

H(x) — это эвристический показатель стоимости пути от вершины «х» и до конечной вершины.

Вычисляя F(x), за «х» берется соседняя вершина с уже посещенной. Поэтому получается, что алгоритм вычисляет стоимость всего пути у всех соседних вершин и останавливает выбор на той, у которой стоимость минимальна [8].

**Как работает 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.

Алгоритм «A Star» находит кратчайший путь между вершинами, основываясь на стоимости и «весе» ребер. Поэтому путь, который находит «A Star», можно по праву назвать «самым быстрым» или «самым простым». По этой причине алгоритм «A Star» очень часто применяется как раз для планирования кратчайших путей, однако его также часто применяют в играх [8].

**Недостаток алгоритма “A Star”**

Алгоритм «A Star» довольно популярен при поиске кратчайшего пути, однако его применение ограничивается его главным недостатком — потребностью в большом количестве памяти, потому что алгоритм «A Star» хранит всю информацию об известных и исследованных вершинах. Из этого получается, что «A Star» не всегда пригоден для использования, поэтому нужно знать о его альтернативах, которым не нужен большой объем памяти для работы. Такими альтернативами являются: алгоритм Дейкстры, «жадный» алгоритм, IDA Star, RBFS, MA Star [8].

**Структурное программирование**

Практика программирования показала необходимость научно обоснованной методологии разработки и документирования алгоритмов и программ. Эта методология должна касаться анализа исходной задачи, разделения ее на достаточно самостоятельные части и программирования этих частей по возможности независимо друг от друга. Такой методологией, зародившейся в начале 70-х годов и получившей в последнее время широкое распространение, является структурное программирование. По своей сути оно воплощает принципы системного подхода в процессе создания и эксплуатации программного обеспечения ЭВМ [3].

Структурное программирование стало популярным в 1970-х годах, когда были разработаны языки программирования, такие как ALGOL, FORTRAN, COBOL. Эти языки предлагали программистам использовать только ограниченное количество структур управления и отказаться от операторов типа GOTO, которые могли бы создать запутанный и трудночитаемый код.

**Структурное программирование** – технология разработки программного обеспечения, в основе которой лежит представление программы в виде иерархической структуры логически целостных фрагментов (блоков).

**Принципы структурного программирования**

* Любая программа строится из трёх базовых управляющих конструкций: последовательность, ветвление, цикл.
* В программе базовые управляющие конструкции могут быть вложены друг в друга произвольным образом.
* Повторяющиеся фрагменты программы можно оформить в виде подпрограмм (процедур и функций). В виде подпрограмм можно оформить логически целостные фрагменты программы, даже если они не повторяются.
* Все перечисленные конструкции должны иметь один вход и один выход.
* Разработка программы ведётся пошагово, методом «сверху вниз» (метод последовательной детализации). Программирование сверху вниз или снизу вверх позволяют человеку постоянно мыслить на предметном уровне, не вникая в уровень конкретных операторов и переменных [10].

Благодаряструктурному программированиюповышается надежность программ (благодаря хорошему структурированию при проектировании, программа легко поддается тестированию и не создает проблем при отладке),повышается эффективность программ (структурирование программы позволяет легко находить и корректировать ошибки, а отдельные подпрограммы можно переделывать (модифицировать) независимо от других),уменьшается время и стоимость программной разработки, улучшается читабельность программ [3].

**Недостатки** структурного программирования проявляются при усложнении программ. Так, структурное программирование  не позволяет в доста­точной степени упростить большие сложные программы. Это связано с неограниченностью доступа функций к глобальным данным и принципом разделения данных и функций [9].

# 2. Реализация алгоритма

В данной программе будет функция, при запуске которой будет работать связно вся программа. Такой функцией будет функция main, которая будет прописана в конце программного кода. Для того чтобы облегчить разработку и отладку, а также не загромождать главную функцию, для каждого действия будет своя функция.

Импортирую класс deque из модуля collections в Python. deque (double-ended queue) - это тип контейнера данных, который предоставляет быстрый доступ к элементам как с начала, так и с конца очереди.

**Функция readMaze** читает файл с лабиринтом и преобразует его в двумерный список, где стены обозначены цифрой "1", а проходы - цифрой "0". Сначала функция открывает файл с именем "maze-for-u.txt". Затем функция читает содержимое файла методом read() и разбивает его на строки с помощью метода split("\n"). Для каждой строки функция заменяет символ "#" на "1" и символ " " на "0" с помощью методов replace().

**Функция key** запрашивает у пользователя координаты ключевой точки в лабиринте. Функция принимает на вход двумерный массив maze, который представляет собой лабиринт, и запрашивает у пользователя координаты X и Y ключевой точки в пределах размеров лабиринта. Если введенные значения находятся в диапазоне от 0 до размера лабиринта по соответствующей оси, и значение в этой точке не равно 1 (то есть не является стеной), то функция возвращает кортеж с координатами ключевой точки. Если введенные значения не соответствуют условиям, то функция выводит сообщение об ошибке и запрашивает координаты еще раз. Если пользователь вводит некорректное значение (например, строку вместо числа), то функция также выводит сообщение об ошибке и запрашивает координаты еще раз.

**Функция end** принимает на вход двумерный массив maze, который представляет собой лабиринт, и ищет координаты выхода из лабиринта. Для этого она проходит по всем элементам последней строки лабиринта ищет элемент со значением '0'. Если такой элемент найден, то функция возвращает кортеж с координатами этой точки.

**Функция start** принимает на вход двумерный массив maze и ищет координаты стартовой точки в лабиринте. Для этого она проходит по всем элементам первой строки лабиринта и ищет элемент со значением '0'. Если такой элемент найден, то функция возвращает кортеж с координатами этой точки.

Далее функциия bfs и функция get\_neighbors реализуют алгоритм поиска кратчайшего пути в лабиринте, используя алгоритм поиска в ширину (BFS).

**Функция bfs** принимает на вход двумерный массив maze, представляющий лабиринт, координаты начальной вершины start и координаты конечной вершины end. Внутри функции создается очередь queue, в которую добавляется начальная вершина start. Также создается множество visited, в которое добавляется начальная вершина, чтобы не посещать ее повторно. Создается словарь path, который будет хранить путь до каждой посещенной вершины.

Далее начинается основной цикл while queue, который выполняется до тех пор, пока очередь не станет пустой. Из очереди извлекается первый элемент vertex. Если этот элемент равен конечной вершине end, то путь найден и функция возвращает список координат пути path[vertex]. Иначе происходит перебор всех соседних вершин neighbor с помощью функции get\_neighbors.

**Функция get\_neighbors** принимает на вход двумерный массив maze и координаты вершины vertex. Она возвращает список координат соседних вершин, которые являются коридорами (то есть имеют значение 0 в массиве maze). Для этого функция создает список candidates с четырьмя соседними вершинами (верхняя, нижняя, левая и правая) и проходит по каждой из них. Если координаты вершины находятся в пределах границ лабиринта и значение в массиве maze равно 0, то эта вершина добавляется в список result.

В функции bfs проверяется, не посещалась ли уже вершина neighbor. Если нет, то она добавляется в множество visited, в очередь queue и в словарь path, который хранит путь до этой вершины. Путь до этой вершины формируется путем конкатенации пути до предыдущей вершины и текущей вершины: path[neighbor] = path[vertex] + [neighbor].

Если цикл while завершается без нахождения пути до конечной вершины, то функция возвращает None.

Таким образом, данный код реализует алгоритм поиска кратчайшего пути в лабиринте с помощью алгоритма BFS и возвращает список координат этого пути.

Далее я реализую алгоритм A\* для поиска кратчайшего пути в лабиринте, заданном в виде двумерного массива.

**Функция get\_neighbors(maze, vertex)** находит всех соседей вершины vertex в лабиринте maze, которые являются проходимыми (значение "0" в массиве maze). Таким образом, данная функция позволяет определить все возможные направления движения из текущей вершины.

**Функция heuristic(start, end)** вычисляет эвристическую оценку расстояния от вершины start до вершины end. Эта функция используется для оценки расстояния от текущей вершины до целевой вершины.

**Алгоритм A\* (def A\_star)** использует эвристическую оценку для выбора следующей вершины на пути к целевой вершине end. В процессе выполнения алгоритма, для каждой вершины хранятся значения g\_score и f\_score, которые представляют длину кратчайшего пути от начальной вершины до текущей вершины и оценку длины пути от начальной вершины через текущую вершину до целевой вершины соответственно. Алгоритм A\* выбирает следующую вершину с наименьшим значением f\_score, которое является суммой g\_score и эвристической оценки расстояния от текущей вершины до целевой вершины.

В процессе выполнения алгоритма, список open\_set содержит вершины, которые еще не были обработаны, список closed\_set содержит вершины, которые уже были обработаны, а словарь came\_from содержит информацию о том, откуда пришли в каждую вершину.

Алгоритм A\* продолжает работу до тех пор, пока список open\_set не станет пустым или пока не будет найден кратчайший путь до целевой вершины. Если кратчайший путь был найден, то алгоритм возвращает список вершин, составляющих этот путь. Если кратчайший путь не был найден, то алгоритм возвращает None.

**Функция main()**. Использует такие функции как:

- readMaze() – функция для создания лабиринта

- key(maze) – функция для объявления ключевой точки

- bfs(maze,start(maze),keyPoint) – функция для нахождения пути от старта до ключевой точки алгоритма поиска в ширину

- A\_star(maze, keyPoint, end(maze)) – функция для нахождения пути от ключевой точки до выхода алгоритмом A\*

Далее заносим путь от старта до ключа, от ключа до выхода и ключевую точку в лабиринт. В конце генерируем файл с пройденным лабиринтом. Полученный результат заносится в файл “maze-for-me-done.txt ”.

# Пример работы алгоритма

При запуске программы она просит ввести координаты ключевых точек.

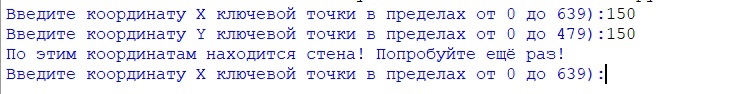
****

Рисунок 7 Пример неудачного ввода

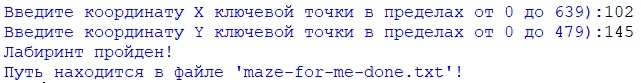
****

Рисунок 8 Пример корректного ввода

****

Рисунок 9 Отрывок из файла по введенным координатам из Рисунка 8

# Заключение

В результате выполнения данной работы была изучена теория алгоритмов обхода в ширину и A\*, а так же основы структурного программирования.

После чего была написана программа на языке Python для поиска маршрута в лабиринте с помощью алгоритма обхода в ширину и A Star. Результат программы был сохранен в файл “maze-for-me-done.txt”. Число строк кода программы составляет примерно 128 строк.

В результате работы с кодом, было выявлено, что алгоритм A\* более эффективен, он гораздо быстрее находит оптимальный путь, в отличие от поиска в ширину. Что объясняется тем, что алгоритм A\* использует эвристику для принятия решений.

В результате выполнения данной работы были получены теоретические знания по структурному программированию и области алгоритмов поиска пути, которые были в дальнейшем закреплены на практике, в ходе написания программы. Данные знания будут полезны при решении дальнейших задач.

Цель курсовой была достигнута. Была создана программа, которая может находить путь в лабиринте с помощью алгоритмов обхода графа в ширину и A Star.

# Список литературы

1. Алгоритмы поиска пути в графе // tprger.ru / [Электронный ресурс]. URL: <https://tproger.ru/articles/pathfindings/> (Дата обращения: 20.04.2023)
2. Лекция 1: Понятие алгоритма. Виды алгоритмов // intuit.ru / [Электронный ресурс]. URL:  <https://intuit.ru/studies/courses/16740/1301/lecture/25624?ysclid=lhztsaw25h126134258> (Дата обращения: 20.04.2023)
3. Основы структурного программирования. Методы структурного программирования // stepik.org / [Электронный ресурс]. URL: <https://stepik.org/lesson/326788/step/1> (Дата обращения: 22.04.2023)
4. Поиск в ширину // ru.algorithmica.org / [Электронный ресурс]. URL: <https://ru.algorithmica.org/cs/shortest-paths/bfs/?ysclid=lhzw7h0lnv146598182> (Дата обращения: 21.04.2023)
5. Поиск в ширину | Формальное описание // pos.jofo.me / [Электронный ресурс]. URL: <https://pos.jofo.me/630473.html> (Дата обращения:21.04.2023)
6. Поиск в ширину (BFS) // algowiki-project.org / [Электронный ресурс]. URL: <https://algowiki-project.org/ru/%D0%9F%D0%BE%D0%B8%D1%81%D0%BA_%D0%B2_%D1%88%D0%B8%D1%80%D0%B8%D0%BD%D1%83_(BFS)> (Дата обращения: 21.04.2023)
7. Поиск в ширину (Breadth first search, BFS) // evileg.com / [Электронный ресурс]. URL: <https://evileg.com/ru/post/512/> (Дата обращения: 21.04.2023)
8. Поисковый алгоритм A Star: что это и как эффективно его использовать? // codernet.ru / [Электронный ресурс]. URL: <https://codernet.ru/articles/web/poiskovyij_algoritm_a_star_chto_eto_i_kak_effektivno_ego_ispolzovat/?ysclid=lhzwqkw5c486421032> (Дата обращения: 27.04.2023)
9. Структурное программирование // labs-org.ru / [Электронный ресурс]. URL: <https://labs-org.ru/strukturnoe-programmirovanie/?ysclid=lhyqmfrbw7790053486> (Дата обращения:22.04.2023)
10. Структурное программирование // skobelevserg.jimdofree.com / [Электронный ресурс]. URL: <https://skobelevserg.jimdofree.com/%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0-1/11-%D0%BA%D0%BB%D0%B0%D1%81%D1%81-%D1%84%D0%B3%D0%BE%D1%81/%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D0%BD%D0%BE%D0%B5-%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5/> (Дата обращения: 22.04.2023)
11. BFS // blog.skillfactory.ru / [Электронный ресурс]. URL: <https://blog.skillfactory.ru/glossary/bfs/?ysclid=lhzwdyefyl823868331> (Дата обращения: 21.04.2023)

**Приложения**

## Приложение 1

### Листинг программы

from collections import deque

# Импорт с заменой символов и последующим превращением в двумерный массив исходного файла

def readMaze(file="maze-for-u.txt"):

maze = [list(line.replace("#", "1").replace(" ", "0")) for line in open(file).read().split("\n")[:-1]]

return maze

# Определение ключевой точки

def key(maze):

while 1:

try:

x = int(input(f"Введите координату X ключевой точки в пределах от 0 до {len(maze)-1}):"))

if x >= 0 and x < len(maze):

y = int(input(f"Введите координату Y ключевой точки в пределах от 0 до {len(maze[0]) - 1}):"))

if y >= 0 and y < len(maze[0]):

if str(maze[x][y]) == "1":

print("По этим координатам находится стена! Попробуйте ещё раз!")

else:

return (x, y)

else:

print("Введённое значение вне диапазона! Попробуйте ещё раз!")

continue

else:

print("Введённое значение вне диапазона! Попробуйте ещё раз!")

continue

except Exception:

print("Вы ввели неверное значение! Попробуйте ещё раз.")

# Функция определения координат выхода

def end(maze):

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

if maze[len(maze)-1][y] == '0':

return (len(maze)- 1,y)

# Функция определения координат старта

def start(maze):

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

if maze[0][y] == '0':

return (0,y)

# Алгоритм поиска в ширину

def bfs(maze, start, end):

queue = deque([start])

visited = set([start])

path = {start: [start]}

while queue:

vertex = queue.popleft()

if vertex == end:

return path[vertex] # путь найден

for neighbor in get\_neighbors(maze, vertex):

if neighbor not in visited:

visited.add(neighbor)

queue.append(neighbor)

path[neighbor] = path[vertex] + [neighbor]

return None # путь не найден

# Функция для нахождения свободных ячеек поблизости

def get\_neighbors(maze, vertex):

rows, cols = len(maze), len(maze[0])

r, c = vertex

candidates = [(r-1,c), (r+1,c), (r,c-1), (r,c+1)]

result = []

for row, col in candidates:

if 0 <= row < rows and 0 <= col < cols and maze[row][col] == "0":

result.append((row, col))

return result

# Эвристическая оценка расстояния от вершины до вершины

def heuristic(start, end):

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

# Алгоритм А\*

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

open\_set = [start]

closed\_set = []

came\_from = {}

g\_score = {start:0}

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

while len(open\_set) > 0:

current = min(open\_set, key=lambda x:f\_score[x])

if current == end:

path = []

while current in came\_from:

path.append(current)

current = came\_from[current]

path.append(start)

return list(reversed(path))

open\_set.remove(current)

closed\_set.append(current)

for neighbor in get\_neighbors(maze, current):

if neighbor in closed\_set:

continue

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

if neighbor not in open\_set:

open\_set.append(neighbor)

elif tentative\_g\_score >= g\_score[neighbor]:

continue

came\_from[neighbor] = current

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

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

return None

def main():

# Создаём лабиринт

maze = readMaze()

# Объявляем ключевую точку

keyPoint = key(maze)

# Находим путь от старта до ключевой точки алгоритмом поиска в ширину

pathBfs = bfs(maze,start(maze),keyPoint)

# Находим путь от ключевой точки до выхода алгоритмом А\*

pathAstar = A\_star(maze, keyPoint, end(maze))

# Заносим путь от старта до ключа в лабиринт

for point in pathBfs:

maze[point[0]][point[1]] = "."

# Заносим путь от ключа до выхода в лабиринт

for point in pathAstar:

maze[point[0]][point[1]] = ","

# Заносим ключевую точку в лабиринт

maze[keyPoint[0]][keyPoint[1]] = "\*"

# Генерируем файл с пройдённым лабиринтом

with open("maze-for-me-done.txt", "w") as f:

f.write('\n'.join([''.join(map(str, line)) for line in maze]))

print("Лабиринт пройден!\nПуть находится в файле 'maze-for-me-done.txt'!")

# Запуск работы программы

main()